4tH, the *friendly* Forth compiler

J.L. Bezemer

June 4, 2017

Contents

1 What's new **17**

I Getting Started **19**

2 Overview **20**

2.1 Introduction 20

2.2 History 20

2.3 Applications 21

2.4 Architecture 22

2.4.1 The 4tH language 23

2.4.2 H-code 24

2.4.3 H-code compiler 25

2.4.4 Error handling 26

2.4.5 Interfacing with C 26

3 Installation Guide **27**

3.1 About this package 27

3.1.1 Example code 27

3.1.2 Main program 27

3.1.3 Unix package 28

3.1.4 Linux package 29

3.1.5 Android package 32

3.1.6 MS-DOS package 35

3.1.7 MS-Windows package 36

3.2 Setting up your working directory 37

3.3 Now what? 38

3.4 Pedigree 38

3.5 Contributors 39

3.6 Questions . . . 39
3.6.1 4tH website . . . 39
3.6.2 4tH Google group . . . 39
3.6.3 Newsgroup . . . 40

4 A guided tour **41**
4.1 4tH interactive . . . 41
4.2 Starting up 4tH . . . 41
4.3 Running a program . . . 41
4.4 Starting an editing session . . . 42
4.5 Writing your first 4tH program . . . 43
4.6 A more complex program . . . 46
4.7 Advanced features . . . 50
4.8 Suspending a program . . . 57
4.9 Calculator mode . . . 59
4.10 Epilogue . . . 59

5 Frequently asked questions **60**

II Primer **62**

6 Introduction **63**

7 4tH fundamentals **64**
7.1 Making calculations without parentheses . . . 64
7.2 Manipulating the stack . . . 65
7.3 Deep stack manipulators . . . 66
7.4 Passing arguments to functions . . . 66
7.5 Making your own words . . . 67
7.6 Adding comment . . . 68
7.7 Text-format of 4tH source . . . 68
7.8 Displaying string literals . . . 68
7.9 Creating variables . . . 69
7.10 Using variables . . . 69
7.11 Built-in variables . . . 69
7.12 What is a cell? . . . 70
7.13 What is a literal expression? . . . 70
7.14 Declaring arrays of numbers . . . 70

7.15 Using arrays of numbers 71
7.16 Copying arrays of numbers 71
7.17 Declaring and using constants 71
7.18 Built-in constants 72
7.19 Using booleans 73
7.20 IF-ELSE constructs 73
7.21 FOR-NEXT constructs 73
7.22 WHILE-DO constructs 74
7.23 REPEAT-UNTIL constructs 75
7.24 Infinite loops 75
7.25 Including source files 75
7.26 Getting a number from the keyboard 76

8 4tH arrays 77

8.1 Aligning numbers 77
8.2 Creating arrays of constants 77
8.3 Using arrays of constants 77
8.4 Using values 78
8.5 Creating string variables 78
8.6 What is an address? 79
8.7 String literals 79
8.8 String constants 80
8.9 Initializing string variables 80
8.10 Initializing a NULL string variable 81
8.11 Getting the length of a string variable 81
8.12 Printing a string variable 81
8.13 Copying a string variable 82
8.14 The string terminator 82
8.15 Slicing strings 82
8.16 Appending strings 83
8.17 Comparing strings 84
8.18 Finding a substring 84
8.19 Replacing substrings 86
8.20 Deleting substrings 86
8.21 Removing trailing spaces 86
8.22 Removing leading spaces 87
8.23 Upper and lower case 87

8.24 String literals and string variables 88
8.25 Printing individual characters 88
8.26 Distinguishing characters 89
8.27 Getting ASCII values 89
8.28 Printing spaces 89
8.29 Fetching individual characters 90
8.30 Storing individual characters 91
8.31 Getting a string from the keyboard 92

9 Character Segment **93**

9.1 The Character Segment 93
9.2 What is the TIB? 93
9.3 What is the PAD? 94
9.4 How do I use TIB and PAD? 94
9.5 Simple parsing 94
9.6 Converting a string to a number 96
9.7 Controlling the radix 96
9.8 Pictured numeric output 99
9.9 `printf()` like formatting 102
9.10 Converting a number to a string 103
9.11 Aborting a program 104
9.12 Opening a file 105
9.13 Reading and writing from/to a file 106
9.14 Closing a file 107
9.15 Writing text-files 107
9.16 Reading text-files 108
9.17 Reading long lines 108
9.18 Reading binary files 109
9.19 Writing binary files 109
9.20 Reading and writing block files 110
9.21 Parsing textfiles 111
9.22 Parsing binary files 112
9.23 Parsing comma-delimited files 113
9.24 Parsing fixed-width text files 114
9.25 Advanced parsing 116
9.26 Appending to existing files 116
9.27 Using pipes 117

9.28 Opening a file in read/write mode 118
9.29 Using random access files 118
9.30 The layout of the I/O system 120
9.31 Using a printer 121
9.32 The layout of the Character Segment 121

10 Integer Segment and Code Segment 123

10.1 The Code Segment 123
10.2 The address of a colon-definition 123
10.3 Vectored execution 124
10.4 Execution tokens 125
10.5 The Integer Segment 125
10.6 A portable way to access application variables 126
10.7 Returning a result to the host program 126
10.8 Using commandline arguments 127
10.9 The layout of the Variable Area 127
10.10 The stacks 128
10.11 Saving temporary values 129
10.12 The Return Stack and the DO..LOOP 130
10.13 Other Return Stack manipulations 131
10.14 Altering the flow with the Return Stack 131
10.15 Leaving a colon-definition 132
10.16 The layout of the Stack Area 133
10.17 Booleans and numbers 134
10.18 Using ’ with other names 135
10.19 Deleting files 136
10.20 Querying environment variables 136
10.21 What is not implemented 137
10.22 Known bugs and limitations 137

11 Advanced programming 138

11.1 Compiletime calculations 138
11.2 Conditional compilation 139
11.3 Checking the environment at compiletime 141
11.4 Checking a definition at compiletime 142
11.5 Exceptions 142
11.6 Enumerations 145

11.7 Forward declarations 145
11.8 Recursion 147
11.9 Private declarations 147
11.10 Aliases 148
11.11 Changing behavior of data 149
11.12 Multidimensional arrays 151
11.13 Binary string constants 151
11.14 Binary string variables 153
11.15 Records and structures 153
11.16 Unions 156
11.17 Complex control structures 157
11.18 Optimization 158
11.19 Static variable pointers 161
11.20 Assertions 162
11.21 Breakpoints 164
11.22 Debugging 165
11.23 Running 4tH programs from the Unix shell 166
11.24 Embedding 4tH programs in a batch file 167

12 Standard libraries 168

12.1 Adding your own library 168
12.2 Adding templates 170
12.3 Parsing the command line 172
12.4 Mixing character and number data 173
12.5 Dynamic memory allocation 174
12.6 Tweaking dynamic memory 175
12.7 Using two heaps 177
12.8 Garbage collection 178
12.9 Application stacks 179
12.10 Local variables 180
12.11 Random numbers 182
12.12 Sorting 184
12.13 Bitfields 186
12.14 Bit arrays 187
12.15 Associative arrays (using hash tables) 188
12.16 Associative arrays (using binary trees) 191
12.17 Lookup tables with integer keys 191

12.18 Lookup tables with string keys . . . 194
12.19 Lookup tables with multiple keys . . . 195
12.20 Lookup tables with duplicate keys . . . 196
12.21 Fixed point calculation . . . 197
12.22 Double numbers . . . 199
12.23 Floating point numbers (unified stack) . . . 201
12.24 Floating point numbers (separate stack) . . . 202
12.25 Floating point functions . . . 205
12.26 Floating point configurations . . . 206
12.27 Forth Scientific Library . . . 207
12.28 Statistical functions . . . 207
12.29 Numerical integration . . . 208

13 Special libraries **214**
13.1 Infix formula translation . . . 214
13.2 Evaluating infix formulas at runtime . . . 215
13.3 Converting infix formulas . . . 216
13.4 Portable Bitmap graphics . . . 217
13.5 Turtle graphics . . . 220
13.6 Annotating Portable Bitmap images . . . 221
13.7 Color Palettes . . . 222
13.8 3D plotting . . . 223
13.9 3D turtle graphics . . . 226
13.10 Interpreters . . . 226
13.11 Menus . . . 228
13.12 Finite state machines . . . 230
13.13 Virtual memory . . . 232
13.14 Triple numbers . . . 233
13.15 Timers . . . 234
13.16 Time & date . . . 235
13.17 Tokenizing strings . . . 236
13.18 Regular expressions . . . 238
13.19 String pattern matching . . . 240
13.20 Wildcard pattern matching . . . 243
13.21 Escape characters . . . 244
13.22 Chinese characters . . . 245
13.23 Managing INI files . . . 246

13.24 Extract, transform and load . . . 248
13.25 Writing spreadsheet files . . . 249
13.26 Writing other tabular formats . . . 251
13.27 Writing LaTeX files . . . 252
13.28 Writing RTF files . . . 254
13.29 Writing HTML files . . . 255
13.30 Converting to XML and HTML . . . 257
13.31 Databases . . . 258
13.32 Indexing a database . . . 261
13.33 Binding the indexes . . . 262
13.34 Speech synthesis . . . 264
13.35 GUI applications . . . 265
13.36 Card games . . . 266

14 Preprocessor libraries **268**
14.1 Introduction . . . 268
14.2 Stack instructions . . . 269
14.3 Coroutines . . . 270
14.4 Interpretation . . . 271
14.5 Closures . . . 272
14.6 Object orientation . . . 273
14.6.1 Encapsulation . . . 273
14.6.2 Subtype polymorphism . . . 275
14.6.3 Inheritance . . . 277
14.6.4 Determining the type and size of an object . . . 279
14.7 This is the end . . . 280

III Reference guide **281**

15 Glossary **282**

16 Editor manual **382**
16.1 Introduction . . . 382
16.2 Selecting a screen and input of text . . . 382
16.3 Line editing . . . 382
16.4 Line editing commands . . . 383
16.5 Screen editing commands . . . 383
16.6 Cursor control and string editing . . . 383

16.7 Commands to position the cursor 384
16.8 String editing commands 384
16.9 Saving and exiting 384
16.10 Calculator mode 384

17 Shell manual **386**
17.1 Introduction 386
17.2 Loading and saving 387
17.3 Task management 387
17.4 Scripting 387
17.5 Stack, I/O and arithmetic 388

18 Preprocessor manual **389**
18.1 Introduction 389
18.2 Macros 390
18.2.1 Back quoted strings 390
18.2.2 Registers 390
18.2.3 Parsing strings 391
18.2.4 The string stack 391
18.2.5 Phony variables 392
18.2.6 Branching and looping 392
18.2.7 Functions 393
18.3 Invocation (script) 394
18.3.1 Options 394
18.4 Invocation (executable) 395
18.4.1 Options 395
18.5 Preprocessor commands 396
18.6 Error messages 396
18.7 Known bugs and limitations 397
18.8 Preprocessor libraries 398

19 uBasic manual **399**
19.1 Introduction 399
19.2 Statements 399
19.3 Functions 403
19.4 Error messages 404
19.4.1 System messages 405

20 ANS Forth statement **406**
20.1 ANS-Forth Label . . . 406
20.2 Unsupported CORE words . . . 407
20.3 Supported ANS Forth word sets . . . 408
20.3.1 Core Extensions word set . . . 408
20.3.2 Block Extensions word set . . . 409
20.3.3 Double number word set . . . 409
20.3.4 Double number Extensions word set . . . 410
20.3.5 Facility Extensions word set . . . 410
20.3.6 File-Access word set . . . 410
20.3.7 File-Access Extensions word set . . . 410
20.3.8 Floating-Point word set . . . 411
20.3.9 Floating-Point Extensions word set . . . 411
20.3.10 Programming-Tools word set . . . 412
20.3.11 Programming-Tools Extensions word set . . . 412
20.3.12 String word set . . . 413
20.3.13 XCHAR word set . . . 413
20.3.14 XCHAR Extensions word set . . . 413

21 Porting guide **414**
21.1 Introduction . . . 414
21.2 General guidelines . . . 414
21.3 Differences between 4tH and ANS-Forth . . . 415
21.3.1 Strings . . . 415
21.3.2 Double numbers . . . 415
21.3.3 Booleans . . . 415
21.3.4 CREATE..DOES> . . . 417
21.3.5 HERE . . . 418
21.3.6 2>R and 2R> . . . 418
21.3.7 Interpretation and compilation mode . . . 418
21.3.8 BEGIN..WHILE..REPEAT . . . 419
21.3.9 CASE..OF..ENDOF..ENDCASE . . . 419
21.3.10 DO..LOOP . . . 421
21.3.11 I/O . . . 422
21.4 Easy 4tH . . . 423
21.4.1 Enabling the String Space . . . 424
21.4.2 The structure of Easy 4tH . . . 424
21.5 The preprocessor . . . 425
21.6 Converting ANS-Forth programs to 4tH . . . 426

22 Errors guide **427**
22.1 How to use this manual . . . 427
22.2 Interpreter (exec_4th) . . . 427
22.3 Compiler (comp_4th) . . . 433
22.4 Loader (load_4th) . . . 439
22.5 Saver (save_4th) . . . 441

23 4tH library **442**
23.1 4tH library files . . . 442
23.2 Library dependencies . . . 456

24 Change log **463**
24.1 What's new in version 3.62.4 . . . 463
24.2 What's new in version 3.62.3 . . . 464
24.3 What's new in version 3.62.2 . . . 467
24.4 What's new in version 3.62.1 . . . 470
24.5 What's new in version 3.62.0 . . . 472
24.6 What's new in version 3.61.5 . . . 474
24.7 What's new in version 3.61.4 . . . 476
24.8 What's new in version 3.61.3 . . . 479
24.9 What's new in version 3.61.2 . . . 481
24.10 What's new in version 3.61.1 . . . 483
24.11 What's new in version 3.61.0 . . . 484
24.12 What's new in version 3.60.1 . . . 487
24.13 What's new in version 3.60.0 . . . 489
24.14 What's new in version 3.5d, release 3 . . . 491
24.15 What's new in version 3.5d, release 2 . . . 493
24.16 What's new in version 3.5d . . . 493
24.17 What's new in version 3.5c, release 3 . . . 495
24.18 What's new in version 3.5c, release 2 . . . 495
24.19 What's new in version 3.5c . . . 497
24.20 What's new in version 3.5b, release 2 . . . 498
24.21 What's new in version 3.5b . . . 499
24.22 What's new in version 3.5a, release 2 . . . 500
24.23 What's new in version 3.5a . . . 501
24.24 What's new in version 3.3d, release 2 . . . 505
24.25 What's new in version 3.3d . . . 507
24.26 What's new in version 3.3c . . . 509
24.27 What's new in version 3.3a . . . 510
24.28 What's new in version 3.2e . . . 512
24.29 What's new in version 3.1d . . . 514

IV Development guide 518

25 Compiling the source 519

25.1 Introduction 519
25.2 Recommended and preferred compilers 519
25.3 Compiling 4th 520
25.4 Compiling the library 521
25.5 Choosing Makefiles 522
25.6 Shared library 523
25.7 64-bit platforms 523
25.8 Regenerating the include files 524
25.9 Optimizations 525
25.10 GCC specific optimizations 525
25.11 Using the library 526

26 Using the 4tH API 528

26.1 Introduction 528
26.2 A sample program 528
26.3 A first look at open_4th() 530
26.4 A closer look at H-code 531
26.5 A closer look at HX-code 531
26.6 A first look at comp_4th() 535
26.7 A first look at exec_4th() 536
26.8 A first look at free_4th() 538
26.9 A first look at save_4th() 540
26.10 A first look at load_4th() 541
26.11 A first look at error-trapping 541
26.12 A first look at dump_4th() 543
26.13 A first look at cgen_4th() 545
26.14 Converting HX-files 546
26.15 A first look at fetch_4th() 546
26.16 A first look at store_4th() 547
26.17 Examples of embedded HX code 547
26.18 Suspended execution 549
26.19 Useful variables 554

27 Modifying 4tH **555**

27.1 Introduction 555
27.2 Understanding 4tHs versioning 555
27.3 A closer look at comp_4th() 556
27.4 Adding a constant 558
27.5 Adding a word 559
27.6 A closer look at exec_4th() 561
27.7 A first look at name_4th() 563
27.8 Extending the compiler 564
27.9 Making aliases 566
27.10 Giving a name to an application variable 566
27.11 Adding new variables 567
27.12 Resizing the 4tH environment 569
27.13 Tuning pipe failure detection 570
27.14 Adding new error messages 572
27.15 Sizing the Code Segment 573
27.16 Adding inline macros 575
27.17 Adding string words 576
27.18 Adding words with arguments 578
27.19 Packing several words into one token 580
27.20 Adding conditionals 581
27.21 Extending the I/O subsystem 586
27.22 Using the symbol table 587
27.23 Using variables and datatypes 589
27.24 Other tools 591
27.25 Patching 4tH 591
27.25.1 Tokens 592
27.25.2 Words 592
27.25.3 The virtual machine 593
27.25.4 Immediate words 594
27.25.5 Applying the patches 594
27.25.6 Error messages 595

List of Figures

2.1 Integer segment layout 22
2.2 Character segment layout 23
2.3 Hcode structure 24

4.1 Editor architecture 43

9.1 Character segment 93
9.2 The 4tH I/O system 120

10.1 Integer segment 126

13.1 Mirrored boxes 224
13.2 Two boxes 225
13.3 GTK demo 266

23.1 Basic FP architecture 461
23.2 Double, mixed and floating point word dependencies (ANS) 461
23.3 Double, mixed and floating point word dependencies (Zen) 462

26.1 Hcode structure 531

List of Tables

8.1 Character typing words 89

9.1 Picture library formatting characters 102
9.2 List of supported `printf()` flags 104
9.3 Width and precision of `printf()` 105
9.4 List of supported `printf()` specifiers 106

12.1 NELL equivalents 174
12.2 gmkiss randomizers and their registers 184
12.7 Examples of single and double number counterparts 200
12.8 Range and digits of precision 203
12.9 Examples of single and floating point number counterparts 204
12.10 IEEE 754 FP math errors 205
12.3 4tH randomizers 211
12.4 4tH sorting algorithms (address based) 212
12.5 4tH sorting algorithms (index based) 212
12.6 Fraction words 213
12.11 ANS-Forth functions 213
12.12 Floating point configurations 213
12.13 Statistical functions 213

13.4 3D plotting wordset 225
13.5 3D plotting examples 226
13.6 KPRE supported metacharacters 239
13.7 KPRE character classes 239
13.8 Supported control characters 244
13.9 Spreadsheet formats supported by 4tH 250
13.10 Example spreadsheet 251
13.11 Spreadsheet words 251

13.12 Other tabular formats 251
13.13 LaTeX table format 253

16.2 DC commands 385

17.1 4tsh commands 388

21.1 Dumb words 418

24.1 Database index conversion 466
24.2 Forth-79 to ANS conversion 514

25.1 List of compilers 520

26.1 API functions 529
26.2 HX type-byte encoding 532

27.1 comp_4th() variables 556
27.2 exec_4th() basic API 561
27.3 comp_4th() basic API 565
27.4 Examples of aliases 566
27.5 Mapping between 4tH and C variables 567
27.6 Mapping between 4tH and C variable names 567
27.7 Accessing 4tH data from C 579
27.8 exec_4th() data access API 579
27.9 Example execution plan 582
27.10 Branch resolving API 583
27.11 Members of Stream[] structure 586
27.12 Device status macros 586
27.13 Symboltable API 587
27.14 Table search API 589

Chapter 1

What's new

What's new in version 3.62.5

Words

- None.

Functionality

- Introsort was added to the sort algorithms.
- Several new randomizers were added.
- Indexes can now be bound to their database files.
- The new "Extract, transform and load" library parses .CSV files automatically.
- The `padding.4th` library was partly redesigned.
- Singleton objects are supported.
- The library files now support binary trees, `printf()` like output, INI file management and full blown associative arrays.

Bugfixes

- None.

Developer

- Several LINT fixes have been applied to the source code.
- Introsort was added to the sort algorithms.
- Several new randomizers were added.
- Indexes can now be bound to their database files.
- The new "Extract, transform and load" library parses .CSV files automatically.

- The `padding.4th` library was partly redesigned.
- Singleton objects are supported.
- The library files now support binary trees, `printf()` like output, INI file management and full blown associative arrays.

Documentation

- A section on associative arrays with binary trees was added.
- A section on binding indexes to database files was added.
- A section on parsing fixed-width text files was added.
- A section on managing INI files was added.
- A section on `printf()` like formatting was added.
- A section on extract, transform and load was added.
- A section on triple numbers was added.
- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.62.4 programs to V3.62.5 shouldn't be any problem. Source files will compile correctly without modification and executables will run unmodified. There are two things to consider:

PADDING.4TH redesign

If you used the words "+PAD" or "+FILL" anywhere in your programs, you will have to change them in order to compile properly. The easiest fix is to add these lines at the beginning of your program:

```
: +fill rot dup >r swap >r count 2dup chars + >r
  rot over max dup rot - r> swap r> fill r> place ;
: +pad bl +fill ;
```

No other changes are required.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[1]) ANS-Forth standard, except for porting purposes.

This release is dedicated to Wil Baden, my mentor, without whom 4tH would have been an entirely different beast.

[1] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Part I

Getting Started

Chapter 2

Overview

2.1 Introduction

Like Forth, 4tH is a compiler and a interpreter. Unlike Forth you cannot switch between the two. Like Forth, 4tH runs Forth-programs. Not all of them but some. But in a quite different way.

Most things have already been written. There have been Forths written in a high level language. There have been portable Forths. There have been Forths that could interface with C. Different architectures have been used to implement Forth. There have been Forths that were 16 kB or even less.

Well, all of that has been done. But here is a compiler/interpreter that's all of the above. And none of them either. It sounds like an ancient Greek riddle, but it isn't. It's 4tH.

In essence, it is Forth without the typical Forth architecture - a conventional compiler that happens to compile (almost) standard Forth to portable bytecode and will seamlessly blend in your C development environment.

2.2 History

To understand 4tH you have to know how it came to be. As most things in life, 4tH developed slowly. Its predecessor is a C-function called `strcalc()`. This function is an implementation of a RPN calculator in one very compact function (about 6 kB source). It works with signed 32 bits integers and has about 20 commands and 20 variables. The C-programmer can add additional variables.

Using it in a C-program is very easy too. Just pass the source as a string and add any variables you need. It will return the result of that calculation.

Well, although primitive it can still be very useful. You can implement an interactive RPN calculator in less than 5 lines of C. It can also be used to make calculations from sources stored elsewhere, like in a file or an environment-variable. If you can store a string there, you can store `strcalc()` source.

But we were not satisfied. We wanted to create some successor to `strcalc()` that could be used to create applets, small applications that can be embedded in an application. Like `strcalc()` it had to be fast and compact and easy to use. All these requirements and 'Reverse Polish Notation'. What language comes to mind first? Forth.

There were a few advantages and disadvantages to that approach. First, if it looked like Forth, it had to be compatible with Forth up to a certain point. Second, if it looked like Forth, we wouldn't have to write thick manuals and explain how to use the language. Third, if it looked like Forth, could we make it crash-proof?

A user can easily crash a Forth-system. Store something at a wrong address and your system hangs. We don't like that, even when the user is at fault. So we had to make a few concessions somewhere, since adding checks means the program will be less compact and slower.

For a very long time we just didn't get the right idea. Then on a dark night in October 1994, it happened. The baby was called 4tH and could do everything `strcalc()` did.

It took quite a while before 4tH had successfully got away from its `strcalc()` roots. The very first version was very buggy and little more than an RPN calculator with (incompatible) flowcontrol and some string facilities. It required two passes to compile a source and the resulting bytecode could not be saved. The I/O was C-based and very primitive. There was no Character Segment.

The second version got string and file facilities. The I/O and flowcontrol was completely rewritten, so they now were fully Forth-compatible. The second pass was discarded and H-code could finally be saved. The first move to ANS-Forth was made.

The third version came to be when the H-code eXecutable was created. This fileformat made it possible to port bytecode across platforms. At the same time, 4tH moved more and more toward ANS-Forth. Exception-handling and assertions were introduced. And in the spring of 1997, version 3.1c was released to the general public.

Of course, 4tH didn't stop there. Since then, conditional compilation, enumerations, structures, unions, binary strings, forward declarations, inline-macros, pipes, source file inclusion, threads, private declarations and a small IDE have been added. The compatibility with ANS-Forth has been significantly improved. Neither the compactness nor the speed of 4tH have been compromised. It uses less memory than previous versions and is 50% faster.

2.3 Applications

4tH is an excellent platform to learn Forth. It looks and behaves like a conventional compiler, but essentially is Forth. A Forth that detects virtually every error and reports what was wrong and where it went wrong, but still is quite fast and compact.

But like any good teacher 4tH is quite strict. Forth allows constructions that should be avoided. 4tH on the other hand, either does not implement these words or restricts their usage.

Other Forth concepts are hard to handle, like the different wordsets for different kinds of numbers. 4tH only uses signed 32 (or 64[1]) bit integers, which enables the programmer to make a wide range of applications without being bothered by overflow. Pointers, integers and characters are transparently converted.

That doesn't mean that 4tH cannot be used as a scripting language anymore. There are still excellent facilities in 4tH to do just that. They are just modified in order to allow programmers to use 4tH as a stand-alone language. If you wonder how we did all that, here is the answer.

[1] When compiled on a 64 bit platform.

2.4 Architecture

4tH is a segmented Forth. There are different segments for constant strings, characters, cells and tokens. This shows you where each data-type is located:

- Return stack (Integer Segment)
- Data stack (Integer Segment)
- Variables & values (Integer Segment)
- String variables (Character Segment)
- Temporary storage (Character Segment)
- Compiled code (Code Segment)
- Compiled constants (Code Segment)
- String constants (String Segment)

The return-stack, data-stack and variables are allocated in one large array of signed 32 (or 64) bit integers. On top of that 4tHs primitives check all parameters. This makes 4tH a very safe environment.

4tH also propagates clean programming. E.g. storing and fetching of the data-stack is not allowed. You can only store and fetch in the Variable Area.

In effect, as far as we know 4tH cannot be crashed by a user-program. The memory layout of the Integer Segment looks like figure 2.1.

The allocation of variables is totally transparent to the C-programmer. He can also transfer C-variables to the user-program (application variables). These variables can be used like any other variable.

Combining return- and data-stack means the C-programmer only has to worry about the size of the stack and not the sizes of both stacks, thus allowing a wider range of user-applications with different requirements.

The Code Segment contains words. A word is a structure that contains a unsigned byte (the token) and a signed long integer (the argument). Only the argument can be accessed by the 4tH programmer. He cannot change the program in memory, since we never really liked self-modifying code.

True, this scheme has some redundancy, but a more elaborate scheme means a more code to encode and decode the tokens and arguments. That means the memory-space we saved by compacting the program-code will make the compiler and interpreter less

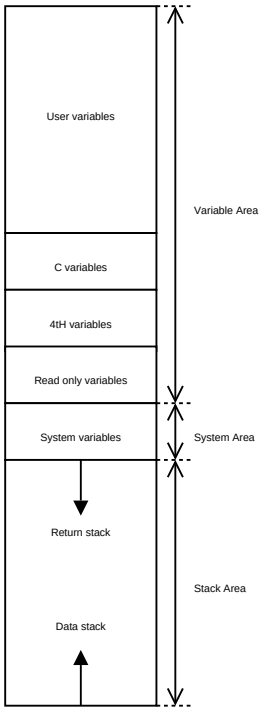

Figure 2.1: Integer segment layout

compact. And it certainly won't run any faster!

The String Segment contains all string constants. The words which use strings contain an offset to the ASCIIZ strings in the String Segment. The 4tH programmer can copy strings from this segment, but cannot write any. Constants are constants.

Finally there is a chunk of memory the user can manipulate at will. It contains the TIB, the PAD and all string variables (if any). The memory layout of the Character Segment looks like figure 2.2.

The 4tH programmer can store and fetch anything here. Since 4tH uses some C-functions ASCIIZ strings are used. The words that act on counted strings take the same parameters and deliver functionally the same results.

File I/O is supported too in a more Forth-like way than Forth itself. You can have six concurrently open files and/or pipes. 4tH has threads too. A thread can be saved to disk and reloaded. The only restriction is that all files are closed when the execution of a thread is suspended.

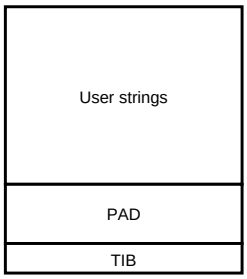

Figure 2.2: Character segment layout

2.4.1 The 4tH language

Most Forths use four different datatypes: signed 16 bit numbers, unsigned 16 bit numbers, signed 32 bit numbers and unsigned 32 bit numbers. The latter two are usually called "double numbers". Unlike C they all have their own operators. On top of that there are mixed operators too. Highly confusing!

We never liked that in the first place. Application programmers want to make an application. They don't want to worry whether any intermediate result could possibly be larger than 32767. So 4tH gets rid of most data- types and operators. It uses signed 32 (or 64) bit numbers. That's it. No mixed, double or unsigned operators.

Second, a Forth programmer has to know how much address-units a cell takes. Since every data-type in 4tH has its own segment, the address-unit of a segment is always one, regardless the data-type. Consequently, ANS- Forth words like 'CELLS' and 'CHARS' are 'NOOP's. Which fits 4tH nicely.

Although 4tH has different words for storing and fetching different data- types, most of its vocabulary is still compatible with Forth. E.g. the word "C!" takes an address in the Character Segment and "!" takes an address in the Integer Segment. Since the Code Segment and String Segment do not allow any writing, there is no need for such operators.

Each segment has its own allocation operators too. 'VARIABLE', 'ARRAY' and 'VALUE' allocate space in the Integer Area. 'STRING' allocates space in the Character Area. Other words like "' and 'CREATE' have restricted functionality and compatibility with Forth.

4tH was originally loosely based on the Forth-79 standard, but now it supports most of the CORE wordset of ANS-Forth. Note that compatibility never had the highest priority. 4tH was designed to write applets, not to be the next "fully ANS-Forth compatible compiler with a little difference". If that is what you want, 4tH is not for you.

2.4.2 H-code

Long before the dawn of the original IBM-XT there was a language called UCSD Pascal. Like Forth, it was a compiler and an interpreter. In fact, it didn't compile source into object-code for some silicon-based processor. Instead it made P-code. So if you wanted to execute it, you needed a P-code interpreter for your system.

Such an interpreter can run faster than an ordinary interpreter since it doesn't interpret source-statements with all of its symbolic labels intact, but optimized P-code. It seems to have been discovered again, since Java and previous versions of Visual Basic work the same way. Visual Basic hides the interpreter in a DLL, but basically it doesn't work any different.

The 4tH uses the same basic architecture. First the source is compiled into H-code. Then the H-code interpreter is run. A token is a very simple structure. It's got a single byte instruction and an argument. Here's a sample of disassembled H-code:

```
[62] CR (0)
[63] VALUE (2)
[64] +LITERAL (-1)
[65] DUP (0)
[66] TO (2)
[67] 0BRANCH (62)
```

BTW, building a decompiler for tokenized code is quite simple. There is one for Visual Basic and it seems like one emerged for Java too. The H- code was the result after compiling this little piece of source code:

```
cr begin times @ 1- dup times ! until
```

You can clearly see that everything is actually compiled. Flow-statements are compiled into BRANCH and 0BRANCH instructions pointing to addresses in the Code Segment.

Compiled H-code can be used on its own. It can be kept in memory, loaded, saved, decompiled and executed. H-code is a combination of the String Segment, the Code Segment and a header (figure 2.3). The header contains all the information to set up the runtime environment and some information on the String- and the Code Segments. The Integer Segment and the Character Segment are created at runtime.

Although speed was an issue when 4tH was designed, it is beaten by some other Forths. There are several possible explanations.

- 4tH uses 32 (or 64) bit numbers, while some Forths still use 16 bit numbers;
- 4tH checks all parameters, while other Forths depend on signals or don't do any checking at all;
- 4tH is written in C, while some other Forths are written in assembler.

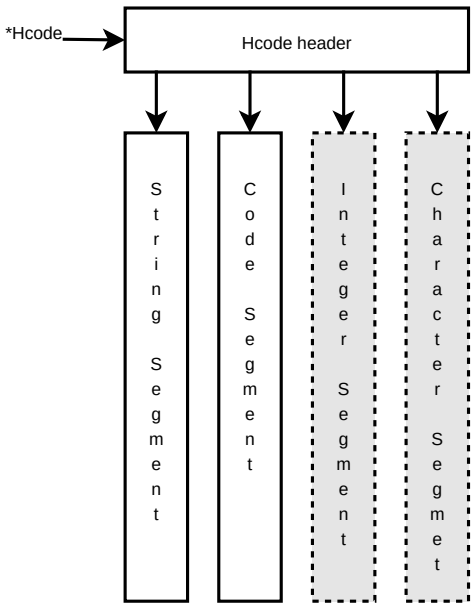

Figure 2.3: Hcode structure

When 4tH is compiled with a 32 (or 64) bit compiler it outruns Python, Ruby, Perl and most other C-based Forths (upto 4 times) or has a comparable performance (with the possible exception of GCC optimized Forth compilers). In real life applications the difference is barely noticeable.

To make compiled H-code portable, a separate scheme was developed: the Hcode-eXecutable. Or HX-file for short. It contains all the information in the header, a compacted Code Segment, the String Segment and some additional information on compatibility and integrity. Numbers are stored in an architecture-independant way.

2.4.3 H-code compiler

The H-code compiler looks a lot like any conventional compiler or assembler. Basically it is a simple one-pass compiler. In order to understand the workings of 4tH you have to know that not all H-code instructions are equal:

- Immediate words (flow control, declarations, etc.)
- Predefined constants (addresses, aliases, etc.)
- Simple words (do not require an argument)
- Symboltable entries (user-definitions)

To determine the initial size of both the Code Segment and the symbol table the source is parsed first and the actual number of words counted. This determines the initial size of the Code Segment with a high degree of accuracy, so extending the Code Segment is never necessairy. After compilation the Code Segment will be shrunk to its actual size.

The parser can distinguish between directives and string constants. The size of the symbol-table is determined by simply counting all definitions. Every definition needs one symbol-table entry. That makes determining the size of the symbol-table very easy.

During compilation all simple words are compiled into tokens without a valid argument. When a definition is encountered, like a colon-definition or a variable-declaration, a symbol is added to the symbol-table.

There are four compiler directives which determine how a number is interpreted. '[BINARY]' interprets numbers as binary numbers, '[HEX]' interprets them as hexadecimal numbers. '[DECIMAL]' and '[OCTAL]' are available too. The "simple words" 'HEX', 'DECIMAL' and 'OCTAL' only act during execution and do not determine how a number is interpreted during compilation.

During compilation the compiler also resolves all flow words. It simply matches the correct instruction and enters the jump-address into the argument of the 'BRANCH', '?DO', 'LOOP', '+LOOP', 'CALL' or '0BRANCH' word. The way 4tH handles flow control is almost completely identical to Forth.

It may sound strange, but colon-definitions are also treated like flow-words. The colon simply compiles into a 'BRANCH' instruction that skips the colon definition.

When the user calls a colon definition, it simply compiles into a 'CALL' instruction that puts the current address on the return-stack and jumps inside the colon definition, after the 'BRANCH'. The semi-colon works like a RETURN instruction that pops the return address from the return-stack. Yes, like a subroutine in BASIC or assembler!

2.4.4 Error handling

When 4tH finds an error during compilation or execution it stops and sets the H-code member ErrNo. It works like `errno` in C. You can optionally link in an array of error-messages. ErrNo is an index to this array, which makes issuing the correct error message very simple. The instruction pointer is frozen at the point where the error occured, so it is very easy to find out where the error occured.

2.4.5 Interfacing with C

A minimal compiler would take only a few lines of C-code. The C-programmer can send C-variables to the interpreter, just like `strcalc()`.

E.g. a compile takes a string-pointer as argument and returns a pointer to H-code:

```
object = comp_4th (source);
```

Executing H-code is easy too:

```
ReturnVal = exec_4th (object, argc, argv, 3, Var1, Var2, Var3);
```

Which would preload variables `Var1`, `Var2` and `Var3`. You must specify how many variables are preloaded. Also `argc` and `**argv` are available from the 4tH program.

The value returned by `exec_4th()` and stored into `ReturnVal` is the value of the 4tH variable 'OUT', which initially contains `CELL_MIN`. If an error occurs `exec_4th()` will *always* return `CELL_MIN`, regardless the value stored in 'OUT'.

Chapter 3

Installation Guide

3.1 About this package

4tH will compile ordinary text-files (MS-DOS and Unix) as well as block-files produced by the 4tH editor. The user-interface of this line-editor is highly compatible with conventional Forth block-editors.

4tHs special architecture almost forces you to write "clean" code, so you will learn Forth the proper way. This does not mean that you can't write portable code with 4tH. In fact, because Forth is so flexible you can usually write a small interface to your well-written 4tH-code in a matter of minutes.

You can use 4tH in virtually every environment, from Linux to MS-Windows. You don't even have to recompile your applications since 4tH uses a special executable format, that is interpreted by the 4tH virtual machine.

3.1.1 Example code

There are a lot of example programs, written in 4tH. From line-editors and calculators to adventure-games. Not all have been especially written for 4tH. There are quite a few programs from the hand of people like Professor C.H. Ting and Leo Brodie that started their existence as Forth-programs.

Most are available in source. That means they have the extension *'.4th'*. You can examine or edit them like any other source-file. Source-files written with the 4tH editor get the extension *'.scr'*. They can only be edited with the 4tH editor or other Forth blockfile editors. Executables have the extension *'.hx'* (Hcode eXecutable).

3.1.2 Main program

You will find a binary program within this package called 4tH. You can copy this binary to any directory. 4tH is a small development system by itself. When you start it, it will automatically enter interactive mode and show you a menu not unlike early versions of Turbo Pascal. You can edit, compile, run and debug programs from the 4tH prompt. Please read chapter 4 for more details.

You can also use 4tH from the commandline:

```
4th <commands> <file> [file | argument .. argument]
```

It takes most combinations of these ten commands:

m enter interactive mode

e edit a 4tH screenfile

c load a sourcefile (.4th) and compile it

l load an objectfile (.hx)

d decompile a 4tH program

g generate a C sourcefile (default: out.c)

s save a 4tH program (default: out.hx)

x execute a 4tH program

v enter verbose mode

q suppress copyright message

A few examples:

- To compile a 4tH program and save the object code: `4th csv <source.4th> [object.hx]`
- To compile a 4tH program and execute it: `4th cx <source.4th>`
- To decompile object code: `4th ld <object.hx>`
- To convert object code to C source: `4th lg <object.hx> [source.c]`
- To load and execute object code: `4th lx <object.hx> [arguments]`
- To load and execute object code without arguments: `4th <object.hx>`
- To edit a 4tH screenfile: `4th e <source.scr>`
- To enter interactive mode: `4th m <source.scr>`
- To enter interactive mode (without loading a screenfile): `4th`

Note: don't include the "[]" and "<>" in your commandline. They are just there to show whether an argument is optional ([arg]) or mandatory (<arg>).

3.1.3 Unix package

It is not possible for us to provide Unix binaries for all possible platforms, not now and not in the future, simply because we don't have access to them all. Here is a list of the Unix (like) platforms that are known to compile 4tH:

- Intel - FreeBSD
- Intel - Coherent

- Intel - BeOS
- Intel - Plan9
- RS/6000 - AIX
- NeXT - NS
- Apple - OS/X
- Sun - Solaris
- ARM - RISC/OS
- ARM - Android
- Intel - Linux
- Zaurus - Linux
- Raspberry Pi - Linux
- Ben Nanonote - Linux
- Zipit Z2 - Linux
- Nokia N810 - Linux
- Apple - Linux

If your platform is not listed, give it a try anyway. The chances are it will compile flawlessly, since we've never had a report of a Unix platform that refused to compile or run 4tH. Please send us an email with your results, so we can add it (or remove it) from our list.

You have to compile 4tH yourself, which is not difficult if you read the 'Developers Guide'. Usually this will do the trick:

```
make
make install
```

If you have any special needs, feel free to edit the makefile.

3.1.3.1 Updating

Simply install the package. Unless you've used a different location or different options the previous time, it will simply overwrite the previous executables. If you still have `4thd`, `4thg`, `4thx` or `4thc` somewhere on your drive, delete them. That's ancient stuff!

3.1.4 Linux package

You will find Linux binaries in this package. They will run under *most modern Linux distributions for Intel*. If the Linux binary doesn't run, you can easily recompile it. Just enter:

```
make
make install
```

You don't have to run './configure'. If you have any special needs, feel free to edit the makefile, e.g. compiling for the Zaurus means you have to add the '-DZAURUS' option.

You'll also find some icons for KDE or GNOME and a 'man' page. However, you have to install them manually. If you want to embed 4tH in KDE or GNOME you have to do that manually as well. Please consult your KDE or GNOME documentation.

You can place the 4tH executable any place you want. It doesn't require *any* external files.

3.1.4.1 Copying the tarball to your platform

If you're using an exotic platform like the ones listed here, you may experience some problems getting the tarball onto your device. We will give you some directions on selected platforms.

Ben Nanonote

The easiest way is to copy the tarball onto a micro-SD card and boot your Ben Nanonote. You'll find the tarball under `/card`. Copy it to the location of your choice and proceed as usual.

If not, perform this procedure. First, be sure you have set the password of "root" under your Ben Nanonote. Second, you have to connect the Ben Nanonote to the USB port of your host computer. Finally, enter the following commands (as root) on your host computer:

```
ifconfig usb0 192.168.254.100
scp <4tH tarball> root@192.168.254.101:~/
```

The 4tH tarball will end up in the `$HOME` directory of "root" on your Ben Nanonote. Proceed as usual.

3.1.4.2 /etc/magic

If you want Linux to recognize your 4tH files, you have to add the following lines to your /etc/magic file:

```
# From The.Beez.speaks@gmail.com
# These are the magic numbers for 4tH HX files

0       belong          0x01020400      4tH eXecutable
>9      leshort x       \b, version %x
```

E.g. if you enter:

```
file editor.hx
```

It will respond:

```
editor.hx: 4tH eXecutable, version 362
```

3.1.4.3 Using binfmt_misc

There is a module in Linux that will allow you to execute 4tH programs from the prompt without explicitly calling the 4tH interpreter. It is called 'binfmt_misc'. 4tH has built-in support for this module. Just add the following lines to your 'boot.local'[1] file:

```
insmod binfmt_misc
cd /proc/sys/fs/binfmt_misc
echo ':HX:M::\x01\x02\x04\x00\xff\xff\xff\x7f\x04\x62\x03\x08:
      :/usr/local/bin/4thx:' >register
```

If you use a kernel version later than 2.4.13 you have to add these lines:

```
insmod binfmt_misc
mount -t binfmt_misc none /proc/sys/fs/binfmt_misc
cd /proc/sys/fs/binfmt_misc
echo ':HX:M::\x01\x02\x04\x00\xff\xff\xff\x7f\x04\x62\x03\x08:
      :/usr/local/bin/4thx:' >register
```

You can find out whether 4tH support has been properly installed by issuing:

```
cd /proc/sys/fs/binfmt_misc
cat HX
```

And Linux should answer:

```
enabled
interpreter /usr/local/bin/4thx
offset 0
magic 01020400ffffff7f04620308
```

Finally, you should go to the directory where 4tH has been installed (usually /usr/local/bin) and enter:

```
ln -s 4th 4thx
```

Now, after you've compiled a program you should make it executable and it will run like it is a native executable, e.g.:

```
4th cs asc2html.4th asc2html
chmod 755 asc2html
asc2html ascii7.4th ascii7.html
```

Note you have to be *root* in order to run some of these commands!

3.1.4.4 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, the compiler[2] will try to get it here. You can set this environment variable in your `.profile` or `.bashrc` file. Simply login into your default user account and type:

[1] On SuSE 'boot.local' is located in the /sbin/init.d directory.
[2] That is: the *compiler*, not the editor or anything else.

```
cd
vi .profile
```

or:

```
cd
vi .bashrc
```

This will launch the editor and allow you to edit the appropriate file. In this example your default 4tH directory is `/home/joe/4th`:

```
export DIR4TH=/home/joe/4th/
```

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `/home/joe/4th-/lib/anscore.4th`. *Do not forget to add the trailing slash*. If you do, it will not work properly.

3.1.4.5 EDIT4TH environment variable

This variable is used by the 4tH executable to launch an external editor, e.g. `vi`, instead of the built-in block editor. You can set this environment variable in your `.profile` or `.bashrc` file. Simply login into your default user account and type:

```
cd
vi .profile
```

or:

```
cd
vi .bashrc
```

This will launch the editor and allow you to edit the appropriate file. In this example the external editor is `vi`:

```
export EDIT4TH=vi
```

It works exactly the same as the familiar `EDITOR` environment variable[3].

3.1.4.6 Updating

Simply install the package. Unless you've used a different location or different options the previous time, it will simply overwrite the previous executables. If you still have `4thd`, `4thg`, `4thx` or `4thc` somewhere on your drive, delete them. That's ancient stuff!

3.1.5 Android package

First, install the "Android Terminal Emulator" by Jack Palevich[4]. Note that on some devices `/sdcard` can *only* be accessed by `/mnt/sdcard`. Unzip the archive on your Linux, OS/X or Windows PC if your Android device lacks this capability.

[3] https://en.wikibooks.org/wiki/Guide_to_Unix/Environment_Variables#EDITOR

[4] You can find this free application in your Android Market. Homepage: `https://github.com/-jackpal/Android-Terminal-Emulator/wiki`

3.1.5.1 Android 2.2

Support for Android 2.2 is *experimental*. Follow the instructions in section 3.1.5.2. Please report any problems you experience.

3.1.5.2 Android 2.3+

This version will be installed in the `shared_prefs` folder of ATE.

> **Important!**
> The `shared_prefs` folder is *required*. It *only* exists after you change the preferences in the terminal emulator. So before commencing, change the text colour, font or *something*.

Copy the following files from the `install` directory to any directory on the SD card of your ARM Android device:

- 4th
- 4tsh
- pp4th
- install23.sh

Start up the "Android Terminal Emulator" and issue the following command:

```
cd /sdcard
cd <your directory>
sh install23.sh
```

You will be asked two vital questions:

1. Are the `4th`, `4tsh` and `pp4th` files in your current working directory;
2. Is there an `OK` file in the ATE directory under `/data/data`.

If you can answer these questions affirmative, press the "Enter" key, otherwise exit ATE. Finally, you will be given a last chance to quit the installation script. If you want to continue, press "Enter". The application should respond:

```
(INFO ) Installing Android 4tH..
(INFO ) Android 4tH installed..
```

4tH is now installed on your Android device and you can close the "Android Terminal Emulator". You can remove the files from the SD card if you wish.

3.1.5.3 Android 3+, 4+

This version will be installed in the `app_HOME` folder of ATE.

> **Important!**
> Some Android 3+ devices don't feature a full shell, so in some cases the following procedure won't work and the script will fail with a shell error. This is *not* harmful to your device. If that happens to you, please use the script `install23.sh` instead and follow the procedure described in section 3.1.5.2.

Copy the following files from the `install` directory to the `/sdcard/Download` directory of your ARM Android device:

- 4th
- 4tsh
- pp4th
- install.sh

Start up the "Android Terminal Emulator" and issue the following command:

```
cd /sdcard/Download
sh install.sh
```

The application should respond:

```
(INFO ) Installing Android 4tH..
(INFO ) Android 4tH installed..
```

4tH is now installed on your Android device and you can close the "Android Terminal Emulator". You can remove the files from `/sdcard/Download` if you wish.

3.1.5.4 Postinstallation

Copy the `4th` subdirectory to your Android device, e.g. `/sdcard/4th`. Start up the "Android Terminal Emulator" and choose "Preferences" from the top right menu. Scroll to "Shell" and choose "Initial command". Edit it so it reads:

```
export PATH=<original setting>:/data/data/jackpal.androidterm/
  shared_prefs:$PATH;export DIR4TH=/sdcard/4th/
```

When you installed the 2.3+ version. Or, when you installed the Android 3+ version:

```
export PATH=<original setting>:/data/data/jackpal.androidterm/
  app_HOME:$PATH;export DIR4TH=/sdcard/4th/
```

Make sure `$PATH` is declared only once, e.g.:

```
export PATH=/data/local/bin:/data/data/jackpal.androidterm/
  app_HOME:$PATH;export DIR4TH=/sdcard/4th/
```

The installation is finished now. You can now restart the "Android Terminal Emulator" and access 4tH by simply typing:

```
4th
```

At the prompt. If you use 4tH on a tablet you may also consider installing the "Hackers keyboard" by Klaus Weidner[5]. Working with the "Android Terminal Emulator" becomes a lot easier that way!

3.1.5.5 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, the compiler[6] will try to get it here. If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `/sdcard/4th-/lib/anscore.4th`. *Do not forget to add the trailing slash*. If you do, it will not work properly.

3.1.5.6 EDIT4TH environment variable

This variable is used by the 4tH executable to launch an external editor, instead of the built-in block editor. Use only if you *exactly* know what you are doing.

3.1.5.7 Updating

Simply install the package. Unless you've used a different location or different options the previous time, it will simply overwrite the previous executables.

3.1.6 MS-DOS package

The "4th.exe" that is included in the MS-DOS package is a 32-bit MS-DOS version of the main Unix utility. It will only run on 80386 class machines and up. It allows you to compile and run very large 4tH programs. It requires CWSDPMI.EXE somewhere in your path. It is also available as "4th86.exe", which will run on any IBM-PC with 256 KB memory. This version is a bit slower and you may experience some memory restrictions.

3.1.6.1 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, the compiler[7] will try to get it here. You can set this environment variable in your `autoexec.bat` file. In this example your default 4tH directory is `C:\4th`:

[5] http://code.google.com/p/hackerskeyboard/ Note Android will issue a warning about keyloggers. Don't worry, it always does that when you install a non-standard input device. The "Hackers keyboard" is Open Source and there are *no* indications it contains any malware.

[6] That is: the *compiler*, not the editor or anything else.

[7] That is: the *compiler*, not the editor or anything else.

```
set DIR4TH=C:\4th\
```

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `C:\4th\-lib\anscore.4th`. *Do not forget to add the trailing backslash*. If you do, it will not work properly.

3.1.6.2 EDIT4TH environment variable

This variable is used to indicate which external editor the 4tH executable should load if you do not want to use its built-in block editor. You can set this environment variable in your `autoexec.bat` file. In this example you're using `edit`:

```
set EDIT4TH=edit
```

If the external editor is not in the `PATH`, you should provide the entire path.

3.1.6.3 Updating

Simply install the package. Unless you've used a different location or different options the previous time, it will simply overwrite the previous executables. If you still have `4thd.com`, `4thg.com`, `4thx.com` or `4thc.com` somewhere on your drive, delete them. That's ancient stuff!

3.1.7 MS-Windows package

Run "setup.exe" to install the package. It runs with Windows 95 OSR2 and up, Windows NT 4.0, Windows 2000, Windows XP and Windows Vista.

You can launch Explorer and double-click an HX-file. Windows will complain it doesn't recognize the file and tell you what to do. Browse to "4th.exe" and select it. After that you can click on an HX-file and it will be executed. You can even add HX-files to your desktop where they will start and run like ordinary Windows applications.

This is a true 32-bit version, so it does take long filenames, but you can't run it with Windows V3.x and early versions of Windows 95. It is a console application, so you'll need an MS-DOS box to run and use it. Note that it will exit immediately once a program has halted. We recommend you run 4tH from the MS-DOS prompt when you're using 4tH as a development environment.

3.1.7.1 Environment variables

The `DIR4TH` variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, the compiler[8] will try to get it here. In this example your default 4tH directory is `C:\4th`:

```
set DIR4TH=C:\4th\
```

[8] That is: the *compiler*, not the editor or anything else.

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `C:\4th\-lib\anscore.4th`. *Do not forget to add the trailing backslash*. If you do, it will not work properly.

The `EDIT4TH` variable is used to indicate which external editor the 4tH executable should load if you do not want to use its built-in block editor. In this example you're using `notepad`:

```
set EDIT4TH=notepad
```

If the external editor is not in the `PATH`, you should provide the entire path.

MS-Windows 9x While it is possible to set environment variables in the same way as for MS-DOS by editing `autoexec.bat`, it is easier to use `msconfig`. First run `msconfig` from the task bar by selecting "Run ".

Select the "Autoexec.bat" pane, then go to the bottom of the window, select the last entry and click the "New" button. A small input window appears below the last entry, and in this you should type a new entry with the exact syntax as shown in the example above. Then click "OK" and a small pen appears against the entry, indicating that `autoexec.bat` will be modified. You may have to reboot afterwards.

MS-Windows NT+ Click on the "My computer" icon or the "Start" menu, then click on the "Control panel". Click on the "System" icon to get the "System Properties" dialog box. For Windows NT+ use the "Environment" tab instead of the "Advanced" tab. Click on the "Environment Variables" button and select "New". Enter the `DIR4TH` and its value in the boxes and then click "OK". You can repeat the procedure for `EDIT4TH` if you wish.

If there are several users on the PC, it is probably better to set the variables as "System variables", rather than "User variables" since they will then automatically be accessible for all users. You will need to have Administrator rights to do this.

3.1.7.2 Updating

Be sure to properly uninstall the previous package before installing the new one. If you don't it may suggest to overwrite the package and simply refresh the group. Be sure the installation is verbatim. If there are any changes, the uninstaller may refuse to uninstall the package.

3.2 Setting up your working directory

The best thing to do is to create a directory under your home directory. In Windows, your home directory is called `My documents`, in Unix-like environments simply type '`cd`' and you're there.

The rest largely depends whether you are the only one developing 4tH programs on your system or whether you want a system wide installation. If you are the only one developing programs you could create a `lib` subdirectory and copy all library files there. Your `DIR4TH` environment variable can now simply point at your own 4tH directory.

If you want a system wide installation or your Linux distribution already installed the library files for you, the smartest thing is to let your `DIR4TH` environment variable point to the 4tH system directory, e.g. `/usr/share/4th` or `C:\Program files\4tH`.

Most people find it a lot easier to use 4tH from the prompt, so if you want to start a session, start up your favorite command line shell and navigate manually to your personal 4tH directory. When you just want to run a 4tH program, it depends on whether you want to run it as a script, a bytecode image, a shell script or a native executable. Please consult either your Operating System manual or the appropriate sections of this document.

3.3 Now what?

After you've installed and played around with the utilities, we suggest you either click the 4tH icon on your desktop or start an interactive session by entering:

```
4th m session1.scr
```

And start reading the Primer. When you've thoroughly read and understood the very first section you're ready to go on. Start up your favourite editor (or use the built-in editor if you don't have one) and make your own very first 4tH program. If you don't know how to use the built-in editor, read chapter 4.

If you encounter an error during compilation or execution, refer to the 'Errors Guide' for a detailled description what it means, what probable causes are and how you can fix it.

3.4 Pedigree

4tH is basically an original work. However, some concepts have been derived from the work of other, much smarter people.

- The pictured numeric output and flow-control routines are based on Abersoft Forth.
- The exception handler is based on the dpANS-6 implementation.
- The enumerations are based on the Swift-Forth implementation.
- The structures are based on the GForth implementation and suggested by Stephen Pelc.
- The 'ASSERT(' and ')' words are based on an idea implemented in GForth.
- The implementation of '[DECIMAL]', '[HEX]', '[OCTAL]' and '[BINARY]' was suggested by William Tanksley.
- The implementation of ':REDO' and 'DOES>' was suggested by Astrobe.
- The implementation of unions was suggested by Tim Trussel and Bruce McFarlane.
- The HX-format was suggested by Mikael Cardell.

4tH was discussed in Volume XVIII, Number 3 of Forth Dimensions. Thank you, Marlin Ouverson for giving me that opportunity.

3.5 Contributors

You may get the suggestion that I did all this myself, but that is hardly true:

Will Baden	4tH to ANS-Forth interface;
G.B. Stott	4tH Makefile;
AltLinux team	Shared libraries;
Ed	ANS-Forth compatible floating point I/O libraries;
Bill Cook	George Marsaglia random number libraries;
David Johnson	Zenfloat floating point, Zenfloat SQRT, Gem4tH, Portable Bitmap graphics, Turtle graphics, 3D plotting, infix formula translation, selected library members and example programs.

Furthermore, I'd like to thank all those people who have helped me to port 4tH to different machines, expecially Wim Slangewal, Zbigniew, Ron K. Jeffries, Rubén Berenguel and Greg Schmidt. Finally, I humbly apologize to all those who have been forgotten in the course of 4tH's 20+ years history or whose contributions have been superseded by other developments.

3.6 Questions

We tried to provide you with all the documentation you'll probably ever need. That doesn't mean that you'll never have any questions. NEVER EMAIL THE PEOPLE WHOSE SITE YOU GOT THIS FROM! THEY DON'T KNOW EITHER! INSTEAD, MAIL TO:

```
The.Beez.speaks@gmail.com
```

You'll usually get fast answers, although when your question is very complex we'll probably give you just some general directions. We have to stress that *any* comment is welcome, always.

3.6.1 4tH website

You can visit our website, which is dedicated to 4tH:

```
http://thebeez.home.xs4all.nl/4tH/
```

3.6.2 4tH Google group

We've got a Google group for discussions about 4tH. You will find all the latest information there, including additions and bugfixes. If you want to interact with other 4tH users, we recommend you subscribe to this group. You will also have to become a Google member if you are not already, e.g. when you already have a *gmail* account:

```
http://groups.google.com/group/4th-compiler
```

Important! Your posts will not be accepted by the server if you don't subscribe first! Your first messages will be moderated.

3.6.2.1 Conditions of use

This group has been created as a service to, and in support of, the 4tH (and Forth) community. As in most discussion groups, there are a few rules to ensure the survivability of the group for the future.

1. This group is for discussions of 4tH problems, 4tH questions and answers. It is not to be used for non-4tH discussions.
2. This is not an 4tH advocacy group. Stick to 4tH questions and problem-solving or move your discussion to an appropriate channel. i.e. alternative site or private e-mail.
3. Flames, insults, foul language will not be tolerated. You will be unsubscribed and barred from re-subscribing under your present e-mail address.

3.6.2.2 What to discuss?

Well, Problems, wishes, needs, solutions (how you did something) basically anything 4tH related.

3.6.3 Newsgroup

There is no special newsgroup for 4tH. However, comp.lang.forth will prove to be able to answer most of your questions.

Chapter 4

A guided tour

4.1 4tH interactive

4tH's interactive mode was introduced with version 3.3c, but it is still fully compatible with previous versions, so you can still use all your external IDE's and script files. The interactive mode is especially useful when you are using an environment where other tools are not available or impossible to use. This document shows you how to use interactive mode and get the most out of it.

4.2 Starting up 4tH

You can enter 4tH's interactive mode by just clicking the icon (when you are using MS-Windows) or by issuing this command on the Unix or MS-DOS commandline:

```
4th
```

4tH will respond by showing you this screen:

```
(S)creen file: new.scr
(O)bject file: out

(E)dit   (C)ompile   (R)un    (A)rguments

(Q)uit   (G)enerate  (B)uild  (D)ecompile

>_
```

This is the main menu. It is slightly reminiscent to the earlier versions of Turbo Pascal. At the bottom is the prompt. Just press the appropriate key and hit enter, e.g. "S", which stands for the name of the screenfile. 4tH will now prompt you for the name of the screenfile. Note that 4tH is not case sensitive, so both "s" and "S" will do.

4.3 Running a program

We assume you've installed 4tH according to the instructions. If not, this might not work. Now press "S" and hit enter. 4tH will prompt you for the name of a screenfile:

```
Screen file name:
```

Answer by typing ”examples/romans.scr”[1] and hit answer. 4tH will return to the menu:

```
Screen file name: examples/romans.scr

(S)creen file: examples/romans.scr
(O)bject file: out

(E)dit   (C)ompile   (R)un    (A)rguments
(Q)uit   (G)enerate  (B)uild  (D)ecompile

>
```

Now hit ”R” and press enter. What now appears is your program that is actually running:

```
>r
Enter number: 2005
Roman number: MMV
```

After the program has ended, you will return to the menu. Well, that wasn’t too hard, was it?

4.4 Starting an editing session

We start by entering the editor mode. Just type ”e” and hit enter. Ignore any file opening errors. The ”OK” prompt shows you you’re now in the editor. Now type:

```
0 clear
```

This will erase the first screen and select it for editing. 4tH’s editor is a typical Forth editor. Forth organizes its mass storage into "screens" of 1024 characters. Forth may have one screen in memory at a time for storing text. The screens are numbered, starting with screen 0.

Each screen is organized as 16 lines with 64 characters. The Forth screens are merely an arrangement of virtual memory and do not correspond to the screen format of the target machine.

Depending on memory model and operating system, you have either 28, 32 or 64 screens available. This will be sufficient in most situations. These screens correspond to a region in memory, which acts like a RAM drive.

The actual editing is done in an area that is called the ’workspace’. With the word ’clear’ you wipe all information in the workspace. With the word ’list’ you can select a certain screen for editing and load its information from the RAM disk into the workspace. The figure below shows you how to transfer information between the screenfile, the RAM disk and the workspace (figure 4.1).

When you enter the editor the file is automatically loaded into the RAM disk. With ’list’ you transfer the source from a screen in the RAM disk into the workspace. Since we started a new file (that’s why you got the error message) all screens are empty. That why we cleared screen 0 and selected it for editing. You can quit the editor without changes by pressing ”q” and hitting the enter key.

[1] This works for both Windows and Unix type Operating Systems.

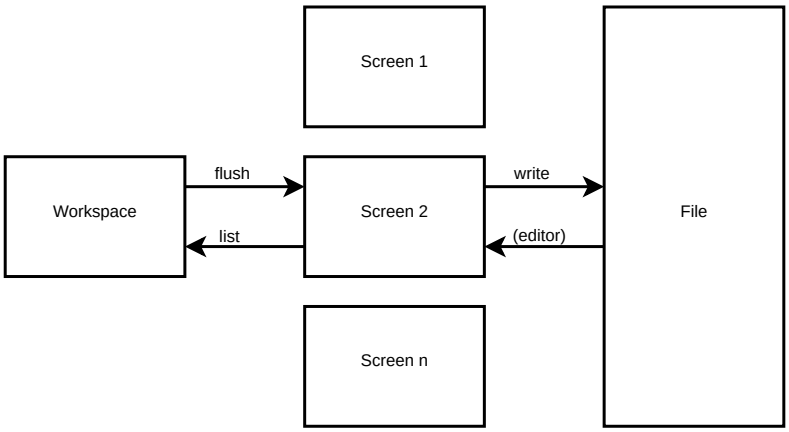

Figure 4.1: Editor architecture

4.5 Writing your first 4tH program

We start our program by giving it a name. Press ”s” and enter ”hello.scr”. Again, ignore any file opening errors. Now we're going to enter the source text, so we start up the editor by pressing ”e” (you know by now you have to press the enter key afterwards). Then we select screen 0 for editing by entering:

```
0 clear
```

If you want to know what you've entered so far you can list the editing screen by entering:

```
l
```

The editor will now show you a full listing:

```
Scr # 0
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^                                                          0 OK
```

The first line will tell you which screen you're working on, which is screen 0. Then all sixteen lines are listed, all blank of course. Finally it will show you the current line, which is line 0. The ”^” is the cursor, which is at the beginning of the line. You can move the cursor around with the ”m” command. Try:

```
10 m
```

The editor will respond with:

```
^                                                        0 OK
```

And shows you this way that the cursor has moved 10 positions. If you want to move the cursor backwards, you can do that too. Just enter a negative value, like:

```
-5 m
```

And the cursor will move back five positions:

```
^                                                      0 OK
```

If you enter a larger value, that is perfectly acceptable too:

```
128 m
```

Note that every line is 64 characters long, so the editor will tell you you've just moved to line 2:

```
^                                                      2 OK
```

Don't be afraid that you'll do something wrong and lose your source. Note that this is 4tH, not Forth. If you try something funny like entering a very large value, the editor will just issue an error message:

```
1024 m
Off screen OK
```

You just tried to go beyond the workspace and the editor won't allow you to do that. Okay, we've moved around enough. How about writing that program? You can enter text with the "p" command, which stands for "PUT". Just provide the editor with the appropriate linenumber and the text:

```
0 p ." Hello world!" cr
```

Let's list our screen:

```
l

Scr # 0
  0 ." Hello world!" cr
  1
  2
  3
  4
  5
  6
  7
```

```
  8
  9
 10
 11
 12
 13
 14
 15

^." Hello world!" cr                                          0 OK
```

That's it. That's it? What about all that red tape like "Program Hello" or "int main()", opening parenthesis or closing braces? Hey, this is Forth[2], not C or something. You've just told the compiler it has to print the text "Hello world!" and write a newline. Isn't that what you wanted?

According to the figure in section 3, we first have to save the workspace in the RAM disk by entering "flush", then save it to disk by entering "write" and subsequently leave the editor by entering "q". Although perfectly correct, it is a lot of typing for just saving and exiting. You can do that a lot faster by just entering "wq", which stands for "Write and Quit".

Now we're back in the main menu and we want to see our program run. Just hit "R" and press enter. Don't we have to compile it first? Sure, but 4tH will notice your program hasn't been compiled yet and will compile it automatically for you. If you get an error message like this:

```
Compiling;      Word 0: Undefined name
```

Then you know you've just made a classical beginners error: there is a space between ." and the text. You'll have to go back to the editor to correct it. Reload screen 0 by entering:

```
0 list

Scr # 0
  0 ."Hello world!" cr
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
OK
```

Now let's see where our cursor is:

```
0 m

 ^."Hello world!" cr                                          0 OK
```

[2]If you are not familiar with Forth and want to learn it, please read the primer. Everything you want to know is explained there in detail.

Now we know we have to move our cursor two positions and enter a space. Entering text at the cursor position is done by the ”c” command, which stands for ”COPY”. Note that you have to add a space after each command, so adding a space at the cursor position is done by entering a ”c” with two spaces:

```
2 m

 ."^Hello world!" cr                                          0 OK
c

 ." ^Hello world!" cr                                         0 OK
```

Now we can exit the editor again and rerun our program. Yes, 4tH will know you’ve changed the text and recompile your program automatically:

```
wq

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>r
Hello world!
```

That’s it! You’ve just successfully entered, compiled and ran your very first 4tH program!

4.6 A more complex program

Note that this is not a tutorial on Forth. If you do not know the language you’ll probably won’t understand the statements we’re going to enter. You don’t have to, but if you need to please refer to our highly acclaimed 4tH primer.

Okay, let’s presume you’re looking at your 4tH prompt. We want to write a program which converts Unix ASCII files to DOS ASCII files. Unix ASCII files use a single linefeed to signify the end of a line while DOS ASCII files use an carriage return/linefeed pair for that purpose.

First, we need to name our program, so we press ”s” to enter the name of the screen file. We’ll call it ”convert.scr”. Then we enter the editor by pressing ”e” and are greeted by the ”OK” prompt. First we’ll define a word (that’s what a subroutine is called in Forth) that converts a file:

```
0 clear
0 p : ProcessFile
1 p   begin
2 p     refill
3 p   while
4 p     0 parse-word
5 p     type 13 emit 10 emit
6 p   repeat
7 p ;
```

Note that 4tH confirms you after each line that everything is ”OK”, but we left those messages out. When we list our program it looks like this:

```
l

Scr # 0
  0 : ProcessFile
  1   begin
  2     refill
  3   while
  4     0 parse-word
  5     type 13 emit 10 emit
  6   repeat
  7 ;
  8
  9
 10
 11
 12
 13
 14
 15

 ^: ProcessFile                                                    0 OK
```

It is a good custom to start each screen with a comment line, so others will know what we've been doing. However, line 0 is already taken. To insert a blank line we use the "s" command, which stands for "SPREAD". All lines following it will move down. If you happen to use line 15 you're in trouble since that one will be lost:

```
0 s 0 s l

Scr # 0
  0
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12
 13
 14
 15

 ^                                                                 0 OK
```

Yes, as long as you're not entering a command with a trailing text parameter, you can enter multiple commands on a single line. So this one tells the editor "spread at line 0, spread at line 0, list". Now we're going to enter our comment line:

```
0 p ( Conversion from UNIX ASCII files to DOS ASCII files - I)
OK
l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
```

```
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

That will do nicely. Although this word will do the job, we still have to open the input- and the output file. Since we want to test our program quickly we make a quick and dirty word that will do the job:

```
11 p : test s" code.txt" inpud open s" out.txt" outpud open
12 p error? rot error? rot or abort" Error!" use use;
13 p ttest ProcessFile
wq
```

When we try to compile this program by entering ”c”, it doesn’t work:

```
Compiling;      Word 17: Undefined name
```

Oops, we’ve obviously made an error, but where? Word 17? Where is word 17? We can find that out by decompiling the program and see where it went wrong. Just press ”d”:

```
Object size: 17 words
String size: 9 chars
Variables  : 0 cells
Strings    : 0 chars
Reliable   : No

[   8] type      (0)
[   9] literal   (13)
[  10] emit      (0)
[  11] literal   (10)
[  12] emit      (0)
[  13] branch    (0)
[  14] exit      (0)
[  15] branch    (0)
[  16] s"        (0)    code.txt
```

The last thing it compiled was the start of the ’TEST’ definition. It must have gone wrong right after that one. So we go back to the editor and find out. Sure, ”inpud” must be ”input”. We can even find out it we made more errors like this:

```
f pud

 : test s" code.txt" inpud^ open s" out.txt" outpud open          11 OK
n

 : test s" code.txt" inpud open s" out.txt" outpud^ open          11 OK
n
Not found OK
```

And yes, we did. On lines eleven and twelve to be exact. With the ”f” command (which stand for ”FIND”) we can find a string. By entering ”n” (which stands for ”NEXT”) we can find the same text again. Now we have to correct it. We’ll get back to the top of the screen and find the offending word:

```
top f pud

 : test s" code.txt" inpud^ open s" out.txt" outpud open          11 OK
```

Note that the cursor is positioned at the end of ”input”. We only have to wipe one character and insert the correct one:

```
1 w c t

 : test s" code.txt" inpu^ open s" out.txt" outpud open           11
 : test s" code.txt" input^ open s" out.txt" outpud open          11 OK
```

With the command ”1 w” we destructively backup the cursor by one position. Then we enter the ’t’ at the cursor position by using the ”c” command. However, there is a quicker way to do this:

```
x pud

 : test s" code.txt" input open s" out.txt" out^ open             11 OK
c put

 : test s" code.txt" input open s" out.txt" output^ open          11 OK
```

The ”x” command works very much like ”f”, but it does not only *find* the string, it also *deletes* it. Still, there are other errors left in the source:

```
f test

 ttest^ ProcessFile                                               13 OK
b

 t^test ProcessFile                                               13 OK
1 w
```

Yes, ”test” has an extra ”t”. So we find the next occurrance of ”test”. Note that a search is always performed from the cursor position, so the definition of ”test” is not found. The ”b” command will move the cursor backwards up to the point where ”test” begins and we can delete the superfluous ”t” with the command ”1 w”. The final typo we have to correct is a lacking space between ”then” and the semicolon. That can be fixed pretty quickly:

```
top f use

 error? rot error? rot or abort" Error!" use^ use;                12 OK
n

 error? rot error? rot or abort" Error!" use use^;                12 OK
till ;

 error? rot error? rot or abort" Error!" use use^                 12 OK
c  ;

 error? rot error? rot or abort" Error!" use use ;^               12 OK
```

Now the cursor is positioned right after then. The ”till” command deletes everything from the current cursor position (indicated by the caret, remember?) to the end of the following string. In this case the semicolon but you can use any string. Finally, we copy the correct string into the text, which is a space followed by a semicolon. Four errors corrected. Let’s write the screen back to RAM disk and see what we have got:

```
flush l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 : test s" code.txt" input open s" out.txt" output open
 12 error? rot error? rot or abort" Error!" use use ;
 13 test ProcessFile
 14
 15

 ^test ProcessFile                                          13 OK
```

Seems to be okay. Let's go back to the main 4tH screen by issuing the "wq" command. We recompile the source by pressing "c" and presto: we got a program! Simply hit "r" to run it. After we've run the program we find a file named "out.txt" in our working directory and examine it with a hex editor:

```
4261 6445 7865 6375 7465 2028 202d 2d20
2920 2d34 2045 5845 4355 5445 203b 0d0a
3a20 4261 6441 6464 7265 7373 2028 202d
2d20 2920 2d34 2040 2044 524f 5020 3b0d
0a3a 2042 6164 416c 6967 6e20 2820 2d2d
2029 2031 2040 2044 524f 5020 3b0d 0a
```

It seems our program is working perfectly. However, it doesn't seem very practical to copy a textfile to your working directory, rename it, start up 4tH, load your program and finally run it. That can be fixed. How? Well, you'll read that in the next section.

But first we have to stress that you don't *have* to use 4tHs editor. You can use any editor you like. Shame you've already entered and saved your source. But there is a way out. And you don't have to go too far. Just start up the editor again and enter:

```
OK
export convert.4th
OK
```

You'll find an ordinary file called "convert.4th" in your working directory hat you can modify with any text editor you like.

4.7 Advanced features

What we actually want is a program we can run from the prompt, something like:

```
convert in.txt out.txt
```

And if you do not provide the required parameters it has to issue an error message:

```
Usage: convert infile outfile
```

We will get there, but we still have some coding to do. First of all, we have to structure our program. We already have a working word[3] called "ProcessFile". It seems like a good idea to define two others, one that opens the files and one that closes the files. And we have to get rid of our "test" word. So let's fire up the editor and take care of that right now:

```
OK
0 list

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 : test s" code.txt" input open s" out.txt" output open
 12 error? rot error? rot or abort" Error!" use use ;
 13 test ProcessFile
 14
 15
OK
11 d l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 error? rot error? rot or abort" Error!" use use ;
 12 test ProcessFile
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

You can remove lines with the "d" command, which stands for "DELETE". This will remove the line and move all remaining lines up. Line 15 becomes blank. But there is another way to get rid of unwanted lines:

```
11 e l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
```

[3] A subroutine in Forth is called a "word", remember?

```
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12 test ProcessFile
 13
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

The ”e” command, which stands for ”ERASE”, will leave every line at exactly the same position. It just *blanks* that line. Let’s finish this:

```
11 p : Convert OpenFiles ProcessFile ;
OK
12 e
OK
13 p Convert
OK
l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 : Convert OpenFiles ProcessFile ;
 12
 13 Convert
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

Seems neat enough, but we still haven’t got a ”OpenFiles” word. This has to be defined before ”Convert”, but do we have still have room for that on screen 0? No, we haven’t. Fortunately, you can insert screens with the 4tH editor[4]. Don’t forget to flush. That is not only a good practice when you’ve visited the bathroom, but also when you’re working with a Forth editor:

```
flush 0 insert
OK
```

We start our screen with a comment of course. We’ll use the same comment as in our previous screen, so why not copy it?

```
1 list 0 h 0 list 0 r l
```

[4]Note that this command is usually *not* available in other Forth editors!

```
Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

What did we do here? First, we switched to screen 1, which is our previous screen 0. Then we used the "h"[5] command, which copied line 0 into PAD. PAD is a buffer, which is able to hold the contents of a single line. Note that line 0 of screen 1 remains intact. It is only copied.

Then we switched back to screen 0 and issued the "r" command, which stands for "REPLACE". It replaces whatever is there with the contents of the PAD. Finally, we listed the screen. Let's play around a little with this PAD thing:

```
1 r 1 t

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              1 OK
```

Yes, the line we copied was still in PAD! We also used the command "t" to "TYPE" line 1. This command is very similar to "h", since it copies line 1 to PAD. But is also moves the cursor to the beginning of the line and types it. Let's see if you can explain this one:

```
1 d 2 r 2 t

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              2 OK
```

Sure, the "d" command not only *deletes* the line, it also *copies* it to PAD. So when the "r" command is issued, it replaces line 2 with the contents of the line we deleted. Let's do one final test:

```
0 i l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1 ( Convert UNIX ASCII files to DOS ASCII files - I)
  2
  3 ( Convert UNIX ASCII files to DOS ASCII files - I)
  4
  5
  6
  7
  8
  9
 10
```

[5] In case you wondered, "h" stands for "HOLD".

```
 11
 12
 13
 14
 15

 ^                                                                2 OK
```

Here we used the ”i” command, which stands for ”INSERT”. It inserted the contents of PAD at line 0 and moved all the remaining lines down. Note that the cursor didn’t move a bit. That’s enough play for one day, let’s get back to work:

```
1 e 3 e 2 p : OpenFile
OK
3 p   args 2dup 2>r rot open error?
OK
4 p   if
OK
5 p      drop ." Cannot open " 2r> type cr abort
OK
6 p   else
OK
7 p      dup use 2r> 2drop
OK
8 p   then
OK
9 p ;
OK
l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : OpenFile
  3   args 2dup 2>r rot open error?
  4   if
  5      drop ." Cannot open " 2r> type cr abort
  6   else
  7      dup use 2r> 2drop
  8   then
  9 ;
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

Hmm, it seems like we’re going to need another screen. It is always wise to leave some room for future extensions, so this screen is full enough. But don’t forget the commentline. We don’t want to enter that one again, so let’s store it in PAD:

```
0 h flush 1 insert 0 r l

Scr # 1
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2
  3
  4
  5
  6
```

```
  7
  8
  9
 10
 11
 12
 13
 14
 15

^                                                                2 OK
```

Hold the line in PAD, flush the screen, insert screen 1 and replace line 0 with the contents in PAD. But the commentline is not entirely correct, so let's fix it:

```
top x I)

 ( Convert UNIX ASCII files to DOS ASCII files - ^                0 OK
c II)

 ( Convert UNIX ASCII files to DOS ASCII files - II)^             0 OK
```

The cursor is still on line 2, so we move it to the top again. Then we find and delete "I)". Finally we copy in "II)". We can do that since the cursor is at the right position. Now let's enter our final word:

```
2 p : OpenFiles
OK
3 p   argn 3 < abort" Usage: convert infile outfile"
OK
4 p   input  1 OpenFile
OK
5 p   output 2 Openfile
OK
6 p ;
OK
l

Scr # 1
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - II)
  1
  2 : OpenFiles
  3   argn 3 < abort" Usage: convert infile outfile"
  4   input  1 OpenFile
  5   output 2 Openfile
  6 ;
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - II)     0 OK
```

Almost there! We just have to fix the commentline in screen 2:

```
flush 2 list
OK
top x I)
```

```
 ( Convert UNIX ASCII files to DOS ASCII files - ^                0 OK
c III)

 ( Convert UNIX ASCII files to DOS ASCII files - III)^            0 OK
```

The current screen is flushed, then screen 2 is listed. We position the cursor at the top, find and delete ”I)” and copy ”III)” in at the cursor position. Done! Let’s leave the editor and see what we have got. It compiles cleanly and when we run it it answers:

```
Usage: convert infile outfile
```

Sure, but what we actually want is to convert a file. Well, you can do that too without leaving 4tH. Just press ”a” and enter the filenames, just like you would do at the prompt:

```
(S)creen file: convert.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>a
Arguments: code.txt out.txt
```

When you press ”r” now, the arguments entered will be passed to your 4tH program, just like they would at the prompt. To clear the arguments, press ”a” again and just hit enter when prompted for arguments.

But how *do* we run it from the prompt? Easy, just press ”o” and enter ”convert.hx” at the prompt. Now press ”b”:

```
(S)creen file: convert.scr
(O)bject file: convert.hx

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>b
```

If 4tH has nothing to complain about, it doesn’t complain, so you can safely assume that everything is okay. Now we can go to the prompt[6] and run it:

```
user@linux:~ > 4th lxq convert.hx code out.txt
Cannot open code
user@linux:~ >
```

That was to be expected. Our file was called ”code.txt”, not ”code”. It is always a good idea to test all exceptions as well. There could be a bug in that code too.

```
user@linux:~ > 4th lxq convert.hx code.txt out.txt
user@linux:~ >
```

[6]Windows users can do this by starting an MS-DOS session.

Well, it seems to work.. But what we really want is a standalone program. One that can be run without invoking 4tH and shared with our friends and families. Why this ".hx" thing? HX-files do have their merits. First of all, it is very small, less than 200 bytes. But most importantly, you can take this file and run it on a Windows NT, MS-DOS or other Unix machine without modification or recompilation, provided a 4tH is available for that platform.

If you still want a standalone program, startup 4tH and reload "convert.scr". Then press "o" and enter "convert.c". Isn't that the extension of a C-program? Yes, it is. 4tH is able to generate C code. Just press "g" and you've created a C program. You don't even have to know C. If you know how to compile a C program that's more than enough[7]. We assume you've installed the 4tH library and header files, since those are needed to compile "convert.c"[8]:

```
user@linux:~ > cc -o convert convert.c -l4th
user@linux:~ > convert
Usage: convert infile outfile
user@linux:~ >
```

Is that all? No that's not all 4tH can do. We have a few surprises left.

4.8 Suspending a program

We've entered this program:

```
Scr # 0
  0 ." Is everybody in? The ceremony is about to begin.." cr
  1 44596 36 base !
  2 pause
  3 ." Wake up! Do you remember where it was?" cr
  4 ." Has this dream stopped? " . cr
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

The first line is simply a string we print to screen. The next line, we push a number on the stack and we change the radix. Then we go to sleep. After that, we wake up again, print a few lines and retrieve the number on the stack. Let's run it:

```
(S)creen file: new.scr
(O)bject file: out

(E)dit   (C)ompile   (R)un    (A)rguments

(Q)uit   (G)enerate  (B)uild  (D)ecompile
```

[7] Windows users need to consult the documentation that came with their C compiler. Some Windows compilers may not be able to compile standard C programs. MS-DOS users are encouraged to use the 'DJGPP' compiler, which is free.

[8] Read the "Developers Guide" if you are not sure how to do this.

```
>r
Is everybody in? The ceremony is about to begin..

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>
```

At first, it seems like 'PAUSE' is nothing more than an alias for 'ABORT', but that is not entirely true. Let's save the executable and enter "r" one more time:

```
>b

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>r
Wake up! Do you remember where it was?
Has this dream stopped? YES

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>
```

Now the *second* part of the program is run, that is the part after 'PAUSE'. Note that both the stack *and* the radix have remained intact. Every time 'PAUSE' is invoked, it will return you to the prompt. When you enter "r" again, it will continue where it left off, until it meets one of the following three conditions:

1. It encounters another 'PAUSE'; entering "r" will continue where it left off.
2. It encounters 'ABORT', 'QUIT' or 'ABORT"'; entering "r" will restart the program.
3. There are no more instructions to execute; entering "r" will restart the program.

But why did we save an executable? We'll have to go back to the shell to show you:

```
user@linux:~> 4th lxq out
Wake up! Do you remember where it was?
Has this dream stopped? YES
user@linux:~>
```

Entering "b" during suspension will save the program in its suspended state. When you run the resulting executable, it will behave like you've entered "r". That's neat, isn't it?

4.9 Calculator mode

Startup 4tH and enter the editor. We're going to show you this baby can do a lot more than just editing:

```
OK
.( Hello world!) cr
Hello world!
OK
```

Hey, that is a lot like the very first program we ran! Yes, it is. You can enter a subset of the 4tH language at the editor prompt, so you can test simple programs like this without getting into the "edit-compile-run" cycle. You can even make some simple calculations:

```
OK
23 45 + .
68 OK
```

Simple? Aren't the operators and operands entered in the wrong order? No, they aren't. 4tH uses Reverse Polish Notation, which is also used by HP calculators and the Unix "dc" command. 4tH has even eight built-in variables in which you can store numbers:

```
23 45 + A. !
OK
A. ?
68 OK
```

It even understands binary, hexadecimal and octal numbers:

```
23 45 + binary .
1000100 OK
1000011111 hex FACE octal 765 + + decimal .
65250 OK
```

This is called the "calculator mode" and you don't have to do anything if you want to use it. It is part of the editor command set. You can mix editor commands and calculations[9] as you like. Nice extra, isn't it?

4.10 Epilogue

This concludes our tour of the 4tH interactive mode. We hope we've shown you what you can do with it and how to use it. Of course, you don't have to use 4tH's interactive mode. It will happily reside and cooperate with existing external IDE's, editors and the like. But if memory is tight and you have nothing else, 4tH will prove to be a completely selfcontained environment.

If you're still wondering what you can do with Forth and 4tH in particular, let me tell you this: if you worked your way through this tour, you've been working with Forth all the time. The entire editor is a 4tH program, embedded in the 4tH executable, taking up less than 3 KB. It is run by the very same interpreter as your initial "Hello world!" program. Have fun!

[9] The full calculator command set is listed in the "Editor reference guide".

Chapter 5

Frequently asked questions

QUESTION: Why has `exec_4th()` this enormous `switch()` statement? Why wasn't a structure used with pointers to functions?

ANSWER: That one was built too, but it proved to be upto four times slower on all platforms. Which is perfectly understandable, because every time you evaluate a token, you have to take the overhead of calling a function into account.

QUESTION: Why are the tokens of `exec_4th()` listed in a random order?

ANSWER: We have statistically analyzed which tokens are used more often. They are up front. Some C-compilers generate a jumptable. Others generate a repeated "if .. elif .. endif" construction. Compilants produced by the latter perform better when tokens are ordered this way.

QUESTION: Do I have to use 4tHs builtin editor?

ANSWER: No. You can use any editor you like. 4tH will happily compile all vanilla MS-DOS, MS-Windows and Unix text files and most block files as well. Just use your favorite editor to create your source and compile it at the command line. Note the editor is capable of exporting vanilla text files.

QUESTION: I get "`I/O error`" all the time. What am I doing wrong?

ANSWER: If you're working with Windows and you're trying to compile some example programs, type '`S`' and enter the relative path to the example file, e.g. "`examples/romans.4th`". If you're working on some other Operating System, be sure you've also installed the library files. Now go to the directory where you have installed them, e.g. "`cd /home/john/4th/lib`" and type "`cd ..`". From this working directory use the relative or absolute path to your source file when you compile it. Be sure the path of the environment variable `DIR4TH` is correct.

QUESTION: When I use 4tHs builtin editor I get "`Cannot open file`" all the time. What am I doing wrong?

ANSWER: Probably nothing. 4tH just informs you it cannot load the file you issued at the main menu or the command line. This is always the case when you start a new file.

QUESTION: When I try to use 4tHs builtin editor or run 4tsH I get "`Bad object`" all the time. What am I doing wrong?

ANSWER: You're probably running 4tH under a 64 bit operating system. The only thing you can do is to recompile 4tH (see sections 25.7 and 25.8).

QUESTION: When I try to load a screen file or execute an *.hx* file, 4tH doesn't seem to take the `DIR4TH` environment variable into account.

ANSWER: The `DIR4TH` environment variable is used by the compiler when it tries to pull all source files from their different locations at compiletime. Note that it only *reads* files. When the editor would start writing files where you don't expect it to, things might get very dangerous. The *.hx* files are executables and your Operating System already offers several ways to find them.

QUESTION: When I open up the editor in 4tH, it takes most 4tH code like an actual Forth compiler, but not my colon definitions. Why?

ANSWER: The 4tH editor mimics Forth, that's true. But it is actually a Forth like environment *on top of 4tH*. It may *seem* like you're working on a Forth prompt, but you're not. You can use the editor *only* for editing or some quick calculations, but if you want to use the full capability of 4tH, you're stuck to the menu.

QUESTION: Can I load shared libraries with 4tH and call the external functions defined there?

ANSWER: No. 4tH is (almost) entirely composed of ANSI-C and since ANSI-C doesn't define loading and using shared libraries you can't. Furthermore, it would violate 4tH's "uncrashable" design objective, since it is impossible to trap all possible errors when you call external functions. However, if you know C, it shouldn't be too difficult to add that functionality yourself.

QUESTION: When I compile a 4tH program, I get messages like: "`Word 381: Undefined name`" . How do I know where that is?

ANSWER: 4tH is a single pass compiler and keeps the partial compilant in memory for you to examine. Simply decompile it by pressing "D" or add the "d" option to the command line. Section 11.22 will give you all the details.

Part II

Primer

Chapter 6

Introduction

Don't you hate it? You've just got a new programming language and you're trying to write your first program. You want to use a certain feature (you know it's got to be there) and you can't find it in the manual.

I've had that experience many times. So when I wrote 4tH I promised myself, that would not happen to 4tH-users. In this manual you will find many short features on all kind of topics. How to input a number from the keyboard, what a cell is, etc.

I hope this will enable you to get quickly on your way. If it didn't, email me at 'The.Beez.-speaks@gmail.com'. You will not only get an answer, but you will help future 4tH users as well.

You can use this manual two ways. You can either just get what you need or work your way through. Every section builds on the knowledge you obtained in the previous sections. All sections are grouped into levels. We advise you to use what you've learned after you've worked your way through a level.

There are six levels. First, 4tH fundamentals. It assumes a working knowledge of programming and covers the basics. Second, 4tH arrays. We'll try to explain to you what an address is and teach you basic string handling.

Third, 4tHs Character Segment. We'll explain you how it is laid out and what you can do with it. Fourth, 4tHs Integer Segment and Code Segment. We'll explain you how it is laid out and what you can do with it.

Finally, advanced programming techniques. First the builtin facilities 4tH offers and after that the extra features the 4tH library offers. We'll teach you how to program multilevel exits, write interpreters, use jump-tables, emulate floating point calculation and a lot more!!

I don't think it is enough to teach you Forth, from which 4tH was derived, but you can always get a good textbook on Forth, like "Starting Forth" by Leo Brodie. Have fun!

Chapter 7

4tH fundamentals

7.1 Making calculations without parentheses

To use 4tH you must understand Reverse Polish Notation. This is a way to write arithmetic expressions. The form is a bit tricky for people to understand, since it is geared towards making it easy for the computer to perform calculations; however, most people can get used to the notation with a bit of practice.

Reverse Polish Notation stores values in a stack. A stack of values is just like a stack of books: one value is placed on top of another. When you want to perform a calculation, the calculation uses the top numbers on the stack. For example, here's a typical addition operation:

```
1 2 +
```

When 4tH reads a number, it just puts the value onto the stack. Thus 1 goes on the stack, then 2 goes on the stack. When you put a value onto the stack, we say that you push it onto the stack. When 4tH reads the operator '+', it takes the top two values off the stack, adds them, then pushes the result back onto the stack. This means that the stack contains:

```
3
```

after the above addition. As another example, consider:

```
2 3 4 + *
```

(The '*' stands for multiplication.) 4tH begins by pushing the three numbers onto the stack. When it finds the '+', it takes the top two numbers off the stack and adds them. (Taking a value off the stack is called popping the stack.) 4tH then pushes the result of the addition back onto the stack in place of the two numbers. Thus the stack contains:

```
2 7
```

When 4tH finds the '*' operator, it again pops the top two values off the stack. It multiplies them, then pushes the result back onto the stack, leaving:

```
14
```

The following list gives a few more examples of Reverse Polish expressions. After each, we show the contents of the stack, in parentheses.

```
7 2 -          (5)
2 7 -          (-5)
12 3 /         (4)
-12 3 /        (-4)
4 5 + 2 *      (18)
4 5 2 + *      (28)
4 5 2 * -      (-6)
```

7.2 Manipulating the stack

You will often find that the items on the stack are not in the right order or that you need a copy. There are stack-manipulators which can take care of that.

To display a number you use '.', pronounced "dot". It takes a number from the stack and displays it. 'SWAP' reverses the order of two items on the stack. If we enter

```
2 3 . . cr
```

4tH answers:

```
3 2
```

If you want to display the numbers in the same order as you entered them, you have to enter:

```
2 3 swap . . cr
```

In that case 4tH will answer:

```
2 3
```

You can duplicate a number using 'DUP'. If you enter:

```
2 . . cr
```

4tH will complain that the stack is empty. However, if you enter:

```
2 dup . . cr
```

4tH will display:

```
2 2
```

Another way to duplicate a number is using 'OVER'. In that case not the topmost number of the stack is duplicated, but the number beneath. E.g.

```
2 3 dup . . . cr
```

will give you the following result:

```
3 3 2
```

But this one:

```
2 3 over . . . cr
```

will give you:

```
2 3 2
```

Sometimes you want to discard a number, e.g. you duplicated it to check a condition, but since the test failed, you don't need it anymore. 'DROP' is the word we use to discard numbers. So this:

```
2 3 drop .
```

will give you "2" instead of "3", since we dropped the "3".

The final one I want to introduce is 'ROT'. Most users find 'ROT' the most complex one since it has its effects deep in the stack. The thirdmost item to be exact. This item is taken from its place and put on top of the stack. It is 'rotated', as this small program will show you:

```
1 2 3                \ 1 is the thirdmost item
. . . cr             \ display all numbers
                     ( This will display '3 2 1' as expected)
1 2 3                \ same numbers stacked
rot                  \ performs a 'ROT'
. . . cr             \ same operation
                     ( This will display '1 3 2'!)
```

7.3 Deep stack manipulators

No, there are no manipulators that can dig deeper into the stack. A stack is NOT an array! So if there are some Forth-83 users out there, I can only tell you: learn Forth the proper way. Programs that have so many items on the stack are just badly written. Leo Brodie agrees with me.

If you are in 'deep' trouble you can always use the returnstack manipulators. Check out that section.

7.4 Passing arguments to functions

There is no easier way to pass arguments to functions as in 4tH. Functions have another name in 4tH. We call them "words". Words take their "arguments" from the stack and leave the "result" on the stack.

Other languages, like C, do exactly the same. But they hide the process from you. Because passing data to the stack is made explicit in 4tH it has powerful capabilities. In other languages, you can get back only one result. In 4tH you can get back several!

All words in 4tH have a stack-effect-diagram. It describes what data is passed to the stack in what order and what is returned. The word '*' for instance takes numbers from the stack, multiplies them and leaves the result on the stack. It's stack-effect-diagram is:

```
n1 n2 -- n3
```

Meaning it takes number n1 and n2 from the stack, multiplies them and leaves the product (number n3) on the stack. The rightmost number is always on top of the stack, which means it is the first number which will be taken from the stack. The word '.' is described like this:

```
n --
```

Which means it takes a number from the stack and leaves nothing. Now we get to the most powerful feature of it all. Take this program:

```
2     ( leaves a number on the stack)
3     ( leaves a number on the stack on top of the 2)
*     ( takes both from the stack and leaves the result)
.     ( takes the result from the stack and displays it)
```

Note that all data between the words '*' and '.' is passed implicitly! Like putting LEGO stones on top of another. Isn't it great?

7.5 Making your own words

Of course, every serious language has to have a capability to extend it. So has 4tH. The only thing you have to do is to determine what name you want to give it. Let's say you want to make a word which multiplies two numbers and displays the result.

Well, that's easy. We've already seen how you have to code it. The only words you need are '*' and '.'. You can't name it '*' because that name is already taken. You could name it 'multiply', but is that a word you want to type in forever? No, far too long.

Let's call it '*.'. Is that a valid name? If you've programmed in other languages, you'll probably say it isn't. But it is! The only characters you can't use in a name are whitespace characters (<CR>, <LF>, <space>, <TAB>). Note that 4tH is not case-sensitive!

So '*.' is okay. Now how do we turn it into a self-defined word. Just add a colon at the beginning and a semi-colon at the end:

```
: *. * . ;
```

That's it. Your word is ready for use. So instead of:

```
2 3 * .
```

We can type:

```
: *. * . ;
2 3 *.
```

And we can use our '*.' over and over again. Hurray, you've just defined your first word in 4tH!

7.6 Adding comment

Adding comment is very simple. In fact, there are two ways to add comment in 4tH. That is because we like programs with a lot of comments.

You've already encountered the first form. Let's say we want to add comment to this little program:

```
: *. * . ;
2 3 *.
```

So we add our comment:

```
: *. * . ;          This will multiply and print two numbers
2 3 *.
```

4tH will not understand this. It will desperately look for the words 'this', 'will', etc. However the word '\' will mark everything up to the end of the line as comment. So this will work:

```
: *. * . ;        \ This will multiply and print two numbers
2 3 *.
```

There is another word called '(' which will mark everything up to the next ')' as comment. Yes, even multiple lines. Of course, these lines may not contain a ')' or you'll make 4tH very confused. So this comment will be recognized too:

```
: *. * . ;        ( This will multiply and print two numbers)
2 3 *.
```

Note that there is a whitespace-character after both '\' and '('. This is mandatory! However the closing paren) does not have to have a leading blank space. It is optional.

7.7 Text-format of 4tH source

4tH source is a simple ASCII-file. And you can use any layout as long a this rule is followed:

> All words are separated by at least one whitespace character!

Well, in 4tH everything is a word or becoming a word. Yes, even '\' and '(' are words! And you can add all the empty lines or spaces or tabs you like, 4tH won't care and your harddisk supplier either.

7.8 Displaying string literals

Displaying a string is as easy as adding a comment. Let's say you want to make the ultimate program, one that is displaying "Hello world!". Well, that's almost the entire program. The famous 'hello world' program is simply this in 4tH:

```
." Hello world!"
```

Compile this and it works. Yes, that's it! No declaration that this is the main function and it is beginning here and ending there. May be you think it looks funny on the display. Well, you can add a carriage return by adding the word 'CR'. So now it looks like:

```
." Hello world!" cr
```

Still pretty simple, huh?

7.9 Creating variables

One time or another you're going to need variables. Declaring a variable is easy.

```
variable one
```

The same rules for declaring words apply for variables. You can't use a name that already has been taken. A variable is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive!

7.10 Using variables

Of course variables are of little use when you could not assign values to them. This assigns the number 6 to variable 'ONE':

```
6 one !
```

We don't call '!' bang or something like that, we call it 'store'. Of course you don't have to put a number on the stack to use it, you can use a number that is already on the stack. To retrieve the value stored in 'ONE' we use:

```
one @
```

The word '@' is called 'fetch' and it puts the number stored in 'one' on the stack. To display it you use '.':

```
one @ .
```

There is a shortcut for that, the word '?', which will fetch the number stored in 'ONE' and displays it:

```
one ?
```

7.11 Built-in variables

4tH has only three built-in variables. They are called 'BASE', '>IN' and 'OUT'. 'BASE' controls the radix at run-time, '>IN' is used by 'PARSE' and 'OUT' returns a value to the host program.

7.12 What is a cell?

A cell is simply the space a number takes up. So the size of a variable is one cell. The size of a cell is important since it determines the range 4tH can handle. It also helps make code portable across machines with different cell sized, for example 16 bit and 32 big systems. We'll come to that further on.

7.13 What is a literal expression?

A literal expression is simply anything that *compiles* to a literal. All numbers, all defined constants and some expressions are compiled to a literal. In the glossary you can find what compiles to a literal, but we list them here too:

```
'           <name>
['']        <name>
[DEFINED]   <name>
[UNDEFINED] <name>
CHAR   <char>
[CHAR] <char>
<literal> [NOT]
<literal> [SIGN]
<literal> NEGATE
<literal> 1+
<literal> 1-
<literal> 2*
<literal> /FIELD
<literal> +FIELD <name>
<literal> ENUM   <name>
<literal> <literal> *
<literal> <literal> /
<literal> <literal> +
<literal> <literal> -
<literal> <literal> [=]
<literal> <literal> [MAX]
```

7.14 Declaring arrays of numbers

You can make arrays of numbers very easily. It is very much like making a variable. Let's say we want an array of 16 numbers:

```
16 array sixteen
```

That's it, we're done! You must omit the word 'CELLS', since 'ARRAY' implicates that you want an array of numbers, not characters. The size is a literal expression. You can't take it from the stack, so this is invalid:

```
3 dup + array sixteen
```

4tH will let you know that this is not a valid construction, but in case you wonder..

7.15 Using arrays of numbers

You can use arrays of numbers just like variables. The array cells are numbered from 0 to N, N being the size of the array minus one. Storing a value in the 0th cell is easy. It works just like a simple variable:

```
5 sixteen 0 th !
```

Which will store '5' in the 0th cell. So storing '7' in the 8th cell is done like this:

```
7 sixteen 8 th !
```

Of course when you want to store a value in the first, second or third cell you have to use 'TH' too, since it is a word. If you don't like that try defining 'ST', 'ND' and 'RD' yourself:

```
: st th ;
: nd th ;
: rd th ;
4 sixteen 1 st !
5 sixteen 2 nd !
6 sixteen 3 rd !
```

Isn't 4tH wonderful? Fetching is done the same of course:

```
sixteen 0 th @
sixteen 4 th @
```

Plain and easy.

7.16 Copying arrays of numbers

If you want to move chunks of data around, there is 'SMOVE':

```
1024 array a
1024 array b

a b 512 smove
```

This wil define two arrays of 1024 cells. 'SMOVE' will move the first 512 cells of array "a" to array "b".

7.17 Declaring and using constants

Declaring a simple constant is easy too. Let's say we want to make a constant called "FIVE":

```
5 constant five
```

Now you can use "FIVE" like you would '5'. E.g. this will print five spaces:

```
five spaces
```

The same rules for declaring words apply for constants. You can't use a name that already has been taken. A constant is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive. By the way, '5' is a literal expression. You can't take it from the stack or calculate it.

A special kind of constant is the so-called "plus-constant". This constant will automatically add itself to the top of the stack when executed, e.g.:

```
10 +constant tenplus
20 tenplus .
```

Will print "30". First we define a '+CONSTANT' named "tenplus", then we throw "20" on the stack, finally we execute "tenplus" and print the result. It is equivalent to:

```
10 constant ten
20 ten + .
```

Yes, you guessed it, a '+CONSTANT' is a constant with built-in addition! Cool, huh? And if you thought it couldn't get any better, there are also '*CONSTANT' and '/CONSTANT', which have built-in multiplication and division.

7.18 Built-in constants

There are several built-in constants. Of course, they are all literals in case you wonder. Here's a list. Refer to the glossary for a more detailed description:

```
/PAD
/TIB
/HOLD
/CELL
/CHAR
MAX-N
MAX-CHAR
CHAR-BITS
(ERROR)
BL
FALSE
LO
APP
PAD
STACK-CELLS
TIB
TRUE
VARS
WIDTH
INPUT
OUTPUT
STDOUT
STDIN
APPEND
PIPE
FILES
4TH#
```

7.19 Using booleans

Booleans are expressions or values that are either true or false. They are used to conditionally execute parts of your program. In 4tH a value is false when it is zero and true when it is non-zero. Most booleans come into existence when you do comparisons. This example will determine whether the value in variable 'VAR' is greater than 5. Try to predict whether it will evaluate to true or false:

```
variable var
4 var !
var @ 5 > .
```

No, it wasn't! But hey, you can print booleans as numbers. Well, they are numbers. But with a special meaning as we will see in the next section.

7.20 IF-ELSE constructs

Like most other languages you can use IF-ELSE constructs. Let's enhance our previous example:

variable var

```
4 var !
var @ 5 >
if ." Greater" cr
else ." Less or equal" cr
then
```

So now our program tells you when it's greater and when not. Note that contrary to other languages the condition comes before the 'IF' and 'THEN' ends the IF-clause. In other words, whatever path the program takes, it always continues after the 'THEN'. A tip: think of 'THEN' as 'ENDIF'..

7.21 FOR-NEXT constructs

4tH has FOR-NEXT constructs as well. The number of iterations is known in this construct. E.g. let's print the numbers from 1 to 10:

```
11 1 do i . cr loop
```

The first number represents the limit. When the limit is reached or exceeded the loop terminates. The second number presents the initial value of the index. That's where it starts off. So remember, this loop iterates at least once! You can use '?DO' instead of 'DO'. That will not enter the loop if the limit and the index are the same to begin with:

```
0 0 ?do i . cr loop
```

'i' represents the index. It is not a variable or a constant, it is a predefined word, which puts the index on the stack, so '.' can get it from the stack and print it.

But what if I want to increase the index by two? Or want to count downwards? Is that possible. Sure. There is another construct to do just that. Okay, let's take the first question:

```
11 1 do i . cr 2 +loop
```

This one will produce exactly what you asked for. An increment by two. This one will produce all negative numbers from -1 to -10:

```
-11 -1 do i . cr -1 +loop
```

Note that the step is not a literal expression. You can change the step if you want to, e.g.:

```
32767 1 do i . i +loop
```

This will print: 1, 2, 4, 8, all up to 16384. Pretty flexible, I guess. You can break out of a loop by using 'LEAVE'. Note that 'LEAVE' only sets the index to the value of the limit: it doesn't branch or anything. Make sure that there is no code left between 'LEAVE' and 'LOOP' that you don't want to execute. So this is okay:

```
10 0 do i dup 5 = if drop leave else . cr then loop
```

And this is not:

```
10 0 do i dup 5 = if drop leave then . cr loop
```

Since it will still get past the '.' before leaving. In this case you will catch the error quickly, because the stack is empty.

7.22 WHILE-DO constructs

A WHILE-DO construction is a construction that will perform zero or more iterations. First a condition is checked, then the body is executed. Then it will branch back to the condition. In 4tH it looks like this:

BEGIN <condition> WHILE <body> REPEAT

The condition will have to evaluate to TRUE in order to execute the body. If it evaluates to FALSE it branches to just after the REPEAT. This example does a Fibbonaci test.

```
: fib 0 1
  begin
    dup >r rot dup r> >     \ condition
  while
    rot rot dup rot + dup . \ body
  repeat
  drop drop drop ;          \ after loop executed
```

You might not understand all of the commands, but we'll get to that. If you enter "20 fib" you will get:

```
1 2 3 5 8 13 21
```

This construct is particularly handy if you are not sure that all data will pass the condition.

7.23 REPEAT-UNTIL constructs

The counterpart of WHILE-DO constructs is the REPEAT-UNTIL construct. This executes the body, then checks a condition at 'UNTIL'. If the expression evaluates to FALSE, it branches back to the top of the body (marked by 'BEGIN') again. It executes at least once. This program calculates the largest common divisor.

```
: lcd
  begin
    swap over mod          \ body
    dup 0=                 \ condition
  until drop . ;
```

If you enter "27 21 lcd" the programs will answer "3".

7.24 Infinite loops

In order to make an infinite loop one could write:

```
begin ." Diamonds are forever" cr 0 until
```

But there is a nicer way to do just that:

```
begin ." Diamonds are forever" cr again
```

This will execute until the end of times, unless you exit the program another way.

7.25 Including source files

4tH has a vocabulary of over 200 words. If you use them in one of your 4tH programs 4tH will recognize them instantly. These words are *internal*.

But if you take a look at the glossary, you'll find that there are a lot of other words too. Words that 4tH will not recognize; they have to be *included* first. These words are *external*.

These words are defined in an *include file*. An include file is just an ordinary ASCII file with 4tH source. You can read them if you want. In order to use these words, you have to tell 4tH where it can find the include file.

This is done by the '[NEEDS' directive, which is equivalent to the COMUS word 'INCLUDE' (which 4tH also supports). Everything up to the next "]" is considered to be a filename, so the path may contain embedded spaces. You can use absolute paths or relative paths, *just make sure that you're starting 4tH from the proper directory*. E.g. this one includes additional ANS-Forth CORE-words from the directory just above 'lib'[1]:

```
[needs lib/anscore.4th]
```

Or:

```
include lib/anscore.4th
```

4tH comes with a rich library of words, which covers a large part of ANS-Forth and COMUS[2] standard words and beyond. They are all located in the 'lib' directory. In the next level we're going to need a lot of these words, so you'd better know how to include them.

[1] If you're not sure where that is, enter the 'lib' directory and execute "cd ..".

[2] In case you wonder, COMUS stands for COMmon USage.

7.26 Getting a number from the keyboard

The word to enter a number from the keyboard can be found in the 'lib' directory and is defined in the `enter.4th` file. To include it you have to tell 4tH. We *assume* your working directory is just above the 'lib' directory[3]:

```
[needs lib/enter.4th]
```

That's all! Now you can use 'ENTER' just like any 4tH word. This will allow you to enter a number and print it:

```
[needs lib/enter.4th]
enter . cr
```

By the way, this is the end of the first level. Take our advise and give it a try!

[3] As a matter of fact, we will *always* assume that! If you don't know what we mean, execute "cd <path to lib directory>" and then "cd ..". Now you're there for sure! Note none of this is any concern to you if you set the **DIR4TH** variable.

Chapter 8

4tH arrays

8.1 Aligning numbers

You may find that printing numbers in columns (I prefer "right-aligned") can be pretty hard. That is because the standard word to print numbers ('.') prints the number and then a trailing space. That is why '.R' was added.

The word '.R' works just like '.' but instead of just printing the number with a trailing space '.R' will print the number right-aligned in a field of N characters wide. Try this and you will see the difference:

```
140 . cr
150 5 .r cr
```

In this example the field is five characters wide, so '150' will be printed with two leading spaces.

8.2 Creating arrays of constants

Making an array of constants is quite easy. First you have to define the name of the array by using the word 'TABLE' or 'CREATE' (which is ANS-Forth). Then you specify all its elements. Note that every element is a literal expression. All elements (even the last) are terminated by the word ','. An example:

```
create sizes 18 , 21 , 24 , 27 , 30 , 255 ,
```

Please note that ',' is a word! It has to be separated by spaces on both ends.

8.3 Using arrays of constants

Accessing an array of constants is very much like accessing an array of numbers. In an array of numbers you access the 0th element like this:

```
sixteen 0 th @
```

When you access the first element of an array of constants you use this construction:

```
sizes 0 th @c
```

The only difference is the word '@C', which is exclusively used to access arrays of constants.

8.4 Using values

A value is a cross-over between a variable and a constant. May be this example will give you an idea:

declaration:

```
variable a                  ( No initial value)
1 constant b                ( Literal expression assigned at compiletime)
2 b + value c               ( Expression assigned at runtime)
```

fetching:

```
a @                         ( Variable throws address on stack)
b                           ( Constant throws value on stack)
c                           ( Value throws value on stack)
```

storing:

```
2 b + a !                   ( Expression can be stored at runtime)
                            ( Constant cannot be reassigned)
2 b + to c                  ( Expression can be stored at runtime)
```

In many aspects, values behave like variables and can replace variables. The only thing you cannot do is make arrays of values. A value is *not* a literal expression either, so you can't use them to size arrays. In fact, a value is a variable that behaves in certain aspects like a constant.

Note that although 'VALUE' and 'TO' are aliases, it is more portable and more readable to use 'VALUE' for declaration and 'TO' for reassignment.

8.5 Creating string variables

In 4tH you have to define the maximum length of the string, like Pascal:

```
10 string name
```

You cannot add the 'CHARS' keyword, since 'STRING' already implies that you are creating an array of characters. Note that the string variable includes the terminator. That is a special character that tells 4tH where the string ends (see section 8.14). You usually don't have to add that yourself because 4tH will do that for you. But you will have to reserve space for it.

That means that the string "name" we just declared can contain up to nine characters *AND* the terminator. These kind of strings are usually referred to as ASCIIZ strings.

E.g. when you want to define a string that has to contain "Hello!" (without the quotes) you have to define a string that is at least 7 characters long:

```
7 string hello
```

8.6 What is an address?

An address is a location in memory. Usually, you don't need to know addresses, because 4tH will take care of that. But if you want it, you can retrieve them as we will show you later. Think of memory like a city. It has roads and houses and inhabitants. There are three roads in 4tH city:

1. INTEGER SEGMENT, that is where the cells live;
2. CHARACTER SEGMENT, that is where the strings live;
3. CODE SEGMENT, that is where the instructions that form your program live.

If you want to visit a certain person, you go to the city where he lives, find the right street and knock on the door. If you want to retrieve a certain string or integer, you do the same.

When you define a string, you actually create a constant with the address of that string. When you later refer to the string you just defined its address is thrown on the stack. An address is simply a number that refers to its location. As you will see you can work with string-addresses without ever knowing what that number is. But *because* it is a number you can manipulate it like any other number. E.g. this is perfectly valid:

```
16 string hello

hello                       \ address of string on stack
dup                         \ duplicate it
drop drop                   \ drop them both
```

Later, we will tell you how to get "Hello!" into the string.

8.7 String literals

In 4tH a string literal is created by the word 'S"'. The word 'S"' is very much like '."', but instead of printing it to the screen you will just be defining a string literal.

```
s" This is a string"
```

4tH is a stack oriented language, so what does 'S"' leave on the stack? In 4tH, a string is usually represented by on the stack by its address and its count. So in order to get its length, you only have to get the first value on the stack. In order to get its address you have to get the second value on the stack, which is demonstrated by this small program:

```
s" This is a string"        \ create a temporary string
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

And what about string literals with quotes. Easy, there is an equivalent to 'S"' that does the same thing:

```
s| "This is a string with quotes"|
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Instead of a quote, the string is delimited by a bar. And what about string literals that include them both? Bad luck? Well, almost but not quite. Just take a look at section 13.21.

8.8 String constants

String constants work the same way as numeric constants:

```
10 constant ten             \ define a string constant
ten . cr                    \ equivalent to: 10 . cr
```

In fact, you give a name to a literal value. After that, you can refer to that literal throughout your program by using its name. String constants do the same thing. Take a look at this little piece of code:

```
s" This is a string"        \ create a temporary string
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Now we do the same thing, but this time we define a string constant by using 'SCONSTANT':

```
s" This is a string" sconstant mystring
                            \ define a string constant
mystring                    \ now we use the string constant
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Why use string constants? Well, first of all, if you use a string constant throughout your program, it will save you some editing when you have to change your program for one reason or another. Second, it will make your program a little smaller.

8.9 Initializing string variables

You can initialize a string with the 'S"' word. If you want the string to contain your first name use this construction:

```
s" Hello!" name place
```

The word 'PLACE' copies the contents of a string literal into a string-variable.

If you still don't understand it yet, don't worry. As long as you use this construction, you'll get what you want. Just remember that assigning a string literal to a string that is too short will result in an error or even worse, corrupt other strings.

8.10 Initializing a NULL string variable

If you're not sure what that means, it means we're initializing a string variable to an empty string. Well, that's very easy:

```
0 name c!
pad 0 name place
0 dup name place
```

Choose any of these three. And all these constucts are compatible with ANS-Forth.

8.11 Getting the length of a string variable

You get the length of a string variable by using the word 'COUNT'. It will not only return the length of the string variable, but also the string address. It is illustrated by this short program:

```
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting count              \ get string length
." String length: " . cr    \ print the length
drop                        \ discard the address
```

Most string handling words return or take an address/count pair. One of the exceptions is the string variable itself (see section 8.9). To copy the contents of an address/count pair represented string into a string variable, we use 'PLACE'. In order to convert a string variable back to an address/count pair represented string, we use 'COUNT':

```
32 string my-string         \ create a string variable
                            \ create an address/count
s" This is a string"        \ pair represented string
my-string place             \ copy it into the variable
my-string count             \ convert it into an address/count pair
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Note that the contents of the string variable do not change by a 'COUNT' conversion!

8.12 Printing a string variable

Printing a string variable is pretty straight forward. The word that is required to print a string variable is 'TYPE'. It requires an address/count pair. Yes, that are the values that are left on the stack by 'COUNT'! So printing a string means issuing both 'COUNT' and 'TYPE':

```
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting count type cr      \ print the string
```

If you don't like this you can always define a word like 'PRINT$':

```
: print$ count type ;
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting print$ cr          \ print the string
```

8.13 Copying a string variable

You might want to copy one string variable to another. Let's take a look at this example:

```
32 string one                \ define the first string
32 string two                \ define the second string

s" Greetings!" one place     \ initialize string one
one count                    \ get the length of string one
two place                    \ and copy it into string two
two count type cr            \ print string two
```

First we place the string "Greetings!" into a string variable. 'S"' will put an address/count pair on the stack, that is consumed by 'PLACE'. Variable "ONE" only puts its address on the stack, that is converted into an address/count pair by 'COUNT'. After it has been consumed again by 'PLACE' we need 'COUNT' again to provide 'TYPE' with an address/count pair.

8.14 The string terminator

In order for 'COUNT' to work, it has to know where the string stops. So a special character at the end of the string, the string terminator, is used to indicate the end of an ASCIIZ string. It has nothing to do with Arnold Schwarzenegger obliterating innocent strings! It is simply a character, having the ASCII value zero. It may also be referred to as the NULL-character. Although most strings in 4tH will be terminated automatically it is considered bad style to rely on that.

If you have doubts, you can always convert an address/count pair to a terminated string by applying '>STRING', although you have to take care that nothing is overwritten and enough space is available. As a rule of the tumb you might say that '>STRING' can safely be applied to all string variables.

8.15 Slicing strings

Slicing strings is just like copying strings. We just don't copy all of it and we don't always start copying at the beginning of a string. We'll show you what we mean:

```
32 string one                \ define string one
s" Hans Bezemer" one place   \ initialize string one
one count 2dup type cr       \ duplicate and print it
1 /string                    \ move one character forward
2dup type cr                 \ duplicate and print it again
1 /string                    \ move one character forward
2dup type cr                 \ duplicate and print it again
1 /string                    \ move one character forward
type cr                      \ print it for the last time
```

First it will print "Hans Bezemer", then "ans Bezemer", then "ns Bezemer" and finally "s Bezemer". The word '/STRING' adjusts the address/count pair by a given number of characters, in this case one character. The word '2DUP' is much like 'DUP', but it copies the top *two* values on the stack. It is functionally equivalent to:

```
over over
```

If we want to discard the first name at all we could even write:

```
32 string one                \ define string one
s" Hans Bezemer" one place   \ initialize string one
one count 5 /string type cr  \ print sliced string
```

The five characters we want to skip are the first name (which is four characters) and a space (which adds up to five). There is a special word for slicing strings in the library member `slice.4th`. You call it with:

```
address count position-to-start position-to-end
```

Both positions start counting at zero. So this will copy the first name to string "two" and print it:

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 0 3 slice          \ slice the first name
two place                    \ copy it to string two
two count type cr            \ print string two
```

This will slice the last name off and store it in string "two":

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 5 11 slice         \ slice the last name
two place                    \ copy it to string two
two count type cr            \ print string two
```

Since the last name is seven characters long and starts at position five (start counting with zero!).

8.16 Appending strings

The word '+PLACE[1]' appends two strings. In this example string "one" holds the first name. The second string literal is appended to string "one" to form the full name. Finally string "one" is printed.

```
32 string one                \ define string one

s" Hans " one place          \ initialize first string
s" Bezemer" one +place       \ append 'Bezemer' to string
one count type cr            \ print first string
```

[1]There is a COMUS word called 'APPEND' which works exactly the same.

8.17 Comparing strings

If you ever sorted strings you know how indispensable comparing strings is. As we mentioned before, there are very few words in Forth that act on strings. Here is a word that can compare two strings. It is located in the library member `compare.4th`.

```
[needs lib/compare.4th]
                              \ compare two chars
32 string one                 \ define string one
s" Hans Bezemer" one place    \ initialize string one
32 string two                 \ define string two
s" HANS BEZEMER" two place    \ initialize string two

one count two count compare   \ compare two strings
if
  ." Strings differ"          \ message: strings ok
else
  ." Strings are the same"    \ message: strings not ok
then
cr                            \ send CR
```

Simply pass two strings (represented by their address/count pairs) to 'COMPARE' and it will return a TRUE flag when the strings are different. This might seem a bit odd, but `strcmp()` does exactly the same. If you don't like that you can always add '0=' to the end of 'COMPARE' to reverse the flag.

You'll soon find out that ANS-Forth's 'COMPARE' is case sensitive. Lucky for you, you can modify the behaviour of 4tH's 'COMPARE'. Just define this *before* the '[NEEDS' directive:

```
[pragma] casesensitive
[needs lib/compare.4th]
                              \ compare two chars
32 string one                 \ define string one
s" Hans Bezemer" one place    \ initialize string one
32 string two                 \ define string two
s" HANS BEZEMER" two place    \ initialize string two

one count two count compare   \ compare two strings
if
  ." Strings differ"          \ message: strings ok
else
  ." Strings are the same"    \ message: strings not ok
then
cr                            \ send CR
```

Now 'COMPARE' will do a case sensitive comparison.

8.18 Finding a substring

Sometimes you need to find a string within a string. ANS-Forth has defined a word for that too. It is called 'SEARCH'. You need to include `search.4th` in order to use it. Now lets find "the" in this string:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search                \ search for 'the'
0= if ." not " then ." found: "
type                          \ print the result
```

'SEARCH' always returns a flag and a address/count pair. If it returns true, the substring was found; if it returns false, the substring was *not* found. Now that's pretty straightforward, isn't it? That means that the small program above will print:

```
found:
```

When the substring was found and:

```
not found:
```

When the substring was not found. But what kind of string does it return when the substring was not found? Well, the entire string you fed it, so this *would have been* its output if we had been looking for the substring "now" instead of "the":

```
not found: How the cow catches the hare
```

But in this specific example we are looking for "the". When found, 'SEARCH' returns the string after the first occurrence of the substring we were looking for:

```
found: the cow catches the hare
```

Why that? Why not a position? Well, first of all, you can look for the same substring again:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search drop          \ drop the flag
2dup type                    \ print the string
s" the" search drop          \ now search again
type                         \ print the string
```

This will print:

```
the cow catches the hare
the hare
```

But if you still want to see a position instead of a string, you can simply define this:

```
[needs lib/search.4th]

: position
  2>r over swap 2r> search 0= >r drop swap - r> if 1- then
;

s" How the cow catches the hare"
s" the" position . cr
```

That will take care of your problems. If the substring was found, "POSITION" will return a positive number. If it wasn't found, it will return a negative number. Note that 'SEARCH' can be persuaded to do a case-sensitive comparison, just like 'COMPARE':

```
[pragma] casesensitive
[needs lib/search.4th]
```

Now 'SEARCH' will do a case sensitive comparison, just like 'COMPARE'.

8.19 Replacing substrings

Sometimes finding is not enough. You have replace it by something else. You can do that very easily with 4tH. Just include `replace.4th`. It contains a word that will do all that. Take this example:

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the" s" a"
replaceall type cr
```

It will print:

```
How a cow catches a hare
```

Yes, this one replaces all occurrences of ”the” by ”a”. Note that like ’COMPARE’ and ’SEARCH’ this one can be made case sensitive too:

```
[pragma] casesensitive
[needs lib/replace.4th]
```

8.20 Deleting substrings

Yes, we even got a word for ’search-and-destroy’ missions. You only have to include `replace.4th`:

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the"
deleteall type cr
```

This will print:

```
How cow catches hare
```

Yes, it deletes all occurrences of ”the”. Note that like ’COMPARE’, ’SEARCH’ and ’REPLACE’ this one can be made case sensitive too:

```
[pragma] casesensitive
[needs lib/replace.4th]
```

8.21 Removing trailing spaces

You probably know the problem. The user of your well-made program types his name and hits the spacebar before hitting the enter-key. There you go. His name will be stored in your datafile with a space and nobody will ever find it.

In 4tH there is a special word called ’-TRAILING’ that removes the extra spaces at the end with very little effort. Just paste it after ’COUNT’. Like we did in this example:

```
32 string one                \ define a string
s" Hans Bezemer    "         \ string with trailing spaces
one place                    \ now copy it to string one

one dup                      \ save the address

." ["                        \ print a bracket
count type                   \ old method of printing
." ]" cr                     \ print bracket and newline

." ["                        \ print a bracket
count -trailing type         \ new method of printing
." ]" cr                     \ print a bracket and newline
```

You will see that the string is printed twice. First with the trailing spaces, second without trailing spaces.

8.22 Removing leading spaces

And what about leading spaces? Patience, old chap. You've got a lot of ground to cover. There is no built-in word for that, but we can use a library member like we did in this example:

```
[needs lib/leading.4th]

32 string one                \ define a string
s"     Hans Bezemer"         \ string with leading spaces
one place                    \ now copy it to string one

one dup                      \ save the address

." ["                        \ print a bracket
count type                   \ old method of printing
." ]" cr                     \ print bracket and newline

." ["                        \ print a bracket
count -leading type          \ new method of printing
." ]" cr                     \ print a bracket and newline
```

You will see that the string is printed twice. First with the leading spaces, second without leading spaces. Happy?

8.23 Upper and lower case

Sometimes you will have to convert a string to upper or lower case. 4tH has a library member for that too. Just include:

```
[needs lib/ulcase.4th]
```

This will define several easy to use conversion words. E.g. in order to convert a string to upper case, just enter:

```
s" Convert this!" s>upper    \ convert addr/count string to uppercase
type cr                      \ type the string
```

Its lower case counterpart is:

```
s" Convert this!" s>lower    \ convert addr/count string to lowercase
type cr                      \ type the string
```

Like most string words it takes and returns an address/count pair. Note that the string in question is *modified*, so if you still need the original, copy it first. You can also convert an individual character:

```
char A char>lower emit       \ convert a character and show it
```

And consequently, its counterpart is:

```
char a char>upper emit       \ convert a character and show it
```

These words take an ASCII value from the stack, convert it and put the converted ASCII value back on the stack. If the value does not represent a alphabetic character, it is left unchanged.

8.24 String literals and string variables

Most computer languages allow you to mix string literals and string variables. Not in 4tH. In 4tH they are two distinct datatypes. To print a string literal you use the word '."'. To print a string variable you use the 'COUNT TYPE' construction.

There are only three different actions you can do with a string literal. First, you can define one using 'S"'. Second, you can print one using '."' Finally, you can compile a string into your program using ',"'.

This may seem a bit mind-boggling to you now, but we'll elaborate a bit further on this subject later.

8.25 Printing individual characters

"I already know that!"

Sure you do. If you want to print "G" you simply write:

```
." G"
```

Don't you? But what if you want to use a TAB character (ASCII 9)? You can't type in that one so easily, huh? You may even find it doesn't work at all!

Don't ever use characters outside the ASCII range 32 to 127 decimal. It may or may not work, but it won't be portable anyway. the word 'EMIT' may be of some help. If you want to use the TAB-character simply write:

```
9 emit
```

That works!

8.26 Distinguishing characters

Like in a novel, not all characters are created equal. There are upper case characters, lower case characters, control characters, whitespace, etc. Sometimes it is necessary to find out what kind of character we are dealing with. Of course, 4tH can help you there. You need to include `istype.4th` in order to use it:

```
char a is-lower . cr
char a is-upper . cr
```

4tH will first print a TRUE value (because 'a' is a lower case character) and then a FALSE value. This table tells you what words 4tH offers and the ranges of valid characters:

WORD	RANGE (ASCII)	DESCRIPTION
IS-ASCII	0 - 127	All 7-bit ASCII characters
IS-PRINT	32 - 127	As above, without control characters
IS-WHITE	0 - 32	All control characters plus space
IS-DIGIT	'0' - '9'	All digits
IS-LOWER	'a' - 'z'	All lower case characters
IS-UPPER	'A' - 'Z'	All upper case characters
IS-ALPHA	'a'-'z', 'A' - 'Z'	All alphabetic characters
IS-ALNUM	'0' - '9', 'a' - 'z', 'A' - 'Z'	All alphanumeric characters

Table 8.1: Character typing words

8.27 Getting ASCII values

Ok, 'EMIT' is a nice addition, but it has its drawbacks. What if you want to emit the character "G". Do you have to look up the ASCII value in a table? No. 4tH has another word that can help you with that. It is called 'CHAR'. This will emit a "G":

```
char G emit
```

The word 'CHAR' looks up the ASCII-value of "G" and leave it on the stack. You can also use '[CHAR]'. It does exactly the same thing. It is included for compatibility with ANS-Forth versions. Note that 'CHAR' only works with printable characters (ASCII 33 to 127 decimal).

8.28 Printing spaces

If you try to print a space by using this construction:

```
char   emit
```

You will notice it won't work. Sure, you can also use:

```
."   "
```

But that isn't too elegant. You can use the built-in constant 'BL' which holds the ASCII-value of a space:

```
bl emit
```

That is much better. But you can achieve the same thing by simply writing:

```
space
```

Which means that if you want to write two spaces you have to write:

```
space space
```

If you want to write ten spaces you either have to repeat the command 'SPACE' ten times or use a DO-LOOP construction, which is a bit cumbersome. Of course, 4tH has a more elegant solution for that:

```
10 spaces
```

Which will output ten spaces. Need I say more?

8.29 Fetching individual characters

Take a look at this small program:

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
```

What is the second character of string "one"? Sure, its an "a". But how can you let your program determine that? You can't use '@' because that word can only access variables.

Sure, you can do that in 4tH, but it requires a new word, called 'C@'. Think of a string as an array of characters and you will find it much easier to picture the idea. Arrays in 4tH always start with zero instead of one. So accessing the first character might be done with:

```
one 0 th c@
```

We do not recommend using this construction, although it will work perfectly. If you never want to convert your program to Forth you might even choose to keep it that way. We recommend the construction:

```
one 0 chars + c@
```

Which is slightly more wordy. 4tH will compile both constructions in exactly the same way. Anyway, accessing the second character is easy now:

```
one 1 chars + c@
```

This is the complete program:

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
one 1 chars + c@              \ get the second character
emit cr                       \ print it
```

8.30 Storing individual characters

Storing individual characters works just the same. Keep that array of characters in mind. When we want to fetch a variable we write:

```
my_var @
```

When we want to store a value in a variable we write:

```
5 my_var !
```

Fetching only requires the address of the variable. Storing requires both the address of the variable *and* the value we want to store. On top of the stack is the address of the variable, below that is value we want to store. Keep that in mind, this is very important.

Let's say we have this program:

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
```

Now we want to change "Hans" to "Hand". If we want to find out what the 4th character of string "one" is we write:

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
one 3 chars + c@                \ get the fourth character
```

Remember, we start counting from zero! If we want to store the character "d" in the fourth character, we have to use a new word, and (yes, you guessed it right!) it is called 'C!':

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
one 3 chars +                   \ address of the fourth char
char d                          \ we want to store 'd'
swap                            \ get the order right
c!                              \ now store 'd'
```

If we throw the character "d" on the stack before we calculate the address, we can even remove the 'SWAP':

```
32 string one                   \ define string one
char d                          \ we want to store 'd'
s" Hans" one place              \ initialize string one
one 3 chars +                   \ address of the fourth char
c!                              \ now store 'd'
```

We will present the very same programs, but now with stack-effect-diagrams in order to explain how this works. We will call the index 'i', the character we want to store 'c' and the address of the string 'a'. By convention, stack-effect-diagrams are enclosed by parenthesis.

If you create complex programs this technique can help you to understand more clearly how your program actually works. It might even save you a lot of debugging. This is the first version:

```
32 string one                  ( --)
s" Hans" one place             ( --)
one 3 chars                    ( a i)
+                              ( a+i)
char d                         ( a+i c)
swap                           ( c a+i)
c!                             ( --)
```

Now the second, optimized version:

```
32 string one                  ( --)
char d                         ( c)
s" Hans" one place             ( c)
one 3 chars                    ( c a i)
+                              ( c a+i)
c!                             ( --)
```

8.31 Getting a string from the keyboard

Of course, you don't want to initialize strings all your life. Real applications get their input from the keyboard. We've already shown you how to get a number from the keyboard. Now we turn to strings.

When programming in BASIC, strings usually have an undefined length. Some BASICs move strings around in memory, others have to perform some kind of "garbage-collection". Whatever method they use, it takes up memory and processor-time.

4tH forces you to think about your application. E.g. when you want to store somebodies name in a string variable, 16 characters will be too few and 512 characters too many. But 64 characters will probably do.

But that poses a problem when you want to get a string from the keyboard. How can you prevent that somebody types a string that is just too long? And how do you terminate it?

The word 'ACCEPT' takes two arguments. First, the string variable where you want to save the input and second, the maximum number of characters it can take. It automatically terminates the string when reading from the keyboard. But there is a catch. This program can get you into trouble:

```
64 constant #name              \ length of string
#name string name              \ define string 'name'

name #name accept              \ input string
name swap type cr              \ swap count and print
```

Since 64 characters *plus* the terminator add up to 65 characters. The word 'ACCEPT' always returns the number of characters it received. You will find that you won't need that information most of the time.

This is the end of the second level. Now you should be able to understand most of the example programs and write simple ones. I suggest you do just that. Experience is the best teacher after all.

Chapter 9

Character Segment

9.1 The Character Segment

Wonder where all these strings are created? I bet you do. Well, when you define a string, memory is allocated in the Character Segment. When you define another one, space is allocated after the first string. That means that if you go beyond the boundaries of the first string, you'll end up in the space allocated to the second string.

After the second string there is a void. If you end up there your program will end with an error-message. And what about the space before the first string? Well, take a look at figure 9.1.

The lower memory is at the bottom. Yes, before your strings there are two other areas, the TIB and the PAD. We'll elaborate on that in the next section.

The Character Segment is created at run-time. That means that it isn't there when you compile a program. The compiler just keeps track of how much memory would be needed to create such a Character Segment and stores that information in the header.

When you run the program the header is read first. Then the Character Segment is created, so it is already there when your program starts executing. When you exit the program, the Character Segment is destroyed and all information stored there is lost (unless you save it first).

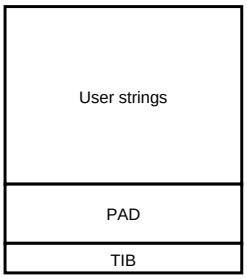

Figure 9.1: Character segment

9.2 What is the TIB?

The TIB stands for "Terminal Input Buffer" and is used by one single, but very important word called 'REFILL'. In essence, 'REFILL' does the same thing as 'ACCEPT', except that it has a dedicated area to store its data and sets up everything for parsing. Whatever you type when you call 'REFILL', it is stored in the TIB.

9.3 What is the PAD?

The PAD is short for "scratch-pad". It is a temporary storage area for strings. It is heavily used by 4tH itself, e.g. when you print a number the string is formed in the PAD. Yes, that's right: when you print a number it is first converted to a string. Then that string is 'COUNT'ed and 'TYPE'd. You can even program that subsystem yourself as we will see when we encounter formatted numbers (see section 9.8).

Furthermore, string constants (compiled by 'S"' or ',"') are temporarily stored in the PAD. Finally, 'NUMBER' and 'ARGS' also use the PAD. The PAD is actually a circular buffer. That means that strings are stored in the PAD until it runs out of space. Then it starts to overwrite the oldest strings. Usually, they have turned into garbage that is no longer used, but sometimes they still have some significance to your program. In that case, you'll have to save the string that was overwritten into a variable. Don't rely on the PAD to keep your strings alive!

9.4 How do I use TIB and PAD?

In general, you don't. The TIB is a system-related area and it is considered bad practice when you manipulate it yourself. The PAD can be used for temporary storage, but beware! Temporary really means temporary. A few words at the most, provided you don't use any string constants.

Think of both these areas as predefined strings. You can refer to them as 'TIB' and 'PAD'. You don't have to declare them in any way. This program is perfectly alright:

```
s" Hello world" pad place      \ store a string in pad
pad count type cr              \ print contents of the pad
```

If you want to know how big TIB and PAD are, you can use the predefined constants '/TIB' and '/PAD':

```
." Size of TIB: " /TIB . cr    \ print sizeof TIB
." Size of PAD: " /PAD . cr    \ print sizeof PAD
```

Note, this does *not* print the length of a string stored in the area, but the maximum size of the string that can be stored there. Some space of the PAD is reserved for number generation (see section 9.3). You can get the size of this area by the predefined constant '/HOLD'. This will print the size of this area and the size of PADs circular buffer:

```
." Size of HOLD  : " /HOLD . cr \ print sizeof HOLD
." Size of buffer: " /PAD /HOLD - . cr
```

If that area did not exist even printing a number could corrupt the circular buffer. In some unusual circumstances, the PAD can get corrupted. If so, identify the temporary string that gets corrupted and store it explitly into a string variable.

9.5 Simple parsing

We have already discussed 'REFILL' a bit. We've seen that it is closely related to 'ACCEPT'. 'REFILL' returns a true flag if all is well. When you use the keyboard it usually is, so we can safely drop it, but we will encounter a situation where this flag comes in handy.

If you want to get a string from the keyboard, you only have to type:

```
refill drop                    \ get string from keyboard
```

Every next call to 'REFILL' will overwrite any previously entered string. So if you want to do something with that string you've got to get it out of there, usually to one of your own strings.

But if accessing the TIB directly is not the proper way, what is? The use of 'REFILL' is closely linked to the word 'PARSE-WORD', which is a parser. 'PARSE-WORD' looks for the delimiter, whose ASCII code is on the stack.

If the string starts with the delimiter, it will skip this and all subsequent occurrences until it finds a string. Then it will look for the delimiter again and slice the string right there. It then returns its address and count.

This extremely handy when you want to obtain filtered input. E.g. when you want to split somebodies name into first name, initials and lastname:

```
Hans L. Bezemer
```

Just use this program:

```
." Give first name, initials, lastname: "
refill drop                    \ get string from keyboard
bl parse-word                  \ parse first name
." First name: "               \ write message
type cr                        \ type first name
bl parse-word                  \ parse initials
." Initials  : "               \ write message
type cr                        \ type initials
bl parse-word                  \ parse last name
." Last name : "               \ write message
type cr                        \ write last name
```

You don't have to parse the entire string with the same character. This program will split up an MS-DOS filename into its components:

```
." DOS filename: " refill      \ input a DOS filename
drop cr                        \ get rid of the flag

char : parse-word              \ parse drive
." Drive: " type ." :" cr
                               \ print drive
begin
  char \ parse-word            \ parse path
  dup 0<>                      \ if not a NULL string
while                          \ print path
  ." Path : " type cr
repeat                         \ parse again
drop drop                      \ discard string
```

If 'PARSE-WORD' reaches the end of the string and the delimiter is still not found, it returns the remainder of that string. If you try to parse beyond the end of the string, it returns a NULL string. That is an empty string or, in other words, a string with length zero.

Therefore, we checked whether the string had zero length. If it had, we had reached the end of the string and further parsing was deemed useless.

9.6 Converting a string to a number

We now learned how to parse strings and retrieve components from them. But what if these components are numbers? Well, there is a way in 4tH to convert a string to a number, but like every number-conversion routine it has to act on invalid strings. That is, strings that cannot be converted to a valid number.

4tH uses an internal error-value, called '(ERROR)'. The constant '(ERROR)' is a strange number. You can't negate it, you can't subtract any number from it and you can't print it. If 4tHs number-conversion word 'NUMBER' can't convert a string it returns that constant. 'ERROR?' checks the return value and leaves an additional true flag if an error occured (which means: '(ERROR)' was returned). Let's take a look at this program:

```
." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  ." The number was: " . cr
then
```

You first enter a string, then it is parsed and 'PARSE-WORD' returns the address and count. 'NUMBER' tries to convert it. If 'NUMBER' returns '(ERROR)' it wasn't a valid string. Otherwise, the number is right on the stack, waiting to be printed. That wasn't so hard, was it?

9.7 Controlling the radix

If you are a programmer, you know how important this subject is to you. Sometimes, you want to print numbers in octal, binary or hex. 4tH can do that too. Let's take the previous program and alter it a bit:

```
." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  hex
  ." The number was: " . cr
then
```

We added the word 'HEX' just before printing the number. Now the number will be printed in hexadecimal. 4tH has a number of words that can change the radix, like 'DECIMAL' and 'OCTAL'. They work in the same way as 'HEX'.

4tH always starts in decimal. After that you are responsible. Note that all radix control follows the flow of the program. If you call a self-defined word that alters the radix all subsequent conversion is done too in that radix:

```
: .hex hex . ;                  \ print a number in hex

." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  ." The number was: " .hex cr
then
```

In this example not only that single number is printed in hex, but also all subsequent numbers will be printed in hex! A better version of the ".HEX" definition would be:

```
: .hex hex . decimal ;
```

Since that one resets the radix back to decimal. Words like 'HEX' do not only control the output of a number, but the input of numbers is also affected:

```
." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
hex                             \ convert hexadecimal
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  dup
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'NUMBER' will now also accept hexadecimal numbers. If the number is not a valid hexadecimal number, it will return '(ERROR)'. You probably know there is more to radix control than 'OCTAL', 'HEX' and 'DECIMAL'. No, we have not forgotten them. In fact, you can choose any radix between 2 and 36. This slightly modified program will only accept binary numbers:

```
: binary 2 base ! ;

." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
binary                          \ convert hexadecimal
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  dup                           \ both decimal and hex
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'BASE' is a predefined variable that enables you to select any radix between 2 and 36. This makes 4tH very flexible. However, this won't work:

```
hex 02B decimal . cr
```

4tH will try to compile "02B", but since it isn't a word or a valid decimal number, it will fail. Words like 'HEX' and the 'BASE' variable work only at run-time, not at compile-time! Isn't there a way to compile non-decimal numbers?

Sure, there is, although it is not that flexible. There are four words that control the interpretation of numbers at compile-time:

1. [BINARY]
2. [OCTAL]
3. [DECIMAL]
4. [HEX]

They work fundamentally different than their run-time equivalents. First, they only work at compile-time. Second, they are interpreted sequentially and do not follow the flow of the program. Let's take a look at these two programs:

```
[binary] 101 . cr
[octal] 101 . cr
[decimal] 101 . cr
[hex] 101 . cr
```

This will print the decimal numbers "5", "65", "101" and "257", since each one of them is compiled with a specific radix.

```
: binary 2 base ! ;
binary 101 . cr
octal 101 . cr
decimal 101 . cr
hex 101 . cr
```

Now the decimal number "101" is printed in four different radixes, since at compile-time the radix was set to decimal (which is the default). Now take a look at this program:

```
: do_binary [binary] ;
: do_decimal [decimal] ;
do_binary 101 decimal . cr
do_decimal 101 decimal . cr
```

The program will print "101" two times! Haven't we selected binary at compile-time? No, both '[BINARY]' and '[DECIMAL]' are interpreted sequentially!

When '[BINARY]' is encountered at the first time, it will set the radix at compile-time to binary. When '[DECIMAL]' is encountered in the second line, it will set the radix to decimal. When the third line it compiled, the radix is still set to decimal. If you want to make this program work, try this:

```
[binary]
101 decimal . cr
[decimal]
101 decimal . cr
```

When the first line is encountered, it sets the radix (at compile-time) to binary. So the number "101" at line two is compiled as a binary number. 'DECIMAL' will just be compiled. It will only influence the radix at run-time. The third line sets the radix at compile-time to decimal. So the number "101" at line four is compiled as a decimal number.

Since the run-time of 4tH starts up in decimal, both occurrences of 'DECIMAL' have little value. We can even eliminate 'DECIMAL' from the program altogether without affecting the result:

```
[binary] 101 . cr
[decimal] 101 . cr
```

Note that both the compile-time radix control words and the run-time radix control words stay in effect until they are superseded by others:

```
[binary]                        \ compile-time binary
101                             \ first binary number
1011                            \ second binary number
[decimal]                       \ compile-time decimal
5                               \ decimal 5
do                              \ set run-time radix
  i base !                      \ to loop-index
  dup . cr                      \ print number
loop
drop                            \ clean stack
```

9.8 Pictured numeric output

You probably have used this before, like when writing Basic. Never heard of "PRINT USING.."? Well, it is a way to print numbers in a certain format. Like telephone-numbers, time, dates, etc. Of course 4tH can do this too. In fact, you've probably used it before. Both '.' and '.R' use the same internal routines. They are called just before a number is printed.

This numeric string is created in the PAD and overwritten with each new call. But we'll go into that a bit later on.

What you have to remember is that you define the format reverse. What is printed first, is defined last in the format. So if you want to print:

```
060-5556916
```

You have to define it this way:

```
6196555-060
```

Formatting begins with the word '<#' and ends with the word '#>'. A single number is printed using '#' and the remainder of the number is printed using '#s' (which is always at least one digit). Let's go a bit further into that:

```
: print# <# #s #> type cr ;
256 print#
```

This simply prints a single number (since only '#S' is between the '<#' and the '#>' and goes to a new line. There is hardly any difference with '.'. You can try any (positive) number. Note that the values that '#>' leaves on the stack can directly be used by 'TYPE'.

This is a slightly different format:

```
: print3# <# # # # #> type cr ;
256 print3#
1 print3#
1000 print3#
```

This one will print "256", "001" and "000". Always the last three positions. The '#' simply stands for 'print a single digit'. So if you want to print a number with at least three digits, the format would be:

```
#s # #
```

That is: print the remainder of the number (at least one digit) and then two more. Now reverse it:

```
# # #s
```

Enclose it by '<#' and '#>' and add 'TYPE CR':

```
<# # # #s #> type cr
```

And that's it! Is it? Not quite. So far we've only printed *positive* numbers. If you try a negative number, you will find it prints garbage. This behavior can be fixed with the word 'SIGN'.

'SIGN' simply takes the number from the stack and prints a "-" when it is negative. The problem is that all other formatting words can only handle positive numbers. So we need the same number twice. One with the sign and one without. A typical signed number formatting word looks like:

```
: signed# dup abs <# #s sign #> type ;
```

Note the 'DUP ABS' sequence. First the number is duplicated (for 'SIGN') and then the absolute value is taken (for the other formatting words). So we got the number on the stack twice. First with sign (for 'SIGN'), second without sign (for the other formatting words). Does that make sense to you?

We can place 'SIGN' wherever we want. If we want to place the sign after the number (like some accountants do) we would write:

```
: account# dup abs <# sign #s #> type ;
```

But that is still not enough to write "$2000.16" is it? Well, in order to do that there is another very handy word called 'HOLD'. The word 'HOLD' just copies any character into the formatted number. Let's give it a try:

```
$2000.16
```

Let's reverse that:

```
61.0002$
```

So we first want to print two numbers, even when they are zero:

```
# # .0002$
```

Then we want to print a dot. This is where 'HOLD' comes in. 'HOLD' takes an ASCII code and places the equivalent character in the formatting string. We don't have to look up the ASCII code for a dot of course. We can use 'CHAR':

```
# # char . hold 0002$
```

Then we want to print the rest of the number (which is at least one digit):

```
# # char . hold #s $
```

Finally we want to print the character "$". Another job for 'HOLD':

```
# # char . hold #s char $ hold
```

So this is our formatting word:

```
: currency <# # # char . hold #s char $ hold #> type cr ;
```

And we call it like this:

```
200016 currency
```

You can do some pretty complex stuff with these formatting words. Try to figure out this one from the master himself, Leo Brodie:

```
: sextal 6 base ! ;
: :00 # sextal # decimal 58 hold ;
: time# <# :00 :00 #S #> type cr ;
3615 time#
```

Yeah, it prints the time! Pretty neat, huh? Now try the telephone-number we discussed in the beginning. That shouldn't be too hard. Still, you may think it's all a bit too complicated for your taste.

Well, there is a solution to that, but you'll need to include a library. Let's say you want to format that darn telephone number - or that currency thing:

```
include lib/picture.4th

605556916 s" ###-#######" picture type cr
200016 s" $?.##" picture type cr
```

That's much easier, isn't it? All you have to do is to issue a string and the library handles the whole thing. The library works very simple: there are five special formatting characters, the rest is copied verbatim. And yes, it handles the sign for you automatically - unless you've specified the position yourself with a '+' character.

You can also use this library to print number in a fixed width field, e.g.:

```
include lib/picture.4th

605616 s" $_____.__" picture type cr
```

This will print the number in a field nine characters wide, including the decimal point and the currency sign. Note you can print *any* character you want by postfixing it with a '!' - yes, even a '!'.

CHARACTER	MEANING
#	Prints a single digit
?	Prints the remainder of the number
_	Prints a single digit, unless all digits have been printed. Then a space is printed
+	Prints a '-' if the number is negative
!	Prints the previous character verbatim, even if it is a formatting character

Table 9.1: Picture library formatting characters

9.9 `printf()` like formatting

Some people can just not be pleased, neither by 4tH's native pictured numeric output, nor by the libraries we offer to make it all a bit easier. They just want `printf()`. And that's alright with me.

Although `printf()` is quite bulky, much slower and is known to have a few quirks of its own, it's not too hard too implement. 4tH has three `printf()` like functions:

printf Supports some basic specifiers, width and writes to 'STDOUT';

sprintf Supports some basic specifiers, one flag, width and writes to a string buffer;

fsprintf Supports most specifiers, all flags, width, precison and writes to a string buffer.

But how do you use it? First you have to include the proper file, either `printf.4th`, `sprintf.4th` or `fsprintf.4th`:

```
include lib/sprintf.4th
```

`printf()` like functions require a format string, optionally with embedded format specifiers. A format specifier has the following layout:

```
%[flags][width][.precision]specifier
```

Flags, width and precision are optional. If you want to print a decimal number, you use the "d" specifier. If you want to print a string, you use the "s" specifier, e.g.:

```
80 string mybuf
s" World" 100 s" A %d hello's from the %s!" mybuf sprintf type cr
```

And this will print(f):

```
A 100 hello's from the World!
```

You may wonder why the number comes before the string, but that is easily explained. First, this is 4tH - we do everything the other way around. Second, the format string is evaluated from left to right, so the number is consumed first. Consequently, it is on the top of the stack.

Now let's say we want to place the number in a field, five positions wide and pad it with spaces where needed:

```
80 string mybuf
s" World" 100 s" A %5d hello's from the %s!" mybuf sprintf type cr
```

Now it comes out this way:

```
A   100 hello's from the World!
```

Of course we can do the same thing for the string, but this time we make it ten positions wide:

```
80 string mybuf
s" World" 100 s" A %5d hello's from the %10s!" mybuf sprintf type cr
```

And this is what comes out of that one:

```
A   100 hello's from the      World!
```

Want to specify that width on the stack? You can:

```
80 string mybuf
s" World" 10 100 s" A %5d hello's from the %*s!" mybuf sprintf type cr
```

Again, the width of "10" on the stack is consumed before the string is printed. Default, everything is right-aligned, but you can change that with the "-" flag:

```
80 string mybuf
s" World" 10 100 s" A %5d hello's from the %-*s!" mybuf sprintf type cr
```

Which gives:

```
A   100 hello's from the World     !
```

Note that although some people may find it easier to use, these words are quite fragile. Make an error in the format string, the order or number of arguments and everything goes haywire. It's your choice.

It goes far beyond the scope of this manual to discuss all the possibilities of `fsprintf.4th`. Even more, because you can only use it in a floating point environment. And that is still to come[1]. For the time being you'll have to do with the following tables.

9.10 Converting a number to a string

Since there is no special word in 4tH which will convert a number to a string, we'll have to create it ourselves. In the previous section we have seen how a numeric string is created in the PAD. We can use this to create a word that converts a number to a string.

Because the PAD is highly volatile, we have to save the string immediately after its creation. So we'll create a word that not only creates the string, but places it directly in its proper location:

```
( n a -- )
: n>string >r dup abs <# #s sign #> r> place ;
```

It takes a number, the address of a string and returns nothing. Example:

```
16 string num$
-1024 num$ n>string
num$ count type cr
```

[1] In sections 12.23 and 12.24, to be exact.

flags	description	printf.4th	sprintf.4th	fsprintf.4th
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).	N	Y	Y
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.	N	N	Y
(space)	If no sign is going to be written, a blank space is inserted before the value.	N	N	Y
#	Used with o or x specifiers the value is preceeded with 0 or 0x respectively. Used with e, f, or g it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.	N	N	Y
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).	N	N	Y

Table 9.2: List of supported `printf()` flags

9.11 Aborting a program

Some conditions are so grave you can consider them to be fatal errors. In such cases the only thing you can do is abort the program as soon as possible. Of course, there is a way in 4tH to do just that. You can use either 'ABORT' or 'QUIT'. Same thing. Both will terminate your program immediately. This small program prints nothing:

```
abort
." This will never be printed." cr
```

But there is more. Let's say you only want to exit a program when a certain condition is met, e.g. a word left a non-zero value on the stack. In that case you would have to write something like this:

```
if
    ." We have an error condition!" cr quit
then
```

You can write that much shorter by using the word 'ABORT"':

```
abort" We have an error condition!"
```

'ABORT"' will print the message following it and abort, but *only* when there is a non-zero value on the stack. So this program does *not* abort:

```
false abort" This will not be printed!"
." This will be printed!"
```

You will find that 'ABORT"' is a very handy tool when processing error conditions.

width	description	printf.4th	sprintf.4th	fsprintf.4th
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.	Y	Y	Y
*	The width is not specified in the format string, but as an additional integer value argument following the argument that has to be formatted.	Y	Y	Y
.precision				
.number	For integer specifiers (ld, d, o, u, x): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. For e, g and f specifiers: this is the number of digits to be printed after the decimal point. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.	N	N	Y
.*	The precision is not specified in the format string, but as an additional integer value argument following the argument that has to be formatted.	N	N	Y

Table 9.3: Width and precision of `printf()`

9.12 Opening a file

You probably don't want to write programs that only write to the screen and read from the keyboard. So 4tH has a few words that allow you to work with files. Since 4tH is a scripting language, its capabilities are limited. But you will find that you can perform most common operations.

One of the limitations is that you can have a limited number of open files, but it will do in most situations.

Opening a output-file is pretty simple. Just throw the address and length of a filename and a file access mode on the stack and execute the word 'OPEN'. The value 'OPEN' returns is a simple number which bears little significance. However, you have to save it to a variable or value, for you will need it later. We'd like to use values for storing file pointers, so we created the word 'FILE'. 'FILE' simply creates a value and initializes it, so if you use it prematurely 4tH will issue an error message.

```
file myfile
s" outfile.dat" output open error?   ( a1 n1 fam -- h f)
abort" File could not be opened"     ( h)
to myfile                            ( --)
```

'OUTPUT' is a file access mode and will open a file for writing. 'OPEN' leaves a value on the stack. If it equals '(ERROR)', something was not quite right. If not, the file was

specifier	output	example	printf.4th	sprintf.4th	fsprintf.4th
d	Signed decimal number	392	Y	Y	Y
c	Character	a	Y	Y	Y
s	String of characters	sample	Y	Y	Y
%	Single % character	%	Y	Y	Y
ld	Signed decimal double number	7235	N	N	Y
u	Unsigned decimal number	7235	N	N	Y
o	Signed octal number	610	N	N	Y
x	Signed hexadecimal number	7fa	N	N	Y
f	Decimal floating point, lowercase	392.65	N	N	Y
e	Scientific notation (mantissa/exponent)	3,93E+06	N	N	Y
g	Use the shortest representation: %e or %f	392.65	N	N	Y

Table 9.4: List of supported `printf()` specifiers

successfully opened. 'FILE' is nothing but an initialized value, so you can assign it with 'TO'. 'ERROR?' leaves the handle intact, but leaves an additional true flag if an error occurred, which makes it much easier to evaluate.

The syntax for opening an input file is the same, except for the read-flag 'INPUT' of course:

```
file myotherfile
s" infile.dat" input open error?
abort" File could not be opened"
to myotherfile
```

9.13 Reading and writing from/to a file

There are no special words to read from or write to a file. You can use all the words you used for keyboard-input and screen-output.

But if you open a file and do some I/O you will notice nothing has changed. Of course not. You should be able to determine whether you write to a file or to the screen. There are special words to do just that:

```
file OutFile                  \ file variable
s" outfile.dat" output open error?
abort" File could not be opened"
to OutFile                    \ open the file

OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
```

After you've opened the file, the program will still write to the screen. When 'USE' executes, all output will be redirected to the file. When 'USE' executes again, but this time with the 'OUTPUT' flag, all output will go to the screen again, but the output-file will not be closed! Both words take the same read/write-flags as 'OPEN'.

You can call 'USE' again and again, without closing or opening any files. Here is an example using an input-file:

```
file OutFile
s" outfile.dat" output open error?
abort" File could not be opened"
to OutFile                    \ open output file

OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
OutFile close                 \ close file

s" outfile.dat" input open error?
abort" File could not be opened"
to OutFile

OutFile use                   \ read from disk
pad dup 32 accept             \ read 32 characters
type                          \ write string to screen
stdin use                     \ read from keyboard
OutFile close                 \ close file
```

The output of this program is:

```
This is written to screen
This is written to disk
```

Note that files are always opened in binary mode. If you're a Microsoft user and you worry about your text files, don't. 4tH is much smarter than that as you will learn later on.

9.14 Closing a file

There is usually no need to close any files. When you quit the program all files are closed. It seems like there is no need at all to close files manually, but that is a mistake.

If you want to open a file for reading to which you've just written, you will find it doesn't work. Of course, you can open a file only once.

No, there is a word which closes either the input- or the output-file, using the same read/write-flags. You've already seen it, it is called 'CLOSE'. When you close an active file, the input (or output) is redirected to the keyboard (or screen).

9.15 Writing text-files

Writing text to a file is just as easy as writing text to screen. Open the file, redirect the output, and write like you would write to the screen:

```
file OutFile                  \ value for file
s" outfile.dat"               \ put the filename on the stack
output                        \ add the modifier
open error?                   \ open the file
abort" File could not be opened"
to OutFile

OutFile use                   \ write to file
." This is written to disk" cr
```

That's all! Note that if you execute your program on a Microsoft Operating System, it will write a Microsoft text file. If you do so on a Unix Operating System, it will write a Unix text file. If you want to override that you'll have to issue the end-of-line sequences yourself using 'EMIT'.

9.16 Reading text-files

Reading text-files is pretty straightforward. You don't even have to open a file in text-mode contrary to other languages. Just open the file and call 'REFILL' until it signals end-of-file (EOF):

```
\ Example program. It reads a file line by line
\ and prints it to the screen.

file InFile
s" readln.4th" input open error?
abort" Could not open file"   \ open file
to InFile                     \ save handle
InFile use                    \ read from file

begin
  refill                      \ read a line
while                         \ while EOF not found
  0 parse-word                \ parse the entire line
  type                        \ print it
  cr                          \ terminate line
repeat                        \ read next line
```

You will find that if you run this program, it will print itself to the screen.

'REFILL' will return a non-zero value if EOF was *not* detected. By using the word '0=' you can invert this value. Finally, it will read Unix ASCII-files as well as DOS ASCII-files, no matter where your program is executed.

9.17 Reading long lines

The TIB is only /TIB characters long. If you read a line that is longer than that, only /TIB - 1 characters are read. The rest of the line is read when you invoke 'REFILL' again. Although you don't lose any information that way, it might not be what you want. Fortunately, you can define your own TIB:

```
2048 constant /mytib           \ length of your TIB
/mytib string mytib            \ define your own TIB

mytib /mytib source!           \ tell the system about your TIB
```

The next time you invoke 'REFILL', it will use your TIB instead of the system TIB, so it will now read lines up to 2047 characters. 'SOURCE!' takes an address/count pair and makes it the current TIB. So if you want to use the system TIB again you issue:

```
tib /tib source!
```

And if you have forgotten which TIB you're using try this:

```
source . . cr
```

'SOURCE' will return the address/count pair of the TIB you're currently using. In fact, this definition does absolutely *nothing*:

```
: doesnothing source source! ;
```

For the simple reason that it reassigns the TIB it is already using.

9.18 Reading binary files

If you process binary files, you won't get far reading it line by line. You want to read chunks of data. 4tH can do that too by using 'ACCEPT'. You feel there must be a catch, since 'ACCEPT' terminates strings automatically. Well, there isn't. When 'ACCEPT' does not read from the keyboard, it won't add that extra byte.

Reading blocks of data usually means defining buffers. If maintainability is an issue, define a constant for the sizes of these buffers. You cannot only use this constants when defining buffers, but also when calling 'ACCEPT'.

Furthermore, 'ACCEPT' returns the number of characters actually read. If this value is compared to the number of characters we actually wanted to read, we can determine whether a reading error or EOF occurred:

```
1024 constant bufsize         \ actual buffersize
bufsize string buffer         \ define buffer
file InFile                   \ value for file
                              \ open input file
s" infile.dat" input open error?
abort" File could not be opened"

to InFile                     \ save handle
InFile use                    \ redirect input

begin                         \ using bufsize
  bufsize                     ( n1)
  buffer over                 ( n1 a n1)
  accept                      ( n1 n2)
  <>                          ( f)  \ make EOF flag
until                         \ until EOF
```

Note that "BUFFER" is actually not a string, but a chunk of memory. But since a character in 4tH takes up a single address-unit (=byte), raw chunks of memory are allocated in the Character Segment. This is not an uncommon practice in both Forth and C.

9.19 Writing binary files

Writing binary files is very easy. Of course you need a buffer, like we discussed in the previous section. The program is not much different than the previous one:

```
1024 constant bufsize         \ actual buffersize
bufsize string buffer         \ define buffer
file OutFile                  \ value for file

buffer bufsize char H fill    \ fill the buffer
                              \ open output file
s" infile.dat" output open error?
abort" File could not be opened"

to Outfile                    \ save handle
OutFile use                   \ redirect input
buffer bufsize type           \ write to file
```

This will write 1024 "H"s to "infile.dat". The actual command that does all writing is 'TYPE'. The word 'TYPE' does not return anything. You can be assured that everything was alright, since if it wasn't, 4tH would have caught the error itself.

9.20 Reading and writing block files

Block files are a special kind of files used by Forth compilers. In the old days Forth controlled the *entire* computer and directly communicated with all peripherals, including disks. To Forth, a disk is just a bunch of numbered blocks. Each block is divided into 16 lines of *exactly* 64 characters. A block file simply mimics that layout.

Before we can begin, you need to create a block file. Well, that's easy:

```
include lib/ansblock.4th
4 s" blocks.scr" create-blockfile
```

This creates a file with four blocks. Then we have to tell 4tH which file to use:

```
s" blocks.scr" open-blockfile
```

Note that apart from creating the file, we haven't performed any I/O yet. First, we have to request a block. When a block is requested, its contents are transferred to a memory buffer. You can manipulate this buffer any way you want with the standard words. If you request another block its contents are transferred to the buffer too, overwriting whatever is there. All changes you have made are lost, *unless* you have flagged the block as dirty, which means its contents are different from the block on disk. If a block is dirty, it is written to disk *before* the next block is read. 'CLEAR' is a special word, assigning an empty buffer to a block without reading it first. The buffer is BLANKed. 'UPDATE' will flag the buffer as dirty. 'FLUSH' writes the dirty buffer to disk and unassigns the buffer. So, first we clear block 0:

```
0 clear
```

Then we clear block 1, copy a string to it and flag it as dirty:

```
1 clear
s" Hello world!" >r 1 buffer r@ cmove update
```

'BUFFER' returns the address of the buffer assigned to that block[2]. If the buffer is dirty, it is FLUSHed before assigning a buffer to that block. We can also write the dirty buffer to disk, without unassigning it:

```
save-buffers
```

Note that the buffer is not dirty anymore, since it has been synchronized. Let's write something to another block:

```
s" Goodbye cruel world!" 0 block swap cmove update
```

It is always a nice game to figure out what will happen now. The current block is block 1. Since we haven't UPDATEd it since 'SAVE-BUFFERS', it is clean. That means that 4tH won't perform a write. Since block 0 isn't current, 'BLOCK' reads it into the buffer. The 'UPDATE' will flag the buffer as dirty.

[2]That is particularly handy if your implementation can handle multiple buffers. In this implementation we have only one buffer, so we always return the same address.

```
1 block r> type cr
```

This is fun! The current block is block 0. It is dirty, so it is written to disk. Since block 1 isn't current, it is read into the buffer. You catch my drift? If you want to print the contents of a block, you can use 'LIST'. Of course, 'LIST' uses 'BLOCK' and applies to the same rules:

```
0 list
." This block has been listed: " scr ? cr
```

'SCR' is a variable containing the last screen LISTed. Note that is not the same thing as the current block! Finally, we can discard all our changes:

```
empty-buffers
```

'EMPTY-BUFFERS' does not perform any I/O nor does it change the contents of the buffer. It just unassigns the buffer and flags it as clean. Note that you don't *have* to close a block file since all I/O is block-oriented. Just don't forget to "FLUSH". You can use different block files within the same program, but you'll have to close the current block file - since it performs "FLUSH" as well and it's portable:

```
close-blockfile
```

It is also important to know that ANS-Forth doesn't define any error conditions, so if anything happens all the poor thing can do is throw an exception[3].

Note that the standard block file library implements the absolute bare minimum required by ANS-Forth. The most important restriction is that is has just one single buffer. There is also a libary that is able to handle *thousands* of buffers very efficiently. Standard, it will allocate 16 buffers. If you want a different number of buffers, you'll have to define a constant before including it:

```
32 constant #blocks           \ allocate 32 block buffers
include lib/multiblk.4th
400 s" blocks.scr" create-blockfile
```

It is a drop-in replacement, so you won't have to change your code too much.

9.21 Parsing textfiles

As we've already seen, it is very easy to enter a line using 'REFILL' and parse it. You can also use 'REFILL' to read lines from a text-file. It is quite similar to reading lines from the keyboard, except that you have to open a file first. This little program prints all the words of a textfile on a new line:

```
file InFile                   \ value for file
s" file.txt" input open error?
abort" File could not be opened"
                              \ open the file
to InFile                     \ save handle
InFile use                    \ redirect input to file
```

[3]See section 11.5.

```
begin
  refill                      \ get a line from file
while                         \ check if EOF
  begin
    bl parse-word             \ if not, parse line
    dup 0<>                   \ check if zero length
  while
    type cr                   \ if not, print word
  repeat                      \ parse next word
  drop drop                   \ drop address/count
repeat                        \ get next line
```

Now that flag left by 'REFILL' makes sense! If it is zero, we have reached the end of the line. Note that you don't have to open a file in text-mode and both Microsoft ASCII and Unix ASCII files are supported.

9.22 Parsing binary files

And what about binary files, like classic Forth blockfiles? Well, you could use 'REFILL' in that context too, but it would probably break up words since it can't find an end-of-line marker and its buffer is smaller than 1024 characters. Does that mean it can't be done? No! But 'REFILL' makes it easier for you, because it handles a few tasks automatically.

First, it has its own buffer (TIB). When you're not using 'REFILL' you have to define one yourself. Second, it terminates the string for you. You don't want 'PARSE-WORD' to wander into new territory, do you? Third, it sets '>IN' for you every time its receives new input. You have to take care of that one too.

Never heard of '>IN'? Well, the only way for 'PARSE-WORD' to know on what position the previous scan ended is to store that information into a variable. This variable is called '>IN'.

Not all internal 4tH variables are accessible, mostly because we can't imagine what use they could have to you. Some variables are just better left alone. But '>IN' is available for some very obvious reason: you can reset it and make 'PARSE-WORD' work for you. Note that for '>IN' to work, you have to make the buffer the parsing area by using 'SOURCE!'

The following program will read the first screen of a block-file for you and print out all the words. You will see that all spaces are eliminated and every word is printed on a new line, just the behavior you would expect from 'PARSE-WORD'.

```
1025 constant /buffer         \ screensize + terminator
/buffer 1- constant c/scr     \ size of the block
file InFile                   \ value for file

/buffer string buffer         \ 1: our own buffer

: openfile                    \ open the block file
  s" romans.blk" input open error?
  abort" Cannot open file"
  to InFile                   \ save handle
  InFile use                  \ read from file
;

: readfile                    \ fill the buffer
  buffer c/scr 2dup           \ address and count
  bl fill                     \ clear the buffer
  accept drop                 \ fill the buffer
  InFile close                \ close the file
```

```
;

: initparse                    \ configures parsing
  0 buffer c/scr chars + c!    \ 2: terminate screen
  buffer /buffer source!       \ 3: make buffer the parse area
  0 >in !                      \ 4: reset >IN
;

: parseblock
  begin
    bl parse-word              \ get word
    dup 0<>                    \ length zero?
  while
    type cr                    \ if so, print it
  repeat
  2drop                        \ else drop addr/cnt
  ." End of block" cr          \ signal "End of block"
;

: parsefile                    \ do it all
  openfile                     \ open the file
  readfile                     \ read it
  initparse                    \ set up parsing
  parseblock                   \ parse it
;

parsefile
```

Note there is no need to reset '>IN' if you use 'REFILL', since it will be reset *automatically*. In this case, if you want to parse another block, you will have to reset '>IN' again.

9.23 Parsing comma-delimited files

'PARSE-WORD' is a powerful and very useful word, but it is less than useful when parsing comma-delimited files. Why? Well, because 'PARSE-WORD' skips leading delimiters. So when you have a file like this it doesn't work:

```
FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX
Hans,Bezemer,The.Beez.speaks@gmail.com,,http://hansoft.come.to,
```

Again, 'PARSE-WORD' skips leading delimiters, so instead of an empty string we get the homepage when we're trying to read the (non-existant) telephone number. Fortunately, we got a word like 'PARSE'. 'PARSE' also takes a delimiter from the stack, just like 'PARSE-WORD', but it acts on leading delimiters. Take a look at this program:

```
file OutFile                   \ value for output file
file InFile                    \ value for input file

: WriteCommaFile               ( --)
  s" address.csv" output open error?
  abort" Could not write CSV file"
  to OutFile                   \ save handle
  OutFile use                  \ redirect output to file
  ." FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX" cr
  ." Hans,Bezemer,,,http://hansoft.come.to," cr
  OutFile close                \ close file
  stdout use                   \ redirect output to screen
;
                               \ reads a field
: field> [char] , parse ;      ( -- a n)
```

```
: ProcessLine                 ( --)
  refill                      \ get line
  0= abort" Read error"
  field> type cr              \ parse first name
  field> type cr              \ parse name
  field> type cr              \ parse email
  field> type cr              \ parse telephone
  field> type cr              \ parse homepage
  field> type cr cr           \ parse fax
;

: ReadCommaFile               ( --)
  s" address.csv" input open error?
  abort" Could not read CSV file"
  to InFile                   \ save handle
  InFile use                  \ redirect input to file
  ." _Headerline_" cr         \ this is the headerline
  ProcessLine                 \ now process headerline
  ." _First record_" cr       \ this is the first record
  ProcessLine                 \ now process first record
  InFile close                \ close file
;

WriteCommaFile                \ write the CSV file
ReadCommaFile                 \ read the CSV file
```

With "WriteCommaFile" we write a simple comma-delimited file to disk. We got to read something, don't we? Then we read the file we've just written with "ReadCommaFile". "ProcessLine" does the actual job. Since we have six fields we use 'PARSE' six times - that is: each time we call "FIELD>". We cannot do this with a loop. Why not?

Well, 'PARSE' not only returns a NULL-string when we've reached the end of a line, but also when a field is empty. So we've got to know how many fields we actually want to read. Of course, you could parse the headerline with 'PARSE-WORD' to find that out, but you already know how to do this.

9.24 Parsing fixed-width text files

A fixed-width text file is a file consisting of records with a fixed-width fields. They are not as common anymore as they used to be in the mainframe era, but you still find them every now and them. They look like this:

```
New Jersey          NJTrenton       8958013 22592  12
Texas               TXAustin        27469114695660 36
Washington          WAOlympia       7170351 184661 10
```

And yes, you can process them with 4tH. First thing you need to do is to include a small library:

```
include lib/fixedfld.4th
```

Next, you need to create a small array containing all the field lengths, first one first, and terminate it with a zero:

```
create states-fields 20 , 2 , 14 , 8 , 7 , 2 , 0 ,
```

Note there are fixed-width files with line termination. You can read those with 'REFILL'. If they are not line terminated, don't dispair[4]. Let's say a record is 64 bytes, then you can read a record at the time like this:

```
64 constant /record           \ length of record
/record buffer: record        \ define a buffer
                              \ use like REFILL
: read-record record /record accept /record = ;
```

But in this example we will assume our file is line terminated. The next thing to do is to integrate the library into your reading routine. Since we use a line-terminated file, 'REFILL' will do:

```
/layout array states-layout   \ this is the layout variable
                              \ enhance REFILL
: !refill states-fields tib states-layout fields! refill ;
```

The "FIELDS!" word takes three parameters. First, the array with the field lengths. Second, your read buffer. If we would have used 'ACCEPT' from our previous example, we would have used the "RECORD" buffer. Here we use 'REFILL', so we use the standard buffer, 'TIB'.

Finally the so-called "layout variable". This is variable you have to define for each fixed-width file you want to process with this library, so if you parse two different files at the same time, you have to define two "layout variables".

Now the only thing you have to do is to use "!REFILL" instead of 'REFILL' and we're all set. "FIELDS!" will make sure the library is properly set up each time we read a fresh line.

When we construct our "FIELD>" routine[5] we do something similar. "NEXT-FIELD" will ensure you get the next field, provided you feed it the proper "layout variable", of course. Note that "NEXT-FIELD" returns the *entire* field, which is probably padded. Let's assume it's padded with spaces, which we can easily remove with '-TRAILING':

```
: field> states-layout next-field -trailing ;
```

Since the stack diagram of this "FIELD>" is identical to the one in section section 9.23, the entire thing becomes almost completely transparent:

```
: ProcessLine                 ( --)
  !refill                     \ get record
  0= abort" Read error"
  field> type cr
  field> type cr
  field> type cr
  field> type cr
  field> type cr
  field> type cr cr
;
```

Yes, it takes some setting up, but in the end reading a fixed-width text file is not much harder than reading an ordinary .CSV file.

[4]See also section 9.22.
[5]See section 9.23.

9.25 Advanced parsing

Let's think of something difficult to parse, e.g.:

```
;;;;;;;;;.........Can you parse me.
;;;;,,,,And me too, huh?
```

If we would use 'PARSE' we would have to know how many semicolons to skip and there is a different number of them on each line. If we would use 'PARSE-WORD' we'd lose all the semicolons, but the parsed string would have all these nasty leading dots.

Even worse, if we were able to skip the semicolons and use 'PARSE-WORD' with the leading delimiter we'd get "Can you parse me" and "And me too" instead of "Can you parse me." and "And me too, huh?". What can we do?

Fortunately, 4tH doesn't really know about 'PARSE-WORD' but translates it into a sequence of words[6]. We can also use them directly. 'OMIT' is very handy. It doesn't actually do anything, it just skips leading delimiters and sets '>IN' accordingly. It takes an ASCII value from the stack as its delimiter. This will correctly parse the first line:

```
char ; omit                    \ omit the semicolon
char . omit                    \ omit the dot
0 parse                        \ parse the remainder of the line
```

This will correctly parse the second line:

```
char ; omit                    \ omit the semicolon
char , omit                    \ omit the comma
0 parse                        \ parse the remainder of the string
```

Please note that this are special 4tH words! Unfortunately you cannot port this to ANS-Forth, where only a limited version of 'PARSE-WORD' and 'PARSE' are available.

9.26 Appending to existing files

You can use a so-called modifier to signal 4tH, it shouldn't overwrite the file it opens, but append to it:

```
file OutFile                   \ value for output file

s" outfile.dat" output open error?
abort" Cannot open file"       \ open the file
to OutFile                     \ save handle
OutFile use                    \ now write to disk
10 0 do i . loop               \ write 0 to 9
OutFile close                  \ close the file

s" outfile.dat" output append + open error?
abort" Cannot open file"       \ reopen in append mode
to OutFile                     \ save handle
OutFile use                    \ now write to disk
20 10 do i . loop              \ write 10 to 19
OutFile close                  \ close the file
```

Take a look at the contents of this file after you've run the program and you'll find it contains the number 0 to 19.

[6] If you're really curious, 'PARSE-WORD' is equivalent to 'DUP OMIT PARSE'.

9.27 Using pipes

If you're using Windows 95 OSR2 (and up), Windows NT (and up), OS/X, Linux or another Unix system you're in for a treat! With Unix you can do neat tricks like this:

```
ls | mail root
```

Which means you can redirect the output of 'ls' to 'mail', so in effect you send an email to root with the contents of your current working directory. Yes, 4tH can do this too, but you can do even more. You can start 'ls' and read its output line by line as if it were a file. You can also start 'mail' and write the output of a 4tH program to it. We do that by opening a *pipe* to a program.

If you've ever written a program using C, you know this is a bit cumbersome, since you've got to use special functions to use pipes. In 4tH you don't. Just let 4tH know it's a pipe that you're opening and not a file:

```
file InFile                    \ value for input file
file OutFile                   \ value for output file

s" ls" input pipe + open error?
s" mail hans" output pipe + open error? rot or
abort" Cannot open pipe"
to OutFile to InFile
```

The only thing you have to do to signal 4tH that you're using a pipe is add the word 'PIPE', just like 'APPEND'. The filename is replaced by the command you want to execute. That's all. If one of the pipes in this program fails, the program aborts.

```
InFile use OutFile use
." These are the contents of my current working directory:"
cr cr
```

Now we can treat our pipes just as if they are ordinary files. We redirect input and output and write a nice header to our email. Now we can start to process the output of 'ls':

```
begin
  refill
while
  0 parse-word type cr
repeat
```

Note that we don't have to signal that we're reading the pipe to 'ls' as a text file. We just read it line by line until 'REFILL' returns zero. Then we can parse the line and 'TYPE' it to 'mail'.

```
InFile close OutFile close
```

Of course you don't *have to* close the pipe, but it won't harm when you don't. 4tH knows what to do. After executing this program, Hans will receive this email:

```
From: Hans Bezemer <The.Beez.speaks@gmail.com>
Message-Id: <200202252017.VAA00712@gmail.com>
To: hans@localhost.org
Status: RO
```

```
These are the contents of my current working directory:

4th.c
4thc.c
4thd.c
4thg.c
4thx.c
```

Well, that wasn't too hard, was it?

9.28 Opening a file in read/write mode

For special purposes you might want to open a file in read/write mode. That's quite easy:

```
file InOutFile                \ value for output file

s" outfile.dat" output input + open error?
abort" Cannot open file"      \ open the file
to InOutFile                  \ save handle
```

Just add both together like you adding a modifier. Note that once you 'USE' this file, you're both reading *and* writing to this file. Furthermore, the file has to exist otherwise you get an error. If you want to write to a new file, you first have to open it in write mode:

```
file InOutFile                \ value for output file
                              \ create a new file
s" outfile.dat" output open close
s" outfile.dat" output input + open error?
abort" Cannot open file"      \ open the file
to InOutFile                  \ save handle
```

9.29 Using random access files

Upto now we've always accessed a file sequentially, but it is also possible to use random access files. Two words are crucial here, 'SEEK' and 'TELL'. 'SEEK' will seek for the desired file position and 'TELL' will tell you that you're there. It is as simple as that..!

Let's take a look at this example. We've got a block-file called "Messages.scr" with the following contents:

```
Scr # 0
  0 (0) No errors
  1 (1) Out of memory
  2 (2) Bad object
  3 (3) Stack overflow
  4 (4) Stack empty
  5 (5) Return stack overflow
  6 (6) Return stack empty
  7 (7) Bad string
  8 (8) Bad variable
  9 (9) Bad address
 10 (A) Divide by zero
 11 (B) Bad token
 12 (C) Bad radix
 13 (D) Undefined name
 14 (E) I/O error
 15 (F) Assertion failed
```

First, let's define a word that reads a message and then displays it:

```
: next-msg pad dup 64 accept -trailing type cr ;
```

Since this is a simple program we can safely use the 'PAD' to store our messages. Every message has the length of a standard block-file line, which is 64 characters. Trailing spaces are stripped by '-TRAILING'. Now we need a word that tells us what our file position is:

```
: tell-msg cr ." Current position: " dup tell . cr ;
```

'TELL' needs a file pointer and leaves the current position of that file pointer on the stack. This word assumes that the top of the stack contains a valid file pointer. Finally we need a word that sets the file position:

```
: seek-msg over seek abort" Seek failed" ;
```

'SEEK' needs a file position and a file pointer. If it returns false, it was successful; if it returns true there was an error. This word assumes that the top of the stack contains a valid file pointer. We're ready now, let's play. First we open the file and use it:

```
s" Messages.scr" input open dup use
```

This leaves a file pointer on the top of the stack, assuming everything went OK. Now let's read some messages:

```
next-msg next-msg next-msg tell-msg
```

You'll see these messages appear on the screen:

```
(0) No errors
(1) Out of memory
(2) Bad object

Current position: 192
```

After reading three messages we've obviously reached position 192 in the file. That makes sense, since 3 lines of 64 characters makes 192 characters in total. Let's see what 'SEEK' does:

```
0 seek-msg tell-msg next-msg
```

This should take us back to the very beginning of the file, as if we've freshly opened it. And yes, it does:

```
Current position: 0
(0) No errors
```

After executing 'SEEK', 'TELL' confirms that we've actually returned to the very beginning of the file. Reading the next message reconfirms that again. When you feed 'SEEK' positive values, it always starts seeking from the *beginning* of the file. When you feed 'SEEK' negative values, it seeks from the *end* of the file. So this one takes you to the last line:

```
-64 seek-msg tell-msg next-msg
```

On screen it looks like this:

```
Current position: 960
(F) Assertion failed
```

Finally, we clean up the mess we made:

```
close
```

This will consume the file pointer we left on the stack and close the file. Note that 'SEEK' and 'TELL' come with a few restrictions. Pipes are out of the question and so are the standard streams 'STDIN' and 'STDOUT'. Apart from that you can pretty much do with them what you want.

Another tip: if you use '(ERROR)' as an offset for 'SEEK', it will take you to the very end of the file, since 'MAX-N' won't do the trick.

9.30 The layout of the I/O system

You're probably quite confident manipulating files now, so I guess it is time to offer you a view under the hood. 4tH has two channels, an input channel and an output channel. All words read from the input channel or write to the output channel. At startup, the input *channel* is connected a *stream* that reads from the keyboard ('STDIN') and the output *channel* is connected to a *stream* that writes to the screen ('STDOUT').

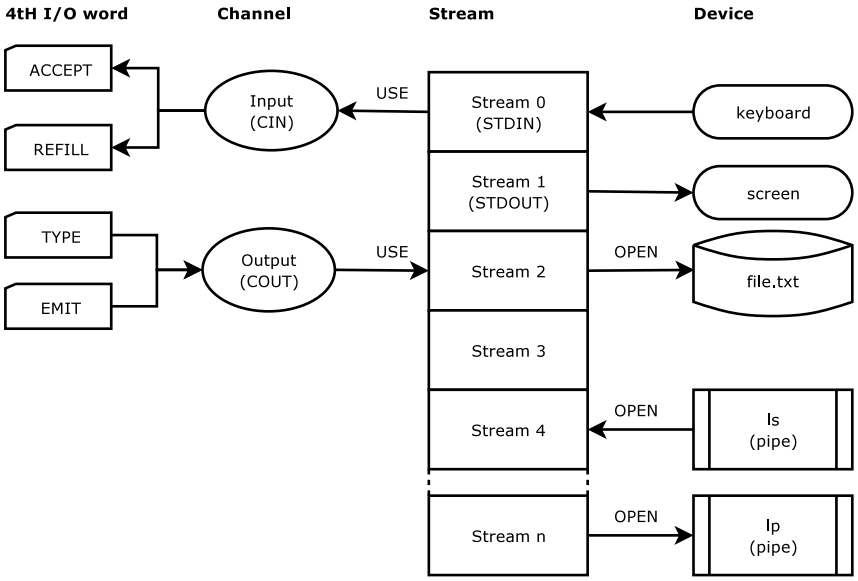

Figure 9.2: The 4tH I/O system

With 'OPEN' you can open *additional* streams, which are connected to a file or a pipe. The return value of 'OPEN' points to the stream that was opened. There are few words that directly handle streams, 'USE', 'CLOSE', 'TELL' and 'SEEK' being the exceptions. 'USE' attaches a stream to one or both channels, which results in redirecting all in- and/or

output to that stream. E.g. if a file is opened in read/write mode using 'OPEN', a stream is returned. If we 'USE' that stream, both the input and the output channel are connected to that stream. If it had only been opened in read mode, only the input channel would have been connected to the stream.

'CLOSE' closes a stream, even if it is still attached to a channel. If that is the case, the appropriate default streams ('STDIN', 'STDOUT') are reattached. We can find out which streams are currently used by 'CIN' and 'COUT'. 'CIN' returns the stream that is currently attached to the input channel, 'COUT' returns the stream that is currently attached to the output channel.

9.31 Using a printer

How you access a printer depends on the operating system you're working on. That is not a flaw of 4tH, you will encounter this problem with every programming language. If you're working with MS-DOS or MS-Windows it is quite basic:

```
file printer                  \ value for printer

s" lpt1" output open error?
if
  drop
else
  to printer
  printer use
  ." This will be printed." cr
  stdout use
then
```

Just open the port as a file and print to it. Unix isn't that different, but instead of opening a file, you open a pipe:

```
file printer                  \ value for printer

s" lp" output pipe + open error?
if
  drop
else
  to printer
  printer use
  ." This will be printed." cr
  stdout use
then
```

If you're using a different Operating System, you may have to check your manual.

9.32 The layout of the Character Segment

The final topic of this chapter again. You already know that 4tH checks whether an operation is still within the Character Segment. However, sometimes you want to check this yourself.

You already know how you can obtain the size of TIB and PAD. Yes, you can using '/TIB' and '/PAD'. But TIB and PAD have their addresses too. And when you query them, you will find that PAD comes after TIB:

```
." Address of TIB: " tib . cr
." Size of TIB : " /tib  . cr
." Address of PAD: " pad . cr
." Size of PAD : " /pad  . cr
```

And beyond PAD, what is there? Well, allocated memory of course. Things you defined using 'STRING'. There are two words which can give you information about allocated memory. First, 'LO'. 'LO' gives you the lowest address of allocated memory. Second, 'HI'. 'HI' gives you the highest valid address of the Character Segment. That means that:

```
0 hi c!
```

Is always valid and:

```
0 hi char+ c!
```

Is always invalid. It you try it, 4tH will stop executing the program with an error-message. 'LO' and 'HI' are addresses. Addresses are just numbers, so you can print and compare them. E.g.

```
hi char+ lo - . cr
```

Will print how much memory as allocated to your strings. And:

```
lo hi >
```

Will indicate whether you allocated any memory at all. If 'LO' is greater than 'HI', you didn't. If 'HI' is greater or equal to 'LO' you did. Experiment a bit with the knowledge you obtained in this chapter and continue with the next one where we will go much deeper into the secrets of the Integer Segment and Code Segment.

Chapter 10

Integer Segment and Code Segment

10.1 The Code Segment

It is known by designers of microprocessors that a processor can run much faster when every instruction has the same length. In fact, 4tH has his own virtual microprocessor. The compiler is nothing more than an assembler and the interpreter nothing more than an emulator on top of the real microprocessor.

In order to speed up 4tH, all instructions have the same length. They consist of a token (which is the real instruction) and an argument. The argument is a value that gives meaning to the instruction, e.g. the 'LITERAL' token means that a number is compiled here. The argument is the actual number.

Some instructions wouldn't need an argument, but for speeds sake, they have: it is always zero. Isn't that a lot overhead? Not really. Half the instructions in an actual program need an argument. Decoding a more elaborate scheme would need more processor time and more programming. So in the end, it would make hardly any difference. Except for the speed.

A token with its argument is called a word. And the Code Segment is one large array of words. Each of these words has an address and can be accessed by the word '@C'. In fact, '@C' throws the argument on the stack. Where have we seen '@C' before?

Yes, when fetching from an array of constants. These arrays are compiled into the Code Segment. How come that 4tH isn't confused by these arrays? Because they have the token 'NOOP', which does absolutely nothing.

10.2 The address of a colon-definition

You can get the address of a colon definition by using the word "' (tick):

```
: add + ;                    \ a colon definition
' add . cr                   \ display address
```

Very nice, but what good is it for? Well, first of all the construction "' ADD" throws the address of "ADD" on the stack. In fact, it is a literal expression. You can assign it to a variable, define a constant for it, or compile it into an array of constants:

```
' add constant add-address

variable addr
' add addr !

create addresses ' add ,
```

Are you with us so far? If we would simply write "ADD", "ADD" would be executed right away and no value would be left on the stack. Tick forces 4tH to throw the address of "ADD" on the stack instead of executing "ADD".

Note this only works for your own colon-definitions. It doesn't work for 4tHs built-in words. If you try to, you'll get an error-message. What you can actually do with it, we will show you in the next section.

10.3 Vectored execution

This is a thing that can be terribly difficult in other languages, but is extremely easy in Forth. Maybe you've ever seen a BASIC program like this:

```
10 LET A=40
20 GOSUB A
30 END
40 PRINT "Hello"
50 RETURN
60 PRINT "Goodbye"
70 RETURN
```

If you execute this program, it will print "Hello". If you change variable "A" to "60", it will print "Goodbye". In fact, the mere expression "GOSUB A" can do two different things. In 4tH you can do this much more comfortable:

```
: goodbye ." Goodbye" cr ;
: hello ." Hello" cr ;

variable a

: greet a @ execute ;

' hello a !
greet

' goodbye a !
greet
```

What are we doing here? First, we define a few colon-definitions, called "HELLO" and "GOODBYE". Second, we define a variable called "A". Third, we define another colon-definition which fetches the value of "A" and executes it by calling 'EXECUTE'. Then, we get the address of "HELLO" (by using "' HELLO") and assign it to "A" (by using "A !"). Finally, we execute "GREET" and it says "Hello".

It seems as if "GREET" is simply an alias for "HELLO", but if it were it would print "Hello" throughout the program. However, the second time we execute "GREET", it prints "Goodbye". That is because we assigned the address of "GOODBYE" to "A".

The trick behind this all is 'EXECUTE'. 'EXECUTE' takes the address of e.g. "HELLO" from the stack and calls it. In fact, the expression:

```
hello
```

Is equivalent to:

```
' hello execute
```

This can be extremely useful as we will see in the next chapter when we build a full-fledged interpreter. We'll give you a little hint:

```
create subs ' hello , ' goodbye ,
```

Does this give you any ideas?

10.4 Execution tokens

The address of a colon definition is called an "execution token". It's much like a token in a subway: you insert a token into the appropriate slot and the turnstile opens. Here you pass an execution token to 'EXECUTE' and the definition is executed. Of course, you can define an execution token this way:

```
: hello ." Hello world!" ;
' hello constant hello-xt
```

But you can also define it more directly:

```
:token hello-xt ." Hello world!" ;
```

It will behave in exactly the same way our previous definition of "HELLO-XT" did. There is still another way to define execution tokens. We'll come around that in section 11.7.

10.5 The Integer Segment

Wonder where all these variables are created? Or where that infamous stack really is? I bet you do. Well, when you define a variable, memory is allocated in the Integer Segment. When you define another one, space is allocated after the first variable. That means that if you go beyond the boundaries of the first variable, you'll end up in the space allocated to the second variable.

After the second variable there is a void. If you end up there your program will end with an error-message. However, if you define an 'ARRAY', single variable is created with a number of additional cells. You can only access these additional by referring to the array itself.

And what about the space before the first variable? There are other variables and they are not defined by you. Well, take a look at figure 10.1.

Lower memory is at the bottom. The user variables are the variables you defined yourself. The application variables can differ from host program to host program. Refer to your documentation on that subject. You have already seen the 4tH variables, which are 'BASE' and '>IN'. There are also variables you cannot access. These variables are hidden and only used by the system. All these variables are located in the Variable Area.

There is also a Stack Area, which contain the datastack and the returnstack. If you enter a number like "5", it is thrown on the datastack. Most words in 4tH take or put numbers on the datastack. It is very heavily used. We'll come to the returnstack later on.

The datastack and the returnstack share the same memory space. The datastack grows upward and the returnstack downward. If they clash the stack is full and 4tH will issue an error-message.

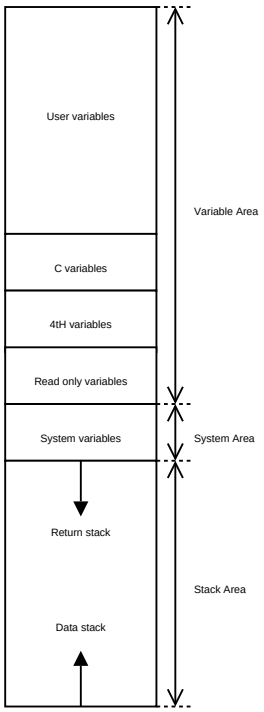

Figure 10.1: Integer segment

10.6 A portable way to access application variables

A host program can add special variables to the 4tH environment. If 4tH is used as a scripting language in e.g. a printer program, the programmer can "send" variables to 4tH. These variables are called "application variables". Do not confuse them with 4tH variables, like 'BASE' or '>IN' which are used internally by 4tH. 4tH doesn't do anything with application variables.

If the creator of the host program provided special names for each of these variables, he will probably have documented them. However, even if he didn't there is another way to access these variables.

They are stored in a predefined array called 'APP' and its values can be fetched like any other array, e.g.:

```
app 1 th @
```

Which fetches the value of the second element in the array. This also enables you to write programs that can be compiled and run under all "standard" versions of 4tH.

10.7 Returning a result to the host program

The 'APP' array can feed values from the host program to yours, but it can't return any. For that you need 'OUT', the third 4tH variable. Returning a value is very easy. Just store it in 'OUT'. Let's assume the host program has send two values to the 'APP' array and you want to return the sum. All you have to do is add them and store the result in 'OUT':

```
app 0 th @ app 1 th @ + out !
```

Nothing to it..

10.8 Using commandline arguments

A host program can also transfer an array of strings to the 4tH environment. Usually, commandline arguments will be transferred this way, although any string array with the correct format can be used. If so, you will probably find it in the documentation of the host program.

If you are familiar with C, the concept is probably quite easy to understand. There are two words, 'ARGN' and 'ARGS'. 'ARGN' will leave the number of commandline arguments on the stack. The commandline arguments itself are numbered from 0 to (ARGN - 1), e.g.

```
argn 0>                       \ test if there are
if                            \ any arguments
  argn 0 do                   \ loop through them
    i args type cr            \ print them
  loop
then
```

First, we test if there are any commandline arguments. Second, if that is the case we loop through them with 'ARGN' as upper limit. Why? Since "ARGN 1- ARGS" is always the last valid commandline argument!

Third, when 'ARGS' executes, it takes a number from the stack as index. Then it leaves the address of the Character Segment (where it is temporarily stored, usually PAD) and its count on the stack.

Using the expression "TYPE CR" we can print that string. Because it is already stored in the Character Segment we can treat it like any other string. Remember, that if you don't save it anywhere else it won't last long!

10.9 The layout of the Variable Area

There are special words that allow you to get information about the layout of the Variable Area. They are called 'VARS', 'APP', 'FIRST' and 'LAST'.

'VARS' is the address of the very first variable. Before that is the Stack Area and other variables you are not allowed to touch. 'APP' is the address of the first application variable. All variables before that are 4tHs own built-in variables. 'FIRST' is the address of the first user-variable, a variable you defined yourself in your 4tH program. 'LAST' is the address of the last accessible variable, so

```
last ?
```

will *never* fail. The first question that will pop in your mind is, what can I do with them. Well, you can use it to see how many variables there are of a certain kind, so you can prevent runtime errors:

```
." number of 4tH variables: "         app vars - . cr
." number of application variables: " first app - . cr
." number of user variables: "        last first - 1+ . cr
```

These tests are possible too:

```
app vars -   0= if ." No 4tH variables" cr then
first app -  0= if ." No application variables" cr then
last first - 1+ 0= if ." No user variables" cr then
```

This is a general test to see whether the address of any variable is within range:

```
dup 0<
dup last >
or
if ." Out of range" cr then
```

You can use this check on numeric arrays too, of course.

10.10 The stacks

The Stack Area contains two stacks. So far we've talked about one stack, which is the Data Stack. The Data Stack is heavily used, e.g. when you execute this code:

```
2 3 + .
```

Only the Data Stack is used. First, "2" is thrown on it. Second, "3" is thrown on it. Third, '+' takes both values from the stack and returns the sum. Fourth, this value is taken from the stack by '.' and displayed. So where do we need the other stack for?

Well, we need it when we want to call a colon-definition. Before execution continues at the colon-definition, it saves the address of the currently executed token in the Code Segment on the other stack, which is called the Return Stack for obvious reasons.

Then execution continues at the colon-definition. Every colon-definition is terminated by ';', which compiles into 'EXIT'. When 'EXIT' is encountered, the address on top of the Return Stack is popped. Execution then continues at that address, which in fact is the place where we came from.

If we would store that address on the Data Stack, things would go wrong, because we can never be sure how many values were on that stack when we called the colon-definition, nor would be know how many there are on that stack when we encounter 'EXIT'. A separate stack takes care of that.

Try and figure out how this algorithm works when we call a colon-definition from a colon-definition and you will see that it works (4tH is proof of that).

It now becomes clear how 'EXECUTE' works. When 'EXECUTE' is called, the address of the colon-definition is on the Data Stack. All 'EXECUTE' does is copy its address on the Return Stack, take the address from the Data Stack and call it. 'EXIT' never knows the difference..

But the Return Stack is used by other words too. Like 'DO' and 'LOOP'. 'DO' takes the limit and the counter from the Data Stack and puts them on the Return Stack. 'LOOP' takes both of them from the Return Stack and compares them. If they don't match, it continues execution after 'DO'. That is one of the reasons that you cannot split a 'DO..'LOOP'.

However, if you call a colon-definition from within a 'DO'..'LOOP' you will see it works: the return address is put on top of the limit and the counter. As long as you keep the Return Stack balanced (which isn't too hard) you can get away with quite a few things as we will see in the following section.

10.11 Saving temporary values

We haven't shown you how the Return Stack works just for the fun of it. Although it is an area that is almost exclusively used by the system you can use it too.

We know we can manipulate the Data Stack only three items deep (using 'ROT'). Most of the time that is more than enough, but sometimes it isn't.

In 4tH there are special words to manipulate stack items in pairs, e.g. '2DUP' (n1 n2 – n1 n2 n1 n2) or '2DROP' (n1 n2 –). Although they are already part of 4tH, we could easily define those two ourselves:

```
: 2dup over over ;
: 2drop drop drop ;
```

You will notice that '2SWAP' (n1 n2 n3 n4 – n3 n4 n1 n2) becomes a lot harder. How can we get this deep? You can use the Return Stack for that..

The word '>R' takes an item from the Data Stack and puts it on the Return Stack. The word 'R>' does it the other way around. It takes the topmost item from the Return Stack and puts it on the Data Stack. Let's try it out:

```
: 2swap ( n1 n2 n3 n4)        \ four items on the stack
  rot   ( n1 n3 n4 n2)        \ rotate the topmost three
  >r    ( n1 n3 n4)           \ n2 is now on the Return Stack
  rot   ( n3 n4 n1)           \ rotate other items
  r>    ( n3 n4 n1 n2)        \ get n2 from the Return Stack
;
```

And why does it work in this colon-definition? Why doesn't the program go haywire? Because the Return Stack is and was perfectly balanced. The only thing we had to do was to get off "n2" before the semi-colon was encountered. Remember, the semi-colon compiles into 'EXIT' and 'EXIT' pops a return-address from the Return Stack. Okay, let me show you the Return Stack effects:

```
: 2swap ( r1)
  rot   ( r1)
  >r    ( r1 n2)
  rot   ( r1 n2)
  r>    ( r1)
;       ( --)
```

Note, these are the Return Stack effects! "R1" is the return-address. And it is there on top on the Return Stack when 'EXIT' is encountered. The general rule is:

> "Clean up your mess inside a colon-definition"

If you save two values on the Return Stack, get them off there before you attempt to leave. If you save three, get three off. And so on. This means you have to be very careful with looping and branching. Otherwise you have a program that works perfectly in one situation and not in another:

```
: this-wont-work ( n1 n2 -- n1 n2)
  >r             ( n1)
  0= if          ( --)
    r>           ( n2)
    dup          ( n2 n2)
  else
    1 2          ( 1 2)
  then
;
```

This program will work perfectly if n1 equals zero. Why? Let's look at the Return Stack effects:

```
: this-wont-work ( r1)
  >r             ( r1 n2)
  0= if          ( r1 n2)
    r>           ( r1)
    dup          ( r1)
  else           ( r1 n2)
    1 2          ( r1 n2)
  then
;
```

You see when it enters the 'ELSE' clause the Return Stack is never cleaned up, so 4tH attempts to return to the wrong address. Avoid this, since this can be very hard bugs to fix.

10.12 The Return Stack and the DO..LOOP

We've already told you that the limit and the counter of a DO..LOOP (or DO..+LOOP) are stored on the Return Stack. But how does this affect saving values in the middle of a loop? Well, this example will make that quite clear:

```
1       ( n)
10 0 do ( n)
  >r    ( --)
  i .   ( --)
  r>    ( n)
loop    ( n)
cr      ( n)
drop    ( --)
```

You might expect that it will show you the value of the counter ten times. In fact, it doesn't. Let's take a look at the Return Stack:

```
1       ( --)
10 0 do ( l c)
  >r    ( l c n)
  i .   ( l c n)
  r>    ( l c)
loop    ( --)
cr      ( --)
drop    ( --)
```

You might have noticed that it prints ten times the number "1". Where does it come from? Usually 'I' prints the value of the counter, which is on top of the Return Stack.

This time it isn't: the number "1" is there. So 'I' thinks that "1" is actually the counter and displays it. Since that value is removed from the Return Stack when 'LOOP' is encountered, it doesn't do much harm.

We see that we can safely store temporary values on the Return Stack inside a DO..LOOP, but we have to clean up the mess, before we encounter 'LOOP'. So, this rule applies here too:

> "Clean up your mess inside a DO..LOOP"

But we still have to be prepared that the word 'I' will not provide the expected result (which is the current value of the counter). In fact, 'I' does simply copy the topmost value on the Return Stack. Which is usually correct, unless you've manipulated the Return Stack yourself.

Note that there are other words beside 'I', which do exactly the same thing: copy the top of the Return Stack. But they are intended to be used outside a DO..LOOP. We'll see an example of that in the following section.

10.13 Other Return Stack manipulations

The Return Stack can avoid some complex stack acrobatics. Stack acrobatics? Well, you know it by now. Sometimes all these values and addresses are just not in proper sequence, so you have to 'SWAP' and 'ROT' a lot until they are.

You can avoid some of these constructions by just moving a single value on the Return Stack. You can return it to the Data Stack when the time is there. Or you can use the top of the Return Stack as a kind of local variable.

No, you don't have to move it around between both stacks all the time and you don't have to use 'I' out of its context. There is a well-established word, which does the same thing: 'R@'. This is an example of the use of 'R@':

```
: delete     ( n )
  >r #lag +  ( a1)
  r@ #lag    ( a1 a2 n2)
  r@ negate  ( a1 a2 n2 n3)
  r# +!      ( a1 a2 n2)
  #lead +    ( a1 a2 n2 a3)
  swap cmove ( a1)
  r> blanks  ( --)
;
```

'R@' copies the top of the Return Stack to the Data Stack. This example is taken from the 4tH-editor. It deletes "n" characters left of the cursor. By putting the number of characters on the Return Stack right away, its value can be fetched by 'R@' without using 'DUP' or 'OVER'. Since it can be fetched at any time, no 'SWAP' or 'ROT' has to come in.

10.14 Altering the flow with the Return Stack

The mere fact that return addresses are kept on the stack means that you can alter the flow of a program. This is hardly ever necessary, but if you're a real hacker you'll try this anyway, so we'd better give you some pointers on how it is done. Let's take a look at this program. Note that we comment on the Return Stack effects:

```
: soup ." soup " ;               ( r1 r2)
: dessert ." dessert " ;         ( r1 r6)
: chicken ." chicken " ;         ( r1 r3 r4)
: rice ." rice " ;               ( r1 r3 r5)
: entree chicken rice ;          ( r1 r3)
: dinner soup entree dessert ;   ( r1)
dinner cr                        ( --)
```

And this is the output:

```
soup chicken rice dessert
```

Before we execute "DINNER" the Return Stack is empty. When we enter "DINNER" the return address to the main program is on the Return Stack (r1).

"DINNER" calls "SOUP". When we enter "SOUP" the return address to "DINNER" is on the Return Stack (r2). When we are done with "SOUP", its return address disappears from the Return Stack and execution continues within "DINNER".

Then "ENTREE" is called, putting another return address on the Return Stack (r3). "ENTREE" on its turn, calls "CHICKEN". Another return address (r4) is put on the Return Stack. Let's take a look on what currently lies on the Return Stack:

-	Top Of Return Stack (TORS)
r4	returns to ENTREE
r3	returns to DINNER
r1	returns to main program

As we already know, ';' compiles an 'EXIT', which takes the TORS and jumps to that address. What if we lose the current TORS? Will the system crash?

Apart from other stack effects (e.g. too few or the wrong data are left on the Data Stack) nothing will go wrong. Unless the colon-definition was called from inside a DO..LOOP, of course. But what DOES happen? The solution is provided by the table: it will jump back to "DINNER" and continue execution from there.

```
: soup ." soup " ;                ( r1 r2)
: dessert ." dessert " ;          ( r1 r6)
: chicken ." chicken " r> drop ;  ( r1 r3 - r4 gets lost!)
: rice ." rice " ;                ( r1 r3 r5)
: entree chicken rice ;           ( r1 r3)
: dinner soup entree dessert ;    ( r1)
dinner cr                         ( --)
```

Since "CHICKEN" gets rid of the return address to "ENTREE", "RICE" is never called. Instead, a jump is made to "DINNER" that assumes that "ENTREE" is done, so it continues with "DESSERT". This is the output:

```
soup chicken dessert
```

Note that this is *not* common practice and we do *not* encourage its use[1]. However, it gives you a pretty good idea how the Return Stack is used by the system.

10.15 Leaving a colon-definition

You can sometimes achieve the very same effect by using the word 'EXIT' on a strategic place. We've already encountered 'EXIT'. It is the actual word that is compiled by ';'.

What you didn't know is that you can compile an 'EXIT' without using a ';'. And it does the very same thing: it pops the return address from the Return Stack and jumps to it. Let's take a look at our slightly modified previous example:

[1] As a matter of fact, it *may* interfere with 4tH's peephole optimizer.

```
: soup ." soup " ;                ( r1 r2)
: dessert ." dessert " ;          ( r1 r6)
: chicken ." chicken " ;          ( r1 r3 r4)
: rice ." rice " ;                ( is never reached)
: entree chicken exit rice ;      ( r1 r3)
: dinner soup entree dessert ;    ( r1)
dinner cr                         ( --)
```

After "CHICKEN" has been executed by "ENTREE", an 'EXIT' is encountered. 'EXIT' works just like ';', so 4tH thinks the colon-definition has come to an end and jumps back to "DINNER". It never comes to calling "RICE", so the output is:

```
soup chicken dessert
```

'EXIT' is mostly used in combination with some kind of branching like IF..ELSE..THEN. Compare it with 'LEAVE' that leaves a DO..LOOP early.

But now for the big question: what is the difference between 'EXIT' and ';'? Both compile an 'EXIT', but they are not aliases. 4tH will try to match every ';' with a ':'. If it doesn't succeed, it will issue an error message. This matching is not performed by 'EXIT'.

10.16 The layout of the Stack Area

Before we tell you how to obtain information on the Stack Area, we first have to explain you how it is laid out. We've already seen that there are two stacks: the Data Stack and the Return Stack. We also know what they are used for.

The next question is what part of the Stack Area is used by the Data Stack and what part is used by the Return Stack. In fact, both stacks share the very same Stack Area.

The Data Stack grows upward from the bottom and the Return Stack grows downward from the top. When they meet, you're in trouble. If the Return Stack causes the overflow, 4tH will report that the Return Stack overflowed. If it was the Data Stack, it will report that the Data Stack overflowed.

If an overflow happens, you can't say which stack actually overflowed. If the Data Stack filled up the Stack Area and a colon-definition tries to put a return address on the Return Stack, the Return Stack will get the blame.

Now for the good news. Because of this shared stack space, programs with different requirements can run without having to modify stack sizes (you can't do that; only the programmer of your application can). It can be a program that heavily uses the Return Stack (recursive colon-definitions) or a program that needs lots of data on the Data Stack.

What you can check is how big the Stack Area actually is. It is a constant named 'STACK'. It will report the size in cells. Every value on any stack (address or value) takes up a single cell.

You can also ask 4tH how many values are on the Data Stack using 'DEPTH'. It will report the number of values, before you executed 'DEPTH'. Let's elaborate on that a little more:

```
." Begin" cr                  \ no values on the stack
10                            \ 1 value on the stack
5                             \ 2 values on the stack
9                             \ 3 values on the stack
depth                         \ 4 values on the stack
. cr                          \ 4tH reports "3"
```

If you want to know what values the actual stack pointers have, you have to use 'SP@' and 'RP@'. By subtracting 'SP@' from 'RP@' you can see how much space is left in the Stack Area:

```
rp@ sp@ -
." Space left: " . ." cells" cr
```

10.17 Booleans and numbers

You might have expected we had discussed this subject much earlier. But we haven't and for one very good reason. We've told you a few chapters ago that 'IF' branches if the top of the stack is non-zero. Any number will do. So you would expect that this program will print "I'm here":

```
1 2 and
if
  ." I'm here"
then
```

In fact, it doesn't! Why? Well, 'AND' is a BINARY operator, not a LOGICAL operator. That means it reacts on bit-patterns. Given two numbers, it will evaluate bits at the same position.

The number "1" is "01" in binary. The number "2" is "10" in binary. 'AND' will evaluate the first bit (binary digit, now you know where that came from!). The first bit is the rightmost bit, so "0" for the number "2" and "1" for the number "1".

'AND' works on a simple rule, if both bits are "1" the result will be "1" on that position. Otherwise it will be "0". So "1" and "0" are "0". The evaluation of the second bit has the same result: "0". We're stuck with a number that is "0". False. So 'IF' concludes that the expression is not true:

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
01 AND                        \ binary number "1"
. cr                          \ binary result after AND
```

It will print "0". However, "3" and "2" would work just fine:

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
11 AND                        \ binary number "3"
. cr                          \ binary result after AND
```

It will print "10". The same applies to other binary operators as 'OR' and 'INVERT'. 'OR' works just like 'AND' but works the other way around. If both bits are "0" the result will be "0" on that position. Otherwise it will be "1":

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
01 OR                         \ binary number "1"
. cr                          \ binary result after OR
```

It will print "11". We do not encourage the use of 'INVERT' for logical operations. You should use '0=' instead.

'0=' takes the top of the stack and leave a true-flag if it is zero. Otherwise it will leave a false-flag. That means that if a condition is true (non-zero), it will leave a false-flag. Which is exactly what a logical NOT should do.

Take a look at his brother '0<>'. '0<>' takes the top of the stack and leaves a true-flag if it is non-zero. Otherwise it will leave a false-flag.

The funny thing is 'AND' and 'OR' work perfectly with flags and behave as expected. '0<>' will convert a value to a flag for you. So this works:

```
1 0<>
2 0<>
and if
  ." I'm here" cr
then
```

Of course, you don't have to use '0<>' when a word returns a flag. You should check the glossary for details on that.

10.18 Using ' with other names

So far we've only used "' (tick) with colon-definitions, but you can also use it with all constants, variables, values, strings, vectors (see section 11.7) and constant arrays. However, the information it provides is not always useful. E.g. the expression:

```
10 constant ten
' ten
```

Does not compile differently from:

```
10 constant ten
ten
```

The same applies to constant arrays and strings. It will give you possibly information on the address of variables, vectors, arrays and values, e.g.:

```
variable ten
' variable ." Relative address of ten: " . cr
```

Yes, *relative* address! What does that mean? When a 4tH program is compiled it has no idea how many application variables a host program will provide. So it stores a *relative* address. This address is relative to the address returned by 'FIRST'. You might call it an offset if you want to. 4tH provides a word which will convert the relative address of vectors, variables, values and numeric arrays to an absolute address, called '>BODY'. So this piece of code does exactly the same thing:

```
variable ten
ten                          \ throw address of 'ten' on stack
dup                          \ duplicate address
10 swap !                    \ store 10 at address
? cr                         \ show value stored at address
```

As this piece of code:

```
variable ten
' ten >body                    \ calculate address
dup                            \ duplicate address
10 swap !                      \ store 10 at address
? cr                           \ show value stored at address
```

There are not too many occasions where this is useful, but it let's take a look at this one:

```
0 value ten                    \ define a value
' ten >body                    \ calculate address of value
dup                            \ duplicate address
10 swap !                      \ store 10 at address
? cr                           \ show value stored at address
```

We already know that values, numeric arrays and vectors are stored in the very same area of the Integer Segment. This construction makes it possible to access them as variables.

You can access string constants or arrays of string constants with tick, but they will return a value which only has a meaning to 4tH itself. You won't be able to do anything useful with those values.

You should avoid these kind of constructions, but there might be some situations out there where it might come in handy. Note that you can only tick your *own* names. All of 4tHs built-in variables, strings, words, etc. cannot be accessed by tick.

10.19 Deleting files

4tH can also delete files on disk. That's very easy, just feed it the name or path of the file in question:

```
s" delete.me" delete-file .
```

If the deletion was successful, it prints zero, otherwise non-zero.

10.20 Querying environment variables

You can also retrieve the contents of every environment variable. Just feed 4tH with the name of your environment variable and it will return the address and count of its contents:

```
s" DIR4TH" environ@ type cr
```

This will print the contents of your DIR4TH environment variable. Note that the contents may be truncated if they would overflow the PAD. In such cases, you may want to use the libraryfile `getenv.4th`. This allows you to determine the string variable where the contents will be written to. However, it doesn't work on systems that do not support pipes (see section 9.27) and is a bit slower:

```
include lib/getenv.4th
1024 constant /buffer
/buffer string buffer
s" DIR4TH" buffer /buffer getenv type cr
```

Note that the contents are truncated as well if you don't make the buffer large enough. Like 'ENVIRON@' it will return an empty string on error.

10.21 What is not implemented

When writing a product like 4tH that is modelled after an existing programming language like Forth one has to cut a few corners somewhere.

Forth has a fundamentally different architecture, which allows you to extend the compiler with ease. 4tH is much more like conventional programming languages and many still wonder how we got this far.

When you're learning 4tH to learn Forth you will find there are things you can't do in 4tH. This section sums up most of the restrictions 4tH has in comparison to Forth and other languages.

Datatypes There are no words that allow you to define your own datatypes, although you can change the behaviour of individual variables.

Interpreter Since 4tH is a conventional compiler, you won't find a built-in interpreter. There is a library-source, which will enable you to make an interpreter for specific applications with ease. Next chapter we will show you how to use it.

If you have more questions concerning the functionality of 4tH, please read the ANS-Forth document. This describes the compliance of 4tH to the ANS-Forth standard. Further information can be obtained by studying the glossary.

10.22 Known bugs and limitations

Like every software product, 4tH has bugs. Because a work-around is available, fixing these bugs has no high priority.

- When you use '\' without any actual comment in a Unix ASCII file the complete next line will be marked as comment. With MS-DOS ASCII files 4tH will correctly detect a null string and terminate with an error. Add at least a dot or something, just to be safe.
- There can be only *one space* between '[DEFINED]', '[UNDEFINED]', '[CHAR]', 'CHAR' and the string following it. If you don't comply, 4tH will complain about empty string constants.
- You cannot use 'HIDE' conditionally. If you specify a 'HIDE' it must *always* be executed in 4tH.

Chapter 11

Advanced programming

11.1 Compiletime calculations

When you've reached this chapter, you must have quite some experience with 4tH. This chapter will help you to use 4tH to its full capacity. You'll be able to use software exceptions, conditional compilation, compiletime calculation, enumerations, private declarations and much, much more.

We've already explained that when you define a string, it has to be preceded by a literal expression:

```
64 constant name
16 constant #names

name #names * string name_space
```

The word '*' takes two subsequent literal expressions and multiplies them to a new single literal expression as if you'd written "1024" yourself! Everything that is evaluated at compiletime, *must* already be known at compiletime, thus a literal expression. Although not all arithmetic words compile to a literal expression, you can use the most common ones, like '*', '/', '+' and '-'. Another useful word is 'NEGATE', e.g. when you need to assign a negative value to a constant:

```
             16 constant +range
+range negate constant -range
```

In this example the value of "-RANGE" is -16. You can even mix and chain compiletime calculations. This will compile to the literal "500":

```
5 25 75 + *
```

Just as if you'd just written "500" in the sourcecode yourself. You can also write:

```
25 75 + 5 *
```

Because it's just simple postfix notation. Note that there must be two subsequent literal expressions available at any time, so this doesn't work:

```
5 5 * dup *
```

Since 'DUP' isn't a literal expression, but a word which is simply compiled. But don't worry: 4tH will notify you when you make an error like this. The final word, we'd like to present you is '[NOT]', which logically inverts a flag at compiletime, just like '0=' at runtime. This expression will compile to a true flag:

```
false [not]
```

You might wonder why we included this one, but that will become clear when you read the next section.

11.2 Conditional compilation

This is something which can be very handy when you're designing a 4tH program for different environments or even different Forth compilers. Let's say you've written a general ledger program in 4tH that is so good, you can sell it. Your customers want a demo, of course. You're willing to give one to them, but you're afraid they're going to use the demo without ever paying for it.

One thing you can do is limit the number of entries they can make. So, you copy the source and make a special demo version. But you have to do that for every new release. Wouldn't it just be easier to have one version of the program and just change one single constant? You can with conditional compilation:

```
true constant DEMO

DEMO [if]
256 constant #Entries
[else]
limit constant #Entries
[then]

variable CurrentEntry
#Entries array Entries
```

We defined a constant, called "DEMO", which is true. So, when the compiler reaches the "DEMO [IF]" line, it knows that it has to compile "256 constant Entries", since "DEMO" is true. When it comes to '[ELSE]', it knows it has to skip everything up to next '[THEN]'. So, in this case the compiler behaves like you've written:

```
256 constant #Entries
variable CurrentEntry
#Entries array Entries
```

Would you change "DEMO" to false, the compiler would behave as if you wrote:

```
limit constant #Entries
variable CurrentEntry
#Entries array Entries
```

The word '[IF]' only works at compile time and is *never* compiled into the object. '[IF]' takes a literal expression. If this expression is true, the code following the '[IF]' is compiled, just as '[IF]' wasn't there. Is this expression false, everything up to '[ELSE]' or '[THEN]' is discarded as if it wasn't there.

That also means you can discard any code that is superfluous in the program. E.g. when you're making a colon-definition to check whether you can make any more entries. If you didn't use conditional compilation, you might have written it like this:

```
: CheckIfFull                 ( n -- n)
  dup #Entries =              ( n f)
  if                          ( n)
    drop                      ( --)

    DEMO                      ( f)
    if                        ( --)
      ." Buy the full version"
    else                      \ give message and exit program
      ." No more entries"
    then                      ( --)

    cr quit
  then                        ( n)
;
```

But his one is nicer and will take up less code:

```
: CheckIfFull                 ( n -- n)
  dup #Entries =              ( n f)
  if                          ( n)
    drop                      ( --)

DEMO [if]                     ( n f)
    ." Buy the full version"
[else]
    ." No more entries"
[then]

    cr quit
  then                        ( n)
;
```

You can also use conditional compilation to discard large chunks of code. This is a much better way than to comment all the lines out, e.g. this won't work anyway:

```
(
: room?                       \ is it a valid variable?
  dup                         ( n n)
  size 1- invert and          ( n f)
  if                          \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
)
```

This is pretty cumbersome and prone to error:

```
\ : room?                     \ is it a valid variable?
\   dup                       ( n n)
\   size 1- invert and        ( n f)
\   if                        \ exit program
\     drop ." Not an element of ROOM" cr quit
\   then
\ ;
```

But this is something that can easily be handled:

```
false [if]
: room?                        \ is it a valid variable?
  dup                          ( n n)
  size 1- invert and           ( n f)
  if                           \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
[then]
```

Just change "false" to "true" and the colon-definition is part of the program again. Note that '[IF]..[ELSE]..[THEN]' can be nested! Conditional compilation is very powerful and one of the easiest features a language can have. And it's ANS-Forth compatible!

11.3 Checking the environment at compiletime

Let's say you've written something which works perfectly on your own machine and you want to use it on the mainframe at work. It turns out to be it doesn't work. Why? Because your program assumed that a cell was four address units wide. And it didn't turn out to be that way.

You could have prevented that if you had used a check at compiletime. You can do that this way:

```
/cell 4 [=] [NOT] [IF]
( do something)
[THEN]
```

'/CELL' is a constant which holds the number of address units in a cell. '/CELL' has got a little brother called '/CHAR', which will tell you how many address units there are in a character. '[=]' will check whether a cell has four address units and '[NOT]' will reverse that flag. Neat huh?

But then again, what do we do if it doesn't turn out to be that way. Any action will first be executed at runtime so a message or 'ABORT' won't do. Further compilation will be useless, so we actually want to stop. You're in luck, since we have a special word that will stop the compiler regardless. It's called '[ABORT]'. So this is our complete snippet:

```
/cell 4 [=] [NOT] [IF]
[ABORT]
[THEN]
```

But suppose you want to check whether a cell is *at least* 4 address units. '[=]' won't do in that case. Of course, you can check every imaginable cellsize, but that is not very pretty. That is where '[SIGN]' comes in. '[SIGN]' will take a previously compiled literal expression and compile -1 if the number was negative, zero if the number was zero and 1 if the number was positive. You may wonder how that does help. Well, consider this one:

```
/cell 4 - [SIGN] -1 [=] [IF]
[ABORT]
[THEN]
```

What have we been doing here? First, we substract the minimal cellsize from the actual cellsize. If the sign of the sum is -1, compilation is aborted. The sum can *only* be negative when '/CELL' is three or smaller. Get it? By using '[SIGN]' you can make all kinds of compiletime comparisons, which makes it a real asset.

11.4 Checking a definition at compiletime

We've already encountered 'COMPARE' in section 8.17. 'COMPARE' is word that compares two strings. It can do that both case sensitive and case insensitive. If you define a constant called `casesensitive` before the '[NEEDS' directive, it will perform a case sensitive comparison. If you don't, it will do a case insensitive comparison by default.

Most approaches would require the definition of `casesensitive`, regardless which mode you select. This one doesn't:

```
: compare                               ( a1 n1 a2 n2 -- f )
  rot over over swap - >r               ( a1 a2 n2 n1)
  min 0 tuck                            ( a1 a2 0 n 0)
  ?do                                   ( a1 a2 f)
    drop                                ( a1 a2)
    over i + c@                         ( a1 a2 c1)
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    over i + c@                         ( a1 a2 c1 c2)
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    - dup                               ( a1 a2 f f)
    if leave then                       ( a1 a2 f)
  loop
  >r drop drop r> r> swap dup           ( f1 f2 f2)
  if swap then drop                     ( f)
;
```

'[UNDEFINED]' checks whether the word following it has been defined and leaves a TRUE flag if it wasn't. It doesn't matter whether the word is built-in, included or defined in your program. It can be a variable, a word, a constant, anything you like.

In this case it checks whether `casesensitive` has been defined. If it isn't, a line of code is compiled. If an 'IF' had been used, the code would always be compiled with the added overhead of testing a constant at runtime. This construction allows for tighter and faster code.

Note that '[UNDEFINED]' has also a counterpart called '[DEFINED]'. It leaves a true flag when the word following it has been defined.

11.5 Exceptions

You know when you violate the integrity of 4tH, it will exit and report the cause and location of the error. Wouldn't it be nice if you could catch these errors within the program? It would save a lot of error-checking anyway. It is quite possible to check every value within 4tH, but it takes code and performance, which makes your program less compact and slower.

Well, you can do that too in 4tH. And not even that, you can trigger your own errors as well. This simple program triggers an error and exits 4tH when you enter a "0":

```
[needs lib/enter.4th]          \ get a number
                               \ if non-zero, return it
                               \ if zero, throw exception
: could-fail                   ( -- n)
  enter dup 0=
```

```
  if 1 throw then
;
                                \ drop numbers and
                                \ call COULD-FAIL
: do-it                         ( -- )
  drop drop could-fail
;
                                \ put 2 nums on stack and
                                \ execute DO-IT
: try-it                        ( -- )
  1 2 ['] do-it execute
  ." The number was" . cr
;
                                \ call TRY-IT
try-it
```

"TRY-IT" puts two numbers on the stack, gets the execution token of "DO-IT" and executes it. "DO-IT" drops both numbers and calls "COULDFAIL". "COULD-FAIL" gets a number and compares it against "0". If zero, it calls an exception. If not, it returns the number.

The expression "1 THROW" has the same effect as calling 'QUIT'. The program exits, but with the error message "Unhandled exception". You can use any positive number for 'THROW', but "0 THROW" has no effect. This is called a "user exception", which means you defined and triggered the error.

There are also system exceptions. These are triggered by the system, e.g. when you want to access an undefined variable or print a number when the stack is empty. These exceptions have a negative number, so:

```
throw -4
```

Will trigger the "Stack empty" error. You can use these if you want but we don't recommend it, since it will confuse the users of your program.

You're probably not interested in an alternative for 'QUIT'. Well, 'THROW' isn't. It just enables you to "throw" an exception and exceptions can be caught by your program. That means that 4tH won't exit, but transfers control back to some routine. Let's do just that:

```
[needs lib/enter.4th]

: could-fail                    ( -- n)
  enter dup 0=
  if 1 throw then
;

: do-it                         ( -- )
  drop drop couldfail
;

: try-it                        ( -- )
  1 2 ['] do-it catch
  if drop drop ." There was an exception" cr
  else ." The number was" . cr
  then
;

try-it
```

The only things we changed is a somewhat more elaborate "TRY-IT" definition and we replaced 'EXECUTE' by 'CATCH'.

'CATCH' works just like 'EXECUTE', except it returns a result-code. If the result-code is zero, everything is okay. If it isn't, it returns the value of 'THROW'. In this case it would be "1", since we execute "1 THROW". That is why "0 THROW" doesn't have any effect.

If you enter a nonzero value at the prompt, you won't see any difference with the previous version. However, if we enter "0", we'll get the message "There was an exception", before the program exits.

But hey, if we got that message, that means 4tH was still in control! In fact, it was. When "1 THROW" was executed, the stack-pointers were restored and we were directly returned to "TRY-IT". As if "1 THROW" performed an 'EXIT' to the token following 'CATCH'.

Since the stack-pointers were returned to their original state, the two values we discarded in "DO-IT" are still on the stack. But the possibility exists they have been altered by previous definitions. The best thing we can do is discard them.

So, the first version exited when you didn't enter a nonzero value. The second version did too, but not after giving us a message. Can't we make a version in which we can have another try? Yes we can:

```
[needs lib/enter.4th]

: could-fail                  ( -- n)
  enter dup 0=
  if 1 throw then
;

: do-it                       ( -- )
  drop drop could-fail
;

: retry-it                    ( -- )
  begin
    1 2 ['] do-it catch
  while
    drop drop ." Exception, keep trying" cr
  repeat
  ." The number was " . cr
;

retry-it
```

This version will not only catch the error, but it allows us to have another go! We can keep on entering "0", until we enter a nonzero value. Isn't that great? But it gets even better! We can exhaust the stack, trigger a system exception and still keep on going. But let's take it one step at the time. First we change "COULD-FAIL" into:

```
: could-fail                  ( -- n)
  enter dup 0=
  if drop ." Stack: " depth . cr 1 throw then
;
```

This will tell us that the stack is exhausted at his point. Let's exhaust is a little further by redefining "COULD-FAIL" again:

```
: could-fail                  ( -- n)
  enter dup 0=
  if drop drop then
;
```

Another 'DROP'? But wouldn't that trigger an "Stack empty" error? Yeah, it does. But instead of exiting, the program will react as if we wrote "-4 THROW" instead of "DROP DROP". The program will correctly report an exception when we enter "0" and act accordingly.

This will work with virtually every runtime error. Which means we won't have to protect our program against every possible user-error, but let 4tH do the checking.

We won't even have to set flags in every possible colon-definition, since 4tH will automatically skip every level between 'THROW' and 'CATCH'. Even better, the stacks will be restored to the same depth as they were before 'CATCH' was called.

You can handle the error in any way you want. You can display an error message, call some kind of error-handler, or just ignore the error. Is that enough flexibility for you?

11.6 Enumerations

Sometimes you need a lot of constants:

```
0 constant Monday
1 constant Tuesday
2 constant Wednesday
3 constant Thursday
4 constant Friday
5 constant Saturday
6 constant Sunday
```

A little error here may ruin your program. This does the very same thing, except it is easier to maintain:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  enum Sunday   drop
```

'ENUM' is much like a 'CONSTANT', but increments and leaves a value after the constant has been created. That is why we need to add 'DROP' after the final enumeration. To show you that 'ENUM' and 'CONSTANT' are much alike, you could also write the declaration above as:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  constant Sunday
```

Since 'CONSTANT' just consumes the value, you don't need the final 'DROP'.

11.7 Forward declarations

It doesn't happen very often, but sometimes you have a program where two colon-definitions call each other. When you look at 4tHs source you find several examples. The throw() function calls the rpop() function, because 'THROW' takes items from the Return Stack. On the other hand, when the Return Stack underflows, it has to call 'THROW'.

There is a special instruction in 4tH to do this, called 'DEFER'. 'DEFER' doesn't create an executable word, but a vector containing an execution token[1], which is executed when called. You might want to consult section 10.3 first to see how this works. But for all purposes you might consider it to be an executable word, because it behaves the same way.

[1] See section 10.4.

```
defer Step2
```

Now we can create "STEP1" without a problem:

```
: Step1 1+ dup . cr Step2 ;
```

But "STEP2" does not have a body yet. Of course, you could create a new colon-definition, tick it and assign the execution token to "STEP2" manually, but it is much neater to use ':NONAME'. ':NONAME' can be used like a normal ':', but it doesn't require a name. Instead, it pushes the execution token of the colon-definition it created on the stack. No, ':NONAME' does *not* create a literal expression, but it is just what we need:

```
:noname 1+ dup . cr Step1 ; is Step2
```

Now we are ready! We can simply execute the program by calling "STEP1":

```
1 Step1
```

Note that if you run this program, you'll get stack errors! Sorry, but the example has been taken from a Turbo Pascal manual ;-). If you have forgotten what a deferred word actually executes, you can retrieve the execution token by using 'DEFER@':

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus is thisword            \ assign the word to the vector
' thisword defer@             \ retrieve the execution token
2 3 rot execute               \ execute the deferred word
. cr                          \ display the result
```

As a matter of fact, this expression:

```
' thisword defer@ execute
```

Is equivalent to this one:

```
thisword
```

You can also reassign a vector without using 'IS'. 'IS' is a parsing version. That means the actual vector to which a certain behaviour is assigned is determined at *compiletime*. 'DEFER!' can be used to assign a certain behaviour at *runtime*. 'DEFER!' takes two execution tokens:

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus ' thisword defer!      \ assign it to a vector
```

This is equivalent to this:

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus is thisword            \ assign it to a vector
```

I guess you'll agree with me that this creates countless possibilities.

11.8 Recursion

Yes, but can she do recursion? Of course she can! It is even very natural and easy. Everybody knows how to calculate a factorial. In 4tH you can do this by:

```
: factorial                    ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    factorial *
  then
;

10 factorial . cr
```

Which is exactly as one would expect. Unfortunately, this is not the way it is done in ANS-Forth. In order to let a colon-definition call itself, you have to use the word 'RECURSE'. 4tH supports this word too:

```
: factorial                    ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    recurse *
  then
;

10 factorial . cr
```

It will even compile to the same code. If you use the word 'RECURSE' outside a colon-definition, the results are undefined. Note that recursion lays a heavy burden on the return stack. Sometimes it is wiser to implement such a routine differently:

```
: factorial
  dup
  begin
    dup 2 >
  while
    1- swap over * swap
  repeat

  drop
;

10 factorial . cr
```

So if you ever run into stack errors when you use recursion, keep this in mind.

11.9 Private declarations

Sometimes you want to hide some definitions from other programmers. This is especially true when you're writing libraries or templates. The Application Programmers Interface must be public of course, but you don't want anyone else to tinker with the internals of your library. And there is the problem of cluttering your name space.

Relax, 4tH has a way to get rid of these internal words. It's easy, just tell 4tH to hide them:

```
VARIABLE #emits                        \ private

: SHOW emit 1 #emits +! ;              \ public
: NL CR 0 #emits ! ;                   \ public

hide #emits
```

After that the name "#EMITS" is no longer recognized and can be reused if you want to, e.g. this is completely valid:

```
: dummy ;
hide dummy
: dummy ." I am no longer a dummy!" cr ;
```

As a matter of fact, the previous declaration of "DUMMY" has been turned into a ':NONAME' declaration by the use of 'HIDE'. Note that 'HIDE' is meant to make definitions *private*; not to optionally override previously defined definitions in order to achieve a Forth-like behavior:

```
[DEFINED] myoption [IF]
hide myoption
[THEN]

true CONSTANT myoption
```

If you want to achieve that, use the "first come, first served" rule:

```
[UNDEFINED] myoption [IF]
true CONSTANT myoption
[THEN]
```

The constant "MYOPTION" is only defined if it hasn't been defined before. You may have to change a few things here and there to make it work, but there are obvious advantages to this construction.: it saves memory and is much easier to understand and maintain. And you won't scratch your head why your perfectly valid 4tH program doesn't compile[2].

11.10 Aliases

Sometimes you want to make an alias for a word. Of course you can embed the word you want to alias in a new definition:

```
: noop ;
: nop noop ;
```

Although this approach works perfectly under all circumstances it has its disadvantages, because calling a word is relatively slow. Unless you're trying to make an alias for an internal word, you'd better use an 'ALIAS ':

```
: noop ;
' noop alias nop
```

[2]The tokenizer evaluates 'HIDE' in order to estimate the size of the symboltable while conditional compilation is first evaluated by the parser.

This is completely equivalent to:

```
defer nop
: noop ;
' noop is nop
```

Although the vector takes up a little space, it will save you from most of the calling overhead. Since you can only alias self-defined executable words, 'ALIAS' is quite limited. 'AKA' does not have that disadvantage:

```
: noop ;
aka noop nop
```

Both words are now completely equivalent and compile to exactly the same code. Even better, you can use 'AKA' with *every* self-defined word, including variables, vectors, files, values, fields and constants. 'AKA' even works with most built-in 4tH words, e.g. constants:

```
aka + plus
```

'AKA' will *not* work with defining, preprocessor, flow control words or inline macros[3]. If you're unsure, just try! If 4tH bombs out with an error message, you'll know. 'AKA' is also known as "also known as".

11.11 Changing behavior of data

One of the most ingenious things Forth can do, is change the behavior of data at runtime. With 4tH, you cannot do this for an entire datatype, but you can do it for individual 'VARIABLE's, 'CREATE's, 'STRING's, 'ARRAY's and 'CONSTANT's. Just use:

```
:REDO <name> <definition> ;
```

Where <name> is a previously defined 'VARIABLE', 'STRING', etc. The definition will behave as if the 'VARIABLE', etc. has just been thrown on the stack, e.g. to make a 'VARIABLE' behave as a 'CONSTANT 'you define:

```
variable me

10 me !
:redo me @ ;
```

The body of the definition will behave as if it said:

```
me @
```

Which boils down to a (rather slow) constant. You cannot change the contents of the variable anymore if you haven't taken precautions, because there is no way to address it. Here is another, more elaborate example:

[3]Rule of the thumb: any word that compiles to a single token. See the glossary for details.

```
create life                    \ create an array of string constants
  ," This is my life!"
  ," This is your life!"

0 constant my                  \ create two constants
1 constant your                \ to address the elements
                               \ now change the behaviour of LIFE
:redo life swap th @c count type cr ;

my life                        \ use it!
your life
```

At runtime, this will print:

```
This is my life!
This is your life!
```

There is even a shorter way to do this by using 'DOES>', e.g. we could also write the example above as:

```
create life                    \ create an array of string constants
  ," This is my life!"
  ," This is your life!"
does> swap th @c count type cr ;

0 constant my                  \ create two constants
1 constant your                \ to address the elements
                               \ now change the behaviour of LIFE
my life                        \ use it!
your life
```

'DOES>' always applies to the last defined 'VARIABLE', 'CREATE', 'STRING', 'ARRAY' or 'CONSTANT', so you don't have to repeat the name. So why ':REDO', you might ask. Well, sometimes you have to define some other words before you encapsulate the data. ':REDO' allows you to do just that.

If you just want to initialize the data, you can do so with 'LATEST'. 'LATEST' will compile to the last word you have defined, so if you want to change the name of your definition, that's the only thing you have to do. E.g. this is the 4tH translation of a double number[4] constant:

```
include lib/anscore.4th        \ for 2@, 2!
include lib/ansdbl.4th         \ for U>D

2 array mydouble               \ define a double number
max-n u>d latest 2!            \ initialize it
does> 2@ ;                     \ now make it a constant
```

Wording has always been very important to Forth. Using this technique, you can make your programs even more readable.

[4] See section 12.22.

11.12 Multidimensional arrays

We've seen two dimensional arrays with 'ARRAY' and 'STRING', but what about multi-dimensional arrays. Well, it's the same thing all over again. C doesn't actually have multi-dimensional arrays either. When you define one, just a chunk of memory is allocated.

In 4tH you can do the same thing, but now you have to do it yourself. E.g. when you want to define a matrix of cells of 4 rows by 5 elements, you have to multiply those and allocate an array of that size:

```
4 5 * array my_array
```

But what if you want to reference the fourth element of the third row? You *cannot* write something like:

```
2 3 my_array @
```

That's right. But you can change the behaviour of "MY_ARRAY" accordingly:

```
:redo my_array                ( n1 n2 -- a)
  rot 5 *                     \ calculate row offset
  rot +                       \ calculate element offset
  cells                       \ calculate number of cells
  +                           \ add to address of my_array
;
```

Or even better:

```
4 5 * array my_array does> rot 5 * rot + cells + ;
```

This word calculates the correct offset for you. Note that the third row is row number two (since we start counting from 0) and the fourth element is element number three:

```
2 3 my_array @
```

You can also use "MY_ARRAY" to initialize an array, since it simply calculates the correct address for you:

```
5 2 3 my_array !              \ sets 3rd row 4th element to 5
```

You can add more dimensions if you want. This works basically the same way: create an array of a size that equals the products of its dimensions and design a word that calculates the correct address.

11.13 Binary string constants

A binary string constant is an unterminated string that doesn't necessarily contain characters. Creating binary string constants is easy. Just compile them by their ASCII value into the String Segment with 'C,':

```
char H c, char i c, char ! c, 0 c,
```

The fun of it all is that 4tH doesn't allow you to access the String Segment directly, so you can *never* retrieve them. You need 'OFFSET' to define a word which does all the hard work for you. At runtime it takes an index and leaves the ASCII value of the character in question on the stack. 'OFFSET' is used just before you compile the ASCII values:

```
offset greet char H c, char i c, char ! c, 0 c,
```

Note that you have to terminate a binary string constant *manually* if you need to, although it is perfectly legal to create binary string constants with no termination at all. Retrieving characters is easy. This will print "Hi!":

```
0 greet emit
1 greet emit
2 greet emit cr
```

And so will this:

```
0 begin                        \ setup index
  dup greet dup                \ retrieve character
while                          \ if not terminated
  emit 1+                      \ emit and increase index
repeat drop drop               \ clear stack
```

You can use binary string constants for compact tables, bitstrings or any other raw data as long as each element doesn't exceed the size of a single character. Although useful in itself, you'll get into trouble when you try to pack several binary strings in a single datastructure. Sure, you can give each binary string its own 'OFFSET' but binding their addresses into some form of array structure? No, because an 'OFFSET' is not a literal expression and hence cannot be compiled that way.

A solution is to "tag" individual strings. Tags are literal expressions and compilable, so you can do this:

```
OFFSET names
  TAG names Hans
    char H c, char a c, char n c, char s c, 0 c,
  TAG names Wil
    char W c, char i c, char l c, 0 c,
  TAG names Phil
    char P c, char h c, char i c, char l c, 0 c,
```

Or this, if you want to make it a little less "wordy":

```
OFFSET names
  TAG names Hans c" Hans" 0 c,
  TAG names Wil  c" Wil"  0 c,
  TAG names Phil c" Phil" 0 c,
```

Both expressions are completely equivalent. 'C"' simply offers you a much easier way to enter a *printable* string. Note that 'C|' is available too, if you happen to use embedded quotes. And yes, you can mix 'C,' and 'C"' as much as you like.

You can feed the 'TAG' as an index to an 'OFFSET', so this will return "W" (from "Wil"):

```
Wil names emit
```

And as promised, this will work as well:

```
CREATE developers
  Hans , Phil , Wil , NULL ,
```

Contrary to popular belief, this construct is as memory efficient as:

```
CREATE developers
  ," Hans" ," Phil" ," Wil"  NULL ,
```

It is just a bit more cumbersome to set up. You can even wrap up your binary string array into a ':REDO' definition, giving:

```
:redo developers
  swap cells + @c
  begin dup names dup while emit 1+ repeat
  drop drop cr
;
```

Which means you can write:

```
1 developers
```

Which will display "Phil". This way, embedding e.g. UTF-8 code not only becomes possible, but also very easy to set up.

11.14 Binary string variables

You can define raw chunks of memory with 'BUFFER:'. This will allocate a raw chunk of data of 1024 bytes:

```
1024 buffer: a
```

Finally, if you want to move raw chunks of data around, there is 'MOVE':

```
1024 buffer: a
1024 buffer: b

a b 512 move
```

This wil define two raw buffers of 1024 bytes and 'MOVE' will copy the first 512 bytes of buffer "a" to buffer "b".

11.15 Records and structures

The easiest way is to allocate a structure in the Character Segment. Just define the structure like this:

```
struct
  32 +field Name
  64 +field Address
  32 +field City
   4 +field Age
end-struct /Person
```

This might be a familiar example to you. We'll store information on a single person in this structure. Now we got the fields, the length of the fields and the length of the entire structure, stored in "/Person". Both the fields and the entire structure are nothing more than a set of constants, e.g. the offset of the field "Name" is stored in a CONSTANT named "Name". However, we still haven't allocated any memory. We can allocate room for the structure we've just defined by just using the word 'STRING'. Note that you can also create a cell-based structure. Then you need the word 'ARRAY' to allocate the memory required.

```
/Person string Person
```

Now we can define a word which initializes the fields:

```
: InitRecord                  \ initialize fields
  s" Hans Bezemer" Person -> Name    place
  s" Lagendijk 79" Person -> Address place
  s" Den Helder"   Person -> City    place
  s" 44"           Person -> Age     place
;
```

Of course, you can also use 'ACCEPT' to enter the contents of the fields. Fields act like ordinary strings. Note that numbers are stored as strings as well. This is not too much of a problem since 'NUMBER' can convert them back to numbers anyway.

This is a very simple use of structures. You can also use structures within structures:

```
struct
  64 +field Address
  32 +field City
end-struct /Location

struct
  32 +field Name
  /Location field Location
end-struct /Person

/Person string Person

s" Delft" Person -> Location -> City place
```

If you want to make an array of structures, that can be done as well:

```
struct                      \ create structure
  32 +field Name
  64 +field Address
  32 +field City
   4 +field Age
end-struct /Person

32 constant #Person         \ size of array of structs
                            \ now allocate the room
#Person /Person * string Persons
                            \ make it behave properly
:redo Persons swap /Person * + ;
```

```
                              \ initialize the first record
s" Hans Bezemer" 0 Persons -> Name    place
s" Lagendijk 79" 0 Persons -> Address place
s" Den Helder"   0 Persons -> City    place
s" 44"           0 Persons -> Age     place
```

You can also extend an already existing structure:

```
struct                        \ create structure
  32 +field Name
  64 +field Address
  32 +field City
  12 +field Age
end-struct /Person
                              \ now extend the structure
/Person
  32 +field Job
  16 +field Emp-number
end-struct /Employee
```

You now got two different structures, ”/Person” and ”/Employee”, that share the first four fields. Defining a structure within a structure is possible too:

```
struct                        \ create structure
  32 +field Firstname
  64 +field Lastname
  struct                      \ structure within a structure
    64 +field Address
     8 +field Zip
    32 +field City
  +field Location             \ let's give it a name
  4  +field Age
end-struct /Person
```

Let’s use it:

```
/Person string Person         \ allocate some space
                              \ initialize it
s" Hans"         Person -> Firstname place
s" Bezemer"      Person -> Lastname  place
s" Lagendijk 79" Person -> Location -> Address place
s" Den Helder"   Person -> Location -> City    place
s" 44"           Person -> Age  place
```

And finally, if you want to define a structure and allocate the memory it occupies at the same time, you can do that one too. After all, ’+FIELD’ is literal expression[5]:

```
struct                        \ define the structure
  32 +field Firstname
  64 +field Lastname
   8 +field Initials
string Person                 \ allocate the structure

s" Hans" Person -> Firstname place
s" JL"   Person -> Initials place
```

Well, if that isn’t a complete implementation, I don’t know what is..

[5] See section 7.13.

11.16 Unions

A union is a bunch of variables that share the same memory. Defining it in 4tH is quite easy:

```
include lib/ncoding.4th     \ include NCODING library

struct                      \ start defining the union
  32   /field               \ first field
  nell /field               \ second field
end-struct /union           \ end union definition

/union string myunion       \ allocate some space
                            \ now use it
s" Hans Bezemer" myunion place
1960 myunion n!
```

The union has the size of its *largest* field. You may wonder why the individual fields don't have their own names, but there is a reason for that. In other languages the field determines the type, so the compiler can figure out if you're using it properly. Since 4tH has no typechecking names are pretty useless, consequently there aren't any. You can use a union within another union or structure:

```
include lib/fp1.4th        \ include the FP library

struct                     \ define a union
  1 cells /field           \ sharing an integer
  float   /field           \ and a floating point
end-struct /number         \ number

struct                     \ define a structure
  field: type_tag          \ with some fields
  /number +field x         \ and the union concerned
array z                    \ allocate some space

23     z -> x !            \ use it as a number
23 s>f z -> x f!           \ use it as an FP number
```

You can also embed the union in the structure if you like:

```
include lib/fp1.4th        \ include the FP library

struct                     \ define a structure
  field: type_tag          \ with some fields
  struct                   \ define a union
    1 cells /field         \ sharing an integer
    float   /field         \ and an FP number
  +field x                 \ embed the union
array z                    \ allocate some space

23     z -> x !            \ use it as a number
23 s>f z -> x f!           \ use it as an FP number
```

If you really want to differentiate beween the floating point number and the integer, you can make an alias for field "X":

```
aka x y
```

And use it accordingly:

```
23     z -> x !            \ use it as a number
23 s>f z -> y f!           \ use it as an FP number
```

It is only cosmetic though, since it will compile to the same code and 4tH won't issue an error if you don't use it properly.

11.17 Complex control structures

Sometimes, the normal control structures of 4tH are not enough. Take this implementation of '-TRAILING':

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup                       \ quit if length is zero
  while
    2dup 1- chars + c@ bl <>  \ is it still a space?
    if exit else 1- then      \ if not, quit
  repeat                      \ if so, decrement length
;
```

No one will tell you that this is elegant. You have to perform a test and quit the word. And this is still palatable. Imagine you have to test several conditions like this! It will become horrible pretty soon! Therefore, 4tH supports extended control structures. We've seen the basic control structures in sections 7.22, 7.23 and 7.24. Now we're expanding those into:

```
BEGIN .. WHILE .. WHILE .. AGAIN | REPEAT
BEGIN .. WHILE .. WHILE .. UNTIL
```

Yes, that's right: 'REPEAT' and 'AGAIN' are actually aliases. But what can we do with them? Well, take a look at our modified '-TRAILING' word:

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup
  while                       \ quit if length is zero
    2dup 1- chars + c@ bl =
  while                       \ quit if it is not a space
    1-                        \ decrement length
  repeat
;
```

You have to admit that the latter version is much more elegant and readable. Although very elegant and usuable, in some circumstances you want to know which 'WHILE' terminated the loop and take additional action. This means you either have to set an additional flag or repeat the test, which is neither elegant nor efficient. For that purpose 4tH offers the DONE..DONE construct. In this example "-TRAILING" tells you why it terminated:

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup
  while                       \ quit if length is zero
    done ." All spaces or NULL string" cr done
    2dup 1- chars + c@ bl =
  while                       \ quit if it is not a space
    1-                        \ decrement length
    done ." String trimmed" cr done
  repeat
;
```

A DONE..DONE clause *always* belongs to the last defined 'WHILE', so it doesn't really matter where you put it. What 'DONE' actually does is resolve the last defined 'WHILE' or 'DONE'. The last 'DONE' always jumps to 'REPEAT', so a single 'DONE' is a 'NOOP'. You can actually define something like this:

```
begin
  ( condition)
  while
    ( true)
    done ( false) done
    ( true)
    done ( false) done
    ( true)
    done
    ( true)
repeat
```

But you'll have to agree it's not very useful[6].

11.18 Optimization

The best way to optimize a program is to look out for certain patterns and obliterate them where possible, e.g.:

```
swap 2drop
```

Is a useless sequence of words that could easily be replaced by a single "2DROP". You don't need an optimizer to see that one. 4tH does very little optimization itself. One thing 4tH does for you is *tail call optimization*. That means that if the last instruction before the semicolon is a call to another word, it will change the 'CALL' instruction to a 'BRANCH' instruction. The advantage is twofold:

1. It will save you an expensive 'CALL' - 'EXIT' sequence;
2. It will save you Return Stack space.

Consequently, your programs will run a little bit faster[7]. If you want any further improvement, you will have to do that yourself. E.g. this will optimize a tail call optimization even more. Simply change:

```
: myfirstword ;
: mysecondword ;
: mythirdword if myfirstword else mysecondword then ;
```

Into this:

```
: myfirstword ;
: mysecondword ;
: mythirdword if myfirstword exit then mysecondword ;
```

If we decompile these programs we see the difference right away. The first compiles to:

[6] You can actually define "`if else else else then`" too, which is not very helpful neither.

[7] Special threading benchmarks show a speed increase of 40%, although real life programs will never see that much improvement.

```
[   0] branch    (1)
[   1] exit      (0)
[   2] branch    (3)
[   3] exit      (0)
[   4] branch    (9)
[   5] 0branch   (7)
[   6] call      (0)
[   7] branch    (8)
[   8] branch    (2)
[   9] exit      (0)
```

You can clearly see the tail call optimization at word 8. However, there is a 'BRANCH' at word 7 and an 'EXIT' at word 9 we can do without. The second version fixes this:

```
[   0] branch    (1)
[   1] exit      (0)
[   2] branch    (3)
[   3] exit      (0)
[   4] branch    (7)
[   5] 0branch   (6)
[   6] branch    (0)
[   7] branch    (2)
```

After the call at word 6 we immediately branch, which is correct because there is no code to execute after that. Note the tail optimizer was able to kick in *twice* here. Disadvantage of the second version is that it is less expressive, which means it is not completely clear why the programmer chose this construction.

In some rare circumstances, e.g. when you're calling user-defined words which manipulate the return stack[8], you *don't* want the tail optimizer to kick in. Suppressing it is very simple, just add the '[FORCE]' directive to the 'EXIT' or change the order of words slightly so that a reserved word is compiled at the end:

```
: return r> drop ;
: test dup if 1+ else return then [force] ;
0 test . cr
```

Another area of optimization is the calculation of constants. If we do it at compile time, we do not have to do it at runtime. Especially when it is in the middle of a loop or a word we frequently use. 4tH features a small peephole optimizer that tries to make the best of it. The rules are simple:

1. If you compile the word '+' or '-' and the previously compiled word was a LITERAL, a +LITERAL will be compiled *unless* rule 6 can be applied;
2. If you compile the word '*' and the previously compiled word was a LITERAL, a *LITERAL will be compiled *unless* rule 7 can be applied;
3. If you compile the word '/' and the previously compiled word was a LITERAL, a /LITERAL will be compiled *unless* rule 8 can be applied;
4. If you compile the word 'NEGATE' and the previously compiled word was a LITERAL, *LITERAL or /LITERAL, it will be negated;
5. If you compile the word '@' or '!' and the previously compiled word was a VARIABLE, a VALUE or TO will be compiled;

[8] E.g. NR> and N>R.

6. If you compile a +LITERAL and the previously compiled word was a LITERAL, +LITERAL or VARIABLE, the +LITERAL will added to it;
7. If you compile a *LITERAL and the previously compiled word was a LITERAL or *LITERAL, it will be multiplied;
8. If you compile a /LITERAL and the previously compiled word was a LITERAL it will be divided. If it was a /LITERAL, it will be multiplied;
9. If you compile a +LITERAL with the value 0 or a /LITERAL or *LITERAL with value 1, nothing is compiled at all;
10. If you compile a *LITERAL and the previously compiled word was a NEGATE, it will be replaced by a *LITERAL with a negative argument;
11. If you compile a /LITERAL and the previously compiled word was a NEGATE, it will be replaced by a /LITERAL with a negative argument;
12. If you compile a +LOOP with the value 1, LOOP is compiled;
13. If a 'THEN' or 'BEGIN' is compiled, the peephole optimizer is disabled[9] and previously compiled code will not be further optimized.

Let's see how that works in practice and examine this simple program:

```
-10 +constant 10-
100 begin 10- dup while 5 + dup . repeat
```

First we create a '+CONSTANT', which subtracts 10 from any number on the stack. Second, we set up a loop which starts at 100 and is subsequently decremented until it hits zero. 4tH will compile this code for you:

```
[   0] literal   (100)
[   1] +literal  (-10)
[   2] dup       (0)
[   3] 0branch   (7)
[   4] +literal  (5)
[   5] dup       (0)
[   6] .         (0)
[   7] branch    (0)
```

Note that although rule 2 seems to apply to the first two instructions, it is overruled by rule 4. With reason, because otherwise the LITERAL "90" would have been compiled which is certainly *not* what we meant. You can clearly see that the expression "5 +" has been condensed to a single +LITERAL, which saves an instruction. Now let's change it slightly and see what the peephole optimizer does:

```
-10 +constant 10-
100 begin 10- dup while 5 + 10- dup . repeat
```

We've added the '+CONSTANT' to the already optimized expression. This is what 4tH compiles:

[9] Although tail optimization will still partly work.

```
[   0] literal   (100)
[   1] +literal  (-10)
[   2] dup       (0)
[   3] 0branch   (7)
[   4] +literal  (-5)
[   5] dup       (0)
[   6] .         (0)
[   7] branch    (0)
```

At first it seems like it's identical, but it's not. The '+CONSTANT' has simply been subtracted from the expression! 4tH's peephole optimizer is not there to clean up your messy code, but to give you a helping hand where you really need it, e.g. consider this code:

```
struct
  1 +field operator
  2 +field operand
end-struct /instruction       \ define a simple structure
                              \ allocate some space
/instruction string instruction
                              \ now initialize it
1 instruction -> operator c!
5 instruction -> operand  c!
```

Which 4tH compiles to:

```
[   0] literal   (1)
[   1] literal   (768)
[   2] c!        (0)
[   3] literal   (5)
[   4] literal   (769)
[   5] c!        (0)
```

Without the peephole optimizer the addresses of the members of the structure would not have been calculated at compile time. Even more, optimizing this code by hand would not have been easy without writing some pretty murky source code. It is in these situations that the peephole optimizer excels.

11.19 Static variable pointers

The peephole optimizer is an integral part of 4tH and quickly and efficiently optimizes a whole range of expressions that can be resolved at runtime, including strings and variables. E.g. if you want to set a static pointer to a classic counted string, you could define it like this:

```
16 string str-len
str-len char+ constant str-string
```

That is because a 'STRING' expression can be optimized to a single literal. However, this is not the case with a variable and consequently this pig won't fly:

```
16 array array-len            \ won't compile!
array-len cell+ constant array-payload
```

It is for this kind of situation that we have 'EQUATES'. It takes a variable and an optional literal expression and create a static pointer that behaves like any other variable, yes, even including a ':REDO' or 'DOES>' definition[10]:

[10] See section 11.11.

```
variable <literal expression> EQUATES name
```

In its most basic form an 'EQUATES' expression is equal to 'AKA'[11]:

```
variable A
A equates B
```

Is equivalent to:

```
variable A
aka A B
```

Now we can rewrite our expression so it *can* be compiled properly:

```
16 array array-len            \ this will compile!
array-len cell+ equates array-payload
```

An 'EQUATES' expression will save you from writing this kind of code:

```
16 array array-len
: array-payload array-len cell+ ;
```

Which not only requires extra code, but will definitely run much slower.

11.20 Assertions

You have probably seen this before: you've made a program, compiled it and it doesn't work. Then you start putting code at strategic places, trying to pinpoint the error. And when you're finally done, you've got to revisit all of these places to remove that code. And you probably forget a few..

4tH has a built-in facility which allows to put that code there, debug your program and remove the debugging code from your program by changing a single line.

It is called "assertion" and those of you who have ever worked with C probably know what we're talking about.

An assertion is a line of code that will evaluate an expression. If the expression evaluates to false, it will exit the program with an error message. Let's take a look at this simple colon-definition:

```
: add                         \ expects two numbers on the stack
  +
;
```

If we call add by writing:

```
1 add
```

it will fail. Now we add this assertion:

[11] See section 11.10.

```
assert( depth 2 >= )
```

It will evaluate to false when there are less than two items on the stack. The program will be terminated and the appropriate error message will be issued. You may think that this is nice, but you still have to remove all assertion manually.

Not true! If you tried this out already you will see that you won't find an assertion anywhere. It's gone! True, if you want to use assertions you have to enable them. You do that with the word '[ASSERT]':

```
[assert]
: add
  assert( depth 2 >= )
  +
;

1 add
```

Now assertions will compile and work. If you remove the word '[ASSERT]' all assertions will disappear like they were comment. '[ASSERT]' works just like '[DECIMAL]', '[HEX]', etc. They work linear and do not follow the program flow. If you put '[ASSERT]' halfway your source-file you will notice that assertions work from that point:

```
: add
  assert( depth 2 >= )        \ assertions disabled
  +
;

[assert]                      \ enable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions enabled
  . cr r> base !
;
```

Assertions are only enabled in "PRINT-HEX". The assertion inside "ADD" will be removed and thus be disabled. But there is more to '[ASSERT]' than the eye meets. It doesn't enable assertions, it toggles them. When the 4tH compiler starts, assertions are disabled. The first '[ASSERT]' enables them. A second '[ASSERT]' will disable them again:

```
[assert]                      \ enable assertions

: add
  assert( depth 2 >= )        \ assertions enabled
  +
;

[assert]                      \ disable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions disabled
  . cr r> base !
;
```

There are many possibilities:

- You can start testing low level colon-definitions and move your way up to the high level definitions by moving '[ASSERT]' down.
- You can enable assertions on certain parts of your code by enclosing them with an '[ASSERT]' pair.
- You can switch the entire context of '[ASSERT]'s by adding a single '[ASSERT]' to the top of your source.

You are not limited to range-checking when using 'ASSERT('. Any expression that evaluates to TRUE is allowed:

```
[assert]
: add
  assert( ." ADD starts at " here . cr true )
  assert( depth 2 >= )
  assert( ." Values: " over over . . cr true )
  +
;
```

We're sure you can come up with more useful ideas. We did too.

11.21 Breakpoints

4tH also offers you the possibility to set breakpoints. It's quite easy to enable this facility. Just add this to the very beginning of your source:

```
[needs lib/debug.4th]
```

Setting a breakpoint is quite easy too, e.g. this piece of code malfunctions:

```
32 string argument
1 args argument place
```

Change it to:

```
32 string argument
1 args argument ~~ place
```

Now the breakpoint is enabled. It will enter a Forth-like shell just before 'PLACE' is executed. Now a host of words are at your disposal. You can examine any region of the Character Segment with "DUMP" or print any string variable with "TYPE". 4tHs internal variables and regions are known by name, like "PAD", "TIB", ">IN", "BASE" and "OUT". You can examine them or any other variable by using "?", "@" and ".".

You have a small calculator, that you can use to multiply, substract, add. You can change 'BASE' by executing "OCTAL", "HEX", "BINARY" or "DECIMAL". It also has a host of binary operators like "OR", "AND", "XOR", "INVERT", "LSHIFT" and "RSHIFT". It also has stack operators like "DUP", "DROP", "OVER" and "SWAP". "CLEAR" will clear the stack for you.

You can examine both stacks. ".S" will show you the data stack (including any rubbish you put there yourself during the debugging session) and "R.S" will show you the return stack. "DEPTH" and "RDEPTH" will tell you how many items there are on the stack. When

you're done, you may leave the debugger by typing "BYE". Your program will continue as usual.

A word of caution: since the debugger is a 4tH program itself, it doesn't actually freeze the virtual machine. It just *seems* like it is frozen. The contents of PAD may be slightly different than you expected. If you *really* need to examine the PAD as it was, don't examine it directly, but use "SPAD" to examine the "shadow PAD". "SPAD" leaves the address for the "shadow PAD" on the stack. The same goes for ">IN", "BASE" and "OUT": never examine these by address, but always by name. Although every effort has been made to catch any errors, some extreme stress tests might fail. It is not recommended to use the debugger when stack space is very tight.

11.22 Debugging

We've all been there. You've written a non-trivial program and when you try to run it, it bombs out with an error message. Then the question arises: "Where did I go wrong?" This proces is called "debugging". Debugging is very hard without any dedicated tools. Modern development systems include a debugger, featuring breakpoints, single stepping and variable tracking.

4tH features the following tools:

- Stack examination
- Assertions[12]
- Breakpoints[13]
- Decompilation
- Preprocessor
- Compiler directives

First of all, you need to understand how the 4tH compiler works. Initially the tokenizer collects all 4tH sources and breaks them down to tokens. A token is a word or a string. This makes it much easier for the compiler to parse the source[14]. 4tH has a single pass compiler, so after the tokenizer has finished successfully, it will start compiling right away. Even if the compiler encounters an error, it has actually compiled *something*. The point where the compiler stops gives you clues on where it went wrong. 4tH doesn't discard this partial compilant, but keeps it in memory so you can examine it. It won't allow any other action.

If you managed to successfully compile the source, it will pass the compilant to the virtual machine to execute it. Again, the point where the virtual machine stops is retained. Having one external editor window open with the 4tH source and another with 4tH itself is one the most comfortable ways to debug 4tH code.

If you get the dreaded I/O error 4tH is probably unable to locate an include file. You can use the preprocessor to find out where you went wrong. The preprocessor mimics the

[12] See section 11.20.

[13] See section 11.21.

[14] The tokenizer ignores the source layout, which means 4tH is able to read almost any ASCII file, regardless format. The downside of it all is that 4tH can't tell you on which line an error occurred - simply because it doesn't tokenize or compile on a line-by-line basis.

include sequence and leaves a 4tH file you can examine. You can also place an '[ABORT]' directive after the inclusion of each file to see when the I/O error pops up.

Tokenizer errors can also be produced by improperly formatted strings, like empty strings or missing delimiters. Highlighting editors like SciTE[15] allow you to find most of these pretty quickly. If not, you have to resort to the '[ABORT]' directive again because 4tH leaves no indication where you went wrong. The easiest way to track compilation errors is by decompiling the partial compilant, e.g.:

```
: one dup ." This is the address of TWO: " . cr drop ;
: two here one ;
: three 10 0 do [char] * emit loop cr ;
: four tree two tree ;
four
```

Doesn't compile. This will tell you where it went wrong:

```
[  11] literal   (10)
[  12] literal   (0)
[  13] do        (16)
[  14] literal   (42)
[  15] emit      (0)
[  16] loop      (13)
[  17] cr        (0)
[  18] exit      (0)
[  19] branch    (0)
```

It may be a bit hard to read, but you can clearly see it went wrong shortly after the 'LOOP'. Yes, "TREE" is misspelled. If you are unable to locate the error, again '[ABORT]' can help you out - but most of the time these drastic measures are not required.

If you suspect that runtime errors may occur in a certain piece of code, you can use assertions[16] to detect them. Although a well placed breakpoint[17] may help you out, it is annoying to stop and restart the program with each interation. For that reason a simple stack dump using ".S" may be more efficient:

```
**********
37 37 (TOS) This is the address of TWO: 37
(TOS) **********
```

".S" can be found in `anstools.4th`. Of course, you can always fallback to the ancient `printf()` practice. Note that you are still required to keep the stack balanced. If you don't you're even worse off.

11.23 Running 4tH programs from the Unix shell

If you're using Unix (which we highly recommend), you can run 4tH programs right from the Unix shell. All you have to do is to add one single line at the top:

```
#! /usr/lbin/4th cxq
." Hello world!" cr
```

[15] http://www.scintilla.org/SciTE.html
[16] See section 11.20.
[17] See section 11.21.

It indicates the way you normally compile and run a 4tH program, but without the filename, e.g.:

```
/usr/lbin/4th cxq hello.4th
```

In this case, you're using the classic 4tH compiler, which is located in the /usr/lbin directory. Note that you can add options if you want. The 'cxq' options tell the compiler to silently compile and execute a program.

Note this trick only works with 4tH sources, not compiled programs. You also have to flag the 4tH source as 'executable'. You can do that by issuing this command:

```
chmod 555 hello.4th
```

Now you can simply enter:

```
hello.4th
```

at the Unix prompt and your program will be compiled and executed. Don't worry about compromising the portability of your program. It will still compile and run happily under other Operating Systems, since '#!' is an alias for '\'. It only has a special meaning to the Unix shell.

11.24 Embedding 4tH programs in a batch file

If you're running a Microsoft Operating System like Windows or DOS[18], you can embed 4tH source code in an ordinary batch file[19]. All you have to do is to make the shell ignore the 4tH code, e.g.:

```
@goto exec
." Hello world!" cr
(
:exec
@4th cxq %0.bat %1 %2 %3 %4 %5 %6 %7 %8 %9
@rem )
```

Now save your file as "EXAMPLE.BAT" in the current working directory[20] and run it:

```
example
```

Don't add the ".BAT" extension or the whole thing won't work. 4tH will now automatically pick up the batch file and execute it. Well, how does it work?

It's simple: the shell silently jumps to the "EXEC" label and executes 4tH. 4tH will compile the batch file. It ignores the line that starts with '@GOTO', since '@GOTO' is an alias for '\'. It compiles anything up to the opening parenthesis, since that is the start of a multiline comment. The shell on its turn ignores the closing parenthesis, since that has been commented out by "@REM".

[18] DOS version 3.3 or higher.

[19] This method was taken from CSL, the "C Scripting Language". You can learn more about CSL at "http://csl.sourceforge.net".

[20] If you want to store it permanently in another directory, you may have to add additional path information.

Chapter 12

Standard libraries

12.1 Adding your own library

This is a lot easier than you might think! As a matter of fact, almost any program can be turned into a specialized library. A well-written program contains a lot of definitions and only *one* executable word. Take that word away and you've got a library!

A library may contain word defintions, variables, constants, almost anything you like. And a program that includes that library will have all these definitions at its disposal. As a matter of fact, the resulting program will behave like you entered the contents of the entire library file at the position of the '[NEEDS' directive, e.g. this is an except of `constant.4th`:

```
-1 constant NULL
```

When it is included in this file:

```
\ This is a sample table using NULL

[needs lib/constant.4th]

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

It will compile to the same code as this:

```
\ This is a sample table using NULL

-1 constant NULL

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

So it is *not* a good idea to make your library files too big, since there will be a lot of superfluous code included in the compilant which 4tH will *not* dispose of automatically.

You can nest '[NEEDS' directives, so one library file may include other library files. This helps to prevent duplicate code, which can be a serious maintenance problem. You can nest them as deep as you want, available memory being the only restriction.

However, when nesting inclusions you always have the problem of multiple inclusions. Don't think that all 4tH users know by heart which library files calls which. Multiple inclusions will lead to errors, unless you take precautions. We have '[DEFINED]' and '[UNDEFINED]' to prevent that:

```
[UNDEFINED] 2drop [IF]
: 2drop drop drop ;
: 2dup over over ;
: 2swap rot >r rot r> ;
[THEN]
```

If you have included this file before, '2DROP' is already defined, so in fact all definitions are skipped when the file is included for the second time. Of course, it will take up some extra memory, but at least it won't generate any errors.

You should *not* automatically resolve library dependencies when the file included:

- Is fairly large[1];
- Is required by many other include files[2];
- Has lots of (nested) dependencies itself.

In that case you should add a check whether a certain file has been loaded and if not, abort compilation, e.g.:

```
[UNDEFINED] F+ [IF] [ABORT] [THEN]
```

If you want to port your library file, it might be a good idea to hide specific 4tH constructions, e.g.:

```
[DEFINED] 4TH# [IF]            ( a n --)
  : string! chars + 0 swap c! ;
[ELSE]                         \ make an ASCIIZ string
  : string! swap 1- c! ;       ( a n --)
[THEN]                         \ make a counted string
```

Since '4TH#' is a 4tH specific constant, it will not be defined in other Forth compilers. This way the compiler will automatically select the correct definition. Finally, you may want to hide all words which you don't want exposed to the outside program:

```
[DEFINED] 4TH# [IF]
  hide buffer
  hide /buffer
[THEN]
```

The reason for that is twofold. First, you don't want the user of your library to meddle with the internal code or variables of your library. Second, if you leave all these words in, you're cluttering the symboltable, which may lead to compilation errors if the user happens to choose a name you already used inside your library.

[1] See sections 12.24 and 13.22.
[2] See sections 12.23 and 12.24.

Finally, you sometimes find yourself in a situation where you're faced with the choice to either make two library files with *slightly* different code or resort to conditonal compilation. If you choose the latter the question remains how the user of your library can switch between both versions. For those situations 4tH offers '[PRAGMA]'. '[PRAGMA]' simply compiles a constant with a TRUE value. E.g. `compare.4th` can be compiled in a case sensitive mode by issuing the `casesensitive` pragma:

```
[pragma] casesensitive
[needs lib/compare.4th]
```

In the library file itself you can treat the pragma just like any other constant:

```
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
```

Where you place your library files is up to you. You can add them to the library files that come with 4tH, you can put them in another directory, whatever pleases you.

12.2 Adding templates

When you include a library file you add some words to your program. When you include a template you add some words to an existing program. That is the major difference between a library file and a template file. We've included a template with 4tH which allows you to create conversion program pretty quickly. The template is called "convert.4th" and it allows you to create a conversion program by defining just three words.

A standard conversion program takes an input file and creates an output file in a different format. When it can't open a file it will issue an error message, e.g.

```
Cant open input.txt
```

When you don't supply an input file and an output file, it will issue an error message e.g.:

```
Usage: myconversion input output
```

And of course, it will read and process the input file. And that's all you have to tell 4tH:

- The usage message
- How to read the file
- How to process the file

So, let's create a program that will convert a block file to a regular text file. How do we do that? First of all we've got to issue a usage message, like:

```
Usage: blk2txt blockfile textfile
```

Well, that is easy. If it comes to that we've got to abort the program, so this will do:

```
: Usage abort" Usage: blk2txt blockfile textfile" ;
```

Then we've got to read the file. A block file contains lines of 64 characters, always. So, we've got to create a buffer and read 64 characters. This will do:

```
64 string buffer
: Read-file buffer 64 accept ;
```

Finally, we've got to write the output file. Adding a 'CR' after typing the line will do, but we don't want any trailing spaces, so we need to strip those trailing spaces:

```
: Process buffer 64 -trailing type cr ;
```

Now we need to include the template and we're done:

```
[needs lib/convert.4th]
```

Wow! Do you know how much coding we need to do when we try to do this in C? This source code takes less than 256 bytes! Compile it and we're done! So how does it work? Well, the template expects us to define "Usage" and "Process". If you don't it will abort compilation:

```
\ Has Usage been defined? If not, abort!
[UNDEFINED] Usage [IF]
[ABORT] [THEN]

\ Has Process been defined?
[UNDEFINED] Process [IF]
[ABORT] [THEN]
```

If you don't define "Read-file" the template assumes that "Process" is completely self-contained, which means it will only open the files. Finally, you can optionally define "PreProcess" and "PostProcess" if you need any special actions at the beginning or the end:

```
: ProcessFile                          \ process the input file line by line
[DEFINED] PreProcess  [IF]
  PreProcess                           \ do any preprocessing
[THEN]
[DEFINED] Read-file   [IF]
  begin Read-file while Process repeat \ read file and process line or buffer
[ELSE]
  Process                              \ self contained processing
[THEN]
[DEFINED] PostProcess [IF]
  PostProcess                          \ do any postprocessing
[THEN]
;
```

If you don't define it, it won't include it. You can use such templates for many different programs, e.g. this will convert a Unix text file to an MS-DOS text file:

```
: Usage abort" Usage: udc infile outfile " ;
: Read-file refill ;
: Process 0 parse-word type 13 emit 10 emit ;
```

You can make them as sophisticated or as simple as you like. You can create other words as well, as long as those three words have been defined. Templates can be handled like any other library file. You can place them where you want, they can hold anything you want. Amaze your collegues by writing programs in a fraction of the time they should need!

12.3 Parsing the command line

The first thing that comes to mind is, of course, `getopts()`. 4tH offers a perfect equivalent of this C-function without copying it. The first thing you have to do is to include it:

```
include lib/getopts.4th
```

After that, you have to create a decision table containing the option characters and the actions that should be executed:

```
create myoptions
  char p , ' _print ,                  \ associate 'o' with _print
  char q , ' _quiet ,                  \ associate 'q' with _quiet
  char v , ' _verbose ,                \ associate 'v' with _verbose
  char f , ' _file ,                   \ associate 'f' with _file
  NULL ,
```

The stack diagram for an "option word" is very simple: assume nothing is on the data stack and leave nothing behind. If an option requires an argument, you can retrieve it by issuing "GET-ARGUMENT". Note these words have to be defined *before* the decision table:

```
256 string myfile                      \ allocate space for filename

false value (print)                    \ 'print' is off
false value (quiet)                    \ 'quiet' is off
false value (verbose)                  \ 'verbose' is off

: _print true to (print) ;             \ enable 'print'
: _quiet true to (quiet);              \ enable 'quiet'
: _verbose true to (verbose) ;         \ enable 'verbose'
: _file get-argument myfile place ;    \ set the filename
```

"GET-ARGUMENT" leaves an address/count string on the stack, which *has* to be consumed before returning. Finally, the decision table has to be passed to "GET-OPTIONS":

```
myoptions get-options
```

When "GET-OPTIONS" returns, it has also set a value called "OPTION-INDEX". This value is an index to the first non-option command line argument. Note that the 4tH program itself has an index of 0, so the first "real" command line argument has index 1. If "OPTION-INDEX" equals 'ARGN', there are no non-option command line arguments.

You can pass "OPTION-INDEX" to "ARG-OPEN", provided you've included it, e.g.:

```
include lib/getopts.4th
include lib/argopen.4th
( options table and option words)

myoptions get-options
option-index dup 2 + argn >
abort" Missing filename"

input over arg-open
output swap 1+ arg-open
```

You can also use `convert.4th` to handle all the details for you, but you *have to* name your decision table "OPTIONS" in order to make it work.

12.4 Mixing character and number data

Sometimes you have to mix character and number data, e.g. when you're porting a Forth program or when the need complex datastructures arises. Since 4tH gives each datatype its own segment this is not easy. However, there is a library that can help you. Let's have a look at this program:

```
16 constant /my                 \ size of array
/my array my                    \ define array
0                               \ set up counter
begin
  dup dup                       \ duplicate counter
  cells my + !                  \ store counter in array
  1+                            \ increment counter
  dup /my =                     \ limit reached?
until drop                      \ drop the counter

my                              \ set up index
begin
  dup @ . cr                    \ print the value
  cell+                         \ next element
  dup my /my cells + =          \ limit reached
until drop                      \ drop the index
```

This simple program defines a small array, fills and displays it. Now, this little thing does the same thing, but is located in the Character Segment:

```
include lib/ncoding.4th
                                \ load the library
16 constant /my                 \ size of array
/my nells string my             \ define array
0                               \ set up counter
begin
  dup dup                       \ duplicate counter
  nells my + n!                 \ store counter in array
  1+                            \ increment counter
  dup /my =                     \ limit reached?
until drop                      \ drop the counter

my                              \ set up index
begin
  dup n@ . cr                   \ print the value
  nell+                         \ next element
  dup my /my nells + =          \ limit reached
until drop                      \ drop the index
```

You see that the code is *very* similar. The 'STRING' declaration clearly indicates that the array is allocated in the Character Segment. But as you can see it is not an array of *cells*, but an array of *nells*. 'NELL' holds the size of a single nell, so we multiply it by the number of nells we want to get the proper size of the array. After that, it is just replacing the Integer Segment words with nell equivalents:

Note that although you can replace every cell with a nell, you *do* pay a penalty in execution speed, so use with caution.

NELL	CELL
/nell	/cell
nells	cells
n@	@
n!	!
nell+	cell+
nell-	cell-

Table 12.1: NELL equivalents

12.5 Dynamic memory allocation

If you don't know what this is, you probably shouldn't bother. Sometimes you don't know how much memory you will actually need, sometimes you know how much you need, but you won't need it during the entire execution of the program. In these cases, you can temporarily allocate a chunk of memory and release it when you no longer need it.

4tH has similar facilities. E.g. if you want to allocate 600 bytes, you simply include `ansmem.4th` and allocate it:

```
[needs lib/ansmem.4th]
600 allocate
```

'ALLOCATE' leaves two items on the stack. The first one is a flag. If it is true, memory allocation has failed, so we can easily add some error checking to our little program:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
```

It it returns false, memory has been allocated. Its address is the second item on the stack. You can pretty much do what you want with it, but remember that memory is usually allocated in the Character Segment, so if you want to store numbers as well over there, read section 12.4 again. Anyway, this is completely valid:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
s" Hello temporary world!" rot place
```

Let's change that one a little bit to prove we've actually stored anything:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
>r s" Hello temporary world!" \ Let's save the address
r@ place                      \ Now store the string
r> count type cr              \ Let's print the string
```

Let's allocate another 100 bytes and free all memory afterwards:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string
```

```
100 allocate
abort" Out of memory" >r      \ Second allocation
s" I'm a little crammed!"
r@ place                      \ Store another string
r@ count type cr              \ Let's print the string

r> free
abort" Cannot free memory"    \ Now free the first block
r> free
abort" Cannot free memory"    \ Now free the second block
```

Yes, that's right: you feed 'FREE' the address that 'ALLOCATE' returned and it returns a flag. If it is a true flag, an error occurred; if not, everything is hunky dory. Let's try to free it twice:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r@ free
abort" First attempt"         \ Now let's free the block
r> free
abort" Second attempt"        \ And try to free it again..
```

Yes, now 4tH terminated with the error message "Second attempt". You can not free a block twice..! But you can reallocate it if you happen to change your mind. You can increase or decrease its size, without losing any data. When the new block is too small to hold all the data, the data is truncated. Let's see it in action:

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r> 100 resize
abort" Out of memory" >r      \ Now resize the block
r@ count type cr              \ Here is your string again

r> free
abort" Cannot free memory"    \ Now free it
```

You'll see that your precious string is still alright. Apart from a flag, 'RESIZE' also returns the address of the reallocated block. If 'RESIZE' fails, *your original data is still alright*, so in some circumstances you might want to save the old address.

12.6 Tweaking dynamic memory

You might find that 4tH doesn't reserve much memory for dynamic allocation. Dynamic memory is allocated on the heap, which is 16 kB. You can increase it, but first you have to know how dynamic memory works. You can determine how much memory has been allocated by using the word 'ALLOCATED':

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

And it will print something like:

```
768 allocates 64 bytes.
```

64 bytes? I thought we allocated 50 bytes! Let's try another one:

```
[needs lib/ansmem.4th]
500 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

This time it prints something like:

```
768 allocates 512 bytes.
```

As a matter of fact, 'ALLOCATED' will *always* return multiples of 32 bytes. That is a consequence of how 4tH handles dynamic memory. 4tH divides dynamic memory into *paragraphs*. When you allocate memory, 4tH allocates as much paragraphs as it needs to provide you with the memory you requested. Then these paragraphs are marked as 'taken'. This marking is done in the Heap Allocation Table, which is located in the Integer Segment. Every paragraph is represented by a cell in the HAT.

You can fine-tune this mechanism by defining some constants before including `ansmem.4th`. This will create a heap with 512 paragraphs of 256 bytes, which is 128 kB:

```
512 constant #paragraph       \ 512 paragraphs
256 constant /paragraph       \ each paragraph is 256 bytes
[needs lib/ansmem.4th]
500 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

Try to keep the number of paragraphs *low*. 1024 seems like a nice upper limit. If you need that much memory, it is *much* better to handle it in larger chunks. This avoids fragmentation and keeps the time to search the HAT within acceptable limits.

If you're using dynamic memory to manage e.g. linked lists, you can force a heap of cells by defining the pragma `forcecellheap` *before* you include `ansmem.4th`. You can even use the node structure definition to make it match a single paragraph, e.g.:

```
[pragma] forcecellheap        \ force a CELL heap

struct                        \ define the node
  field: kind
  field: o1
  field: o2
  field: o3
  field: val
end-struct /paragraph         \ name it "/paragraph"

include lib/ansmem.4th        \ leave the library to it
```

You can now be assured that the size of a paragraph equals a single node. Neat, huh?

12.7 Using two heaps

Due to 4tH's Harvard architecture[3], you cannot mix cells and characters. Of course, you can use a library to store numeric information[4], but sometimes that is not sufficient. Fortunately, 4tH offers another library that allows you to have *two* heaps, one for your cells and one for your strings. Well, your first question probably is: how can I differentiate between the two? It's easy: if you combine the two you allocate cells with "ALLOCATE", "FREE", "RESIZE", just as usual. If you want to allocate strings, you use "CALLOCATE", "CFREE" and "CRESIZE".

In order to achieve that you have to include the cell library *before* the string library:

```
include lib/memcell.4th
include lib/memchar.4th
```

If you don't, `memchar.4th` will default to your usual "ALLOCATE", "FREE" and "RESIZE" words. You can also determine the size of both heaps by setting a constant before including them. The size is in address units:

```
2048 constant /heap           \ a 2K cell heap
include lib/memcell.4th
8192 constant /heap           \ an 8K string heap
include lib/memchar.4th
```

Note the pragma `forcecellheap` is automatically set when you include `memcell.4th`, so you can use that one in your programs as well. Both libraries are drop-in replacements of `ansmem.4th` (for their respective datatype). So why two different implementations? It largely depends what your program needs. If you require:

- A small implementation;
- Fairly robust;
- A few large allocations or a limited number of allocations of a similar size.

You'll be well off with `ansmem.4th`. However, if you require:

- An unknown number of allocations of different sizes;
- A combination of a cell heap and a string heap.

You'll be better off with `memcell.4th` and `memchar.4th`.

[3] http://en.wikipedia.org/wiki/Harvard_architecture
[4] See section 12.4.

12.8 Garbage collection

4tH features a simple form of garbage collection. The only thing between ”real” garbage collection and this implementation is that you remain in control at all times. First you have to decide what kind of datatype you want to put under control of the garbage collector by defining a pragma. This will enable cell garbage collection:

```
[pragma] forcecellgc
include lib/gc.4th
```

If you don’t define this pragma, garbage collection will default to characters. Since the garbage collector uses the same libraries we discussed in section 12.7, you can size them with ”/HEAP”. Next, you have to start garbage collection:

```
start-gc
```

It’s time to start a ”scope”:

```
start-scope >r
```

This will return a single cell scope token which is used to keep track of when to do garbage collection. Now the fun starts: allocate and resize away all you like, but instead of using the usual allocation words, you use these:

```
50 gc-alloc >r
s" Hello temporary world!"
r@ place                        \ Now store the string
r@ count type cr                \ Let’s print the string

r> 100 gc-resize
count type cr                   \ Here is your string again
```

Wow, did you see that?! We lost a pointer in the last line! Well, that isn’t too bad in this case - we’re using a garbage collector after all. Note you may open up new scopes if you like, but you have to close them in the right order, so you may want to use the data stack to keep track of your scope tokens, like we did.

Closing a scope is easy. Just pass the scope token to ”END-SCOPE” and *all* memory you allocated since the last ”START-SCOPE” will be automatically freed:

```
r> end-scope
```

When you’ve ended a scope, the garbage collection will automatically revert to the previous scope, so any allocation you do *after* closing a scope will be handled by the *previous* scope. Finally, when you’re done you can shutdown the entire garbage collector by issuing:

```
end-gc
```

This will free up all garbage collection support structures, which are also dynamically allocated. Famous last words: do *not* free or resize memory allocated with ”GC- ALLOC” or ”GC-RESIZE” with the standard words ”FREE” or ”RESIZE”. Ignore this warning at your own peril.

12.9 Application stacks

Did you ever feel like a second return stack would be nice? Well, you can. As a matter of fact you can have several dedicated stacks. It's quite easy to use:

```
[needs lib/stack.4th]

16 array mystack              \ allocate some space
mystack stack                 \ convert it into a stack

234 mystack >a                \ push 234 on the stack
456 mystack >a                \ push 456 on the stack
mystack a@ . cr               \ examine top of stack
mystack a>                    \ pop 456 from the stack
mystack a>                    \ pop 234 from the stack
. . cr                        \ show the values
```

Note there is a floating point[5] version as well, called `fstack.4th`. It works in a similar way. Wouldn't it be nice to have a string stack too? Yes, 4tH provides that one too! It works the same way:

```
[needs lib/stsstack.4th]

1024 constant /mystack
string mystack                \ allocate some space
mystack /mystack string-stack \ convert it to a string stack

s" Hello" mystack >s          \ push string 'Hello' on stack
s" World" mystack >s          \ push string 'World' on stack
mystack s@ type cr            \ examine top of stack
mystack s>                    \ pop 'World' from the stack
mystack s>                    \ pop 'Hello' from the stack
type cr type cr               \ show the values
```

But if you only need a single stack - but one you can monitor - there is another library member:

```
2048 constant /cstack
[needs lib/stmstack.4th]

s.clear                       \ clear the string stack

s" Hello" >s                  \ push string 'Hello' on stack
s" World" >s                  \ push string 'World' on stack
s@ type cr                    \ examine top of stack
s>                            \ pop 'World' from the stack
s>                            \ pop 'Hello' from the stack
type cr type cr               \ show the values
s.error CSE.NOERRS = . cr     \ show the exit status
```

You can set the size of the string stack by defining "/CSTACK" before including it. "S.CLEAR" clears the stack and resets "S.ERROR". "S.ERROR" is simply a value you can query or set according to your needs. If you cause an underflow or an overflow, the integrity of the stack is maintained.

Note there is a catch: when you've popped a string from the string stack, the string itself is untouched, so the address-count pair is still valid. However, if you push another string onto the same stack, the popped string is clobbered - so a kind of 'SWAP' is out of the question. There is another way to create one (or more) string stack without these disadvantages, called `strstack.4th`, but it is larger and slower. It works the same way as the first library member we presented.

[5] See sections 12.24 and 12.23.

12.10 Local variables

I have to admit something: I *really* don't like local variables. They offer a kind of "random access" that goes totally against everything the stack-based Forth language stands for. And they don't come for free - you pay a performance penalty for setting them up and setting them free. That's why 4tH doesn't offer any native support.

However, sometimes you just don't want to rewrite a routine, but just use it. And then it's handy when *some* support is there. So here we are. Let's start with an example from gForth:

```
: my.max { n1 n2 -- n3 }
  n1 n2 > if
    n1
  else
    n2
  endif ;
```

So, what's happening here? First you see something that *resembles* a stack diagram - which it isn't: it's executable code. Note it uses accolades, *not* parentheses. Now, what does it do? It assigns the top of the stack to "n2" and the second on stack to "n1". That's pretty neat, because if it wouldn't you'd have to read the whole thing backwards - this *really* follows the stack diagram. Everything after "–" is comment, though.

When the first accolade is executed it builds a frame of local variables, so yes, it also works with recursion. The rest you can read as if this were executed:

```
0 value n1                    \ define a value n1
0 value n2                    \ define a value n2

: my.max ( n1 n2 -- n3 )      \ no accolades here
  to n2 to n1                 \ assign TOS to n2
  n1 n2 > if                  \ assign 2OS to n1
    n1                        \ compare them
  else                        \ and then select
    n2                        \ either n1 or n2
  then ;
```

We already told you that a local frame has to be released - and it doesn't. So what's happening here? Well, gForth *modified* the execution semantics of ';', so it *secretly* checks whether locals were used and released them if they did. Every locals scheme I've seen does it. In 4tH, you can't do that, because the compiler won't let you. The consequence is, you'll have to do it yourself. Now let's make a 4tH version out of this:

```
include lib/locals.4th

: my.max   ( n1:A n2:B -- n)
  2 locals
    :A @ :B @ > if
      :A
    else
      :B
    then @
  end-locals ;
```

This does the very same thing. First, there are no accolades. Second, you only define there are two locals - and that's it. 4tH locals *always* bear the same name. You can't choose them. Third, they are variables, not values. Fourth, you are the master of the frame - not the compiler.

When you issue ”LOCALS”, it takes a value from the stack and allocates a locals frame of that size (in this case a frame of 2 local variables). Then it takes *that same number* of values from the stack and places them into the frame. So let’s call it:

```
1 10 my.max . cr
```

It works just like the gForth version, so ”1” ends up in ”:A” and ”10” ends up in ”:B” (hint: that’s also what the stack diagram tells you). The rest is pretty much like the gForth version - except that ”:A” and ”:B” are *variables* of course. And what if you need more local variables? Well, you can go up to *eight*, from ”:A” to ”:H”.

Finally, we reach the end of our word and there it is, the word that releases the frame, ”END-LOCALS”. If you fail to use it, the local variables stay *in scope*, so they remain valid in the caller - and believe me: that’s not what you want. If you leave a word prematurely by using ’EXIT’, you’ll have to terminate the scope manually as well, so instead of e.g.:

```
if exit then
```

You’ll have to write:

```
if end-locals exit then
```

Don’t forget! This way you can do almost anything other local variable schemes can do, e.g. take this one, also from gForth:

```
: strcmp { addr1 u1 addr2 u2 -- n }
  addr1 addr2
  u1 u2 min 0
  ?do { s1 s2 }
    s1 c@ s2 c@ -
    ?dup-if
      unloop exit
    then
    s1 char+ s2 char+
  loop
  2drop
  u1 u2 - ;
```

This one introduces a new scope *within* a **?DO..LOOP**. So how do we tackle this one? Well, like this:

```
: strcmp   ( a1:A n1:B a2:C n2:D -- n )
  4 locals
    :A @ :C @
    :B @ :D @ min 0
    ?do
      2 locals
        :A @ c@ :B @ c@ - dup
        if                    \ we don’t have ?DUP
          end-locals          \ discard LOOP locals
          unloop              \ discard LOOP parameters
          end-locals          \ discard strcmp locals
          exit                \ exit the routine
        else                  \ we don’t have ?DUP
          drop                \ so we have to drop it
        then
        :A @ char+ :B @ char+
      end-locals              \ free LOOP locals
    loop
    2drop
    :B @ :D @ -
  end-locals ;
```

True, it's much more "wordy", but it does the same thing word for word. It's just not that sneaky. First both strings (in ANS Forth fashion) are assigned to locals variables. Since both strings need two values on the stack, we need 4 local variables. The top of the stack (the count of the second string) is put into ":A", the address of the second string ends up in ":B", the count of the first string ends up in ":C" and finally the address of the first string ends up in ":D". No surprises here.

You may also follow how they are used. The length of the smallest string is the loop parameter of the **?DO..LOOP**. After that follow the addresses of both strings. When the **?DO..LOOP** is entered, a new stack frame is created for these addresses. Consequently, when the loop is terminated, the scope has to be terminated as well. That's why you find an "END-LOCALS" just before 'LOOP'. Of course, gForth does this automatically.

And here is the catch: in the middle of the loop we leave the word *entirely* and return to the caller. At that moment we not only have to deal with the scope of the word, but the scope of the loop as well. So first we release the locals frame of the loop (just before 'UNLOOP') and then the locals frame of the word (just before 'EXIT'). Like I said, gForth does this as well - you just don't see it.

There is just one more catch: locals are not taken into consideration when you use 'CATCH'[6]. The fix is easy, though. There is a word called "LFRAME", which is a pointer to the current locals frame. So if you want to preserve the sanity of your locals subsystem, you simply do this:

```
lframe >r catch r> over       \ duplicate CATCH value
if to lframe else drop then   \ CATCH value still on stack
```

What this does is save the address of your locals frame on the return stack (where it is supposed to be save) and restore it after an exception has been caught. If there was no exception at all we can simply discard it. You can even wrap the whole thing in a definition if you want - and be done with it.

It is the most "userfriendly" locals scheme? Probably not, but it is just as versatile and a real life saver if you need it. And - believe it or not - completely portable to gForth and other Forth compilers.

Note that there is a floating point[7] version as well, called `flocals.4th`. It works in a similar way.

12.11 Random numbers

If you want to program a game or a simulation, you'll probably need random number generation. Of course, you can do that too with 4tH. Just include the proper file:

```
include lib/random.4th
```

And initialize it:

```
randomize
```

[6] See section 11.5.
[7] See sections 12.24 and 12.23.

It generates a number between 0 and ”MAX-RAND”, but we’ll teach you how to generate a number for virtually any range below that.

There are two things important when you want to do that: the range limit and the lower limit. Say, we want to simulate a dice with numbers in a range from 1 to 6. The lower limit is 1. We subtract that from the upper limit to get the range limit.

So: upper limit minus lower limit gives a range limit of five (6 - 1 = 5). The general formula looks like this:

*<range-limit> 1+ random * max-rand 1+ / <lower limit> +*

When we apply this to the dice-example, the complete formula is:

```
5 1+ random * max-rand 1+ / 1 +
```

This will give you a dice-simulation, that produces random numbers between 1 and 6. If you don’t like to write this code yourself, you can always include `choose.4th`. After including that you can simply get a predetermined range of number by simply writing:

```
include lib/choose.4th
6 choose 1+
```

Which is in fact your perfect dice simulation, because it returns a number between 1 and 6. ”CHOOSE” in itself provides a number between 0 and 5 in this example.

Sometimes you want the randomizer to provide a preset range of numbers. There is an variable called ”SEED” which determines what range of numbers is provided. ”RANDOMIZE” initializes it with the setting of the internal clock, which is the same value that ’TIME’ returns. Yes, that means that if you call ”RANDOMIZE” twice within a second, the randomizer will return the same range of numbers. Presetting the randomizer is pretty trivial:

```
1234567890 seed !
```

Finally, under the hood it is not a single randomizer, but two! Default a modified[8] BSD algorithm is used, but you can also default to the standard Micrsoft algorithm by defining the appropriate pragma:

```
[pragma] MS-random
include choose.4th
```

You can also select the required version by calling it by name:

```
MSRandom . BSDRandom . cr
```

Are these randomizers very good? No, as a matter of fact, they are not. They are sufficient to power your occasional game, but for more serious applications they simply fall short. That’s why 4tH also provides some very heavy duty randomizers, designed by randomizing expert George Marsaglia[9]. The first is `gmkiss.4th`. Note that although it provides a similar interface as `random.4th`, it is *not* a drop-in replacement, most significantly because it can return the full range of a cell value, including negative numbers. Another difference is that it takes *four* cell size values to initialize:

RANDOMIZER	REGISTERS	TYPE
CONG	x	congruential
SHR3	y	3-shift shift register
MWC	z, w	multiply with carry

Table 12.2: gmkiss randomizers and their registers

```
1234567890 987654321 12893467 1029384756 seed4!
```

As a matter of fact, it is built out of *three different* randomizers, which feed a fourth one. You can call each and every one of these randomizers and set their registers individually.

All those randomizers combined results in the KISS random number generator, which is the randomizer that "RANDOM" calls[10].

If that still isn't enough we got even a better one, `gmskiss.4th`, which is arguably one of the best randomizers in the world. This one uses a large amount of memory, so don't use it for trivial tasks. It can return the full range of a cell value (including negative numbers) and takes three cell size values to initialize:

```
include lib/gmskiss.4th
987654321 12893467 1029384756 seed3!
." Random number: " kiss . cr
```

If you don't want to set the seed yourself, you can always call "RANDOMIZE" instead. This "super KISS" implementation differs from the previous one because the "multiply with carry" generator has now superlong periods ($54767 * 2^{1337279}$).

But hold, that is not everything. There are a *ton* of different randomizers in 4tH. Table 12.3 gives you an overview. Some of them comply by the standard API[11], some of them are so specialized that they don't. So whatever your needs are, there is always a randomizer that fits them.

12.12 Sorting

Yes, 4tH can do that too. You just have to include `qsort.4th` (a "quick sort" implementation) to make it all possible. It works pretty much like the sort routines you've seen in C, which means you have to devise a word to compare two values. Note that `qsort.4th` can only sort *cell arrays*. Setting it up is pretty simple. First you have to include it:

```
include lib/qsort.4th
include lib/random.4th
```

Then you have to create a word that returns a true flag when the second value on the stack is smaller than the top of the stack. In this example we will just compare two integers, so that is pretty easy:

```
: MyPrecedes < ;
```

[8] The standard BSD algorithm is notoriously flawed, especially the lower bytes of the returned random number.
[9] http://en.wikipedia.org/wiki/George_Marsaglia
[10] Unless you've included `random.4th` as well.
[11] They provide `random`, `randomize` and `MAX-RAND`.

`qsort.4th` creates a deferred word[12] called "PRECEDES". Now we have to assign our word to "PRECEDES", so that it is executed when "PRECEDES" is called:

```
' MyPrecedes is Precedes
```

That's it! We're ready to rock 'n roll now. Let's set up a simple testing environment:

```
10 constant #elements
#elements array elements

randomize

: InitElements #elements 0 do random elements i th ! loop ;
: ShowElements #elements 0 do elements i th @ . loop cr ;
```

This creates an array of ten elements, which is filled with random values by "InitElements". "ShowElements" will show on screen what is stored there. The actual sort is straightforward: tell "SORT" which array and how many elements there are to sort and you're done:

```
: SortElements elements #elements sort ;
```

Now let's put it all together:

```
InitElements
ShowElements
SortElements
ShowElements
```

It will initialize the array, show its contents, sort it and show it again. It will output something like this:

```
12717 6028 1389 31870 14234 15884 31062 14788 18186 149
149 1389 6028 12717 14234 14788 15884 18186 31062 31870
```

And what if these were string addresses? Well, "SORT" would have sorted them too, from the lowest addresses up to the highest addresses, but that's probably not what you meant. You wanted to sort the actual strings, not just their addresses. Can 4tH do that too? Sure, you just got to create another "PRECEDES" word. Something like this:

```
: SPrecedes >R COUNT R> COUNT COMPARE 0< ;
```

This will take the two values and treat them as strings. Now the strings are sorted, not just the addresses. Note that the strings themselves will not move in memory. The *pointers* move, the strings themselves don't. This is a so-called "address based sort".

The 4tH library contains several sorting algorithms. There are two families: "index based" and "address based". The first family can sort *any* array[13], the second one only cell arrays, because fetching and exchanging is done within the library member itself.

Another difference is that the "index based" family allows you to combine several different sorting algorithms within the same program, but also *requires* you to define your own element exchange word. You can also combine *one* member of the "address based" family with the "index based" family, as long as you handle your "PRECEDES" definitions properly since they are not compatible.

Apart from their ”family” characteristics each and every algorithm has its own strengths and weaknesses. The larger library members are usually faster, but the table gives you more accurate timings. The factor ”Speed” gives you an indication of how fast they are. 1.0 is the reference speed, higher numbers are proportionally slower[14]. The same for the ”Size” factor - higher numbers are proportionally bigger.

Which one is optimal for your specific problem depends on a lot of things, e.g. the state and condition of the data (like partially sorted sequences, duplicates) are important factors in choosing the right sorting algorithm. There more than adequate resources for you on the web to find out, but here are a few ”rules of the thumb”:

- If you have only a few records to sort, take Simple sort (improved) or Selection sort;
- If you have a significant number of records, need a reasonable, consistent performance and a compact routine, take Circle sort;
- If you have a significant number of records and need a good performance, take Shell sort or Comb sort;
- If you have a large number of records and absolutely need the fastest performance, take Quick sort or Intro sort;
- If you have a large number of records and need a very consistent, good performance, take Intro sort or Heap sort.

If you’re still unsure and size or speed isn’t a major issue, you can’t go wrong with Heap sort.

12.13 Bitfields

Bitfields are always a bit of a controversial subject, since many complain that they are not portable or simply not worth the effort. Anyway, there is a library included if you should ever need them. In its most basic form they are quite simple: you provide the number and a certain number of bits are extracted from a certain position:

```
include lib/bitfield.4th      \ include the library

[binary] 00011000 [decimal]   \ put a number on the stack
3 2 bn@ .                     \ start at bit 3, 2 bits long
```

This will return ”3” - or ”11” binary, if you prefer. Counting starts at the rightmost bit (which is bit 0) so this expression returns bits 3 and 4. Saving it is very easy as well, just add a new value:

```
include lib/bitfield.4th      \ include the library
                              \ store ”2” into bit 3 and 4
2 [binary] 00011000 [decimal] 3 2 bn!
```

Of course, you can also store a pattern into a string or cell variable:

[12]See section 11.7.
[13]Since it works with indexes, not addresses. Consequently, it is a bit slower in general.
[14]For a 100,000 element cell array

```
include lib/bitfield.4th      \ include the library

variable myvar                \ define a variable

0 myvar !                     \ initialize it
3 myvar 3 2 b!                \ store "3" into bit 3 and 4
myvar 3 2 b@ . cr             \ print the bitfield
```

Its character counterparts are named "BC@" and "BC!", which may not be all that surprising.

Some may argue that these are not true bitfields, because you always have to provide the starting bit and the number of bits. With a bit of effort though we can make these bitfields behave like the real thing. First, include `constant.4th` in addition to the bitfield library:

```
include lib/bitfield.4th      \ include the library
include lib/constant.4th      \ include "cell-bits"
```

Let's define the fields in the usual "starting bit, length in bits" form and add the expression "CELL-BITS * +". After that, turn the whole thing into a constant. Just make sure the bitfields don't overlap:

```
0 3 cell-bits * + constant ink    does> cell-bits /mod ;
3 3 cell-bits * + constant paper  does> cell-bits /mod ;
6 1 cell-bits * + constant bright does> cell-bits /mod ;
7 1 cell-bits * + constant flash  does> cell-bits /mod ;
```

This will scale the bitfield "length" parameter in such a way, that it stays arithmetically separated from the "starting bit" parameter. Finally, wrap it into a simple 'DOES>' definition. This will split the constant into the two components again when it's used. Now let's use it in a real world example:

```
0 enum black  enum     blue   \ enumerate all colors
  enum red    enum     magenta
  enum green  enum     cyan
  enum yellow constant white

24 constant /lines            \ length of the display
32 constant /rows             \ width of the display
                              \ allocate display buffer
/lines /rows * buffer: attributes
  does> swap rot /rows * + chars + ;
                              \ make it a two dim. array
white 5 6 attributes -> ink    bc!
  red 5 6 attributes -> paper  bc!
 true 5 6 attributes -> bright bc!
false 5 6 attributes -> flash  bc!
                              \ set colors at line 5, row 6
```

This program will set the colors in the display buffer of a Sinclair ZX Spectrum. As you can see for yourself, the bitfields behave as you would expect. Only defining them is a little clumsier - nothing a preprocessor can't fix.

12.14 Bit arrays

A bit array (also known as a bitmap, a bitset, or a bitstring) is an array data structure that compactly stores individual bits (boolean values). Setting it up is a bit awkward, but using it is very simple. First we have to include the libary:

```
include lib/bitarray.4th
```

Then we have to set up the bit array itself. The *minimal* size of a bit array is one cell. The *total* size of a bit array has to be a multiple of cells. Although you can use cells to size the array, the most portable way is to use bits (128 in this example, to be exact):

```
128 cell-bits / array mybits
```

Then we have to declare the execution semantics. Note from that moment on, the array is no longer accessable as a cell array. So if you want to initialize the array, the easiest way is to do it *before* that, otherwise you'll have to do it bit by bit:

```
:redo mybits bit-array ;
```

Or even shorter:

```
128 cell-bits / array mybits does> bit-array ;
```

Believe it or not, but that was the hardest part! Manipulating a bit array is dead easy. Bits are numbered zero and up. To set a bit simply issue:

```
0 mybits bit-on
```

Resetting or toggling a bit is no rocket science either:

```
34 mybits bit-off
63 mybits bit-toggle
```

Of course, you can query the status of a bit. If the bit is set it returns true, otherwise false:

```
34 mybits bit?
```

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values. However, bit arrays can also be used for the allocation of memory pages, inodes, disk sectors, etc. In such cases, the term "bitmap" may be used[15].

12.15 Associative arrays (using hash tables)

Associative arrays have been made popular by languages such as Python and PHP. In general they are not very efficient, but make it easy to associate keys with their respective values. 4tH does not have native support for them, but if you want them, they are there.

The first one is small, but limited. First you need to include `hash.4th`, because this implementation uses hash tables:

```
include lib/hash.4th
```

[15]Not to be confused with raster images, which is something entirely different.

Second, you allocate an integer array in which you want to store the keys and initialize it:

```
16 constant /myhash
/myhash array myhash
myhash /myhash hashtable
```

Third, you need to select the hashing algorithm you want to use. ”SDBM” is a decent one, but you can use another one if you want or even write your own:

```
' sdbm is hash
```

Now we’re ready to rock ’n roll! Before we save a value, it is wise to see if its key is free:

```
s" beta" myhash get?
```

”GET?” returns two values. First a flag. If it is non-zero, the key is free. The second one is the value associated with the key. When you’re testing (like now) it is useless, so we can drop it:

```
0= abort" Key taken!" drop
```

Now we can do some business:

```
512 s" beta" myhash put
```

And if we want to retrieve the value we simply write:

```
s" beta" myhash get . cr
```

Well, it’s small, simple and painless, but there is one little catch. What will you do when a collision[16] occurs? Well, frankly that’s up to you. The program does not define any alternative action. You could try to increase the size of the hash table - but not while the program is still running of course. You could also define an alternative hash table, but does that really make life easier?

At least you’ve learned two lessons:

1. Hash tables are not very memory efficient;
2. Hash tables need to take alternative action when a collision occurs.

Of course, there is a version which addresses the latter problem - and uses even *more* memory. In order to resolve a collision, memory is divided into buckets. A ”bucket” is a certain number of associated memory locations where the values of keys with the same hashes are stored. E.g. of you make buckets of eight, you can survive seven collisions before the program gives up. Buckets are four entries deep by default, but you can override that:

```
3 constant /bucket
include lib/hashbuck.4th
```

[16] A ”collision” means a key is already taken.

Now we have to allocate the hash table. The best strategy is to provide space for *many* keys with *small* buckets. In this example we provide space for fifteen keys:

```
include lib/hash.4th
15 /bucket 2 * * 1+ constant /myhash
/myhash array myhash
```

Note that every entry in a bucket takes *two* cells and *one* extra cell is needed for the hash table itself. In this example, we have room for fifteen keys, each having buckets of three entries, each entry taking two cells - and one cell overhead. The rest looks familiar:

```
myhash /myhash hashtable
' sdbm is hash
512 s" beta" myhash put
s" beta" myhash get . cr
```

Note we don't have to test the key ourselves, the program itself transparently handles any collisions. And what when we are out of buckets? Well, the program will issue an error message and quit. If you want to see for yourself if a bucket was full, you need to use "PUT? which returns a non-zero flag if a bucket was full:

```
512 s" beta" myhash put? abort" I want to issue my own!"
```

Finally, there is a version that has *none* of these drawbacks. It is more memory efficient and you don't have to worry about collisions or buckets. But as always, there is a price to pay: it is slower, since it looks up the keys with a binary search. The overhead is negligible, just two cells. So this:

```
include lib/hashkey.4th
include lib/hash.4th

32 constant /myhash
/myhash array myhash
```

Will get you 15 solid key and associated value pairs. Again, the rest looks familiar:

```
myhash /myhash hashtable
' sdbm is hash
512 s" beta" myhash put
s" beta" myhash get . cr
```

You want to store strings instead of numbers? You can do that too if you use `userpad.4th`. This library member stores strings in a circular buffer, much like the internal PAD[17]. After that, storing strings is easy. You only need to define these words:

```
: sput >r 2>r >pad drop 2r> r> put ;
: sget get error? abort" Bad bucket" count ;
```

And this is where it all comes together:

```
s" This is the end" s" beta" myhash sput
s" beta" myhash sget type cr
```

But is it worth all the trouble? In some circumstances where quick access to large tables is required, may be, but you be the judge of that.

[17] Which means they *will* get overwritten at some point.

12.16 Associative arrays (using binary trees)

Sure, it takes much more code and requires dynamic memory, but it has none of the drawbacks associated with hashtable based implementations. Just include the library, define a variable and initialize it:

```
include lib/bintrkv.4th
variable kv
NULL kv !
```

And off you go!

```
15 5 kv put
100 6 kv put
5 kv get .
6 kv get . cr
```

Of course you can define additional arrays or reassign a value:

```
variable mykv
NULL mykv !

15 5 mykv put
16 5 kv put
```

You can also use strings as key values:

```
25 s" alpha" kv put$
35 s" beta" kv put$

s" alpha" kv get$ .
s" beta" kv get$ . cr
```

Don't worry about cleaning up dynamic memory:

```
kv @ destroy-tree
```

That will take care of all that, but of course you have to do it for each and every associative array you created.

12.17 Lookup tables with integer keys

No CASE construct, huh? Now how are we supposed to make those complex decisions? Well, do it the proper way. Leo Brodie wrote: "I consider the case statement an elegant solution to a misguided problem: attempting an algorithmic expression of what is more aptly described in a decision table". And that is exactly what we are going to teach you.

Let's say we want a routine that takes a number and then prints the appropriate month. In ANS-Forth, you could do that this way:

```
: Get-Month
  case
     1 of ."  January " endof
     2 of ." February " endof
     3 of ."   March  " endof
     4 of ."   April  " endof
     5 of ."    May   " endof
     6 of ."   June   " endof
     7 of ."   July   " endof
     8 of ."  August  " endof
     9 of ." September" endof
    10 of ."  October " endof
    11 of ." November " endof
    12 of ." December " endof
  endcase
  cr
;
```

This takes a lot of code and a lot of comparing. In this case (little wordplay) you would be better of with an indexed table, like this:

```
create MonthTable
  ,"  January "
  ," February "
  ,"   March  "
  ,"   April  "
  ,"    May   "
  ,"   June   "
  ,"   July   "
  ,"  August  "
  ," September"
  ,"  October "
  ," November "
  ," December "

: Get-Month                    ( n -- )
  12 min 1- cells MonthTable + @c count type cr
;
```

Which does the very same thing and will certainly work faster. True, you can't do that this easily in ANS-Forth, but in 4tH you can, so use it! The word ',"' compiles a string, whose address can be retrieved by '@C' as if it were a numeric constant. Note that '@C' just returns the *address* of the string, so you have to use 'COUNT' to obtain an address/count pair. Of course, there is also an equivalent to ',"' called ',|'. The latter is delimited by a bar instead of a quote, but essentially works the same way.

But can you use the same method when you're working with a random set of values like "2, 1, 3, 12, 5, 6, 4, 7, 11, 8, 10, 9". Yes, you can. But you need a special routine to access such a table. Of course we designed one for you. It is called "ROW" and you can use it by adding this directive:

```
[needs lib/row.4th]
```

This routine takes four values. The first one is the value you want to search. The second is the address of the table you want to search. The third is the number of fields this table has. And on top of the stack you'll find the excecution token of the word that searches the table. Two have been predefined, "NUM-KEY" searches numeric values and "STRING-KEY" searches string values.

This routine can search zero-terminated tables. That means the last value in the index field must be zero. Finally, it can only lookup positive values. It returns the value you searched,

the address of the row where it was found and a flag. If the flag is false, the value was not found.

Now, how do we apply this to our month table? First, we have to redefine it:

```
create MonthTable
  1 ,  ,"  January "
  2 ,  ," February "
  3 ,  ,"   March  "
  4 ,  ,"   April  "
  5 ,  ,"    May   "
  6 ,  ,"   June   "
  7 ,  ,"   July   "
  8 ,  ," August  "
  9 ,  ," September"
  10 , ,"  October "
  11 , ," November "
  12 , ," December "
  NULL ,
```

The first field must be the "index" field. It contains the values which have to be compared. That field has number zero. Note that this table is sorted, but that doesn't matter. It would work just as well when it was unsorted. Let's get our stuff together: the address of the table is "MonthTable", it has two fields and we want to return the address of the string, which is located in field 1. Field 0 contains the values we want to compare. We can now define a routine which searches our table:

```
: Search-Month                ( n1 -- n2 f)
  MonthTable 2 num-key row    \ search the table
  dup >r                      \ save flag
  if nip cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

Because "ROW" is able to search integer tables and string tables, you have to define which one it is by using either `num-key` or `string-key`. Now, we define a new "Get-Month" routine:

```
: Get-Month                   ( n --)
  Search-Month                \ search table
  if                          \ if month is found
    count type                \ print its name
  else                        \ if month is not found
    drop ." Not found"        \ drop value
  then                        \ and show message
  cr
;
```

Is this flexible? Oh, you bet! We can extend the table with ease:

```
3 Constant #MonthFields

create MonthTable
  1 ,  ,"  January " 31 ,
  2 ,  ," February " 28 ,
  3 ,  ,"   March  " 31 ,
  4 ,  ,"   April  " 30 ,
  5 ,  ,"    May   " 31 ,
  6 ,  ,"   June   " 30 ,
  7 ,  ,"   July   " 31 ,
  8 ,  ," August  " 31 ,
```

```
  9 ,  ," September" 30 ,
  10 , ,"  October " 31 ,
  11 , ," November " 30 ,
  12 , ," December " 31 ,
  NULL ,
```

Now we make a slight modification to "Search-Month":

```
: Search-Month                ( n1 -- n2 f)
  MonthTable #MonthFields num-key row
  dup >r                      \ search table, save flag
  if nip cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

This enables us to add more fields without ever having to modify "SearchMonth" again. If we add another field, we just have to modify "#MonthFields". We can now even add another routine, which enables us to retrieve the number of days in a month:

```
: Search-#Days                ( n1 -- n2 f)
  MonthTable #MonthFields num-key row
  dup >r                      \ search table, save flag
  if nip cell+ cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

Of course, there is room for even more optimization, but for now we leave it at that. Do you now understand why 4tH doesn't have a CASE construct?

12.18 Lookup tables with string keys

But what if the table we're using looks like this:

```
create MonthTable
  ," January" 31 ,
  ," February" 28 ,
  ," March" 31 ,
  ," April" 30 ,
  ," May" 31 ,
  ," June" 30 ,
  ," July" 31 ,
  ," August" 31 ,
  ," September" 30 ,
  ," October" 31 ,
  ," November" 30 ,
  ," December" 31 ,
  NULL ,
```

Sure, 4tH compiled some kind of integer value there, but an address to a string is less than helpful. We have to compare *strings* in order to find the correct entry, not *addresses*. So, we need a word that searches the table and returns the contents of the field that follows the appropriate string. Well, of course there is such a word. It is "ROW" again. You can use it by entering:

```
[needs lib/row.4th]
```

At the *beginning* of your program. ”ROW” takes an address/count pair of the string that has to be found, the address of the table it has to search for that string and the number of fields the table has. It returns the original address/count pair of the string, the address of the row where the search stopped and a flag. That makes it quite a useful word, e.g. how many days has June:

```
: GetDays                     ( a n --)
  MonthTable 2 string-key row \ search the table
  if
    cell+ @c . drop drop      \ if found, display the number of days
  else                        \ else an error message
    drop type ."  is not a month!"
  then
  cr
;

s" June" GetDays
```

If ”ROW” returns true, the value was found. If it returns false, it wasn’t. Note you have to indicate which datatype ”ROW” has to deal with. ”ROW” is quite versatile, but that is not the only merit of ”ROW” as we will see in the next sections.

12.19 Lookup tables with multiple keys

Some tables have multiple keys to search them, e.g. by name or by number. So far all tables we’ve seen dealt with a single key in the first column. It would be a shame if you had to split a table into two tables, simply because you had two different ways to access it. Fortunately, ”ROW” can handle this kind of tables as well as long as you put the key columns up front and add a NULL at the end of the table for every key, e.g.

```
create mytable
  ," Monday"      1 ,
  ," Tuesday"     2 ,
  ," Wednesday"   3 ,
  ," Thursday"    4 ,
  ," Friday"      5 ,
  ," Saturday"    6 ,
  ," Sunday"      7 ,
  NULL , NULL ,
```

This table consists only of key fields. You can search for the name and get a number or search for the number and get the equivalent name. The trick is to keep in mind what the key field is and the relative position of the datafield. In this case we want to search on number, so the corresponding name is the field *before* the key field. When you start the search the pointer you pass to ”ROW” has to point to the key field you want to search. In this case that is equivalent to:

```
mytable cell+
```

Let’s assume we want to search this table both ways:

```
: day>num                     ( a1 n1 -- a1 n1 -f | n2 f)
  mytable 2 string-key row dup >r
  if nip nip cell+ @c else drop then r>
;
```

```
: num>day                     ( n1 -- n1 -f | a1 n2 f)
  mytable cell+ 2 num-key row dup >r
  if nip cell- @c count else drop then r>
;
```

The first word doesn't hold any surprises. It is a vanilla search word. The second one passes a slightly modified pointer to "ROW" and decrements the address it returns, so it now points to the name field. We can use both words quite easily and transparently:

```
s" Friday" day>num if . else type ."  not found" then cr
s" New yearsday" day>num if . else type ."  not found" then cr

5 num>day if type else . ." not found" then cr
8 num>day if type else . ." not found" then cr
```

You will see they work as expected.

12.20 Lookup tables with duplicate keys

Although most tables come with unique keys you may find yourself in a situation where you have to resume a search. "ROW" can handle that situation as well. Let's examine this table:

```
create people
  ," Ritchie"  ," Lionel"
  ," Dijkstra" ," Edsger"
  ," Moore"    ," Henri"
  ," Ritchie"  ," Dennis"
  ," Wirth"    ," Nick"
  ," Hopper"   ," Grace"
  ," Moore"    ," Chuck"
  ," Hopper"   ," Dennis"
  NULL ,
```

There is one key field, since the table is terminated with only one NULL. We also find multiple Hoppers and Moores, so we can't be sure we've found the right one right away. In order to get that one we might have to continue our search. That is exactly what this program does:

```
: >surname 2 string-key row ; ( a n x1 -- a n x2 f)
: first? rot cell+ @c count compare 0= ;
: >next cell+ cell+ >surname ;
: .name type space type ;     ( a1 n1 a2 n2 --)

: >first                      ( a1 n1 x a2 n2 --)
  2>r                         ( a n x)
  if                          ( a n x)
    dup 2r@ first?            ( a n x f)
    if                        ( a n x)
      drop ." Found " 2r> .name cr
    else                      ( a n x)
      >next 2r> recurse       ( a n x a n)
    then                      ( --)
  else                        ( a n x)
    drop 2r> .name ."  not found" cr
  then                        ( --)
;
```

```
: >name -rot 2>r >surname 2r> >first ;

: demo
  s" Ritchie" s" Dennis" people >name
  s" Moore" s" Chuck" people >name
  s" Lovelace" s" Ada" people >name
;

demo                          \ run the demo
```

”>NAME” is a wrapper around this programs most important words ”>SURNAME” and ”>FIRST”. ”>SURNAME” simply searches for a given surname in the table and returns its address. ”>FIRST” takes over and compares the first name. If it checks out we’re done, if not it calls ”>NEXT”. ”>NEXT” increments the pointer, so it now points to the next row. Then it calls ”>SURNAME” again, effectively continuing the search. Finally ”>FIRST” calls itself to check the first name again. Depending on the contents of the table, this process can be repeated several times.

12.21 Fixed point calculation

We already learned that if we can’t calculate it in dollars, we can calculate it in cents. And still present the result in dollars using pictured numeric output:

```
: currency <# # # [char] . hold #s [char] $ hold #> type cr ;
```

In this case, this:

```
200012 currency
```

Will print this:

```
$2000.12
```

Well, that may be a relief for the bookkeepers, but what about us scientists? You can do the very same trick. We have converted some Forth code for you that gives you very accurate results. You can use routines like SIN, COS and SQRT. A small example:

```
[needs lib/math.4th]

45 sin . cr
```

You will get "7071", because the result is multiplied by 10000. You can correct this the same way you did with the dollars: just print the number in the right format. You can also use a delightful little library created by Leo Brodie called ”fraction.4th”. This library allows you to do arithmetic in the range of -13.1072 and 13.1071 with a precision of 0.0001! It seamlessly integrates with the ”math.4th” library:

```
[needs lib/math.4th]
[needs lib/fraction.4th]

45 sin s>v v. cr
```

This one will actually print:

```
0.7071
```

Of course, you can also use it with other math words, as long as they are properly scaled. The scaling constant is called '10K' and is available from "constant.4th". The nice thing about fractions is that you can do all basic math (within its range, of course) without bothering about the decimal point. To convert a fixed point number to a fraction, you have to multiply it by '10K', which is 10,000 and call 'S>V', e.g. in order to convert '0.7071' to a fraction you need to do this:

```
7071 s>v
```

You can also define a fraction yourself, e.g. this converts two-thirds to a fraction:

```
2 3 v/
```

You can add or subtract fractions, e.g.:

```
7071 s>v 2 3 v/ - v. cr
```

This is equivalent to:

```
PRINT 0.7071 - 0.6666
```

You can multiply or divide fractions, e.g.:

```
7071 s>v 2 3 v/ v* v. cr
```

This is equivalent to:

```
PRINT 0.7071 * 0.6666
```

Since a fraction is still a single-cell number, you can manipulate the stack in the usual way. In some cases you can even mix cells and fractions, as the following table will show you:

With the word 'V>S' you can convert a fraction back to a '10K' scaled fixed point number. You can print a fraction using the word 'V.'.

Another example is SQRT from "math.4th". If you enter a number of which the root is an integer, you will get a correct answer. You don't even need a special formatting routine. If you enter any other number, it will return only the integer part. You can fix this by scaling the number.

However, scaling it by 10 will get you nowhere, since "3" is the square root of "9", but "30" is not the square root of "90". In that case, we have to square the scale itself; we need 100, 10,000 or even 1,000,000 to get a correct answer. In order to retrieve the next digit of the square root of "650", we have to multiply it by 100, which is the square of 10:

```
[needs lib/math.4th]

: .fp <# # [char] . hold #S #> type cr ;
650 100 * sqrt .fp
```

Which will print:

```
25.4
```

To acquire greater precision we have to scale it up even further, like 10,000. This will show us, that "25.49" brings us even closer to the correct answer. If you want to use the ”fraction.4th” library, you will have to scale it to the square of ’10K’, which means you’re restricted to numbers upto ”21.475”:

```
include lib/math.4th
include lib/fraction.4th

10K 10K * constant 100M

21 100M * sqrt s>v v. cr
```

All these words and more are included in the library file ”math.4th”. You might not find the word you actually need, because we are not much of a mathematician. When we encounter new routines they will be added to 4tH. We would appreciate your input!

12.22 Double numbers

C’mon, indulge me, run this program:

```
max-n . cr
```

You will probably see some fairly large number displayed on your screen. What is it? Well, it is the largest number that can fit in a cell. Larger numbers and 4tH will start to behave erratically:

```
max-n 1+ . cr
```

Still, it is large enough to do the accounting for a reasonably sized enterprise. But it hasn’t been always like that. Early Forths could barely handle the accounting of an average schoolboy. In order to get some real work done they had to expand the range somehow. And if one cell isn’t enough you simply take two cells. That is what double numbers are all about: they are numbers that are composed of two cells.

The problem is that Forths operators aren’t overloaded. If you try to add up two double numbers with ’+’ you will end up with one double number and the addition of the two parts of the first double number. So in order to add up two double numbers, a separate word had to be defined. If you need a special word for addition, you will also need one for multiplication, subtraction, division and negation.

It is no secret that Charles Moore, the inventor of Forth, thought that double numbers had become superfluous after the introduction of modern processors and modern Forth compilers. That is one of the reasons that 4tH doesn’t have a native double word implementation. But should you need this vastly expanded range for one reason or another 4tH allows you to enter the murky world of double, unsigned and mixed numbers.

So, how does it work. First of all, if you need a full implementation you have to include these two libraries:

```
include lib/todbl.4th          \ double number input
include lib/dbldot.4th         \ double number output
```

Then you probably need some variables. But hey, if double numbers take up two cells, you can't use ordinary variables. That's true. You will need small arrays:

```
2 array dVar1                  \ double variable one
2 array dVar2                  \ double variable two
```

And here comes the next problem. How do you enter double numbers? Depending on the size of the number, you can use two approaches. The easiest one is to convert a single number to a double number with "U>D". The only catch is this only works for *positive* numbers. If you want to enter a *negative* number, you have to negate it afterwards:

```
500 u>d 60000 u>d d+           \ add 500 and 60000
2dup d. cr                     \ print the double number
dVar1 2!                       \ store it in variable one
```

Yes, every single operation has a *double* counterpart:

SINGLE	DOUBLE
+	d+
negate	dnegate
.	d.
2/	d2/
max	dmax
min	dmin
dup	2dup
@	2@
!	2!

Table 12.7: Examples of single and double number counterparts

But what if you want to enter a very large number right away? In that case you will have to convert a string to a double number with "S>DOUBLE[18]".

```
s" 5000000000" s>double        \ convert a string to double
2drop 2dup d. cr               \ print the double number
dVar1 2@ dmax dVar2 2!         \ save the largest in variable two
```

Finally, when you have done all you needed to do and you're left with a number that is small enough you can convert it back to a single number with "D>U". Note this will only work for positive numbers:

```
dVar2 2@ dVar1 2@ d-           \ subtract both double variables
2dup d. cr                     \ print the double number
d2/ d2/ d>u . cr               \ divide by 4 and convert to unsigned
```

Rule of the thumb is: stay away from double numbers if you can. It is slow, cumbersome and error-prone. If you can't, goodnight and good luck!

[18]Note it will bomb out when the string isn't a double number. If you want to play safe, use ">DOUBLE". There is also a single number ">NUMBER" word. In that case you can select the double number version by using its alias ">DNUMBER".

12.23 Floating point numbers (unified stack)

Warning!
This is *really* complex stuff, I *cannot* guarantee that it functions flawlessly. You may *lose accuracy* or get the *wrong result*. **Don't use any of this for any real life applications.**

Ok, if you *really, really, really* want it, 4tH also provides floating point number support. If you only need the basic operations and are willing to settle for limited accuracy and error checking, you should try `zenfloat.4th`. It is small and very easy to use. Just include it[19] and go right ahead:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
314159265 -8 f. cr
```

That will print the first eight decimals of "pi". Just read it like "314159265e-8" or in laymans terms "314159265" with eight places after the decimal point. Of course you can use positive exponents as well. Because ZEN floating point numbers are stored as two numbers[20] on the datastack[21], you can use '2DUP', '2DROP' and '2SWAP' to manipulate them. This is the way to calculate the surface of a circle with a radius of 10.55:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
314159265 -8 1055 -2
2dup f* f* f. cr
```

Which will happily print:

```
349.667115
```

You can also convert a number to a float and back:

```
1960 s>f 2dup f. f>s . cr
```

If you try to convert a floating point number that is bigger than 'MAX-N' to a single number it doesn't work of course. You can store a floating point number in a variable if you want:

```
include lib/zenfloat.4th
include lib/zenfpio.4th

2 array pi

314159265 -8 pi 2!
```

If you want to write ANS Forth compatible code, you can. Just include `zenans.4th` just after `zenfloat.4th`. It will allow you to write code like this:

[19] You *always* have to include `zenfloat.4th` manually before you include any other floating point library member.

[20] The exponent on the TOS, the mantissa on the 20S.

[21] Consequently, this implementation uses what ANS Forth calls a "shared floating point stack". Since 2008, ZEN float is considered a non-standard implementation (see: http://www.forth200x.org/fp-stack.html).

```
[DEFINED] 4TH# [IF]           \ if this is 4tH
include lib/zenfloat.4th      \ include the ZEN fp library
include lib/zenfpio.4th       \ include the ZEN fp I/O library
include lib/zenans.4th        \ make Zen ANS compatible
include lib/zenfsin.4th       \ include the SIN library
[ELSE]                        \ if this is ANS Forth
s" easy.4th" included         \ load the compatibility layer
[THEN]

FLOAT array fVar              \ a floating point variable
FLOAT array Pi                \ a FP variable holding PI
                              \ store PI in the variable
cr s" 314159265e-8" s>float Pi f!
Pi f@ 4 s>f f/                \ get PI and calculate sine
fsin fdup f. cr fVar f!       \ print the result and save it
fVar f@ f. cr                 \ print the variable
```

And this is the way it is run:

```
habe@linux-471m:~/Forth> gforth
Gforth 0.6.2, Copyright (C) 1995-2003 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type 'license'
Type 'bye' to exit
s" fpdemo.4th" included
( Lots of messages you can ignore )
0.707106780551956
0.707106780551956
 ok
bye
habe@linux-471m:~/Forth> 4th cxq fpdemo.4th
0.707106778
0.707106778
habe@linux-471m:~/Forth>
```

Apart from some rounding errors they are identical. Note that this "S>FLOAT" implementation is quite basic[22], but you can feed it most floating point numbers without any problems. If you use `zenans.4th`, most of the constructions in section 12.24 will work with `zenfloat.4th` as well.

The error handling of `zenfloat.4th` is *very* basic. If you try to get the square root of a negative number, it will just stop with an appropriate message. There are lots of floating point routines included[23], just check the glossary for more details.

12.24 Floating point numbers (separate stack)

Warning!
This is *really* complex stuff, I *cannot* guarantee that it functions flawlessly. You may *lose accuracy* or get the *wrong result*. **Don't use any of this for any real life applications.**

If you want something more sophisticated than `zenfloat.4th` you should try `ansfloat.4th`. This library is a full ANS-Forth implementation. In `ansfloat.4th` mantissas are double-cell unsigned. Single-cell exponents contain the mantissa's sign and a

[22] There is also a full ANS Forth implementation available: `fpin.4th` and `fpout.4th`. If you want to use these instead, remove `zenfpio.4th` and include them after you've included `zenans.4th`.

[23] All compatible library members are prefixed with '`zen*`'. See also chapter 23.2.

signed exponent. The format used is non-standard for simplicity. Exponents are almost a whole cell wide, leading to a wider dynamic range than most IEEE formats. Range and digits of precision versus cell width are listed below.

CELL WIDTH	DIGITS OF PRECISION	RANGE = 10^{-x} to 10^{x}
32	18	$x = 323196289$
64	37	$x = 1.3881175 * 10^{18}$
n	$INT((n-1)*0.602)$	$x = 0.301*(2^{n-2})$

Table 12.8: Range and digits of precision

Floating point arithmetic is easy to use, but be careful to watch where your inaccuracies are coming from. Floating point numbers are *approximations*. You can lose up to half a bit of precision in each operation. Differences between large numbers can be trouble spots. If you need transcendental functions, they can often be done in integer arithmetic since you don't need floating point's run-time auto-scaling. Floating point arithmetic uses a dedicated, shallow stack[24]. There is no depth checking, so underflows and overflows may occur unless error checking is added. Since floating point support is implemented in high level 4tH it is also rather *slow* and *big*.

So, how does it work. First of all, if you want to use floating point, you *always* need to include this library *before* any other floating point library:

```
include lib/ansfloat.4th      \ floating point words
```

This will also create the floating point stack. If you want a larger stack, just create this constant accordingly *before* including the library:

```
32 CONSTANT FLOATING-STACK    \ size of float stack
include lib/ansfloat.4th      \ floating point words
```

This will create a stack of 32 floating point items. You probably want to do some I/O, so let's take care of that one too:

```
32 CONSTANT FLOATING-STACK    \ size of float stack
include lib/ansfloat.4th      \ floating point words
include lib/ansfpio.4th       \ floating point I/O words
```

Then you probably need some variables. But hey, if floating point numbers take up more than a cell, you can't use ordinary variables. That's true. You will need small arrays:

```
FLOAT array fVar1             \ double variable one
FLOAT array fVar2             \ double variable two
```

Finally, you have to initialize the library and specify the precision[25]:

```
fclear                        \ initialize library
8 set-precision               \ set precision to eight
```

And here comes the next problem. How do you enter floating point numbers? Depending on the size of the number, you can use two approaches. The easiest one is to convert a single number to a floating point number with "S>F".

[24] ANS Forth defines a six item stack. A larger FP stack is an environmental dependency.
[25] The number of significant digits used by F.

```
500 s>f 60000 s>f f+          \ add 500 and 60000
fdup f. cr                    \ print the floating point number
fVar1 f!                      \ store it in variable one
```

Note the double numbers library is loaded by default as well, so ”D>F” is available too. Every single operation has a *floating point* counterpart:

SINGLE	FLOAT
+	f+
/	f/
*	f*
negate	fnegate
.	f.
max	fmax
min	fmin
dup	fdup
@	f@
!	f!

Table 12.9: Examples of single and floating point number counterparts

But what if you want to enter a floating point number right away? In that case you will have to convert a string to a double number with ”>FLOAT”.

```
s" 5000.575" >float           \ convert a string to a float
drop fdup f. cr               \ drop the flag and print the number
fVar1 f@ fmax fVar2 f!        \ save the largest in variable two
```

A drawback of defining floating point constants that way is that ”>FLOAT” is notoriously slow. However, there is a library that enables you to enter floating point numbers the ”mantissa + exponent” way[26]:

```
include lib/ansfloat.4th      \ include the fp library
include lib/ansfpio.4th       \ include the fp i/o library
include lib/fpow10.th         \ include the F10** library

  5000572 -3 me>f f. cr
314159265 -8 me>f f. cr
```

Defining constants this way is about *four times faster*. This is especially useful when you need a constant in a tight loop. Note that the exponent must be in the range of -24 to 24, otherwise you won’t get the correct result.

Most ANS-Forth floating point words are available, although these words usually have their own library file. For technical reasons they will *not* automatically include the floating point number library for you, but abort instead if it is *not* loaded. So if you want to calculate the sine of 45 degrees, you have to do this:

```
include lib/ansfloat.4th      \ include the fp library
include lib/ansfpio.4th       \ include the fp i/o library
include lib/fsinfcos.4th      \ include the library

fclear                        \ clear the fp stack
8 set-precision               \ set precision to eight

pi 4 s>f f/ fsin f. cr        \ calculate the sine
```

[26]See section 12.23.

Since a full circle (360 degrees) requires 2 times PI, we have to divide it by 4 to get the equivalent in radians[27]. But what if you make an error? What if a conversion overflows, you divide by zero or try to get the square root of a negative number? There is where "FERROR" comes in. It is a variable that holds the last floating point error that occurred. Here you got an example:

```
include lib/ansfloat.4th      \ include the library
include lib/ansfpio.4th       \ include the fp i/o library

fclear                        \ clear the fp stack
8 set-precision               \ set precision to eight
1 s>f 0 s>f f/                \ divide by zero
ferror ? cr                   \ examine FERROR
```

In this case, "FERROR" returns two. But what does that mean? Well, here you got a handy table of all IEEE 754 exceptions. In 4tH, these are predefined constants.

CODE	MEANING
FE.NOERRORS	No error
FE.OVERFLOW	FP overflow
FE.UNDERFLOW	FP underflow
FE.DIVBYZERO	Division by zero
FE.INEXACT	Inexact result
FE.INVALID	Invalid operation

Table 12.10: IEEE 754 FP math errors

You can simply clear any errors by invoking "FCLEAR" again. Note that this clears your floating point stack too, so you have to start all over again. And no, 'THROW' and 'CATCH' do *not* restore your floating point stack. No easy recovery here!

12.25 Floating point functions

ANS-Forth floating point comes with a large number of libraries, sometimes two or more for the same mathematical function. Table 12.11 may help you to select the proper one for your specific problem.

Note several ANS-Forth floating point libraries are used within other libraries. If a library is marked "default", it is the default that will be selected for inclusion if you haven't included another library already. Most of the time you can override the default selection by simply explicitly including it, e.g. instead of:

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fexpint.4th
```

You could set up the program like this:

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fexpt.4th
include lib/flnflogb.4th
include lib/fexpint.4th
```

[27] "FSIN" takes its parameter in radians.

In some cases a library member *may* assume a certain dependency is already resolved and bomb out. In that case, you'll have to figure out yourself which dependency that is and resolve it manually. If that happens, you may find chapter 23.2 quite useful.

Note that Zen float comes with lots of floating point functions as well, but it lacks the more specialized functions like numerical integration, Bessel, Riemann Zeta, elliptic integrals, etc.

12.26 Floating point configurations

In order to simplify setting up floating point support, there are five predefined configurations. The only thing you have to do is to include a single file. All configurations support basic floating point operations and floating point I/O. You can see in the table which configuration meets your particular needs.

For your information: precision is definitely *not* the same thing as accuracy. If you're still not sure what to choose configuration 2 (`fp2.4th`) is a safe bet. Note that "FLOAT" is always defined when having included any of the configurations, which enables you to check whether floating point support has been loaded, e.g.:

```
[UNDEFINED] float [IF] [ABORT] [THEN]
```

If you need to differentiate between Zen floating point or ANS floating point, you just have to check whether the constant "ZENFP" has been defined, e.g.:

```
[DEFINED] ZenFP   [IF] .( Zen float is loaded) cr [THEN]
[UNDEFINED] ZenFP [IF] .( ANS float is loaded) cr [THEN]
```

If you enter the realm of full fledged floating point numbers, you may find it very hard to resolve all dependencies. Well, it is not so hard as you think as long as you apply the following rule of the tumb. Before including any other floating point library file include this one:

```
include lib/fp3.4th
\ now you may include other library files
```

If you don't mind the size or the speed but *do* require full ANS Forth compliance, begin your program like this:

```
include lib/fp4.4th
\ now you may include other library files
```

You will see that this resolves most of your dependency problems[28]. Bottom line: if you don't need floating point numbers, avoid them and apply other techniques. But sometimes it cannot be avoided and I guess you'll agree with me that it's good it's there.

[28] If not, take a look at chapter 23.2.

12.27 Forth Scientific Library

4tH includes serveral library members of the Forth Scientific Library. The FSL contains several very complex and highly specialized mathematical words. Some of these words have been adapted to work with 4tH. All you have to do is to include `fsl-util.4th` after you've included `ansfloat.4th, ansfpio.4th` and preferably also `fpconst.4th`. It doesn't work with `zenfloat.4th`.

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fsl-util.4th
```

Note that like `ansfloat.4th`, `fsl-util.4th` isn't included automatically by other library members, so you have to include it *explicitly*. If not, the compiler will simply abort. Since the FSL uses special datatypes, you have to do some work in order to get them to work with 4tH. If you want to declare an fsl-array of e.g. ten floats:

```
10 FLOAT MARRAY MyFSL
```

You have to declare it like this:

```
10 FLOATS 1+ ARRAY MyFSL            ( allocation)
FLOAT LATEST FSL-ARRAY              ( initialization)
DOES> (FSL-ARRAY) ;                 ( runtime behavior)
```

If you want to declare an fsl-matrix of e.g. 16 by 8 floats:

```
16 8 FLOAT MMATRIX MyFSL
```

You have to declare it like this:

```
16 8 * FLOATS 2 + ARRAY MyFSL       ( allocation)
16 8 FLOAT LATEST FSL-MATRIX        ( initialization)
DOES> (FSL-MATRIX) ;                ( runtime behavior)
```

Surely, this is a bit awkward, but it is required to be as compatible as possible with sources that use the FSL. Note that all restrictions and reservations concerning floating point support also apply to the FSL words.

12.28 Statistical functions

4tH features a small array of statistical functions that can be used in conjunction with both the ANS and the Zen floating point libraries. First you have to declare the floating point stack of your choice:

```
false constant shared-fp            \ select ANS- or ZEN float
                                    \ include the ZEN float library
shared-fp
[IF]   include lib/fp2.4th
[ELSE] include lib/fp4.4th
[THEN]                              \ include the ANS float library
```

After you've included the statistical library itself you have to allocate the statistical structure required:

```
include lib/statist.4th              \ include the statistical library

/st_var array stats                  \ allocate a statistics structure
```

That was the hardest part. Now, initialize the floating point library and clear the structure so it is ready for use:

```
fclear 100 set-precision             \ initialize FP library
stats st.clear                       \ clear the structure
```

You can now add your floating point numbers to the structure by issuing:

```
2 s>f stats st.add
4 s>f stats st.add
4 s>f stats st.add
4 s>f stats st.add
5 s>f stats st.add
5 s>f stats st.add
7 s>f stats st.add
9 s>f stats st.add
```

If you have forgotten how many numbers you added, simply query the structure as if it were a normal variable:

```
stats @
```

A number of statistical functions are now at your disposal, e.g. variance, standard deviation, and several means. You can call them at any time by issuing, e.g.:

```
stats st.stddev f. cr
```

Note you can have several statistical structures at the same time and manipulate or query them independently. You can also clear them individually at any time without any side effects.

12.29 Numerical integration

Although 4tH comes with lots of functions, you may encounter a situation where you need a function and it isn't there. If you're willing to make a full implementation, please do - and send the result to me for inclusion. However, if you need it *fast*, you may want to resort to numerical integration. A well-known method for numerical integration is "Simpson's rule[29]". We offer three versions of it:

1. Standard method;
2. Composite method;
3. The "3/8" rule.

[29] https://en.wikipedia.org/wiki/Simpson%27s_rule

None of these methods is "the best". You simply have to try which one works best for you in a given situation. So how do you go about it? First you have to find the derivative of the function you want to approximate. If you are not very good at math, you may have to find one. In this example we want to approximate $ln(x)$. The integral of the derivative $\frac{1}{x}$ is $\int_1^x \frac{1}{t} dt$. Now we have to turn the derivative into a function:

```
: f() 1 s>f fswap f/ ;
```

That wasn't too hard, was it? Finally, we have to put the entire Simpson construction into place:

```
include lib/fp3.4th
include lib/simpson.4th

: f() 1 s>f fswap f/ ;
: (ln) 1 s>f fswap ['] f() 1000 simpson ;
```

Of course that's all very nice, but where do all these parameters come from? Well, look at the integral. The parameters of the integral state that it goes from "1 to x", where x is the parameter of the $ln(x)$. That's why we do a "FSWAP" in the "(LN)" function. We also pass the execution token of the "F()" function. That has always one parameter, which is the parameter that "SIMPSON" feeds it. The dt simply indicates what the name of this parameter is: t.

Basically, an integral calculates the surface of a function. "SIMPSON" simply chops that curve up in little, tiny segments and t represents the number of chops[30]. The more chops, the more accurate the result will be - but also the more time it will take. The number of chops must be *even* for the composite rule and *divisible by three* for the "3/8" rule. So, let's put the thing to work now:

```
fclear 100 s>f (ln) f.
```

That results in *4.60517038495714201*, which is not half bad when you think of it, since 4tH's own "FLN" returns *4.60517018598809136*.

So let's do another one, the $erf(x)$ function. It's defined as $\frac{2}{\sqrt{\pi}} \int_o^x e^{-t^2} dt$. Since $\frac{2}{\sqrt{\pi}}$ is about *1.1283791671*, we already got that one and since it is not part of the integral we can leave that one out of the equation. So we're left with $\int_o^x e^{-t^2} dt$:

```
: g() fdup f* fnegate e fswap f** ;
```

Now let's assemble the whole thing:

```
include lib/fp3.4th
include lib/falog.4th
include lib/simpson.4th

: g() fdup f* fnegate e fswap f** ;

: (erf)
  0 s>f fswap ['] g() 100 simpson
  s" 1.1283791671" s>float f*
;
```

[30] It's a bit more complex than that, but it gives you an idea.

Why only a hunderd slices? Because it is enough. It returns *0.71115563366031653* for 0.75, where our native "FERF" function returns *0.71115563365344696* and the actual value is *0.71115563365351513*. That ain't half bad!

Sometimes you got a *really* difficult one, like the gamma function. That one is defined as $\int_0^\infty t^{x-1}e^{-t}dt$. There are several problems with this one. First it calculates the entire thing from here to eternity. The only way out of this is to find a sane way to define our own eternity. I won't go into the details, but 10,000 works. The second thing is, that the parameter is now part of our integral. The only way out of this is to save it in a variable and subsequently use it in the formula. Finally, we end up with something like this:

```
include lib/fp3.4th
include lib/falog.4th
include lib/simpson.4th

float array (x)

: h() fdup fnegate fexp fswap (x) f@ 1 s>f f- f** f* ;
: (gamma) (x) f! 0 s>f 10000 s>f ['] h() 10000 simpson ;
```

Ten thousand slices? Yup, that's what's needed here - this one takes a *long* time. No results? No. We're not out of the woods yet, but the fix has nothing to do with numerical integration, so we'll leave that one be. Note the other Simpson variants work exactly the same and you can combine the lot if you want it. But we have another way to do numerical integration - and it's a *lot* faster. If it works.

The only thing you have to do is initialize it - and then you can relax, because you won't be bothered with the question how many slices you want. The method is Gauss-Legendre. And this is how it works.

```
include lib/fp3.4th
include lib/falog.4th
include lib/gauslege.4th

: g() fdup f* fnegate e fswap f** ;

: (erf)
  0 s>f fswap ['] g() lege_inte
  s" 1.1283791671" s>float f*
;

lege_coef lege_roots
3 s>f 4 s>f f/ (erf)
```

You only have to initialize these roots and coefficients *once* before the first use of "LEGE_INTE". After that you're home free and you can do all the calculation you want. The result isn't half bad: *0.71115563365634331*. So why wouldn't use it and leave Simpson be? Because it doesn't always work. Our $ln(x)$ function is a lot worse: *4.52401046228249623*. And when functions got singularities[31], this method does not work at all.

[31] In mathematics, a singularity is in general a point at which a given mathematical object is not defined, or a point of an exceptional set where it fails to be well-behaved in some particular way. E.g. $f(x)=\frac{1}{x}$ has a singularity at zero.

Algorithm	Library	Returns	Standard API
KISS	gmkiss.4th	single	Yes
super KISS	gmskiss.4th	single	No
Mersenne twister	mersenne.4th	single|float	No
MINSTD	minstd.4th	single	Yes
Mifare Crypto-1	mif-prng.4th	single	No
ACM V31N06	plecuyer.4th	float	No
GGUBS	prng.4th	float	No
RAN4	ran4.4th	double	No
Frank Buss	randbin.4th	single	No
Lehmer	randlcg.4th	single	Yes
SimScript	randlcg.4th	single	Yes
EPM2	randlcg.4th	single	Yes
RANDU	randlcg2.4th	single	Yes
CUPL	randlcg2.4th	single	Yes
SuperDuper	randlcg2.4th	single	Yes
Derive	randlcg2.4th	single	Yes
BCPL	randlcg2.4th	single	Yes
Turbo C++	randlcg2.4th	single	Yes
Turbo Pascal	randlcg2.4th	single	Yes
ANSI-C	randlcg2.4th	single	Yes
Simula BS2000	randlcg2.4th	single	Yes
Atari ST	randlcg2.4th	single	Yes
C-RAND	randlcg2.4th	single	Yes
APL	randlcg2.4th	single	Yes
BSPR	randlcg2.4h	single	Yes
EPM	randlcg2.4th	single	Yes
Starting Forth	randlcg2.4th	single	Yes
MWC v/d Horst	randmwcp.4th	single	Yes
BSD	random.4th	single	Yes
Microsoft	random.4th	single	Yes
Doom	randoom.4th	single	Yes
ACM V12N04	runiform.4th	float	No
TT800	tt800.4th	float	No
WELL512	well512.4th	single	No
Wichman Hill	wichill.4th	float	No
ZX Spectrum	zxrandom.4th	float	No

Table 12.3: 4tH randomizers

ALGORITHM	LIBRARY	COMBINE?	SIZE	SPEED
Slow sort	`slowsort.4th`	No	2.3	∞
Stooge sort	`stoosort.4th`	No	2.7	>500,000
Bubble sort	`bublsort.4th`	No	1.4	2500
Cocktail sort	`cocksort.4th`	No	2.6	2800
Cocktail sort (improved)	`coc2sort.4th`	No	2.0	1800
Simple sort	`simpsort.4th`	No	1.0	1800
Simple sort (improved)	`ismpsort.4th`	No	1.1	1300
Oyelami sort	`oyelsort.4th`	No	2.8	600
Insertion sort	`instsort.4th`	No	1.5	1500
Insertion sort (improved)	`ins2sort.4th`	No	1.7	1000
Selection sort	`selcsort.4th`	No	1.2	1200
Cycle sort	`cyclsort.4th`	No	2.7	4100
Pancake sort	`pancsort.4th`	No	2.4	3600
Heap sort	`hea2sort.4th`	No	3.9	2.5
Introsort	`intrsort.4th`	No	11.5	1.0
Quicksort	`qsort.4th`	No	4.0	1.0
Quicksort (unsafe)	`qsort.4th`	No	3.0	1.0
Comb sort	`com2sort.4th`	No	3.0	1.3
Shell sort	`shelsort.4th`	No	2.6	2.5
Shellsort (improved)	`shelsort.4th`	No	4.1	2.0
Circlesort	`circsort.4th`	No	2.0	20
Circlesort (improved)	`cir2sort.4th`	No	5.6	10
Bitonic sort	`bitosort.4th`	No	3.6	10
Merge sort	`mergsort.4th`	No	2.6	20

Table 12.4: 4tH sorting algorithms (address based)

ALGORITHM	LIBRARY	COMBINE?	SIZE	SPEED
Heap sort	`heapsort.4th`	Yes	3.0	3.0
Comb sort	`combsort.4th`	Yes	3.2	2.5
Selection sort	`sel2sort.4th`	Yes	1.3	3900
Bubble sort	`bub2sort.4th`	Yes	1.4	6000
Gnome sort	`gnomsort.4th`	Yes	1.3	6000
Gnome sort (improved)	`gno2sort.4th`	Yes	1.8	4200

Table 12.5: 4tH sorting algorithms (index based)

2OS	TOS	Word	Result
fraction	fraction	V*	fraction
fraction	cell	V*	cell
cell	fraction	V*	cell
fraction	fraction	V/	fraction
cell	cell	V/	fraction
cell	fraction	V/	cell
fraction	fraction	+	fraction
fraction	fraction	-	fraction

Table 12.6: Fraction words

FAMILY	FILE	DEFAULT	SPEED	ACCURACY	SIZE	ALGORITHM
FSIN	`fsinfcos.4th`	Y	+	+	+	Remez
	`fsincost.4th`	N	+	o	+	Taylor
FEXP	`fexp.4th`	Y	+	+	o	Remez
	`fexpt.4th`	N	o	o	+	Taylor
FLN	`flnflog.4th`	Y	+	+	o	Remez
	`flnflogb.4th`	N	-	+	+	Brute force
FERF	`ferf.4th`	*N/A*	+	o	+	Taylor, approx.
	`erf1.4th`	*N/A*	o	-	o	Approximation
	`erf.4th`	*N/A*	-	+	-	Chebyshev
GAMMA	`gamma.4th`	*N/A*	+	o	o	Fourier series
	`gammaln.4th`	N/A	o	+	+	Lanczos
	`fgamma.4th`	N/A	o	++	+	Spouge

Table 12.11: ANS-Forth functions

FILE	Family	FP stack	ANS-FORTH COMPLIANCE	SIZE	PRECISION	PORTING TO ANS-FORTH	PORTING ANS-FORTH
fp0.4th	Zen	Unified	None	Small	Low	No	No
fp1.4th	Zen	Unified	ANS 1994 compatible	Small	Low	Yes	Limited
fp2.4th	Zen	Unified	ANS 1994 compilant	Medium	Low	Yes	Yes
fp3.4th	ANS	Separate	ANS 200x compatible	Medium	High	Yes	Limited
fp4.4th	ANS	Separate	ANS 200x compliant	Large	High	Yes	Yes

Table 12.12: Floating point configurations

FUNCTION	WORD
Standard deviation	st.stddev
Variance	st.variance
Arithmetic mean	st.amean
Geometric mean	st.gmean
Harmonic mean	st.hmean
Root mean square	st.rms

Table 12.13: Statistical functions

Chapter 13

Special libraries

13.1 Infix formula translation[1]

Part of the elegance and simplicity of Forth comes from the use of Reverse Polish or postfix notation. As such, there are no requirements for precedence rules, parentheses, or complicated parsers. However, complicated mathematical and scientific equations that are often written in infix notation can be tedious to convert to Forth code. To simplify such a conversion, Julian V. Noble (one of the founders of the Forth Scientific Library, see section 12.27) wrote a FORmula TRANslator[2] to convert infix code to Forth code. Later Wil Baden, using Noble's concepts implemented a different version. For an implementation in 4tH, Wil Baden's version was chosen as it seemed simpler. The 4tH version of this (`opgftran.4th`) consists of three main words:

`FTRAN`	(a1 n1 – a2 n2) converts a text string from infix to postfix notation
`4TH-MATH`	(–) sets FTRAN to produce 4tH code as the output text
`STANDARD-MATH`	(–) sets FTRAN to produce standard postfix notation for the output text

To see how these work, consider the following code segment:

```
include lib/opgftran.4th

s" (12 - 2)/(6 - 3)" 2dup
forth-math    ftran type cr cr
standard-math ftran type cr
```

Which results in the following output:

```
s" 12E0" s>float
s" 2E0" s>float F-
s" 6E0" s>float
s" 3E0" s>float F- F/
12 2 - 6 3 - /
```

[1] Article written and contributed by David Johnson.
[2] Also known as "Operator Precedence Grammar".

As you can see, the Forth version of the output produces the 4tH code needed to do the calculation, while the standard version just produces the RPN output. Much more complicated expressions can be evaluated in which functions and variables are also handled. Variables begin with a letter and are not followed by a left parenthesis mark; whereas, function-calls are followed by parentheses.

Constants can also be handled by using embedded 4tH code. Line brackets (e.g., lembedded 4tH codel) are used for this, and all the text between the '|' brackets is taken as 4tH code. For example, consider that PI has been defined as a floating point constant (or similar.) Then `|PI|` can be used in the expression to be evaluated as a constant.

```
include lib/opgftran.4th

81 string Eq$
s" Stress=G/(2*|PI|*(1-v)*r)*b*sin(theta)" Eq$ place

Eq$ count forth-math ftran type cr cr
Eq$ count standard-math ftran type cr
```

In the equation for Stress, the variables are `G`, `v`, `r`, `b`, `theta` and `Stress`, while `PI` is a constant and `sin` is a function. These are expressed in the resulting output as:

```
G F@ s" 2E0" s>float PI F*
s" 1E0" s>float v F@ F- F* r F@ F* F/
b F@ F* theta F@ Fsin F* Stress F!
G @  2  PI *  1  v @ - * r @ * /
b @ * theta @  sin * Stress !
```

The 4tH version of the output uses "F!" and "F@" for the variables and assumes that all functions should start with the letter 'F'. If this is not the case, the function is renamed (e.g. `sin` vs. `fsin`). However, the "standard-math" version does not make such an assumption. Note, `PI` is treated the same in both cases as embedded 4tH code.

13.2 Evaluating infix formulas at runtime[3]

In the case in which the end-user my not wish to use RPN notation, the use of `opgftran.4th` allows for the conversion to postfix notation. A simple example is a calculator where the user can use either postfix or infix notation. The 4tH library already supplies an interpretive format; as a minimal example (from `dc.4th`), consider:

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/evaluate.4th
include lib/opgftran.4th

: _+ f+ ;
: _- f- ;
: _* f* ;
: _/ f/ ;

: _.  f.  space ;
: let  [char] :  parse ftran evaluate ;
: _quit quit ;
```

[3]Article written and contributed by David Johnson.

```
create wordlist
  ," +"          ' _+ ,
  ," -"          ' _- ,
  ," *"          ' _* ,
  ," /"          ' _/ ,
  ," ."          ' _. ,
  ," quit"       ' _quit ,
  ," let"        ' let ,
  NULL ,

wordlist to dictionary                \ assign wordlist to dictionary
                                      \ The interpreter itself
: mycalc
  standard-math
  begin                               \ show the prompt and get a command
    ." OK" cr refill drop             \ interpret and issue oops when needed
    ['] interpret catch if ." Oops " then
  again                               \ repeat command loop eternally
;

mycalc
```

Thus, the calculator could be used with infix notation such as:

```
let (2-1)/3: .
```

or with postfix notation such as:

```
2 1 - 3 / .
```

A more complete implementation is given by the `fdc.4th` example.

13.3 Converting infix formulas[4]

Using the 4tH library and `convert.4th`, infix equations can be added to your source code and converted before compiling. An example program to do this is `opgconv.4th`. In Wil Baden's original program, the single end-user word was "LET" based on same word in BASIC. This idea is implemented in `opgconv.4th` where a formula to is found by parsing the text between a "LET" and ":" pair. For example, consider the following program (`mytest.opg`):

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/zenfsqrt.4th

FLOAT array a   FLOAT array b   FLOAT array c
FLOAT array disc                          \ Used for discriminant
                                          \ Example from
: FQUADRATICROOT  ( F: a b c -- r1 r2 )   \ Wil Baden's OPG.TXT
   c F!  b F!  a F!                       \ Pickup coefficients
   LET disc = SQRT(b*b-4*a*c):            \ Set discriminant
   LET (-b+disc)/(2*a), (-b-disc)/(2*a):  \ Put values on f-stack
;

( Solve x*x-3*x+2 )  LET FQUADRATICROOT (1,-3, 2) : F. F. cr
```

[4] Article written and contributed by David Johnson.

This program can be converted to 4tH by using `opgconv.4th`. For example (assuming that it is named `mytest.opg`) one could use:

```
4th cx opgconv.4th mytest.opg mytest.4th
```

Some more complicated examples are also given in the `testopg.opg` file.

13.4 Portable Bitmap graphics[5]

You can make simple graphical drawings in 4tH using a virtual screen or bitmap. For example, consider the following:

```
include lib/graphics.4th
10 20 100 200 line
s" test.ppm" save_image
```

This short program will create a portable Netpbm file which can be converted to other graphical formats using the Netpbm libraries or viewed in numerous imaging programs. The image `test.ppm` is 640*480 pixels in size and simply consists of a white line plotted on a black background. However, one is not limited to such simple images. The PPM-format allows for true color resolution with each pixel consisting of 3 bytes for red, green, and blue. Each byte can have a maximum value or intensity of 255.

The ppm-format is a simple way to describe a "raw" image file in a widely recognized format. The format of the file header consists of a magic word, the size of the image, and the maximum intensity of the pixels. Following this is then the raw data. This format was created by Jeff Poskanzer as part of his Pbmplus graphical utilities (now called Netpbm). The PPM/PGM formats are used in the 4tH library routines, and since the PGM format is the simpler of the two, it will be presented first.

Here is a simple 8*8 PGM image of a diagonal line (white) on black background. The header consists of the magic number "P2" which indicates that the file is PGM image with the data stored as text. The next line is a comment (marked with a '#'), followed by the image size (8x8) and the maximum value of the data (or pixel). If the data after the file header is stored as binary, the the magic number is given by P5.

```
P2
# Simple test of 8x8 PGM grayscale
8 8
255
255 0 0 0 0 0 0 0
0 255 0 0 0 0 0 0
0 0 255 0 0 0 0 0
0 0 0 255 0 0 0 0
0 0 0 0 255 0 0 0
0 0 0 0 0 255 0 0
0 0 0 0 0 0 255 0
0 0 0 0 0 0 0 255
```

Color PPM images are stored with a similar format except 3 bytes are required to specify each pixel (e.g. red, green, blue). Thus PPM images are 3 times larger than the PGM grayscale images. The magic numbers for PPM files are "P3" for text and "P6" for binary. Both text and binary PPM/PGM files can be loaded using the 4tH library. However, images are always saved using the binary (P3, P6) format.

As an example for the P3, P6 color format, try loading and saving `icon_p3.ppm`.

[5] Article written and contributed by David Johnson.

```
include lib/graphics.4th
s" icon_p3.ppm" get_image
cr image_comment$ count type
s" 4th_icon.ppm" save_image
```

Now compare the contents and size of the two files. Better yet, see if you can view the file[6].

When a PPM/PGM image is loaded in 4tH (via "GET_IMAGE"), the data from the image file header is is stored in the 4tH variables: "PIC_HEIGHT", "PIC_WIDTH", and "PIC_INTENSITY". Additionally, both color PPM files (P3, P5) and grayscale PGM (P2, P5) files are supported. The main difference is that there is only one byte per pixel for grayscale image. As another example, consider the following:

```
\ Define, draw and save an image
include lib/graphics.4th

: bar ( -- ) 10 0 do 50 i + 50 50 i + 200 line loop ;

300 pic_width !              \ Define the current image size
300 pic_height !

color_image                  \ Define and draw our image
blue background
yellow bar

s" testimage.ppm" save_image \ saved to file
```

Try using "GRAYSCALE_IMAGE" instead and view the results. Note, that grayscale Netpbm files often have a file extension of *.pgm instead of *.ppm, but most viewing/graphical programs do not care.

After defining the picture type and size, a couple of basic commands can be used to create and manipulate the image. These commands work with both color and grayscale images. However, there is one difference. For color images, each pixel requires three bytes which we'll abbreviate as *rgb;* whereas, grayscale images have only one byte per pixel. Therefore, the stack pictures corresponding to the pixel count will depend upon the type of image in use. For example, if you were to specify a "GRAYSCALE_IMAGE" instead of "COLOR_IMAGE" in the following program , then "COLOR@" would leave just one value on the parameter stack.

```
include lib/graphics.4th

300 pic_width !
300 pic_height !
color_image

0 0 255 color!          \ sets the "current" color to blue (e.g., rgb: 0 0 255)
cr ." current color is " color@ . . .

white                     \ now set the "current color" to white (255 255 255)
background                \ set every pixel in the image to white

blue                      \ current color is blue
100 100 set_pixel         \ will color the pixel at (100,100) to blue
cr ." Pixel at (100,100) is "
100 100 pixel@ . . .      \ check to see if it's blue
255 0 0 100 100 pixel!    \ change it to red (e.g. rgb: 255 0 0)

10 10 98 98 line           \ draw line from (10,50) to (90,90)
s" dotline.ppm" save_image \ save the image to view it
```

[6]Gimp, OpenOffice, xnview and many more should work fine

You may notice that the word ”BACKGROUND” is rather slow. If you just wanted to set the background to white or to some level of gray, you can use the word ”WHITEOUT” which is much faster. That is:

```
255 whiteout              \ set the image to white;
190 whiteout              \ set image to some gray level;
0 whiteout                \ set image to black.
```

To summarize, the basic graphical commands with stack pictures are[7]:

WORD	STACK EFFECT	COMMENT
pic_width	– x	address contains image width
pic_height	– x	address contains image height
pic_intensity	– x	address contains maximum byte value for pixels
save_image	a n –	save portable bitmap to file
get_image	a n –	load file containing portable bitmap
color_image	–	work with color (rgb) pixels
grayscale_image	–	work with grayscale (single byte pixels)
color!	rgb | n –	define current color
color@	– rgb | n	get current color
pixel@	rx cy – rgb | n	get pixel values at specified point
pixel!	rgb | n cr cy –	set pixel values at specified point
set_pixel	rx, cy –	set pixel at specified point using the "current color"
line	rx1 cy1 rx2 cy2 –	draw line using the "current color"
background	–	set image background to current color (slow)
whiteout	n –	set image background to white (n=255) or gray (fast)
red	–	set the "current" color in use for color or grayscale images to red
blue	–	set the "current" color in use for color or grayscale images to blue
green	–	set the "current" color in use for color or grayscale images to green
white	–	set the "current" color in use for color or grayscale images to white
black	–	set the "current" color in use for color or grayscale images to black
yellow	–	set the "current" color in use for color or grayscale images to yellow
magenta	–	set the "current" color in use for color or grayscale images to magenta
cyan	–	set the "current" color in use for color or grayscale images to cyan

There are some graphical words used in converting between image types (color vs. grayscale and so on) and these will be discussed later (also check the example 4tH files). *Lastly,*

[7] rx = row or x; cy = column of y; rgb = red green blue.

*should you wish to work with image that contain more than 800*600 pixels, then simply change the maximum defaults in the* `graphics.4th` *file.*

13.5 Turtle graphics[8]

Turtle graphics provide a simple set of words for drawing simple and complex shapes. There are three basic words: ”XHOME”, ”XROTATE”, and ”XMOVE”. The ’x’ here represents the turtle. A simple program to draw a box can be defined as:

```
include lib/graphics.4th
include lib/gturtle.4th

200 pic_width !
200 pic_height !
color_image

: box ( n -- ) 4 0 do dup xmove 90 xrotate loop drop ;

255 whiteout
xhome
50 blue box
30 green box
s" box.ppm" save_image
```

The motion of the turtle can also be directed by the words: ”FORWARD”, ”BACK”, ”RIGHT”, ”LEFT”, ”XPENUP”, and ”XPENDOWN”. Here is an interesting example where a simple shape can be used to make a more complex drawing.

```
include lib/graphics.4th
include lib/gturtle.4th

300 pic_width !
300 pic_height !
color_image

clear-screen
xhome
xpendown

: octagon ( -- )  8 0 do  25 forward  45 right  loop ;
: tile ( -- )   8 0 do  octagon  45 right  loop ;

tile s” tile.ppm”  save_image
```

A couple of other words that may prove useful. Should the turtle get lost, then ”TURTLE@” will give the coordinates while ”COMPASS@” will give the directional heading. Likewise ”TURTLE!” and ”COMPASS!” will reposition the turtle.

In summary, we have the following:

WORD	STACK EFFECT	COMMENT
turtle!	x y –	set turtle position
turtle@	– x y	get turtle position
compass!	n –	set turtle directional heading in degrees

[8]Article written and contributed by David Johnson.

WORD	STACK EFFECT	COMMENT
compass@	– n	get turtle directional heading in degrees
xhome	–	place turtle in center of image and set the heading to 0 degrees
xrotate	n –	Rotate turtle by n degrees, positive numbers are counter clockwise.
xmove	n –	Move n pixels, can move forward (+) or backwards (-)
forward	n –	Move n pixels forward
backward	n –	Move n pixels backwards
left	n –	Turn left (rotate) by n degrees
right	n –	Turn right (rtate) by n degrees
clear-screen	–	Clear to white background
xpenup	–	Change color to white so that movent of turtle does not leave a trail on white background
xpendown	–	Change color to blue; e.g., same as using the word blue.

13.6 Annotating Portable Bitmap images[9]

To annotate the turtle graphic image from the last example, try the following:

```
include lib/graphics.4th
include lib/gbanner.4th

s" box.ppm" get_image

red horizontal text_up 5 25 s" boxes!" gbanner
black vertical text_up 75 20 s" yea!" gbanner
green vertical text_down 190 190 s" Hello there!" gbanner
magenta horizontal text_down 190 150 s" Bye bye!" gbanner

s" box2.ppm" save_image
```

The general syntax is just the screen position and the string with the key word being ”GBANNER”. To make it interesting, the orientation and direction of the text can easily be changed. Note the starting positions for the text in this example as the text may be written from right to left for some orientations.

The basic words are summarized as follows:

WORD	STACK EFFECT	COMMENT
gbanner	rx cy a n –	print the string a n at position rx cy
horizontal	–	plot the text horizontally
vertical	–	plot the text vertically
textup	–	change the text direction to upright
textdown	–	change the text direction to upside down

[9] Article written and contributed by David Johnson.

13.7 Color Palettes[10]

Since the Netpbm grayscale images are 3 times smaller than the color ones, one way to improve the efficiency (if true color resolution is not needed) is to store the color information in the grayscale format. While the setup shown here is not that useful, it was easily implemented. Instead of converting a color pixel to a grayscale pixel, a 6 level rgb format is used. For this, colors have a maximum intensity of 5, and the conversion is : palette = 38*red + 6*green + blue giving a maximum value of 215. The word "PALETTE_IMAGE" will set the correct defaults, e.g.:

```
include lib/graphics.4th
include lib/gturtle.4th
include lib/palette.4th

300 pic_width !
300 pic_height !
palette_image
( use some graphical routines)
s" ptest.ppm" save_image
```

This will save the image as a PGM image (with a P5 magic word). Thus when you view the image, it will appear as a strange looking grayscale image (as another example, try the `gtest4.4th` example program) To view `ptest.ppm` in color, you must convert the image to a PPM color file. There are two example programs to do this. One is `color2pal.4th` and the other is `pal2col.4th`. Note for this particular color palette, once it is converted back to the color format (with a P6 magic word) it should have a maximum pixel intensity of 5. Unfortunately, many graphical viewers have trouble displaying this correctly (though many work fine). Therefore, the `col2pal.4th` example will scale up the pixels values to the 255 level.

Conversion between color and grayscale pixels can be performed using the words "COLOR>GRAY" and "GRAY>COLOR". If color palette and/or grayscale images are defined, the words "COLOR>N" and "N>COLOR" can be used instead.

WORD	STACK EFFECT	COMMENT
color>gray	rgb – n	Covert 3 byte color format to 1 byte grayscale
gray>color	n – rgb	Convert grayscale to color format. The pixel still has the gray color though
to_color	–	Set the library primitives so that pixel and color data will use 3 bytes.
to_gray	–	Set library primitives so that pixel and color data will use 1 byte
color_image	–	Set defaults for using color (PPM) images
grayscale_image	–	Set defaults for using grayscale (PGM) images
palette_image	–	Set defaults for using color palette. Similar to grayscale and image will be save in PGM format.
color>palette	rgb – n	Convert color information to palette code

[10] Article written and contributed by David Johnson.

WORD	STACK EFFECT	COMMENT
palette>color	n - rgb	Convert palette code to color
to_palette	–	Set library primitives so that pixel and color data will use 1 byte palette code.
color>n	rgb – n	Convert color to palette (or grayscale) format
n>color	n – rgb	Convert palette (or grayscale) to color format

13.8 3D plotting[11]

Plotting or drawing in 3D using the 2D graphics library in 4tH simply requires a few words to map the 3D coordinates to 2D. The orientation of the default coordinate system has the origin towards the top left of the screen with the x-axis running down the screen rows and the y-axis running left across the screen columns.

With a right-handed coordinate system the z-axis is then pointing out of the screen (towards the observer). Two words that show this are “CUBE” and “AXIS”. The mapping from 3D to 2D is done with the word “3D>2D”. The use of these words is shown below:

```
\ Example of cube using the "mirror-left" and "mirror-right" words
\ The blue line from different corner-corner orientations are
\ better viewed using different mappings of 3D to 2D
\ Loading this also loads the graphics.4th and gbanner.4th word-set
include lib/gplot3d.4th

\ Place the 3D axes in center of image
colorimage 255
whiteout
center-plot

\ Line from top-left to bottom-right is clearly visible.
mirror-left
0 0 0 100 black cube
0 0 0 3d>2d 100 100 100 3d>2d blue line
-50 0 0 red axis

\ Conversely, line from top-right to bottom left is clearly visible.
mirror-right
0 150 0 100 black cube
0 250 0 3d>2d 100 150 100 3d>2d blue line
-50 150 0 red axis

s" 2cubes.ppm" saveimage
```

The resulting image is shown in figure 13.1. Note the orientation of the two cubes. The right cube is drawn with the box coming out of the page towards the right side of the image and this is the default orientation. One could have easily drawn the cube with an orientation tilted towards the left size of the image.

The orientation of 3D mapping can be controlled with two other words “MIRROR-LEFT” and “MIRROR-RIGHT” (which is the default). One reason for switching between these orientation is that the blue line drawn in the two cubes would not be easily visible if the orientation was switched.

[11] Article written and contributed by David Johnson.

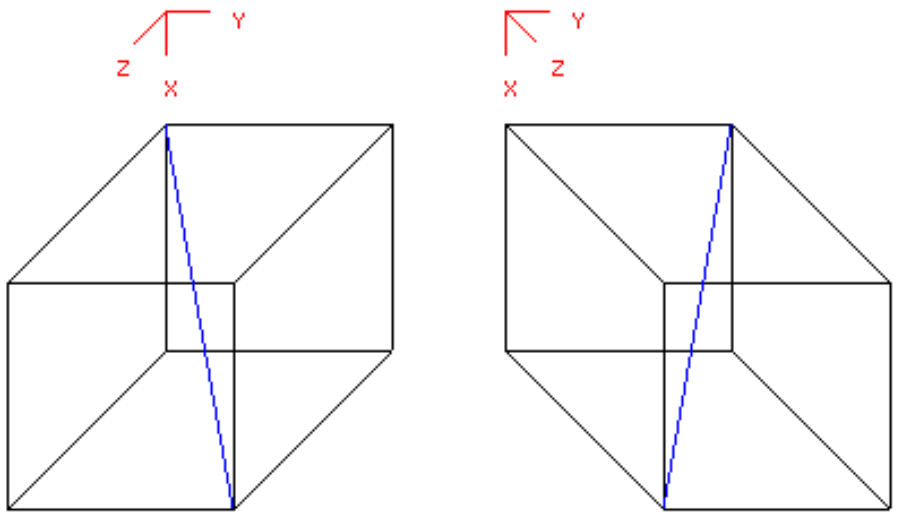

Figure 13.1: Mirrored boxes

For the 3D plotting to be really useful, one would also need to be able to rotate the coordinate axes to any desired orientation. This can be accomplished by defining the Euler angles ϕ_1, Φ, and ϕ_2. These represent a rotation about the *z-axis*, then about the new *x-axis*, and finally about the newest *z-axis*.

Positive values for the Euler angles represent a counter-clockwise rotation. For example, assume the *z-axis* should be placed parallel to the vector $znew = -1i + 2j + 3k$ or $[-1\ 2\ 3]$. This can be accomplished using ϕ_1 and Φ.

The first rotation about the *z-axis* is $\phi_1 = angle$ between $vector = -1i + 2j + 0k$ and the $x-axis + 90°$. That is $\phi_1 = angle$ between $[-1\ 2\ 0]$ and $[1\ 0\ 0] + 90°$ or $\phi_1 = acos\left(\frac{-1}{\sqrt{5}}\right) + \frac{\pi}{2}$ and hence $\phi_1 = 206.6°$.

Now a rotation about this *x-axis* will place the *z-axis* to the desired position, or $\Phi = angle$ between $[-1\ 2\ 3]$ and $[0\ 0\ 1]$. This gives $\Phi = acos\left(\frac{3}{\sqrt{14}}\right) = 36.7°$.

Lastly, a rotation about the newest *z-axis* ϕ_2 can place the the *x-* and *y-axes* to any desired location. Thus with the Euler angles of $\phi_1 = 206.6°$, $\Phi = 36.7°$ will place the *z-axis* parallel to$[-1\ 2\ 3]$ and the rotation ϕ_2 can then be used to set the *x-* and *y-axes*.

In 4tH, the word "EULER!" is used to set the Euler angles and these angles are defined in degrees. Here, integer math is used to calculate the coordinate transformation. Once the Euler angles are set, plotting is done as usual with the new orientation, which can be viewed with the word "AXIS". See the simple demo example below:

```
include lib/gplot3d.4th

colorimage                  \ define bitmap picture
600 picwidth !
400 picheight !
center-plot
255 whiteout

magenta                     \ draw a big box
0 0 0 euler!                \ keep default axis orientation
0 0 -50 100 cube
```

```
0 0 -50 black axis

green                       \ draw a smaller rotated box
90 45 -20 euler!            \ rotate the coordinate system
0 0 0 50 cube
0 0 0 blue axis

s" box.ppm" saveimage
```

WORD	STACK EFFECT	COMMENT
origin!	x y –	set the location of 3D-axis within the 2D image
center-plot	–	place the 3D-axis axis origin at the image center
euler!	phi1 Psi phi2 –	Set Euler angles to define plot orientation
xyznew	xold yold zold – x y z	apply coordinate transformation to map point to the new system
3d>2d	rx cy z1 – rx cy	map x,y,z to 2D for the graphic.4th wordset
axis	x y z –	draw the current axis orientation at point xyz.
cube	x y z size –	draw a cube at xyz
mirror-left	–	when mapping from 3d to 2d use the “left” version of the cube view
mirror-right	–	Use the “right” version of the cube view (default condition)

Table 13.4: 3D plotting wordset

The corresponding image is shown in figure 13.2. To summarize, the 3D plotting word set is to be used in conjunction with those words already defined in the graphics package. The complete wordset is listed in table 13.4.

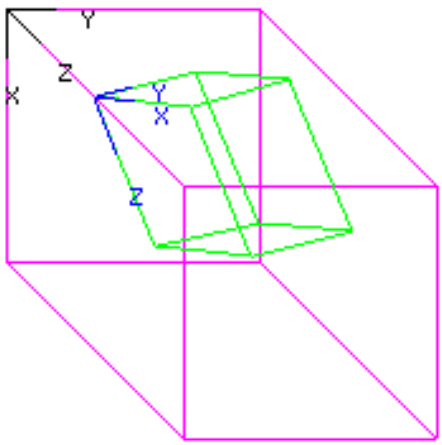

Figure 13.2: Two boxes

13.9 3D turtle graphics[12]

For the turtle graphics library[13] `gturtle.4th`, the word that moves the turtle by the number of pixels issued is "XMOVE". Essentially, a short line of length n is drawn. If the turtle is moving in 3D then this can be specified in "XMOVE" using the defered word "XDSCALE". By default, "XDSCALE" executes the word "2DPLOT", which takes *no* parameters and does absolutely nothing. However, in 3D one *could* define:

```
variable turtlez
0 turtlez !

: zturtle! ( x y z -- ) turtlez ! turtle! ;

\ place the turtle in 3D
: 3dplot ( x1 y1 -- x2 y2 ) turtlez @ 3d>2d ;
' 3dplot is xdscale
```

As a result, when "XMOVE" is executed the turtle moves on the specified z-plane to create an image. Some included examples and demo files using 3D plotting are listed in table 13.5.

File	Description
`gmirex.4th`	Example shows the use of `cube`, `axis`, `3d>2d`, and `mirror-left` and `mirror-right`.
`gdemo3d.4th`	Demo shows use of `euler!` to rotate the coordinate axes
`plt3dex.4th`	Example to plot of a simple 3D surface with two different views.
`3ddemo.4th`	This demo plots $\frac{\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}}$ with many views.
`g3spiral.4th`	Example of using 3D plotting with turtle graphics.

Table 13.5: 3D plotting examples

13.10 Interpreters

Those of you who know Forth will be very surprised to see that 4tH doesn't have a Forth prompt. Some will be even more surprised to see that 4tH does have an interpreter. It is a library routine, written in 4tH, that can easily be adapted and expanded. If you can write 4tH and maintain a table, you can use it. The next question you have to ask yourself, is do you want your interpreter to be case sensitive or not? If it is, "id" will work, but "Id" or "ID" will not. If you want it to be case sensitive, add the pragma `casesensitive`. Example:

```
[pragma] casesensitive        \ don't ignore case
[needs lib/interpret.4th]

: _+ + ;
: _. . ;
: id ." This is 4tH" cr ;
```

[12] Article written and contributed by David Johnson.
[13] See section 13.5.

Well, that isn't very hard, is it. Now we add a table to all that:

```
create wordlist
  ," +"  ' _+ ,
  ," ."  ' _. ,
  ," id" ' id ,
  NULL ,
```

Remember to terminate your table with "NULL"! Every entry consists of a string and an address to your routine. What will happen is that your user enters the string and the appropriate routine will be called. In this case, your interpreter has three commands: "+", "." and "id". We're a hair away from a real interpreter. We just have to assign our table to the dictionary. These lines do the job:

```
wordlist to dictionary
refill drop interpret
```

Now you can compile your application and run it. Enter:

```
45 12 + .
```

And it will print:

```
57
```

Yes, it's just as easy as that! If you enter something the interpreter doesn't recognize it will try to convert it to a number and throw it on the stack. But you will also see that it exits after you've entered that single line. That is because the interpreter is called just once. If you change that to:

```
begin refill drop interpret again
```

It will return with an new prompt. In that case it is wise to add a routine like:

```
: _quit quit ;
```

And add it to your interpreter, because otherwise your user will not be able to leave the application. Note that you have to do all the error-checking. E.g., if your user calls "_+" without putting sufficient items on the stack, 4tH will exit with an error. Of course, you can catch any exceptions. "INTERPRET" has a builtin word, "NotFound", that deals with any unrecognized strings. You can define your own if you want to. The only thing you have to do is to write a word which takes an address/count string and returns nothing, e.g.:

```
:noname 2drop ." I don't understand this!" cr ; is NotFound
```

Or more elaborate:

```
:noname ." I don't what '" type ." ' means!" cr ; is NotFound
```

You could even integrate it with the exception trapping, if you defined one:

```
1 constant #UndefName

:noname #UndefName throw ; is NotFound
```

When a word is not found, a user exception is thrown. This example is taken from `dc.4th`:

```
: dc
  begin                        \ main interpretation loop
    ." OK" cr                  \ print prompt
    refill drop                \ get input from use
    ['] interpret              \ interpret it
    catch dup                  \ catch any errors
    if                         \ if one occurred
      ShowMessage              \ show a message
    else                       \ otherwise
      drop                     \ drop the throw code
    then
  again                        \ loop back
;
```

You can still see the basic structure, but this one is much more advanced. You can also remove the code from the interpreter that decodes numbers. In that case, if a word is not found in the "dictionary" table it will exit immediately and report an error. Simply define this pragma *before* including the library:

```
[pragma] ignorenumbers
include lib/interprt.4th
```

Note that only the use of integers is supported in these interpreters. If you want floating point support you have to include ZEN floating point support[14] *before* including `interprt.4th`, either:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
include lib/interprt.4th
```

Or:

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/interprt.4th
```

The difference between both form is that the former only supports floating point entry in the scientific form. Consequently, the latter is more suited for humans like you and me.

Don't let anybody ever tell you you can't make interactive applications with 4tH. As you have seen, you can with very little effort.

13.11 Menus

The menu is a well-known user interface on the console. Typically, it displays a list of options from which the user can choose. 4tH allows you to create such menus with *very* little effort. First, include the appropriate library file:

```
include lib/menu.4th
```

[14] See section 12.23

Each option needs his own word. Note that an option - any option - can't take nor leave any items on the stack:

```
: helloworld ." Hello world!" cr ;
: calculate ." 1 + 1 = " 1 1 + . cr ;
: bye ." Goodbye!" quit cr ;
```

Now we have to define our menu. The first item is the title of the menu:

```
create MyMenu
  ," My beautiful program"
```

All subsequent entries are options. Each option consists of a description and an execution token. The menu is terminated by "NULL":

```
create MyMenu
  ," My first little programs"
  ," Hello world" ' helloworld ,
  ," 1 + 1"       ' calculate ,
  ," Exit"        ' bye ,
  NULL ,
```

Now all we have to do is to bind it to the main menu and run it:

```
MyMenu MainMenu
RunMenu
```

That's it! But what if we want to nest menus? Well, a nested menu is simply an option which opens up a new menu. For each additional menu we need to do two things. First, define a variable:

```
variable NextMenu
```

The variable will hold the address of the menu table you will declare later on. Second, define a word that allows us to enter the menu:

```
: >NextMenu NextMenu >Menu ;
```

That particular word is needed when we add an extra option to the main menu, which gives the user access to the sub menu we're about to define:

```
create MyMenu
  ," My first little programs"
  ," Hello world"   ' helloworld ,
  ," 1 + 1"         ' calculate ,
  ," More programs" ' >NextMenu ,
  ," Exit"          ' bye ,
  NULL ,
```

Next, we have to define the sub menu itself:

```
create MyNextMenu
  ," More little programs"
  ," Bottles of beer" ' bottlebeer ,
  ," FooBar"          ' foobar ,
  ," Return to main"  ' >MainMenu ,
  NULL ,
```

Hey, where did ">MainMenu" come from? Well, it's part of the library, so it's always at your disposal. And it's included in this menu, because otherwise the user would be stuck in this menu and wouldn't be able to return to the main menu. Finally, you have to add the menu by feeding the table and the variable you declared to "AddMenu":

```
MyNextMenu NextMenu AddMenu
```

Done! Next time you execute "RunMenu", you'll see have an extra option giving you access to the sub menu you just created. BTW, You can have as many menus as you like and can nest them as deep as you want.

Another nice feature is that when an error occurs during execution, the menu will display the appropriate message and simply return. In other words, it won't bomb out, no matter how bad a programmer you are.

13.12 Finite state machines[15]

Some tasks are easy to specify, but hard to program procedurally. Consider the task of accepting numbers from the keyboard. An unfriendly program lets the user re-enter the entire number because he made a small error. A friendly program, by contrast, filters the input so small errors are automatically corrected. For example, a number input routine that only allows signed decimal numbers. Decimal points, numerals and leading signs are all legal, but no other ASCII characters (including spaces) will be recognized.

If you try writing a single routine to 'EMIT' or 'DROP' a key according to these rules and you will find that there are just too many conditions for comfort. A test for each of the three classes of acceptable input, and tests for whether a sign or a decimal point have already been received.

The first state tests for all three possibilities, the second for two, and the last for only one (0-9). Is there a simple way to program that clearly? Yes, instead of nesting words within each other we can pass control as states on the data stack. In this context a state is simply an execution token, which when executed leaves another state on the stack for each possible exit condition. The program is simply a loop which continuously executes states.

Several states are identical in the tests made and actions taken, differing only in the state to be executed next. We need to be able to define the action to be taken and the transition to be made separately. The number returned can be used as an index into a two-dimensional array holding the action to be taken and the index of the state to be executed next. Well, this is what we call a Finite State Machine (FSM). It is very easy to define an FSM in 4tH.

First, you have to declare an array large enough to hold the FSM and its header information:

```
include lib/fsm.4th
3 4 * 2 * fsm.head + array fixed.pt
```

We have three states and four possible inputs. Each entry requires two cells and finally we have to take the overhead into account. But before we can define our FSM, we have to create the execution tokens. These tiny definitions do all the hard work:

```
:token (drop) drop ;
:token (emit) emit ;
```

[15] Part of this section is based on an article in Forthwrite UK by Jenny Brien.

And these take care of the transitions to another state:

```
:token >0 0 ;
:token >1 1 ;
:token >2 2 ;
```

Now we can setup the FSM itself by declaring the number of classes (its width) and the array that holds it. Note the use of the word "WIDE" is *mandatory*:

```
4 wide fixed.pt fsm
\ input   other    ||  digit    ||  - sign   ||  decimal point
\ state  -----------------------------------------------------
  ( 0 )  (DROP) >0 || (EMIT) >1 || (EMIT) >1 || (EMIT) >2   ||
  ( 1 )  (DROP) >1 || (EMIT) >1 || (DROP) >1 || (EMIT) >2   ||
  ( 2 )  (DROP) >2 || (EMIT) >2 || (DROP) >2 || (DROP) >2   ||
```

That's all! Before you can use an FSM, you first have to activate it and set the initial state. Note that upon definition the FSM already is activated with state zero, but is a good habit to explicitly initialize it:

```
fixed.pt to fsm.current
0 fsm.state !
```

We only have to transform the character to the appropriate class in order to make this baby fly:

```
0 enum 'other' enum 'digit' enum 'sign' constant 'point'

create categorize
  char 0 , 'digit' ,
  char 1 , 'digit' ,
  char 2 , 'digit' ,
  char 3 , 'digit' ,
  char 4 , 'digit' ,
  char 5 , 'digit' ,
  char 6 , 'digit' ,
  char 7 , 'digit' ,
  char 8 , 'digit' ,
  char 9 , 'digit' ,
  char + , 'sign'  ,
  char - , 'sign'  ,
  char . , 'point' ,
  char , , 'point' ,
  NULL ,
does>
  2 num-key row if cell+ @c else drop 'other' then nip
;
```

Time to execute, which is hardly rocket science either:

```
s" bad-345.rubbish4467invalid"
begin
  dup
while
  over c@ dup categorize
  fsm.run
  chop
repeat 2drop cr
```

Mission accomplished. Note you can define as many FSMs as you need. If you want to switch between them just activate them before use. Since the state is part of the FSM, it will remain unchanged between calls.

13.13 Virtual memory

Although computers have lots of memory nowadays, you may find yourself in a situation where you don't have enough memory to cater for your needs. In that case you may resort to using 4tH's virtual memory library.

4tH's virtual memory library uses block files, so you may need to create one before you can use it. Since block files are character-oriented you need to include another library if you need numerical data:

```
include lib/vmem.4th
include lib/ncoding.4th
```

That's all so let's define some variables and open the block file we created previously:

```
variable n1 nell vallot latest ! does> paging ;
variable s1   32 vallot latest ! does> paging ;
variable s2  128 vallot latest ! does> paging ;
variable s3  512 vallot latest ! does> paging ;
variable s4  512 vallot latest ! does> paging ;
s" vmemtest.scr" open-blockfile
```

When setting this up, you have to take three things in consideration:

1. Although it isn't required, you'd better define virtual memory variables in order, from the smallest to the largest;
2. The size of a virtual memory variable may *not* exceed "B/BUF" address units;
3. You have to make sure that all data fits in the block file. Take into consideration that no variable can span two blocks, so there'll be some overhead. The easiest solution is not to take any risks and simply oversize it - disk space is relatively cheap.

Now you can access them like any other variable:

```
2960 n1 n! &
s" This is a virtual memory string" s1 place &
```

It looks (almost) completely normal, doesn't it? Except for that ampersand, of course. This ampersand has to be added *every* time you write to a virtual memory variable. It signals to the underlying block management system that the block is now "dirty". If you think that's too cumbersome for your taste, you may adopt another solution - it's not *that* complicated.

Reading a variable is no problem at all: do as you've always done:

```
n1 n@ . s1 count type cr
```

And what about arrays? Well, first of all they have to be character based. And they're restricted to "B/BUF" address units. Or are they? As a matter of fact, they don't - but they do need some special treatment.

First of all, you have to think of the element size, which *must* be a power of two and may *not* exceed "B/BUF" address units. Second, you need to know how many rows you require. Now we can define it:

```
64 constant /row
80 constant #row
variable a1    20 nells vallot latest !
  does> swap nells +paging ;
variable a2 /row #row * vallot latest !
  does> swap /row * +paging ;
```

So "a1" is an integer arrays and "a2" is a string array. So far, so good. But what have we defined at the 'DOES>' part? Well, to address an array we're going to write:

```
5 a1 10 a2
```

Which will return the addresses of the sixth element of "a1" and the eleventh element of "a2" - we're counting from zero. Consequently, we pass the element number and the address of the array to the 'DOES>' part of the definition.

That's were the 'SWAP' comes in, since that one puts the element number on top, which we multiply with the element size to get the offset of that specific element in the array. Then we pass the whole bunch to "+PAGING". That's where the magic begins! If you e.g. prefer a more FSL-like array, now you know what to do..

The same rules apply here, if you want to write to a variable don't forget the ampersand:

```
2960 5 a1 n! &
s" This is a virtual memory string" 10 a2 place &
```

And you can just as easily retrieve it:

```
5 a1 n@ . 10 a2 count type cr
```

That's all! When you're done, you'll have to close your blockfile, of course. But that ain't too tricky:

```
close-blockfile
```

Will do just fine. Tip of the day: there is also a preprocessor library[16] which supports this virtual memory. Not only is it far more efficient, it also allows for much cleaner code.

13.14 Triple numbers

Sometimes you need to handle integers so big, they won't even begin to fit in a double number. Well, not all is lost - 4tH offers triple numbers as a last resort. But don't expect a nice and clean wordset, it is more like a toolbox. We'll tell you here how to use it.

First you have to include the appropriate library:

```
include lib/triple.4th
```

This gives you access to triple word arithmetic and comparison words. If you want to enter triple length words, you need this one:

```
include lib/totriple.4th
```

[16] See section 14.1.

And if you want to print a full triple word number, this one comes in handy:

```
include lib/trisharp.4th
```

Both I/O libraries will automatically load `triple.4th`.

BTW, you get what the name implies: facilities to construct your own triple word numeric output words. You won't find things like "T." or "UT.R" - you want anything like that, you'll have make them yourself.

Finally, you might want to include `3dup3rot.4th`, because that one provides three item stack operators like "3SWAP", "3DUP", etc. But if you already loaded `trisharp.4th`, you got that one automatically anyway.

Now we're loaded and ready. First, let's enter a triple number:

```
s" -295147905179352825855" s>triple
```

Now that's easy isn't it? If you want to convert a double number to a triple, you'll have to do some work, e.g.

```
0 constant ud>t
s" 4294967295" s>double ud>t
```

Adding or subtracting them is easy enough, just use "T+" or "T-". However, when you want to divide or multiply them, you will be confronted with a myriad of words like "UT*", "MT*" and "TU*" and nothing as simple as "T*".

The point is, that calculating the triple result of these operations becomes harder and harder each time a cell is added in order to increase the range of an integer. The best advice I can give you is to study them carefully and pick the most appropriate word for your particular situation.

If you want to print a triple word, you will have to put some work into it. Yes, the complexity here increases as well - e.g. setting a triple number up for signed output is so complex, we defined a special word for it, "(TSIGNED)". So if you want the triple word equivalent of "D.", this is it:

```
: t. (tsigned) <t# t#s tsign t#> type space ;
```

The reason we're not holding your hand here all the time is, that you will rarely need triple word precision - and if you do, you need to know 4tH inside out already in order to avoid errors and bugs. Most likely, you won't need all of the functionality of the triple word libraries - that's why we kept it lean.

13.15 Timers

There is a very low level word in 4tH that keeps track of time. It has several uses. Like a timer that measures how long certain operation takes, like the execution of a colon-definition ("DO-SOME-WORD" in this case):

```
time do-some-word time
swap -
." Do-Some-Word took " . ." seconds." cr
```

There is a somewhat more elaborate library member that does it all for you:

```
[needs lib/timer.4th]

timer-reset
do-some-word
.elapsed
```

This always prints the number of seconds that have elapsed. If you want to create your own display, you can define one easily:

```
[needs lib/timer.4th]

:noname <# # 6 base ! # decimal 58 hold # #> type ."  mins" ;
is timer-stop
```

You define ”TIMER-STOP” *after* inclusion of ”time.4th”, but *before* the first usage of ”.ELAPSED”.

13.16 Time & date

There is also an internal word in 4tH that will tell you what time and what date it is. With a little trouble ;). That word is called ’TIME’ and it will tell you how many seconds have gone since January 1st, 1970. That is the POSIX time format, which probably is of little use to you. However, combined with the appropriate library you can find out how late it is right now:

```
[needs lib/time.4th]

now ." hours:" . ." minutes:" . ." seconds:" . cr
```

Note that it doesn’t know about daylight-saving! It does know about timezones, which may be neccesary on *some* systems. You can determine your timezone by looking at an email message from a local friend. It will probably say somewhere:

```
Date: Mon, 25 Feb 2002 22:28:59 +0100 (CET)
```

The ’+0100’ means that you’re in timezone CET, which is one hour later than GMT. If it said:

```
Date: Sun, 16 Dec 2001 02:19:40 -0800 (PST)
```

This indicates that you’re in timezone PST, which is eight hours earlier than GMT. In that case ’tz’ would be:

```
-8 3600 * +constant tz         \ Pacific Standard Time
```

If you need it, define it accordingly *before* the inclusion of `time.4th`. There are also several words that will allow you to convert any POSIX time:

```
time posix>time . . . cr
```

Which will return the number of seconds (TOS), the number of minutes and the number of hours. ”POSIX>JDAY” will convert any POSIX time to a Julian day. The day of the week is another thing you can easily calculate:

```
[needs lib/time.4th]

: Weekdays
  dup 0 = if drop s" Monday"    exit then
  dup 1 = if drop s" Tuesday"   exit then
  dup 2 = if drop s" Wednesday" exit then
  dup 3 = if drop s" Thursday"  exit then
  dup 4 = if drop s" Friday"    exit then
  dup 5 = if drop s" Saturday"  exit then
  dup 6 = if drop s" Sunday"    exit then
;

today weekday Weekdays type cr
```

By the way, didn’t you hate the way we had to define "Weekdays"? Ugly, isn’t it? Well, there is a better way to do it. You’ll learn that in the next chapter (see section 12.17)! You can also print the full date:

```
[needs lib/time.4th]

today ." year:" . ." month:" . ." day:" . cr
```

Just don’t ask me how this thing works, Everett F. Carter figured this one out. ”TODAY” does the easy work. ”JDATE” converts the Julian day to the Gregorian date. There is also a way to convert a Gregorian date to a Julian day, called ”JDAY”:

```
[needs lib/time.4th]

26 02 2002 jday 64 - jdate
today ." year:" . ." month:" . ." day:" . cr
```

This can be quite handy if you want to calculate which date it was 64 days ago. ANS-Forth also defines a word that does it all called ”TIME&DATE”. This word throws seconds, minutes, hours, day, month and year (TOS) on the stack, but always returns GMT:

```
[needs lib/ansfacil.4th]

time&date . . . . . . cr
```

And finally, we even got a word that returns the date of easter:

```
[needs lib/easter.4th]

2005 easterSunday ." year:" . ." month:" . ." day:" . cr
```

Well, tell me, isn’t that kind of neat?

13.17 Tokenizing strings

Sometimes you want to split up a string in several different parts. This is called ”tokenizing”. Doing it with 4tH is (as usual ;-) quite easy. Just include `tokenize.4th`. Now

you got several words to get what you want. `tokenize.4th` creates a deferred word[17] called "IS-TYPE". It decides whether a character is of a certain type. In this example, we just want to know whether it is a lowercase 'a':

```
include lib/tokenize.4th
:noname [char] a = ; is is-type
```

Now it's time to play ball:

```
s" 01234aBcDe01234" scan> type cr
```

"SCAN>" will now skip all characters unless it is an 'a'. When it is found, it stops and returns the remainder of the string:

```
aBcDe01234
```

Yes, "SCAN>" starts at the beginning of the string. But there is also a word that starts *at the end* of the string:

```
s" 01234aBcDe01234" scan< type cr
```

It returns a different result too:

```
01234a
```

And what about the rest of the string? Well, that is discarded. But if you need it, there is also a word that just splits up the string:

```
s" 01234aBcDe01234" split> type cr type cr
```

So, "SPLIT>" returns *two* strings:

```
01234
aBcDe01234
```

And of course, he's got a little brother that works the other way around:

```
s" 01234aBcDe01234" split< type cr type cr
```

"SPLIT<" returns two strings too:

```
01234a
BcDe01234
```

That was quite easy. But what if you want to find the *first non-digit*? That's what we got "SKIP>" for! "SKIP>" skips all characters of a certain kind. We've already seen how we can distinguish characters in section 8.26. So in this case we just got to include `istype.4th` and assign "IS-DIGIT" to "IS-TYPE":

[17] See section 11.7.

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" skip> type cr
```

”SKIP>” returns a single string:

```
aBcDe01234
```

And he has a counterpart too:

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" skip< type cr
```

May be this result will surprise you, although it is completely correct:

```
01234aBcDe
```

What exactly did we ask for? We wanted the *first non-digit*, starting from *the end*. ’e’ is the first non-digit, so ”SKIP<” is completely correct.

13.18 Regular expressions[18]

Regular expressions (also called ”regex”) represent a way to identify patterns in a text. They can be used to search for patterns. You can define simple ones in 4tH. They don’t even have to take up too much code. The library `tre.4th` for instance, only supports the metacharacters ”*” and ”.”.

In regular expression syntax, a dot means “any single character.” So the regular expression ”b.b” will match “bib”, “bob”, “brb”, or “bub". It will also match two b’s with a space or a tab between them; the dot matches these “whitespace” characters also. The star means “zero or more of the previous character.” So the regex ”Bb*” will match “B”, “Bb”, “Bbb”, “Bbbb”, and so on. (Note that you must have the capital B, or the match will fail; the B must match before b* can try to). So, with library `tre.4th` you can define these queries:

```
s" aa" s" a*" match-reg . cr
s" ab" s" .*" match-reg . cr
s" aab" s" c*a*b" match-reg . cr
```

You can do some pretty sophisticated matching with these two already. If that isn’t enough, you may require `kpre.4th`. That one was originally written by Brian W. Kernighan and Rob Pike. It supports all the metacharacters listed in table 13.6.

The question mark means “zero or one of the previous character.” So the regex ”Bb?” will match “B” (one capital B, zero lowercase b) or “Bb” (one capital B, one lowercase b). The plus means “one or more of the previous character.” So the regex ”Bb+” will match “Bb”, “Bbb”, “Bbbb”, and so on, but will not match “B” by itself. A few other characters have special meanings in a regex:

[18]Part of this text is derived from *http://http://misc.yarinareth.net/regex.html* by Dorothea Salo under the ”Creative Commons Attribution 3.0 United States License”.

META CHARACTER	MEANING
.	Matches any single character.
^	Matches the starting position within the string.
$	Matches the ending position of the string or the position just before a string-ending newline.
*****	Matches the preceding element zero or more times.
?	Matches the preceding element zero or one time.
+	Matches the preceding element one or more times.

Table 13.6: KPRE supported metacharacters

- The ^ character, if it appears at the start of a regex, means "at the beginning of a document."
- The $ character, if it appears at the end of a regex, means "at the end of a document."

These are a few of the expressions you can define with `kpre.4th`:

```
s" 0,9" s" ^0,?9$" match-reg .  cr
s" 0:9" s" ^0,?9$" match-reg .  cr
s" aaaaaabac" s" ab*a$" match-reg .  cr
s" abbd"      s" ab*d$" match-reg .  cr
```

You might exclaim "but where are my bracket expressions". Rightly so, but we don't have 'em. Instead we have predefined (non-standard) character classes. Regex engines allow you to use "character classes" that narrow a search to specific collections of characters. Think of it as a dot, but more specific.

In 4tH, character classes are prefixed with a percent sign. E.g. "**%_**" is about equivalent to "**\s**" and "**%&**" approximates "**\w**". But there is more, as you can see in table 13.7.

CHARACTER CLASS	EQUIVALENT	VIM
%9	[0-9]	**\d**
%a	[a-z]	**\l**
%A	[A-Z]	**\u**
%#	[A-Za-z]	**\a**
%&	[A-Za-z0-9]	**\w**
%_	[\x00-\x20]	**_s**

Table 13.7: KPRE character classes

With character classes, it is very easy to e.g. recognize a floating point number:

```
s" -1234.56" s" -?%9+\.?%9*$" match-reg . cr
s" -1234"    s" -?%9+\.?%9*$" match-reg . cr
s" 1234.56"  s" -?%9+\.?%9*$" match-reg . cr
s" 1234"     s" -?%9+\.?%9*$" match-reg . cr
```

But what if you want to find an actual period (or an actual plus, or an actual question mark, or an actual star) in a regular expression? The backslash signals to a regular expression that

the following character, if it has a special regular expression meaning, should be interpreted literally, not in its special regex sense. So "**.**" by itself means "any single character," while "**\.**" with the backslash means an actual period. This works for any character that has a special meaning in a regular expression: "*****" means an actual star, and "**\+**" means an actual plus. Of course, the backslash works on itself, too; to search for a literal backslash, a regex pattern must contain "****" .

The disadvantage of regular expression is that you have to drag along the entire pattern matching engine, which is written in high level code. Advantage is that you may change the expression you want to evaluate at runtime - you can even enter it at the prompt if you like.

13.19 String pattern matching

If that isn't enough for you, 4tH also offers a small library for string pattern matching. Most regular expressions are static, that means they're not entered at runtime, but hard-coded into the program. That means you're dragging along the entire engine just to parse a string. 4tH offers almost the same possibilities, but you have to do a *little* more effort. Let's say I want to see if a string contains my first name:

```
include lib/chmatch.4th

: myname?
  s" Hh" char-match >r
  s" Aa" char-match r> and >r
  s" Nn" char-match r> and >r
  s" Ss" char-match r> and
;

s" Hans is the creator of 4tH" myname? . type cr
```

Of course, there are easier ways to do this, but you'll see it works. "CHAR-MATCH" will return a "true" value if one of the characters specified matches and a "false" value if it doesn't. E.g.:

```
s" Hans" s" Hh" char-match
```

Will return a "true" value and "ans". On the other hand:

```
s" Hans" s" Aa" char-match
```

Will return a "false" value and "Hans". Now you understand how it works: "MYNAME?" chops off character by character until it is done. But what it I want to check whether "Hans" appears in the middle of a string? Easy:

```
include lib/chmatch.4th

: myname?
  s" Hh" skip-until
  s" Hh" char-match >r
  s" Aa" char-match r> and >r
  s" Nn" char-match r> and >r
  s" Ss" char-match r> and
;

s" In October 1994 Hans created 4tH" myname? . type cr
```

This will wind the string upto the first ”H” and then start scanning. But what if there’s another ”H” in the string? Again, you can handle that:

```
include lib/chmatch.4th

: myname?
  begin
    s" Hh" skip-until
    s" Hh" char-match >r
    s" Aa" char-match r> and >r
    s" Nn" char-match r> and >r
    s" Ss" char-match r> and >r
    dup 0<> r@ 0= and
  while
    r> drop
  repeat r>
;

s" The 4tH compiler was created by Hans in 1994" myname? . type cr
```

You want to know whether it was at the very end of the line? Yes, you can:

```
include lib/chmatch.4th

: myname?
  begin
    s" Hh" skip-until
    s" Hh" char-match >r
    s" Aa" char-match r> and >r
    s" Nn" char-match r> and >r
    s" Ss" char-match r> and >r
    dup 0<> r@ 0= and
  while
    r> drop
  repeat r> over 0= and
;

s" The 4tH compiler was created by Hans" myname? . type cr
```

Let’s try something more difficult now, let’s see if we can parse a number. The first part is very easy:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
s" 0endshere" 'digits' char-match . type cr
```

That works, the correct flag is returned and the string ”endshere” is returned. But a number may be of arbitrary size, how do we do that? Simple, after the first number we can simply discard any trailing digits:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'

s" 0123456789endshere"
'digits' char-match >r
'digits' skip-while
r> . type cr
```

Still, it returns the correct flag and the correct string. But a number may also come with an *optional* sign. Now that complicates life, does it? No it doesn't:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'

s" -123456789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
r> . type cr
```

What? Aren't we evaluating the flag? No, why should we. The sign is *optional*, isn't it? Note that "CHAR-MATCH' not only sets a flag, but also *discards* the character matched. So it it *isn't* a sign, it will simply remain there and be caught by the digit test. But we'll take it a step further. What about a decimal point? A number may come with one or without one. And it may appear anywhere in the number. Now how we do that?

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -123456.789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
'point'  char-equal drop
'digits' skip-while
r> . type cr
```

When parsing stops, because the program reached the point where there are no more digits, we simply check for an optional decimal point. If it is there, we continue checking for digits. If it isn't there, parsing stops. Note that the check for trailing digits fails too, but you may make it optional if you want, it won't change the result:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -123456.789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
'point'  char-equal if 'digits' skip-while then
r> . type cr
```

Note that "CHAR-EQUAL" performs the same function as "CHAR-MATCH", it's only more efficient if there is only one character to compare[19]. Our program still doesn't allow for "-.45". Yes, you can add an optional test for that, but we don't want "-.45.67" to succeed. That one can be handled too:

[19] If you want to speed up a single character search, `scanskip.4th` offers the words you're looking for.

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -.789.000endshere000."
'sign'   char-match drop
'point'  char-equal >r
'digits' char-match r> swap >r >r
'digits' skip-while
r> 0= if 'point' char-equal if 'digits' skip-while then then
r> . type cr
```

Now the test will only be performed if the previous test failed, which is exactly what we want. Try it. Where ever you place the dot, it won't make any difference: it will *always* return ".000endshere000.". Unfortunately, that also applies to:

```
s" -0..000endshere000."
```

If you require that a decimal point is *always* followed by a digit, simply enforce it:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -0..000endshere000."
'sign'   char-match drop
'point'  char-equal >r
'digits' char-match r> swap >r >r
'digits' skip-while r> 0=

if
  'point' char-equal
  if
    'digits' char-match r> and >r
    'digits' skip-while
  then
then

r> . type cr
```

Note this *doesn't* change the string returned (up to the trailing point it was still a valid number), but it *does* change the flag: this is *not* a valid number!

I hope you see that 4tH can do some pretty sophisticated pattern matching with very little effort without dragging along an entire regular expression engine, either in code or in the compiler itself.

13.20 Wildcard pattern matching

If you feel that regular expressions are a bit too complex for your taste, you can also use wildcards for basic pattern matching. First you have to include it:

```
include lib/wildcard.4th
```

Or:

```
include lib/wildcrdr.4th
```

For the recursive version, which is a bit shorter. You've probably used wildcards before. It is very easy. A "*" stands for *zero or more characters* and a "?" stands for *a single character*. E.g. if you're looking for a line that begins with a date in the 21st century, then the word "INVOICE" and finally ends with a name, you could try this:

```
mystring count s" 20??-??-??  == INVOICE ==  *" match-wild
```

"MATCH-WILD" returns true if the string matches and false if the string doesn't match, which is different from "COMPARE". Both strings are consumed, so save their address/count pair if you need them later on. Note that "MATCH-WILD" is *case sensitive*, so if you need a case insensitive comparison you will have to convert them first.

Finally, "MATCH-WILD" is faster than regular expressions, but also less precise. E.g. "gr?y" will not only return "grey" and "gray", but also "groy"; "reg*exp*" will not only return "regular expressions", but also "registered express mail".

13.21 Escape characters

Sometimes you need a string with embedded control characters. You could try to patch them into your sourcecode, but that is usually not a good idea. You could also patch them into a variable, but that is cumbersome. 4tH offers a word that will allow you to use the control characters listed in table 13.8.

ESCAPE CHARACTER	MEANING
\a	Bell
\b	Backspace
\e	Escape
\f	Formfeed
\l	Linefeed
\n	Linefeed
\q	Quote (")
\r	Carriage return
\t	Tab
\v	Vertical tab
\z	Null character

Table 13.8: Supported control characters

Using this facility is quite simple:

```
include lib/escape.4th
s" \tThis is the \qC:\\4tH\q directory" s>escape type cr
```

Which will print:

```
        This is the "C:\4tH" directory
```

Note that other escaped characters are printed ”as is”, like the backslash in this example. If you need even more flexibility, we have another card up our sleeve. The only catch is you’ll have to know the ASCII code of the characters you want to insert, either in binary, octal, decimal or hex:

```
include lib/embed.4th
s" %1001This is the&42C:\4tH#34$20directory" s>embed type cr
```

Which will also print:

```
This is the "C:\4tH" directory
```

- Every number prefixed with a ’**%**’ character is interpreted as a binary ASCII code;
- Every number prefixed with ’**&**’ character is interpreted as an octal ASCII code;
- Every number prefixed with a ’**#**’ character is interpreted as a decimal ASCII code;
- Every number prefixed with a ’**$**’ character is interpreted as a hexadecimal ASCII code.

How much flexibility do you need? Of course, all this flexibility comes at a price. The strings are not expanded at compile-time but at *run-time*, so it will cost you a little more space and a little more time compared to other string literals.

13.22 Chinese characters

There are several encodings for Chinese characters, e.g. Unicode, GBK and Big5. As long as the ASCII-7 range of characters is respected, 4tH is able to handle them. 4tH can even convert between some of these encodings. The `unicdgbk.4th` file forms the centerpiece of this functionality. Basically, it offers conversion between Unicode and GBK/2. Since it is *very* large, it isn’t included automatically.

4tH offers conversion between UTF-8 and GBK/2 - and vice versa. GBK/2 offers about 7,500 characters. UTF-8 is more prevalent under Linux, while a GBK/2 derivate is more prevalent under Windows. Using the UTF-8 to GBK/2 conversion couldn’t be easier:

```
include lib/unicdgbk.4th
include lib/utf8gbk2.4th

s" Some UTF-8 encoded Chinese characters"
utf8>gbk2 type cr
```

Note this is a *destructive* conversion. After this, your original string is gone. If you want to preserve it, you have to save it first to another string variable. The string it returns is *not* terminated, so if you’re converting a string variable this might be a good idea:

```
include lib/unicdgbk.4th
include lib/utf8gbk2.4th
64 string mystring

s" Some UTF-8 encoded Chinese characters"
mystring place

mystring count utf8>gbk2 >string
```

This ”in place” conversion is possible, because a GBK/2 encoding *never* requires more space than an UTF-8 encoding. However, the opposite isn't true, so depending to the size of your original GBK/2 source string, you'll need a larger UTF-8 buffer, e.g.:

```
include lib/unicdgbk.4th
include lib/gbk2utf8.4th
64 string myfirststring
128 string mysecondstring

s" Some UTF-8 encoded Chinese characters"
myfirststring place

myfirststring count mysecondstring 128 gbk2>utf8
>string mysecondstring count type cr
```

Note you don't need any external libraries or programs to accomplish this. 4tH is all you need.

13.23 Managing INI files

An INI file is a plain ASCII file that is used to configure software. It was designed by Microsoft and dates back to the times of MS-DOS. However, you will find it as well nowadays on Linux and Unix systems, because INI files are human-readable and simple to parse.

It's for that reason we chose to support them in 4tH. As a matter of fact, we created two distinct libraries. The first and the most simple one just offers an interface to evaluate an INI file.

It is quite simple. Let's say you just want to store the values in the INI file in your program. First you create a few words to achieve that goal. Note that values are presented to you as strings and your word must consume them entirely:

```
include lib/inifile.4th

variable num1
16 string str1

: getnum1 number num1 ! ;
: getstr1 str1 place ;
```

Then you create a table with the following layout: first the section, then the key and finally the execution token of the word that must process the value:

```
create myini
  ," config" ," number1" ' getnum1 ,
  ," init" ," string1" ' getstr1 ,
  NULL ,
```

And finally you just send the name of this table and the filename (with path if required) to "INIFILE". That's all!

```
myini s" myini.ini" inifile
```

However, if you want to create or modify an INI file, you need something more sophisticated. It's time for the INI file management library. Note that this piece of software is highly dependent on dynamic memory, so if your program is already using the heap or if you want to manage a *really* big INI file, you may want to increase the size of the heap(s).

But first, let's define a new INI file. Of course, you do have to include the library:

```
include lib/inimanag.4th
```

Then you have to signal you want to define an INI file from scratch:

```
begin.inifile
```

Next, define a new section. Yes, you can do without, but I don't think it is really neat:

```
s" Compiler" section.inifile
```

And now you can define your keys and values. Note, *first* the value *then* the key. Let's not forget we're using Forth ;-)

```
s" 4tH" s" Name" keypair.inifile
s" 3.62.5" s" Version" keypair.inifile
s" Forth" s" Language" keypair.inifile
```

Of course, you can start a new section and add some more keys:

```
s" Platform" section.inifile
s" Linux" s" Name" keypair.inifile
s" 4.1" s" Version" keypair.inifile
```

And when you're finally done, you can save it to disk with a single line:

```
s" myfile.ini" put.inifile abort" Cannot write .INI file"
destroy.inifile
```

These are two distinct actions. The first line saves the file to disk. The second line removes the INI file from memory. So yes, you can save your file preliminary and continue working with it - as long as you don't issue "DESTROY.INIFILE", of course.

Reading it back is just as simple:

```
inifile get.inifile abort" Cannot read .INI file"
```

Just don't forget that both "PUT.INIFILE" and "GET.INIFILE" return a non-zero flag when there's any trouble. Ok, we got this thing in memory now, let's make some changes:

```
s" Version" s" Platform" delkeypair.inifile
abort" Cannot delete key from named section"
```

That one deleted a key. It also returns a non-zero flag on error, e.g. because the key or the section could not be found.

```
s" J.L. Bezemer" s" Author" s" Compiler" addkeypair.inifile
abort" Cannot add key to section"
```

This one adds a key to a certain section. You can also use it when defining an INI file, of course, but you can save yourself a lot of typing by using "KEYPAIR.INIFILE" instead.

```
s" Hans Bezemer" s" Author" s" Compiler" set.inifile
abort" Cannot modify key in named section"
```

And of course, you can modify a value as well - initials are so formal. Finally, you can write it to disk - we've seen that one before. Yes, you can also retrieve a value with this library:

```
s" Author" s" Compiler" value.inifile
```

But you have to write your own handling of the string it returns (address and length). And if that's the only reason you're using this library, you're creating a lot of overhead and you'd be better off with `inifile.4th`.

13.24 Extract, transform and load

"Extract, transform and load" is the name of one of the most used conversion methods. First you rip the data from the source, then you manipulate it and finally you move it to its destination. Well, although what we do is more appropriately called "Extract, load and transform" you can apply the same principles when reading a .CSV file with 4tH. It is very handy if you don't know the exact layout of the file, but do have a header line with the correct field names[20]. Of course, first thing you have to do is to include the library:

```
include lib/etl.4th
```

The data has to end up somewhere, so next thing to do is to define the layout and allocate enough space for a record:

```
struct
  16 +field Population
  32 +field State
  16 +field Capital
buffer: States
```

You know I prefer to define a word to parse a field[21], so let's do that now:

```
: field> [char] ; parse-csv csv> ;
```

Now the real fun starts. We have to make a table that maps the column header of the .CSV file to the fieldname of the structure. On top of that, we have to determine if we want to manipulate that field and how:

```
create Mapper
  ," Population" ' Population , ' >comma ,
  ," State"      ' State ,      ' none ,
  ," Capital"    ' Capital ,    ' >upper ,
  NULL ,
```

[20] See RFC 4180, section 2.3
[21] See section 9.23.

Note our field is maniplated after it has been placed in the structure - not before. The reason is simple. We don't want to clobber our source record - which could easily happen when a field expands.

You probably wonder where ">COMMA", "NONE" and ">UPPER" come from. Well, these words manipulate our fields. They all take the address and length from their respective fields and return nothing. E.g. "NONE" could easily be defined as:

```
: none 2drop ;
```

Next, 4tH needs to know how much lines there are in "MAPPER". Of course, you could define:

```
3 constant #etl
```

But you could also let 4tH figure it out for you:

```
here Mapper 1+ - /fieldmap / constant #etl
```

Finally, we need to define the "transformation array" this library requires:

```
#etl /etl * array Transpose
```

Now we're ready to rock 'n roll. Let's read the header:

```
refill 0= abort" Cannot read header"
Transpose Mapper ['] field> read-header
```

Yes, we feed it the "transformation array", the mapping and an execution token to read a field. That's all. After it has executed, the address of the "transformation array" and its actual length are on the stack and 4tH truly knows how to read your file. All you have to do is call "READ-TUPLE" for each following line:

```
2dup States ['] field> read-tuple
```

"READ-TUPLE" takes four parameters: the address of the "transformation array", its length, the address of the target structure and an execution token to read a field. That's all to need to read an entrire record. Note the first two stack items are provided by "READ-HEADER".

You may be asking yourself why you should use this. Well, this library does't care in what order the fields are presented and simply ignores any superfluous ones. With this you are immune to slight changes in the layout of your .CSV files. That's why!

13.25 Writing spreadsheet files

4tH is able to write to a variety of spreadsheet formats, both proprietary and FOSS (see table 13.9). All these formats are known to work with OpenOffice 3.x, Libreoffice 4.x and MS-Excel 20xx. KOffice 1.x can read `.ksp` files. The interface is almost identical and very straight forward:

	Type	*Extension*	*4tH library*	*Worksheets?*	*Integers only?*
XLS 2.1	Binary	.xls	msxls2-w.4th	No	Yes
FODS	XML	.fods	oofods-w.4th	Yes	No
MS-XML	XML	.xml, .xlsx	msxlms-w.4th	Yes	No
KSP	XML	.ksp	koksp-w.4th	Yes	No
CSV	CSV	.csv	csv-w.4th	No	No
HTML	HTML	.html	html-w.4th	No	No

Table 13.9: Spreadsheet formats supported by 4tH

- Open a spreadsheet file;
- Start writing at cell A1;
- If you're finished, go to the next row;
- Write some more cells;
- All done, close the file.

Let's assume you want to write a flat Open Document Sheet. First you have to open the file:

```
include lib/oofods-w.4th
s" mysheet.fods" FODSopen
```

It will return a flag, which is true when an error has occured. No, you don't have to specify it's an output file - you can't read it anyway. It won't return a handle either, the library will take care of that. Note that the moment you've successfully opened the file, it is active. If you want to use another output file, you have to 'USE' it first. And every time you've written to your spreadsheet you have to repeat that procedure. Now, let's handle any errors:

```
abort" Cannot open spreadsheet"
```

Since flat Open Document Sheet supports different sheets, we have to open a worksheet first:

```
s" Sheet1" FODSsheet
```

We're ready to do business now. Let's punch out a few labels. We're starting in cell A1 and continue with cell B1 and C1:

```
s" Label1" FODStype
s" Label2" FODStype
s" Label3" FODStype
```

Let's continue with row 2 and write some data:

```
FODScr
1 FODS.
2 FODS.
3 FODS.
```

We move to row 3 and repeat the same procedure:

```
FODScr
100 FODS.
450 FODS.
325 FODS.
```

Finally, we finish the worksheet and close the file:

```
FODSend
FODSclose
```

	A	*B*	*C*
1	Label1	Label2	Label3
2	1	2	3
3	100	450	325

Table 13.10: Example spreadsheet

That's all! You got a spreadsheet like the one in table 13.10. Writing other formats is just as easy. Take a look at table 13.11 which words are available for which format.

	msxls2-w.4th	*oofods-w.4th*	*msxlms-w.4th*	*koksp-w.4th*	*csv-w.4th*	*html-w.4th*
OPEN	XLSopen	FODSopen	XMLSopen	KSPopen	CSVopen	HTMLopen
sheet	-	FODSsheet	XMLSsheet	KSPsheet	-	-
.	XLS.	FODS.	XMLS.	KSP.	CSV.	HTML.
#	-	FODS#	XMLS#	KSP#	CSV#	HTML#
TYPE	XLStype	FODStype	XMLStype	KSPtype	CSVtype	HTMLtype
CR	XLScr	FODScr	XMLScr	KSPcr	CSVcr	HTMLcr
atxy	XLSatxy	-	-	KSPatxy	-	-
end	-	FODSend	XMLSend	KSPend	-	-
CLOSE	XLSclose	FODSclose	XMLSclose	KSPclose	CSVclose	HTMLclose

Table 13.11: Spreadsheet words

13.26 Writing other tabular formats

There are a few more tabular formats 4tH can write although they cannot be considered "spreadsheets", since there is no major spreadsheet program that actually supports them. However, the API of the library members that support them follows the conventions of those listed in section 13.25.

	json-w.4th	*create-w.4th*
OPEN	JSONopen	CREATopen
.	JSON.	CREAT.
#	JSON#	CREAT#
TYPE	JSONtype	CREATtype
CR	JSONcr	CREATcr
CLOSE	JSONclose	CREATclose

Table 13.12: Other tabular formats

The first format is a new kid in town, called JSON. The second one is able to write a complete 4tH table, the kind you find in section 12.17, which is great for making e.g. program generators.

13.27 Writing LaTeX files

You can write LaTeX files with 4tH very easily. The files it produces can be processed by most popular LaTeX typesetting programs. Although 4tH only supports a tiny LaTeX subset, one could typeset most of this manual with it. If you don't know any LaTeX, don't worry: you don't need to. As a matter of fact, it is very much like HTML.

So let's start by including the library:

```
include lib/latex.4th
```

Next we have to decide what kind of document we want to write. Let's say we want to write an entire book:

```
%texBook
```

If you are a little less ambitious, try writing an article:

```
%texArticle
```

Now you have to give the document a title and claim your rightful place as the author. The flag decides whether you get a "table of contents". If you want one, supply a TRUE flag:

```
%beginTex
s" Hans Bezemer" s" My life with 4tH" true %texTitle
```

Define your first section and you're off:

```
s" Foreword" %subSection
s" First, I want to thank my parents, my teachers," %print
s" my girlfriend, my hairdresser and most of all" %print
s" my dog Snoopy, who never lost faith in me." %print
```

Ok, that was good enough for a Pulitzer Prize. Let's go on with the next section:

```
%endSection

s" The beginning" %subSection
s" It was late at night, I couldn't catch sleep." %print
s" So I decided to create a compiler in order to" %print
s" archieve world peace and love and harmony for" %print
s" all mankind." %print
```

That's enough babble, now let's make a subsection. The difference between making a section and a subsection is that if you want to make a new section you close it with "%endSection". If you want to make a subsection, you don't:

```
s" Basic philosophy" %subSection
s" To make the best product on the market, give the" %print
s" best possible service to our paying customers and" %print
s" squeeze each and every penny out of their pockets." %print
```

Note you have have to balance the "%subSection" and "%endSection" as if they were loop control words. Now let's make a numbered list:

```
%endSection
%endSection

s" Pedigree" %subSection
%enumerate begins
  s" Artic Forth" %item
  s" SpecForth" %item
  s" gForth" %item
%enumerate ends
```

You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

You can even insert source code:

```
%listing begins
  ." : rot >r swap r> swap ;     \ define ROT" cr
  ." 2 3 4 rot                   \ application of ROT" cr
%listing ends
```

Even tables are not much of a problem:

```
%table begins
```

Now you have to decide on a layout. For that you have to compose a string, using a combination of the characters in table 13.13.

CHARACTER	MEANING
l	left justified column
c	centered column
r	right justified column
|	vertical line
||	double vertical line

Table 13.13: LaTeX table format

Let's make this table two left justified columns and one centered column, separated by single vertical lines:

```
s" | l | l | c |" %layout %line
```

The "%line" word draws a single horizontal line. Now we're ready to insert the cells:

```
s" Artic Forth" s" Spectrum" s" 1983" 3 %cells %line
s" FPC"         s" MS-DOS"   s" 1991" 3 %cells %line
s" 4tH"         s" Linux"    s" 1994" 3 %cells %line
```

The "%cells" word needs to know how many cells you're going to insert. Since three strings were supplied, three strings need to be declared. They will be printed in the order you specified, *not* the stack order. Finishing a table is just as uneventful:

```
%table ends
```

If you need links, that's supported too, but it needs the "hyperref" package. If you don't know if that one is installed, enter:

```
kpsewhich hyperref.sty
```

If it is, you may write something like:

```
s" Ok, guys. If you want to know more just" %print
s" google" s" http://www.google.com" %link
s" it! I don't have the time for this." %print
```

You have to enable it, though. All it needs is a '[PRAGMA]' at the beginning of your program:

```
[pragma] UseURLs
```

Ok, that's enough writing for tonight, let's end this:

```
%endSection
%endTex
```

When you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper LaTeX generation. That may not seem like much, but note you've just combined a typesetting language with a powerful programming language, which enables you to handle the most complex document generation tasks with ease.

If you have installed pdfTeX[22] you can make beautiful PDF files out of the box. For those who need to generate office documents, keep an eye on the LaTeX2RTF[23] converter. Both are fully compatible with the LaTeX 4tH generates and easily integrate with 4tH[24].

13.28 Writing RTF files

If you find generating LaTeX is dead easy, you're in for a treat because writing RTF is not much different. There is a catch, though. You can write either RTF *or* LaTeX files, both libraries are incompatible - but don't worry, 4tH will catch that error for you. Like LaTeX, you start with including the library and the kind of document you want to write.

```
include lib/rtf.4th

%RTFbook
%beginRTF
  s" Hans Bezemer" s" My life with 4tH" false %rtfTitle
```

Note the flag is only there for LaTeX compatibility - it doesn't do anything. After that you're in business and if you know how to do a LaTeX file, this should come as no surprise:

[22] http://www.tug.org/applications/pdftex/
[23] http://latex2rtf.sourceforge.net/
[24] Provided your 4tH implementation supports pipes. MS-DOS is not supported.

```
s" Foreword" %subSection
  s" First, I want to thank my parents, my teachers," %print
  s" my girlfriend, my hairdresser and most of all" %print
  s" my dog Snoopy, who never lost faith in me." %println
```

Note RTF is a bit quirky where linebreaks are involved, so it's good practice to terminate the last line of a paragraph with "%println" instead of "%print". If you ever need additional linebreaks, you can add them with "%cr". Now let's make a numbered list:

```
%enumerate begins
  s" Artic Forth" %enumeration
  s" SpecForth" %enumeration
  s" gForth" %enumeration
%enumerate ends
```

Note that the items of a numbered list are terminated by "%enumeration" instead of "%item". You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

Yep, completely identical. Links, listings, sections, quotes, lines, linebreaks, you name it: it's all the same. Tables are only *slightly* different[25]:

```
%table begins
  s" Artic Forth" s" Spectrum" s" 1983" 3 %cells
  s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
  s" 4tH"         s" Linux"    s" 1994" 3 %cells
%table ends
```

Ok, that's enough writing for tonight, let's end this:

```
  %endSection
%endRTF
```

Big surprise.. If you use this library properly it will generate fully *RTF Specification 1.6* compliant files. Like LaTeX, when you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper RTF generation.

13.29 Writing HTML files

If you find generating LaTeX and RTF is dead easy, you're in for a treat because writing HTML is not much different. There is a catch, though. You can write either RTF *or* HTML *or* LaTeX files, all three libraries are incompatible - but don't worry, 4tH will catch that error for you. Like LaTeX and RTF, you start with including the library. After that you go *straight* to the title, after you've defined *one* (and only one) *optional* stylesheet:

[25] Although you can get away with a lot of LaTeX specific stuff, since it's simply ignored. Only `%line` is problematic.

```
include lib/html.4th
%beginHTML
  s" mystyle.css" %styleSheet
  s" Hans Bezemer" s" My life with 4tH" false %htmlTitle
```

Note the flag is only there for LaTeX compatibility - it doesn't do anything. Because if you want to do fancy stuff, you will need to do it with CSS - that's the only way. For arguments sake, let's say you want a very plain-looking table - then define a CSS file `mystyle.css` like this:

```
div.plain table { width: 50%; }
div.plain td { padding: 5px 5px 5px 0; vertical-align: top; }
div.plain th { padding-right: 10px; text-align: left; }
```

Keep that in mind, we'll come back to it later. After that you're in business and if you know how to do a LaTeX file, this should come as no surprise:

```
s" Foreword" %subSection
  s" First, I want to thank my parents, my teachers," %print
  s" my girlfriend, my hairdresser and most of all" %print
  s" my dog Snoopy, who never lost faith in me." %print
```

Now let's make a numbered list:

```
%enumerate begins
  s" Artic Forth" %item
  s" SpecForth" %item
  s" gForth" %item
%enumerate ends
```

Yep. Completely identical. You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

Again, no different. Links, listings, sections, quotes, lines, linebreaks, you name it: it's all the same. Tables are only *slightly* different[26]:

```
%table begins
  s" Artic Forth" s" Spectrum" s" 1983" 3 %heads
  s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
  s" 4tH"         s" Linux"    s" 1994" 3 %cells
%table ends
```

Remember the stylesheet we made? What we actually did was defining a `DIV` class, which is a blocktag as a matter of fact. Within that class you can format whatever tag you like. You're not limited to tables, it can be anything. Now let's apply this class to the table we just defined:

[26] Although you can get away with a lot of LaTeX specific stuff, since it's simply ignored. Only `%line` is problematic.

```
s" plain" %setStyle
  %table begins
    s" Artic Forth" s" Spectrum" s" 1983" 3 %heads
    s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
    s" 4tH"         s" Linux"    s" 1994" 3 %cells
  %table ends
%endStyle
```

You'll see that the format we defined is applied to the table, but after "%endStyle" it returns to the default format. This way you can apply all sorts of styles to different sections of your document. If you defined default styles in your stylesheet they're automatically applied to your document and you don't have to do anything. Needless to say, if you don't use a stylesheet[27] it will be a pretty *plain* looking document altogether.

Ok, that's enough writing for tonight, let's end this:

```
  %endSection
%endHTML
```

Big surprise.. If you use this library properly it will generate fully *HTML 4.01 Transitional* compliant files. Like LaTeX, when you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper HTML generation.

13.30 Converting to XML and HTML

You can't ignore XML and HTML nowadays and 4tH also supports these formats. Let's say you want to write XML. The problem with XML is that there are several characters which aren't allowed in XML files, like apostrophes, quotes and ampersands. How do you know you're dealing with a special character? Well, that's easy:

```
include lib/istype.4th    \ for XML detection
include lib/asciixml.4th  \ we'll need this later
                          \ string with XML characters
s" Here's Johnny! >>> (Johnny)"
bounds ?do                \ let's scan it for XML
  i c@ dup emit is-xml    \ check for XML
  if ."  Yes, it's XML" else ."  Nope" then cr
loop                      \ all done
```

Ok, now we know which characters are XML. But how do we convert them to XML character entities? That is where the second library comes in:

```
include lib/istype.4th    \ for XML detection
include lib/asciixml.4th  \ we'll need this later
                          \ string with XML characters
s" Here's Johnny! >>> (Johnny)"
bounds ?do                \ let's scan it for XML
  i c@ dup is-xml         \ check for XML
  if ASCII>XML type else emit then
loop cr                   \ all done
```

Note that "ASCII>XML" will *always* convert a character to an XML character entity, that's why you need "IS-XML" to determine whether it needs conversion or not.

[27] You will find a stylesheet `html4th.css` in the standard distribution.

If you're converting an ISO-8859-1 or a CP437 encoded document to HTML, things get even more complicated since you'll have to take even more special characters into consideration. Fortunately, 4tH has all the tools you need to get the job done. First, we'll have to determine whether we're dealing with a special character or not. Second, if we *did* find a special character we'll have to convert it.

```
include lib/istype.4th    \ for XML detection
include lib/westhtml.4th  \ the conversion table
include lib/cp437htm.4th  \ the conversion word
                          \ string with HTML characters
: convert-line            ( a n --)
  bounds ?do              \ let's scan it for XML
    i c@ dup is-html      \ check for XML
    if CP437>HTML type else emit then
  loop cr                 \ all done
;
                          \ read and convert a file
: convert-file            ( --)
  begin
    refill                \ read a line
  while                   \ while not end-of-file
    0 parse convert-line  \ parse line and convert it
  repeat
;
( Opening files and writing headers, don't bother..)
```

The library `westhtml.4th` contains the ISO-8859-1 and CP437 conversion tables. Because of its size, you'll have to include it *manually* before you include the `cp437htm.4th` or `i8859htm.4th` libraries. "IS-HTML" works just like "IS-XML", but also reports a HTML character entity when it's not printable or simply outside the ASCII-7 range.

Like "ASCII>XML" the "CP437>HTML" word *always* returns a HTML character entity, even when there is no real equivalent to HTML. E.g. the first CP437 block character renders to "°". If you don't want that you'll either have to maintain an additional table or examine the character entity "CP437>HTML" returns. Needless to say that "ISO8859>HTML" works the same way. The only exception is the NULL character, which is not allowed at all. In that case an empty string is returned.

13.31 Databases

4tH also has a small database package which perfectly blends in with the language. First define a structure. That will be your database buffer:

```
include lib/dbm.4th

struct
  16 +field Firstname
  32 +field Lastname
end-struct /Person

/Person buffer: (Person)
```

Second, create a database file. You have to do this only *once*. If you do it again, it will create another *empty* database file, so take care!

```
s" Persons.dbm" db.create
```

Now add it to the data dictionary. All you have to do is to declare the buffer, its size and the database file. It will return a database handle.

```
(Person) /Person s" Persons.dbm" db.declare to Person
```

After that, you have to tell the datadictionary that you want to use it - or make it current, if you prefer:

```
Person db.use
```

Now we're ready to add records. Simply clear the buffer, fill in the fields and insert the record. When you're done, repeat the operation:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
db.insert

db.clear
s" Chuck"  db.buffer -> Firstname place
s" Moore"  db.buffer -> Lastname  place
db.insert
```

Finally, when you're done simply shut it down:

```
db.close
```

That's all! If you want to use it again, simply define the buffer and declare it to the datadictionary:

```
include lib/dbm.4th

struct
  16 +field Firstname
  32 +field Lastname
end-struct /Person

/Person buffer: (Person)

(Person) /Person s" Persons.dbm" db.declare to Person
```

Of course you can add additional structures and files, the procedure remains the same. If you want to use a table, simply use it:

```
Person db.use
```

Since this is the first time you are accessing the table it is automatically positioned at the first (sequential) record. Just fetch the values:

```
db.buffer -> Firstname count type space
db.buffer -> Lastname count type cr
```

`db.buffer` always points to the current table. Of course, you can also address the buffer directly - that is up to you:

```
(Person) -> Firstname count type space
(Person) -> Lastname count type cr
```

If you explicitly want to go to and fetch the first sequential record, you can also issue:

```
db.first
```

The next (undeleted) sequential record is reached by issuing:

```
db.next
```

If you want to know which record you're at, you can consult `db.rowid`:

```
db.rowid . cr
```

This number is important, because it *never* changes and uniquely identifies a record. If you want to return to a specific record, you can by using this number:

```
1 db.goto
```

Note that whether you access the record random or sequentially, the buffer is *always* updated accordingly. You can also search for a specific string:

```
db.first
db.key Firstname s" Hans" db.find
```

This will return you the first record containing "Hans". In order to find the *next* record simply issue:

```
db.key Firstname s" Hans" db.find
```

If there isn't a second "Hans", the database will return an *End of file* error. You can query the database by using `db.error`:

```
db.error EDB.EOF =
```

You'll find a full list of error codes in the source. You can print the error message by using `db.message`:

```
db.error db.message
```

If you want to delete the current record, simply issue:

```
db.delete
```

Note the record is invalidated at that moment and you will have to navigate away from it. If you want to change the current record, simply make the changes to the buffer and issue `db.update`:

```
s" Chuck" db.buffer -> Firstname place
db.update
```

Finally, you can switch between tables at all times. E.g. you can make changes to a ”Profession” table and return to the ”Person” table at any time. The ”Person” table will be untouched:

```
Profession db.use
( Here we consult or change the ”Profession” table)
Person db.use
( As if we never left it..)
```

Note that `db.close` closes the entire database system, not just the current table. However, if we want to revive it all you have to do is to `db.use` it. You will start at the first record though.

13.32 Indexing a database

Sometimes neither random nor sequential access is sufficient: you want to retrieve the records in order. 4tH supports simple indexes, but don't try to maintain millions of records with it, because are maintained in dynamic memory. If the default allocation is not enough, you can tweak it[28] before including the library file.

Having said that, it is not very difficult. Let's say you already have a database and you want to index it. First you have to decide how many records have to be indexed. Second, which sorting routine do you want to use. You can also use any address based sorting algorithm[29] if you want to, the compiler will figure it out:

```
include lib/shelsort.4th
include lib/dbmidx.4th
```

Now open your database as usual:

```
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use
```

Then you have to index your database. You have to do this only once:

```
8192 {char} db.key Firstname idx.create
abort" Can't make index"
to Pers.First
```

What actually happened is that 4tH stored all the rowids in dynamic memory and sorted them on the field ”Firstname”. In this case you can index a database table with up to 8192 records. The ”{CHAR}” word tells the index to treat it as a string when sorting or searching. The word ”{CELL}” is available for numeric fields, but you can add your own ”field type” if you need to. Access it like this:

```
Pers.First idx.first
```

And it fills the buffer with the very first record. The subsequent record can be reached by issuing:

[28] See section 12.7
[29] See section 12.12.

```
Pers.First idx.next
```

And so on. But how do you know you've reached the last record? Easy, the word "IDX.ERROR" will be set to non-zero. So printing a records in the proper order is as easy as:

```
Pers.First dup idx.first begin
  dup idx.error 0= while
  db.buffer -> Firstname count type cr
  dup idx.next
repeat idx.clear
```

Note you can clear any errors by invoking "IDX.CLEAR". The word "IDX.KEY" will return the address of the key field in the current database buffer. E.g. this is perfectly valid:

```
Pers.First idx.key count type cr
```

In this example it would print the "Firstname". Of course, you will rarely create an index on an already filled table. Normally, you will define your table, define your indexes and off you go. Sure, you can do that too:

```
s" Persons.dbm" db.create
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use

8192 {cell} db.key Firstname idx.init
abort" Cannot create 'First' index" to Pers.First
8192 {char} db.key Lastname idx.init
abort" Cannot create 'Last' index" to Pers.Last
```

Note you *have to* declare indexes when the table is actually "used" at the time, otherwise it will not be properly initialized. The snippet above will create two empty indexes along with an empty database. If creating an index fails, it will return a "true" flag. In any case it will return a reference to the index created, which you will need to pass to almost any word in the index library. Finally, you *don't need* to include a sorting routine, because there won't be any sorting.

13.33 Binding the indexes

Ok, we got two indexes now. How do we update 'em? Well, like this:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
db.insert

Pers.First idx.insert drop
Pers.Last  idx.insert drop
```

It works the same for "DB.DELETE" and "DB.UPDATE": *first* you commit changes to the database buffer, *then* you update all indexes. This is very important. If you don't, some system variables may not be updated properly and you may corrupt your indexes. "IDX.DELETE" does not return a flag. The flag "IDX.INSERT" and "IDX.UPDATE" return indicates that the index now contains duplicate keys. If you're not interested, you can discard it just like we did.

You can also use a dictionary extension. It begins by including a different file:

```
include lib/dbms.4th
```

The second step is to bind the indexes after we've issued the usual red tape. Note the table to bind has to be current:

```
s" Persons.dbm" db.create
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use

8192 {cell} db.key Firstname idx.init
abort" Cannot create 'First' index" to Pers.First
8192 {char} db.key Lastname idx.init
abort" Cannot create 'Last' index" to Pers.Last

Pers.First Pers.Last 2 dbs.bind
abort" Cannot bind indexes"
```

After that you can insert, update and delete records using "DBS.INSERT", "DBS.UPDATE" and "DBS.DELETE" without worrying about updating the indexes. But it comes at a cost. If you're shutting down, you must unbind all the indexes. You can do that one by one:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
dbs.insert

Person dbs.unbind abort" Cannot unbind Person indexes"
```

And repeat that for all tables you've whose indexes you've bound. That's a lot of work, but this way you can mix bound and unbound tables in the same program. You can close them all with one single word, but in that case you have to bind *all* your tables to their indexes - even if they have *no* indexes at all, e.g.:

```
0 dbs.bind abort" Cannot bind table"
```

Now you can use "DBS.RELEASE" to release *all* indexes of *all* tables at once:

```
dbs.release abort" Cannot release indexes"
```

Finding a value in an index is a breeze as well. Again, you have to make sure that the corresponding table is used:

```
s" Hans" Pers.First idx.find 0= abort" Not found"
```

If "IDX.FIND" returns a true flag, your database buffer will contain the first record containing that value. You can use the "IDX.PREVIOUS" word to get any records that are smaller than that value or "IDX.NEXT" to get any records that equal or are greater than that value. If "IDX.FIND" returns false, your value was not found and the contents of the database buffer are undetermined.

If you're not sure about the first value, you can use "IDX.SEARCH". E.g. if you want the first surname beginning with "H", you can issue:

```
s" H" Pers.Last idx.search 0> abort" End of index"
```

If ”IDX.SEARCH” returns a positive value, it is simply not found. If it returns a zero value, the exact value was found in the database and you can use the associated buffer. If it returns a negative value, a value *bigger* than requested was put in the database buffer. If a buffer is valid, you can navigate from there using ”IDX.NEXT” or ”IDX.PREVIOUS”.

Finally, you can search any field using ”DBS.FIND” - whether it is indexed or not. It takes the same parameters as ”DB.FIND” and returns a non-zero value when the record was found. It also returns the index it used or a negative value (”N/A”) when none was available:

```
( find a record by key, abort if not found)
db.key Firstname s" Hans" dbs.find 0= abort" Not found"

( if an index was used, get previous record)
dup N/A = if drop else idx.previous then
```

Of course, it is a shame when you have to rebuild all your indexes when you restart a program. That’s why you can save and reload them. Saving them just takes a single line:

```
Pers.First s" persfrst.idx" idx.save
```

No, it won’t return a flag, but you can examine ”IDX.ERROR” if you want to. Loading takes little code as well:

```
{char} s" persfrst.idx" idx.load abort" Can’t load index"
to Pers.First
```

It returns the same stuff as ”IDX.CREATE” and ”IDX.INIT”, you’re already familiar with that. You don’t have to supply any more information than the type, because the rest of the required information was saved with the index itself. Finally, you will have to close the index. I suggest you close it when you’re closing the table or database itself:

```
dbs.close abort" Cannot release indexes"
Pers.First idx.close abort" Cannot close index"
Pers.Last  idx.close abort" Cannot close index"
```

If it returns a true flag, it failed - which means your dynamic memory is corrupted somehow. Finally, do not forget you can retrieve lots of information on the current status of an index by using the following words:

`idx.error`	Returns the last error encountered;
`idx.size`	Returns the total capacity of an index;
`idx.used`	Returns the number of entries used.

That’s all folks!

13.34 Speech synthesis

4tH can talk! All you need is the ”Festival” speech synthesis package[30] and a small 4tH interface. If you want to imitate old Arnold, this will do:

[30] Homepage: *http://www.cstr.ed.ac.uk/projects/festival/.*

```
include lib/say.4th
s" I'll be back!" say abort" Festival not available"
```

Well, that's cool, isn't it? You can also use the "eSpeak" package[31], which offers even more possiblities. E.g. this will be spoken in a Scottish female voice:

```
include lib/speak.4th
s" I'll be back!" s" en-sc+f1" speak abort" eSpeak not available"
```

Not only English is available, other languages are supported as well. An added advantage is that the package is also available for Windows.

13.35 GUI applications

You can create GUIs with 4tH. Not because 4tH has extensive language bindings, but because it has an interface with GTK-server[32]. After you've successfully installed GTK-server and made sure it's in your path, simply write[33]:

```
include lib/gtkserv.4th
gtk-srv-start
```

GTK-server is now initialized and aways your commands. There are two simple words to do that, "GTK{" and "}GTK". Everything between it is sent to GTK-server. Creating a window is as simple as:

```
gtk{ ." gtk_window_new 0" }gtk
```

Now this command returns a result, the number of the window. Now how can you get it? Very simple: any responses from GTK-server are stored in TIB, which you can process as usual:

```
0 parse number error? abort" Invalid response"
```

Ok, we got the window number now on the stack. Let's give the thing a title:

```
s" My title" gtk{ ." gtk_window_set_title " dup . -rot "type" }gtk
```

""TYPE"" is a word of the interface, which works just like 'TYPE', but places the string within double quotes. Note that after this the window number is still on the stack. You're probably gonna need it..

[31] Homepage: *http://espeak.sourceforge.net/*

[32] Homepage: *http://www.gtk-server.org/*

[33] The following interface only works with Unix-like platforms, except OS/X. This is due to limitations of GTK-server.

Since this is not a tutorial on GTK-server, we simply show you how to shut GTK-server down:

```
gtk-srv-stop
```

This implementation works with named pipes and is pretty fast. However, if it doesn't work for you, you may try an alternative interface which uses an unnamed pipe. Starting it is very similar:

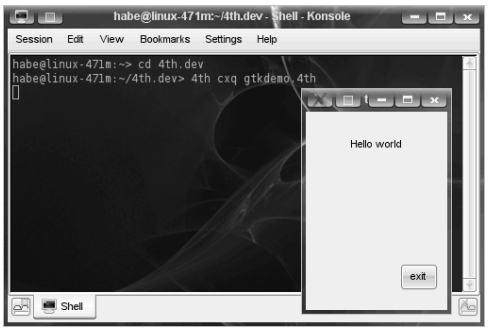

Figure 13.3: GTK demo

```
include lib/gtkipc.4th
gtk-srv-start
gtk{ s" gtk_window_new 0" s>msg }gtk
0 parse number error? abort" Invalid response"
```

The difference is that the GTK command
has to be assembled in memory in its entirety *before* it is sent to GTK-server. For that, three words have been defined: "S>MSG", "'S'>MSG" and "N>MSG". Each one adds a leading blank before the actual payload is appended, so you don't have to worry about that. The first one, "S>MSG" appends a string. Its little brother, "'S'>MSG" adds a quoted string, much like ""TYPE"". Finally, "N>MSG" adds an integer. And yes, any responses are stored in TIB.

So, now let's see how we can give this window a title:

```
s" My title" gtk{ s" gtk_window_set_title" s>msg dup n>msg -rot 's'>msg }gtk
```

Well, that isn't shocking, isn't it? Finally, when you're done shut GTK-server down[34]:

```
gtk-srv-stop
```

Note this interface is *slower*, uses *more* CPU and is slightly more difficult to handle than the default one, so if both interfaces work: use the default.

13.36 Card games

One of the least useful, but funniest libraries may be the playing card library, `cards.4th`, which is dedicated to making card games. It's very easy. After including the library, you take a new deck:

```
include lib/cards.4th
new-deck
```

Like any deck of cards, you have to shuffle it first. You need another library for that, called `shuffle.4th`. It contains two shuffle words, one for a range of cells, the other for a range of characters. The deck is made up of characters. You have to feed it the address of the character array and its length:

[34] This might trigger an error due to 4tHs implementation of `popen()`, but the interface handles it transparently for the programmer.

```
include lib/cards.4th
include lib/shuffle.4th
new-deck
deck /deck cshuffle
```

Ok, now we got a deck of shuffled cards. Now, let's deal:

```
deal
```

This will return an integer, representing the card. If you take modula 13, you will have the value of the card, numbered from 0 to 12 or *Ace* to *King*. If you divide it by 4 you will have the suit of the card, numbered from 0 to 3 or *Diamonds*, *Hearts*, *Clubs* and *Spades:*

```
dup 13 mod . cr
dup 4 / . cr
```

If you want a string representation, simply use "CARD". This will return a string (address/count) containing the card. Simply type it:

```
card type cr
```

Of coure, a deck of cards is limited. If you want to know if it's time for a new deck of cards, call "CARDS-LEFT". This will tell you how many cards are left in the current deck:

```
cards-left . cr
```

Note it's *your* responsibility to track how many cards are left in the deck. If you don't the program will abort with an error! And what when you run out of cards? Well, you simply take a new deck and start all over again. It's simple! This tiny program implements a blackjack game:

```
include lib/shuffle.4th
include lib/cards.4th
include lib/yesorno.4th

: score 13 mod dup if 1+ 10 min else 11 + then ;
: next-card deal dup card space type cr score + ;
: player ." Player: " cr 0 begin next-card s" Hit" yes/no? 0= until cr ;
: dealer ." Dealer: " cr 0 begin next-card dup 16 > until cr ;
: won? dup 21 > if drop ." is bust!" else ." has " 0 .r ." ." then cr ;
: shuffle-deck new-deck deck /deck cshuffle ;
: minijack shuffle-deck player dealer ." Dealer " won? ." Player " won? ;

minijack
```

The beauty is that it is *very* easy to read and understand. All the technical stuff is handled by the `cards.4th` library. So next time you want to make a nice cards game, don't forget about this one!

Chapter 14

Preprocessor libraries

14.1 Introduction

Some people start to frown when you mention preprocessors. But most don't even realize that when you compile C, you're *always* using a preprocessor - and with good reason! Without a preprocessor you really wouldn't want to write C, it would be awkward. C++ started its life as a preprocessor[1]. Several Forth compilers won't even compile without a preprocessor. Face it: there is nothing wrong with preprocessors!

Preprocessors are wonderful tools to hide awful constructs under a silky smooth syntax, enhance your productivity, simplify maintenance and prevent programming errors. 4tH comes with a preprocessor[2] that is more powerful than C's and shares many characteristics with M4[3], like:

- Recursive macro expansion;
- Text replacement;
- Parameter substitution;
- File inclusion;
- Conditional evaluation;
- Arithmetic expressions.

In order to further enhance it, 4tH provides preprocessor libraries, which are in fact predefined macro's. You use a preprocessor library like you would use an ordinary one, e.g.:

```
include 4pp/lib/standard.4pp
```

After you've run the preprocessor, you compile as usual. Note there is no protection against including a preprocessor library twice, but on the other hand it won't be included by (or include) other libraries, so that is pretty academic. If you're running a Unix-like Operating System, you could use `make` to make your life a little easier, e.g. by defining the following rules:

[1] http://www2.research.att.com/~bs/hopl2.pdf
[2] See chapter 18 for more information.
[3] http://en.wikipedia.org/wiki/M4_%28computer_language%29

```
%.c : %.4th
        4th cgq $< $@
%.c : %.hx
        4th lgq $< $@
%.4th : %.4pp
        pp4th -o $@ $<
```

Refer to your system documentation for more information. Finally (and this is very important), *always include preprocessor libraries after including all other libraries*. If you don't, you risk that your macro's are expanded while you're loading 4tH code that wasn't designed for it - which may have adverse effects. You don't want to chase bugs that were never there, don't you?

14.2 Stack instructions

What sets the 4tH preprocessor apart from all other preprocessors is its built-in string stack. And of course, with a stack comes stack instructions. Some, like `@drop`, `@dup`, `@nip` and `@over` are built-in, but some like `@swap` and `@rot` are missing. That is exactly what this library file provides. First of all, you have to include it:

```
include 4pp/lib/standard.4pp
```

Don't forget to use the `.4pp` extension or you'll get an error message, since the preprocessor won't be able to find the file. After that you'll have a full range of stack instructions at your disposal to manipulate the string stack. It allows you to do things like this:

```
include 4pp/lib/standard.4pp
:macro _fac >4> >3> @minus @add @sign @if @drop
       >3> @mul >3> >>> 1 @add <3< _fac ;
:macro __fac >>> 1 @swap >>> 1 @add >>> 2 @max
       <4< >>> 2 <3< _fac ;
:macro factorial >#> __fac @drop <1< #1# ;
." 10! = " factorial 10 . cr
```

Which is nothing less than calculating the factorial of 10 at *compile time*! Now let's break this thing down, how does it work?

- `factorial` puts "10" on the stack and calls `__fac`. After that, it takes the number from the string stack and prints it;
- `__fac` puts "1" on the stack, swaps both values and puts "1" on the stack once more. Then it adds TOS and 2OS[4], puts "3" on the stack and takes the largest value (like 'MAX'). TOS is stored in register ***4*** and the value "2" is stored in register ***3***. Then `_fac` is called.
- `_fac` puts the contents of register ***4*** and ***3*** on the stack, negates TOS and adds TOS and 2OS. If the result of this subtraction is not zero, the contents of register 3 are thrown on the stack once more and multiplied. Finally, the contents of register ***3*** are incremented and a recursive call to `_fac` is made.

[4]"TOS" means: "top of stack" and 2OS: "second of stack". It is a common way to refer to the top two values.

In short, the text `factorial 10` is completely consumed and replaced by the result of the calculation. That's cool, isn't it?

You might ask what's wrong with registers ***1*** and ***2***. Well, nothing.. But most preprocessor libraries use *at least* one register themselves - which is register ***1*** by convention. For the `@rot` macro this library even uses register ***1*** and ***2***. That means if you have stored a value there yourself and you use one of these predefined macro's, it will get clobbered - which is very nasty. In this example, we use `@swap`, which uses just register ***1***, so register ***2***, ***3*** and ***4*** are free to use.

So, save yourself hours of fruitless debugging and check whether a particular macro in a library uses a particular register before using it yourself. Finally, note the preprocessor is powerful enough to calculate the factorial recursively using *only* the string stack. But that's quite complex, you can make your life much easier using `@swap`, `@rot` and registers.

14.3 Coroutines

Calling a subroutine in any computer language is much like launching a program on a single-tasking operating system: you launch it, it takes control and you won't get it back until it relinquishes it. That's quite different from launching a program on a multitasking operating system. You take the control back, give it back to the running program and so on.

Wouldn't it be nice if 4tH could do that too? Well, I may seem like magic, but it can. Take a look at this tiny program:

```
include 4pp/lib/yield.4pp
: fib 0 1 yield begin dup rot + yield again ;
: .fib fib begin dup 100 < while dup . yield repeat drop drop grab ;
: .fibs 0 ?do .fib cr loop ;
5 .fibs
```

So, what happens here? ".FIB" simply prints all Fibonacci numbers smaller than 100. ".FIBS" does that trick a number of times (5 in this case). Finally, "FIB" calculates the Fibonacci numbers. That's all.

It all starts simple enough: ".FIB" calls "FIB", which sets up the Fibonacci sequence. And then it does something peculiar: it *yields*. Yielding is much like calling 'EXIT', but with one small difference: it isn't permanent. As a matter of fact, it returns to the calling word (the "caller"), but it remembers where it was when it gave back the control, much like if it were *calling* the word.

Ok, we're now back at ".FIB" and enter the loop with 'BEGIN', with 0 and 1 at the top of the stack. Sure, 1 is smaller than 100, so we pass the test and print "1". Now we encounter another "YIELD", which *yields* the control *back* to "FIB". We enter the loop there and calculate the next Fibonacci number - and yield again, back to ".FIB".

".FIB" is at 'REPEAT' now and jumps back to 'BEGIN', so we test and print the number before yielding control again. And so on, until we reach the limit and jump to "GRAB". We will see why we need "GRAB" when we examine what's going on.

This is heavy duty stuff. C can't do it without using every dirty trick in the book. We can do it in two lines of preprocessor code, so what's the secret. It's simple, just take a look at section 10.14 and the definition of "YIELD":

```
:macro yield r> execute ;
```

Whenever you yield, you take the return address of the ”caller” from the return stack - just like you would with ’EXIT’. ’EXECUTE’ jumps to it, but *leaves* its own return address on the return stack - like a call would. In fact, the ”caller” and the ”callee” *swap places*, or - to put it in proper perspective - we started out with ”.FIB” calling ”FIB” and now we end up with ”FIB” calling ”.FIB”.

Now, let’s say ”.FIB” encounters ’EXIT’ (which was compiled by ’;’). What do you think is going to happen? It returns to its caller. Which is now ”FIB”.

That’s not what we want. First of all, ”FIB” never ends, so we can wait ’till kingdom come until it relinquishes control. Second, we want to return to the caller of ”.FIB”, which is ”.FIBS”. Note its return address was never changed - it’s just buried beneath the return address of ”FIB”. So we get rid of it and *grab* control. No surprise that this is the definition of ”GRAB”:

```
:macro grab r> drop ;
```

If we had chosen to let ”.FIB” finish ”normally” and relinquish control through ’EXIT’ instead of ”YIELD”, we wouldn’t have needed ”GRAB”, of course. And the same goes for ”FIB”, if it had returned to ”.FIB” by ’EXIT’.

Note this is *extremely* powerful, but also extremely difficult to do right. Whenever control is yielded, the stack diagram has to be exactly what the other coroutine expects. Furthermore, you have to know at any given time what is calling what. For two words, that is feasible - but three or more becomes increasingly difficult. It’s easy to do it wrong, but hard to do it right.

Technically speaking, ”YIELD” does not implement coroutines, but generators. The difference between the two is that true coroutines can choose who they yield to, while generators give you no choice - they’re always bound to one particular word. Finally, Coroutines are *also* supported by an ”ordinary” 4tH library, but that implementation is slower and the optimizer *may* corrupt the return stack if you don’t know what you’re doing.

14.4 Interpretation

By defining just a few macro’s the preprocessor is capable of interpreting and executing a sequence of *user defined* macro’s at runtime. Let’s say you want to define a constant and do some calculations e.g.:

```
include 4pp/lib/interprt.4pp

:macro /line >>> 64 ;
:macro #line >>> 16 ;
:macro @push >#> ;
:macro @* @mul ;
:macro @print @eval ;

[ /line @print ] constant bl_chars
[ #line @print ] constant bl_lines
[ #line /line @* @push 8 @* @print ] buffer: blocks
```

Bottom line: everything between the brackets is interpreted. Of course, there are other (much better) ways to achieve this, but you catch my drift. After you’ve run the preprocessor you will find it has expanded the source to:

```
64 constant bl_chars
16 constant bl_lines
8192 buffer: blocks
```

Note you can use built-in macro's or phony variables as long as you have wrapped them into a macro of your own. This library only uses register *1*, leaving you plenty of registers to play with.

14.5 Closures

Yes, you're reading it correctly: use can use the preprocessor to turn an archaic, low level language like Forth into something very hip and modern. Closures are functions that carry their own, persistent variables along. Some call them "a poor mans objects". Defining a closure is very simple. It is almost like defining a structure:

```
include 4pp/lib/closures.4pp
                                        \ counter creation word
:: counter                              ( n1 n2 addr --)
   vars                                 \ define closure data
     var: start                         \ initial value
     var: inc                           \ increment per call
   end-vars {                           \ only cells may be used!
     this -> inc ! this -> start !      \ initialise closure
     {: this -> inc @ this -> start dup ? +! ;}
   }                                    \ assign default behavior
;
```

You can't use "colon" to define a closure, you have to use the "double colon" notation. Then you start defining the variables of the closure right away. *Note you can only define cell variables*. The rest is quite straight forward: instead of `struct`, you use `vars`. Instead of using `end-struct`, you use `end-vars` to finish your data definition.

Then we start with the initalization routine. Note the *first* parameter of *any* closure routine is *always* its address (just like 'DOES>'), which is put on the stack by "{" and accessible through "THIS". Any additional parameters follow after that. In this case, when creating the closure the initial value of the counter and its increment are set.

Finally, we define the run time behavior - that is the code which is enclosed by "{:" and ";}". Done. Now let's define a few closures of the "COUNTER" type:

```
5 2 sets counter counter1               \ make a counter
```

That's all? Yes, that's all you need to do to create and initialize "COUNTER1". Now let's call it:

```
counter1                                \ 5
counter1                                \ 7
counter1                                \ 9
```

First it will print "5", then "7" and finally "9". Now let's try another "COUNTER:

```
13 3 sets counter counter2              \ make another counter
counter2                                \ 13
counter2                                \ 16
counter2                                \ 19
```

Now that wasn't too hard, was it? Ok, before you start asking stupid questions, consider this: wouldn't you be better off with full fledged object orientation? If you answer this question with "yes", read on. You're in for a surprise.

14.6 Object orientation

We already told you that C++ started its life as a mere preprocessor. In 4tH, all that is required to achieve almost full object orientation is a few humble preprocessor macro's. If you don't like object orientation, that's fine. You take the blue pill, the story ends, you wake up in your bed and believe whatever you want to believe. But if you take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole really goes.

Object orientation comes with an abstract terminology that is incomprehensible to mere humans, but scratch away the shiny surface and you'll understand what it's all about. Stay with me.

14.6.1 Encapsulation

Object orientation defines encapsulation as "a language construct that facilitates the bundling of data with the methods operating on that data". So what are we actually talking about. Let's start with a structure that holds the data. Now add a few fields that hold execution tokens and you're done. That's your "class" definition.

Building an object is nothing more than allocating storage space for your "class". You can reserve that space in static or dynamic memory. That's the easy part. The problem is, the fields for our execution tokens are still empty and when you "instantiate" an object, you want these execution tokens to be initialized. The dirty trick is, you have to do that secretly - in true object oriented tradition.

In 4tH you define a class with a "double colon" definition. A class consists of three parts, the data definition, the "binding" of the "methods" and information hiding, e.g.

```
:: Account
   class                                \ data definition
     ffield:  accountBalance
     virtual: CheckBalance
   end-class {                          \ data initialization

     500 s>f this -> accountBalance f!
     :virtual CheckBalance { this -> accountBalance f@ } ;method
   }                                    \ method binding
   private{ accountBalance }            \ information hiding
;
```

You can only use cells in the data definition. Sorry, you have to store your strings somewhere else. A data definition is almost identical to defining a structure, the only thing is you have to do it within the "double colon" definition. Object orientation also requires a thing called "open recursion", which is nothing more than a "this" or "self" variable, which just holds the address of the "object". That is where the curly braces come in.

The curly braces puts the address (that is always on the top of stack when an object is invoked) on the return stack. Yes, you can still use the return stack, just think of the "THIS" variable between two curly braces as the 'I' in a 'DO..LOOP' and you'll be fine.

The body of a "double colon" definition is actually executed when an object is created, so the floating point "property" (we call it a field) "accountBalance" is set to 500. The colon definition that starts with ":VIRTUAL" is just a ':NONAME' definition, which leaves an execution token. ;METHOD" on its turn pokes this execution token into the equivalent "VIRTUAL:" field of the "class". That's it. We're done, now let's use it.

```
fvariable myBalance                     \ define an FP variable
instance Account myAccount              \ create an Account object
```

The "INSTANCE" word creates an object on the heap. Well, not just that; it also secretly calls "Account", so the object is properly initialized. It also passes the address of the newly created object to "Account", so it knows *what* to initialize. The curly braces in "Account" make sure that "THIS" works.

```
myAccount => CheckBalance myBalance f! \ now pass a message
myBalance f@ f. cr                     \ and store the result
```

You pass a message to an object by using the "=>" operator. You can also use an "adjective"[5]. E.g. you can write the same piece of code like this if you want:

```
myAccount -> virtual CheckBalance myBalance f!
myBalance f@ f. cr
```

If you use '->' on its own you'll just end up with the address of the methods field. When "a message is passed", a "method is invoked". "Passing a message" means calling the object by name and passing all parameters. The object retrieves the *reference* of the actual method and executes it: that's the "invocation of a method". In 4tH, "methods" can be "virtual", which means you can store another execution token in that field at any time.

When a virtual "method" is "invoked", 4tH secretly passes the address of the object, as a matter of fact right on top of the stack. That's why the method itself has curly braces as well - to store the address. Now it starts making sense:

- The opening curly brace puts the address on the return stack;
- "THIS" holds that address, so now "accountBalance" points to the right place;
- The closing curly brace clears the return stack;
- The data stack was uncluttered the whole time.

I told you folks, it's all smoke and mirrors. 4tH also supports static "methods", which means you can't change them later on and they're automatically passed to all subclasses. We could rewrite our previous example like this:

```
:: Account
   class                               \ data definition
     ffield: accountBalance
     method: CheckBalance
   end-class {                         \ data initialization

     500 s>f this -> accountBalance f!
     :method CheckBalance { this -> accountBalance f@ } ;method
   }                                   \ method binding
   private{ accountBalance }           \ information hiding
;
```

You cannot invoke static methods with the "=>" operator, you'll have to use "->" instead. Why? Well, a static "method" is just an ordinary definition, so when you call it, the address of the object is consumed immediately. It is not expanded to an address to the "method" field itself. Worse, static "methods" do not even need a METHOD:" declaration, since they're never changed and consequently don't require any space. It's just a little syntactic sugar that does absolutely *nothing*.

[5] Yes, I know: normal people would call it a *cast*, but allow me to introduce another, completely superfluous concept to the already incomprehesible object orientation vocabulary.

14.6.2 Subtype polymorphism

”Polymorphism” is the ability of ”objects” belonging to different types to respond to ”method” or ”property” calls of the same name, each one according to an appropriate type-specific behavior. Ok, that’s a hard one to swallow. Let’s break it down. We now know that is ”class” is nothing more than a structure.

In section 11.15 we already learned we could *extend* a structure. You may not realize it, but you can pass a pointer to an *extended* structure to a word that takes a pointer to the *unextended* structure and manipulate *any* of the fields they share - without problems. Of course, that includes fields that hold execution tokens. *That* is the basic principle of polymorphism.

Let’s investigate that one a but further. Lots of animals create sounds. Let’s make a ”class” for them:

```
:: Animal
   class
     virtual: Talk
   end-class {}
;
```

That’s all. A ”class” with one ”method”, no ”properties” and no ”binding”. You’d probably call it a ”class with uninitialized methods”, but in object orientation it is usually called an ”interface”. Even if you don’t initialize or bind anything you *have to* use curly braces in order to keep the stack balanced. You can also use ”{}”, which is shorthand for ”{ }”.

Ok, now we have an animal, let’s define a dog:

```
:: Dog
   extends Animal
   end-extends {
     :virtual Talk { ." Woof" cr } ;method
   }
;
```

The ”EXTENDS” word allows you to indicate from which class this type is derived - in this case an ”Animal”. In this case, we don’t add any ”properties” or ”methods”, but we do bind the previously defined method ”Talk”. We can do the same thing for a cat:

```
:: Cat
   extends Animal
   end-extends {
     :virtual Talk { ." Meow" cr } ;method
   }
;
```

If we now create a ”cat” object and a ”dog” object we can make them talk in their own perculiar way:

```
static Cat MyCat
static Dog MyDog

MyCat => Talk
MyDog => Talk
```

That’s what’s called ”dynamic dispatch”: when a method is invoked on an object, the object *itself* determines what code gets executed. You can take that a step further by defining e.g. a figure:

```
:: figure                              \ define an empty class figure
   class                               \ with no properties and two
     virtual: surface                  \ uninitialized methods
     virtual: outline
   end-class {}
;
```

Yes, that's an "interface". Now let's define a rectangle:

```
:: rectangle                           \ define a subtype rectangle
   extends figure                      \ with two specific properties
     field: _width                     \ and a private method
     field: _height
     method: double
   end-extends {
                                       \ now initialize surface and outline
     :method  double { 2* } ;method
     :virtual surface { this -> _width @ this -> _height @ * } ;method
     :virtual outline {
       this -> _width  @ this -> double
       this -> _height @ this -> double +
     } ;method

     private{ double }                 \ make method private
   }
;
```

Note the method "double" is private, which is means it can only be used *within* the class itself. That's why we made it a "static" method, because it's not gonna be redefined any time soon. And let's also define a circle while we're at it:

```
:: circle                              \ define a subtype circle
   extends figure                      \ with one specific property
     field: radius                     \ and a private method
     method: pi*
   end-extends {
                                       \ now initialize surface and outline
     :method  pi* { 103993 33102 */ } ;method
     :virtual surface { this -> radius @ dup * this -> pi* } ;method
     :virtual outline { this -> radius @ 2*    this -> pi* } ;method

     private{ pi* }                    \ make method private
   }
;
```

Both subclasses have very different properties, e.g. "radius" has very little meaning in the context of a rectangle. However, they do share some methods, so if we instantiate them, we can "ask" them to calculate their surface - or their outline:

```
static rectangle MyRectangle           \ make a rectangle instance
static circle    MyCircle              \ make a circle instance

4 MyRectangle -> _width !              \ initialize the rectangle
5 MyRectangle -> _height !
MyRectangle => surface . cr            \ use both methods
MyRectangle => outline . cr

25 MyCircle -> radius !                \ initialize the circle
MyCircle => surface . cr               \ use both methods
MyCircle => outline . cr
```

Nothing to it - if you know what's under the hood.

14.6.3 Inheritance

The final feature true object orientation requires is inheritance. Inheritance is a way to reuse code, which means that classes can inherit attributes and behavior from pre-existing classes called "superclasses". So, how is this done in 4tH? Let's define a traffic light, a simple one:

```
:: two-light                          \ create a two light traffic light
   class
     field:   Red                     \ red color string
     field:   Green                   \ green color string
     field:   State                   \ state of the traffic light
     field:   #lights                 \ number of lights
     field:   Description             \ description of lights
     virtual: Switch                  \ change to the next state
     virtual: Destroy                 \ destroy the object
     method:  Show                    \ show the current state
   end-class {
```

Ok, this is our traffic light, the one you find on pedestrian crossings. It's got a red light and a green light - which totals two lights. It's got a state so we can see what light is on at any time and a few methods. "Switch" makes it change from one state to another, "Show" shows the traffic light and "Destroy" frees the memory the object occupies.

By the way, "Destroy" may walk as a "destructor" and talk as a "destructor", but it's not. A destructor is a method that is *automatically* called when an object is "destroyed". In other words: when the method "Destroy" is invoked, it kills the object. With a "real" destructor, an object is killed and the method is invoked. Not quite the same thing.

Now let's initialize our object:

```
s" Red"   this -> Red   place!        \ save red light color
s" Green" this -> Green place!        \ save green light color
                                      \ we need a table of descriptions
here ['] Green , ['] Red , this -> Description !
2 this -> #lights !                   \ it has two lights
0 this -> State !                     \ set initial state
```

"PLACE!" is a helper word that allocates a bit of dynamic memory for the string and pokes the address in property "Red". The same for "Green", of course. We already concluded that it has two lights, so no surprises there and we set the initial state. "Description" is a pointer to an unnamed table which we use in the next method:

```
 :method Show {                       \ show the current color
    1+ cells this -> Description @ + @c cells this + @ count type cr
 } ;method                            \ note we fix the description offset
```

On a "method" level it just takes an integer, representing the "State" of our traffic light. Next we fetch the item from the table, which is the offset to the appropriate "property" of the "object". So we add it and fetch its contents, which is the address to a string in dynamic memory, so we can type it. Job done. The rest is straight forward:

```
:virtual Switch {
   this -> State dup @ dup            \ get the current state
   this -> Show 1+                    \ show it, go to next state
   this -> #lights @ mod swap !       \ and set it
} ;method                             \ assign it to the Switch method

:default Destroy {                    \ free the green and red light
   this -> Green free? this -> Red free?
   this free abort" Cannot free object"
} ;method
```

The method ”Switch” not only reads property ”State”, it also updates it. In short: it completely encapsulates it. Nothing else, but this class alone, has any business with ”State”, so we can make it private (which we will do later on). Note we use a ":DEFAULT" definition. The difference with a ":VIRTUAL" definition is that we can never refer to the "original" method again once we reassign a virtual method. With ":DEFAULT" you can[6], although you can only do it *once*: multiple defaults for the same virtual method are not allowed.

As a matter of fact, ”:DEFAULT” secretly defines a *static* method and assigns its execution token to the *virtual* method. So, the ”original” static method still has its own symbol in the symbol table, which you can refer to later. The ”Destroy” method we defined simply frees the memory taken by the ”lights” and finally its own. You can do that, because the code that frees the object has no dependencies with the object itself. The variable ”THIS” in the context of ”FREE” only represents the address of the object itself, nothing more.

```
private{ State }
```

We already told you ”State” would become a ”private method”. Now let’s make a heavier duty traffic light:

```
:: three-light                          \ create a three light traffic light
   extends two-light                    \ based on the two light traffic light
     field: Yellow                      \ add the color Yellow
   end-extends {
```

We only need to add the yellow light, don’t we? Sure, we need to add a string for the yellow light:

```
s" Yellow" this -> Yellow place!
```

The configuration may be a bit different, so we have to make ”Description” point to another unnamed table:

```
here ['] Green , ['] Yellow , ['] Red , this -> Description !
```

And of course, we have three lights now:

```
3 this -> #lights !
```

But the rest is just about the same, is it? Not quite. If we destroy our traffic light, we don’t want to leave a yellow light on the sidewalk:

```
:virtual Destroy {                      \ free all lights
   this -> Yellow free?                 \ destroy the yellow color
   this <- Destroy                      \ destroy all previous colors
} ;method                               \ using the static method
```

Yes, we destroy our yellow light first, then we can destroy the rest of the traffic light (including the traffic light itself) by calling its original (default) method. Note the use of ”<-”, which is for ”default” methods what ”=>” is for ”virtual” methods. What we’ve done for ”Destroy” is what is called ”overriding a method”, which means changing its behavior. By the way, a ”default” method has its own ”adjective”, so this is allowed as well:

[6]The reason is that 4tH doesn’t have a virtual method table. See: http://en.wikipedia.org/wiki/Virtual_method_table

```
:virtual Destroy {                      \ free all lights
   this -> Yellow free?                 \ destroy the yellow color
   this -> default Destroy              \ destroy all previous colors
} ;method
```

A two-light traffic light won't behave the same as a three-light traffic light. But we only need to change the bits that are different. The rest is *inherited*. If it's a three-light traffic light, it will behave accordingly. You can see that when you actually use the classes:

```
instance two-light DontWalk             \ define a pedestrian light

." A pedestrian traffic light:" cr cr

DontWalk => Switch                      \ go to the next state four times
DontWalk => Switch
DontWalk => Switch
DontWalk => Switch
DontWalk => Destroy cr                  \ now destroy it

instance three-light TrafficLight       \ define a normal light

." A normal traffic light:" cr cr

TrafficLight => Switch                  \ go to the next state four times
TrafficLight => Switch
TrafficLight => Switch
TrafficLight => Switch
TrafficLight => Destroy cr              \ now destroy it
```

Now you think: "how does he do that?" Well, frankly: by cheating! If you expand the source, you will see what is actually going on:

```
['] two-light ['] three-light (~~parent!)
```

It passes two execution tokens to a secret helper word that invokes the "superclass" initialization before invoking the initialization of the "subclass". If that superclass is a subclass itself, it will call that superclass initialization and so on. When a superclass has finished it relinquishes control to the subclass, so it can do its thing. Finally, the original subclass itself gets the chance to override any "properties" or "methods" it sees fit - and then we're done. Hurray, we got an object - initialized and all.

It's like you make a painting and paint over it a few times. What you see is the final picture. You're unaware of what is hidden under the various layers of paint - but that doesn't mean it's not there or it was never painted. It's that simple.

14.6.4 Determining the type and size of an object

You can determine the type of an object by simply fetching it, e.g.:

```
MyCircle type@
```

You can compare it to a predetermined class by using the "TYPEOF" word:

```
typeof circle
```

You can also determine the superclass of an object:

```
MyCircle parent@
```

And its counterpart on the class side is used like this:

```
parentof circle
```

You can compare these values with the '=' word, nothing special. If you want to go *really* deep and decide whether some interface was incorporated somewhere down the line, you can do that too:

```
MyCircle type@ YourCircle type@ derived?
```

This will leave a 'TRUE' flag if "MyCircle" was derived from the class of the object "YourCircle". Usually you may want to check against a predefined class, like this:

```
MyCircle type@ kindof circle
```

The value all these words return is actually an execution token of the class, so you can use it when you e.g. want to clone objects. It is secretly stored in a hidden field of the class. Of course, what's missing there is the *size* of a class. You can determine that one with the "SIZEOF" word, e.g.

```
sizeof circle
```

If you've allocated objects on the heap, getting their sizes isn't rocket science either:

```
MyCircle allocated
```

For static objects there is no way to determine their sizes (in true 4tH fashion), but that will not be a problem, because you won't be allocating them in a userdefined word.

14.7 This is the end

This is the end of it. If you mastered all we have written about 4tH, you may be just as proficient as we are. Or even better. In the meanwhile you may even have acquired a taste for this strange, but elegant language. If you do, it may be time to step up to Forth, since 4tH does have it limitations. This is in no way an obligation. If you feel comfortable with 4tH, please do stick with it!

If you need any help, you can contact us by sending an email to:

```
The.Beez.speaks@gmail.com
```

Note that we do appreciate *any* input, so if you've written a state of the art application in 4tH, used 4tH in some special way or do have any comments or suggestions on 4tH, we'd like to hear from you! We do also have a web-site:

```
http://thebeez.home.xs4all.nl/4tH/
```

You will find there lots of documentation and news on 4tH. We'd like to thank you for putting so much effort in 4tH. We tried to be of assistance and we hope we did it well!

Part III

Reference guide

Chapter 15

Glossary

This glossary contains all of the word definitions used in version 3.62 of 4tH. The definitions are presented in order of their ASCII sort. Availability of the word in the appropriate ANS-Forth wordset is listed. This does not mean any conformance to the ANS-Forth definition.

PRONUNCIATION: Natural-language pronunciation if it differs from English.

INCLUDE: Following library file provides this word.

COMPILES TO: Describes the transformation of the word to token(s) *without* peephole optimization. Compiler directives will lack this section.

SYNTAX: Describes definition characteristics if non-conformance should lead to a compilation error.

<char>	Character
<string>	String constant, delimited by spaces
<literal>	Expression which compiles to LITERAL (n)
<name>	String of characters, stored in the symboltable
<space>	Space character
<word>	Any valid 4tH word.

COMPILER: Describes special actions the compiler takes when compiling this word.

STACK EFFECTS: Describes the action of the tokens on the parameter stack at runtime. The symbols indicate the order in which input parameters have been placed on the stack. Two dashes indicate the execution point. Any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

n	32 bits signed number	
c	8 bits character	
f	boolean flag	
fam	file access method	
h	file handle (stream)	
d	double number (2 cells)	
sp	stack pointer	Stack Area
x	address of a cell	Variable Area
addr	address of a character	Character Segment
xt	execution token	Code Segment

FLOATING: Describes the action of floating point words on the floating point stack at runtime. The symbols[1] indicate the order in which input parameters have been placed on the stack. Two dashes indicate the execution point. Any parameters left on the floating point stack are listed. In this notation, the top of the stack is to the right.

FORTH: Describes the deviation of 4tH from ANS-Forth and gives suggestions for porting Forth programs.

[1] 'r' stands for real number.

! CORE

PRONUNCIATION: store

COMPILES TO: ! (0)

STACK EFFECTS: n x —

Stores n in the variable at address x.

CORE

PRONUNCIATION: number-sign

COMPILES TO: # (0)

STACK EFFECTS: n1 — n2

FORTH: In Forth a double number is required.

Generate from n1 the next ASCII character which is placed in an output string, stored in PAD. Result n2 is the quotient after the division by BASE, and is remained for further processing. Used between <# and #>.

#! 4TH

SYNTAX: #!<space><string>

The remainder of the line is discarded. This word is used to start a 4tH source program from a Unix type shell. An alias for \.

#> CORE

PRONUNCIATION: number-sign-greater

COMPILES TO: #> (0)

STACK EFFECTS: n1 — addr n2

FORTH: In Forth a double number is required.

Terminates numeric output conversion by dropping n1, leaving the address in PAD and character count n2 suitable for TYPE.

#S CORE

PRONUNCIATION: number-sign-s

COMPILES TO: #S (0)

STACK EFFECTS: n1 — n2

FORTH: In Forth a double number is required.

Generates ASCII text in PAD by the use of # until a zero number n2 results. Used between <# and #>.

#TIB CORE EXT

PRONUNCIATION: number-t-i-b

INCLUDE: obsolete.4th

STACK EFFECTS: — x

X is the address of a cell containing the number of characters in the terminal input buffer (see */TIB*).

' CORE

PRONUNCIATION: tick

COMPILES TO: LITERAL (<argument of symbol>)

SYNTAX: '<space><name>

STACK EFFECTS: — x | xt | n

FORTH: In Forth you can determine the address of variables, constants, etc. In 4tH the contents of the symboltable entry is returned. Of course the token addresses of built-in primitives cannot be determined either. E.g. use

```
: _+ + ; ' _+
```

instead of

```
' +
```

Compile the value contents of the symboltable entry identified as symbol <name> as a literal.

(CORE FILE

PRONUNCIATION: paren

SYNTAX: (<space><string>)

Ignore a comment that will be delimited by a right parenthesis. May occur inside or outside a colon-definition. A blank after the leading parenthesis is required.

(ERROR) 4TH

COMPILES TO: LITERAL (<largest negative integer>)

STACK EFFECTS: — n

Returns 4tHs internal error-flag. This number cannot be printed. Usually -2^31.

) 4TH

COMPILES TO: EQ0 (0)

0BRANCH (<address of THROW>)

LITERAL (<M4ASSERT>)

THROW (0)

STACK EFFECTS: f —

FORTH: Similar constructions are available in GForth and Win32Forth.

If flag f is FALSE, the program will terminate with an error. Its compilation is dependant on the presence of [ASSERT] (see: *[ASSERT]* and *ASSERT(*).

* CORE

PRONUNCIATION: star

COMPILES TO: * (0)

STACK EFFECTS: n1 n2 — n3

Leave the product n3 of two numbers n1 and n2.

*/ CORE

PRONUNCIATION: star-slash

COMPILES TO: */ (0)

STACK EFFECTS: n1 n2 n3 — n4

Leave the ratio n4 = n1*n2/n3.

*/MOD CORE

PRONUNCIATION: star-slash-mod

COMPILES TO: >R (0)

* (0)

R> (0)

/MOD (0)

STACK EFFECTS: n1 n2 n3 — n4 n5

Leave the quotient n5 and remainder n4 of the operation n1*n2/n3.

*CONSTANT 4TH

SYNTAX: <literal><space>*CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for *CONSTANT. The instruction pointer is decremented, actually deleting the literal.

A defining word used to create word <name>. When <name> is later executed, it will multiply the top of the stack with the value of <literal>.

+ CORE

PRONUNCIATION: plus

COMPILES TO: + (0)

STACK EFFECTS: n1 n2 — n3

Leave the sum n3 of n1+n2.

+! CORE

PRONUNCIATION: plus-store

COMPILES TO: +! (0)

STACK EFFECTS: n x —

Add n to the value in variable at address x.

+CONSTANT 4TH

SYNTAX: <literal><space>+CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for +CONSTANT. The instruction pointer is decremented, actually deleting the literal.

A defining word used to create word <name>. When <name> is later executed, it will add the value of <literal> on the top of the stack.

+FIELD FACILITY EXT

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>ENDSTRUCT<space><name>

COMPILER: Take two previous compiled literals. The last literal is added to the first and recompiled. The first literal is the value of a named +CONSTANT. The instruction pointer does not change.

FORTH: This is a Forth-2012 word.

Create a field for STRUCTURE implementations. The created fieldname is an +CONSTANT that memorizes the current offset (see: *+FIELD, /FIELD, STRUCT*, *END-STRUCT*).

+LOOP CORE

PRONUNCIATION: plus-loop

COMPILES TO: +LOOP (<address of matching DO token>)

SYNTAX: DO<space>..<space>+LOOP

STACK EFFECTS: n —

Used in the form DO .. n1 +LOOP. At runtime, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n > 0), or until the new index is less than the limit (n < 0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

+PLACE COMUS

COMPILES TO: COUNT (0)

+ (0)

PLACE (0)

STACK EFFECTS: addr1 n addr2 —

Copies the string at address addr1 with count n to address addr2.

+X/STRING XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: addr1 n1 — addr2 n2

FORTH: This is a Forth-2012 word.

Step forward by one xchar in the buffer defined by addr1 n1. addr2 n2 is the remaining buffer after stepping over the first xchar in the buffer.

, CORE

PRONUNCIATION: comma

COMPILES TO: , (<literal>)

SYNTAX: <literal><space>,

COMPILER: The previously compiled literal is changed into a NOOP instruction. The instruction pointer is not incremented.

FORTH: Forth pops a value from the stack. This is not possible in 4tH. Instead the previously compiled literal has its codefield changed to NOOP.

Store the literal into the next available location.

," COMUS

COMPILES TO: ," (<address of string constant>)

SYNTAX: ,"<space><string>"

FORTH: Compilation characteristics are quite different. 4tH compiles only the address, Forth compiles the entire string.

Compile the string, delimited by " in the String Segment and leave the offset as the address of a string constant (see: @*C*).

,"" 4TH

COMPILES TO: ," (<address of null string constant>)

Compile a null string in the String Segment and leave the offset as the address of a string constant (see: @*C*).

,| 4TH

COMPILES TO: ," (<address of string constant>)

SYNTAX: ,|<space><string>|

Compile the string, delimited by | in the String Segment and leave the offset as the address of a string constant (see: @*C*).

- CORE

PRONUNCIATION: minus

COMPILES TO: - (0)

STACK EFFECTS: n1 n2 — n3

Leave the difference of n1 - n2 in n3.

-> 4TH

COMPILER: The instruction pointer is not incremented. In fact, -> is a dummy.

SYNTAX: <name><space>-><space><name>

Separation between a structure and its member.

-ROT COMUS

COMPILES TO: ROT (0)

ROT (0)

STACK EFFECTS: n1 n2 n3 — n3 n1 n2

Rotate top stack item below the next two items.

-TRAILING STRING

PRONUNCIATION: dash-trailing

COMPILES TO: -TRAILING (0)

STACK EFFECTS: addr n1 — addr n2

Adjusts the character count n1 of a string beginning address to suppress the output of trailing blanks, i.e. the characters from addr+n1 to addr+n2 are blanks.

-TRAILING-GARBAGE XCHAR EXT

INCLUDE: anstools.4th

STACK EFFECTS: addr n1 — addr n2

FORTH: This is a Forth-2012 word.

Examine the last xchar in the string addr n1 - if the encoding is correct and it represents a full xchar, n2 equals n1, otherwise, n2 represents the string without the last (garbled) xchar. -TRAILING-GARBAGE does not change this garbled xchar.

. CORE

PRONUNCIATION: dot

COMPILES TO: . (0)

STACK EFFECTS: n —

Print a number to the current output device, converted according to the numeric BASE. A trailing blank follows.

." CORE

PRONUNCIATION: dot-quote

COMPILES TO: ." (<address of string constant>)

SYNTAX: ."<space><string>"

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device.

.(CORE EXT

PRONUNCIATION: dot-paren

COMPILES TO: ." (<address of string constant>)

SYNTAX: .(<space><string>)

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device. An alias for .".

.R CORE EXT

PRONUNCIATION: dot-r

COMPILES TO: .R (0)

STACK EFFECTS: n1 n2 —

Print the number n1 right aligned in a field whose width is n2 to the current output device. No following blank is printed.

.S TOOLS

PRONUNCIATION: dot-s

INCLUDE: anstools.4th

STACK EFFECTS: —

Copy and display the values currently on the data stack.

.| 4TH

COMPILES TO: ." (<address of string constant>)

SYNTAX: .|<space><string>|

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device.

/ CORE

PRONUNCIATION: slash

COMPILES TO: / (0)

STACK EFFECTS: n1 n2 — n3

Leaves the quotient n3 of n1/n2.

/CELL COMUS

COMPILES TO: LITERAL (<size of a cell>)

STACK EFFECTS: — n

Returns the size of a cell in address units.

/CHAR COMUS

COMPILES TO: LITERAL (<size of char>)

STACK EFFECTS: — n

Returns the size of a character in address units.

/CONSTANT 4TH

SYNTAX: <literal><space>/CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for /CONSTANT. The instruction pointer is decremented, actually deleting the literal.

A defining word used to create word <name>. When <name> is later executed, it will divide the top of the stack by the value of <literal>.

/FIELD 4TH

SYNTAX: STRUCT<space><literal><space>/FIELD<space>END-STRUCT<space><name>

COMPILER: Take two previous compiled literals. The last literal is compared to the first and the larger one of the two is recompiled. The instruction pointer does not change.

Create a field for UNION implementations (see: *+FIELD, /FIELD, STRUCT*, *END-STRUCT*).

/MOD CORE

PRONUNCIATION: slash-mod

COMPILES TO: /MOD (0)

STACK EFFECTS: n1 n2 — n3 n4

Leave the remainder n3 and quotient n4 of n1/n2.

/PAD 4TH

COMPILES TO: LITERAL (<size of PAD>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: /PAD S" /PAD" ENVIRONMENT? DROP ;
```

Returns the size of PAD.

/STRING STRING

PRONUNCIATION: slash-string

COMPILES TO: SWAP (0)

OVER (0)

- (0)

>R (0)

+ (0)

R> (0)

STACK EFFECTS: addr1 n1 n2 — addr2 n3

Adjust the character string at addr1 by n2 characters. The resulting character string, specified by addr2 n3 , begins at addr1 plus n2 characters and is n1 minus n characters long.

/TIB 4TH

COMPILES TO: LITERAL (<size of TIB>)

STACK EFFECTS: — n

Returns the size of the terminal input buffer.

0< CORE

PRONUNCIATION: zero-less

COMPILES TO: 0< (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is less than zero (negative), otherwise leave a FALSE flag in f.

0<> CORE EXT

PRONUNCIATION: zero-not-equals

COMPILES TO: 0<> (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is not equal to zero, otherwise leave a FALSE flag in f.

0= CORE

PRONUNCIATION: zero-equals

COMPILES TO: 0= (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is equal to zero, otherwise leave a FALSE flag in f.

0> CORE EXT

PRONUNCIATION: zero-greater

COMPILES TO: 0> (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is greater than zero (positive), otherwise leave a FALSE flag in f.

1+ CORE

PRONUNCIATION: one-plus

COMPILES TO: +LITERAL (1)

STACK EFFECTS: n — n+1

Increment n by 1.

1- CORE

PRONUNCIATION: one-minus

COMPILES TO: +LITERAL (-1)

STACK EFFECTS: n — n-1

Decrement n by 1.

2! CORE

PRONUNCIATION: two-store

INCLUDE: anscore.4th

STACK EFFECTS: n1 n2 x —

Store the cell pair n1 n2 at x, with n2 at x and n2 at the next consecutive cell.

2* CORE

PRONUNCIATION: two-star

COMPILES TO: *LITERAL (2)

STACK EFFECTS: n — n*2

Multiply n by 2. Performs a left shift.

2/ CORE

PRONUNCIATION: two-slash

COMPILES TO: 2/ (0)

STACK EFFECTS: n — n/2

Divide n by 2. Performs a right shift.

2>R CORE EXT

PRONUNCIATION: two-to-r

COMPILES TO: >R (0)

>R (0)

STACK EFFECTS: n1 n2 —

FORTH: Forth swaps both values before transfering them to the return stack.

Transfer cell pair n1 n2 to the return stack.

2@ CORE

PRONUNCIATION: two-fetch

INCLUDE: anscore.4th

STACK EFFECTS: x — n1 n2

Fetch the cell pair n1 n2 stored at x. n2 is stored at x and n1 at the next consecutive cell.

2DROP CORE

PRONUNCIATION: two-drop

COMPILES TO: DROP (0)

DROP (0)

STACK EFFECTS: n1 n2 —

Drop cell pair n1 n2 from the stack.

2DUP CORE

PRONUNCIATION: two-dupe

COMPILES TO: OVER (0)

OVER (0)

STACK EFFECTS: n1 n2 — n1 n2 n1 n2

Duplicate cell pair n1 n2.

2NIP TOOLBELT

COMPILES TO: ROT (0)

DROP (0)

ROT (0)

DROP (0)

STACK EFFECTS: n1 n2 n3 n4 — n3 n4

Drop the third and fourth items on the stack.

2OVER CORE

PRONUNCIATION: two-over

INCLUDE: anscore.4th

STACK EFFECTS: n1 n2 n3 n4 — n1 n2 n3 n4 n1 n2

Copy cell pair n1 n2 to the top of the stack.

2R> CORE EXT

PRONUNCIATION: two-r-from

COMPILES TO: R> (0)

R> (0)

STACK EFFECTS: — n1 n2

FORTH: Forth swaps both values after transfering them from the return stack.

Transfer cell pair n1 n2 from the return stack.

2R@ CORE EXT

PRONUNCIATION: two-r-fetch

COMPILES TO: R> (0)

I (0)

OVER (0)

>R (0)

STACK EFFECTS: — n1 n2

FORTH: Forth swaps both values after transfering them from the return stack.

Copy cell pair n1 n2 from the return stack.

2ROT DOUBLE EXT

PRONUNCIATION: two-rote

INCLUDE: anscore.4th

STACK EFFECTS: n1 n2 n3 n4 n5 n6 — n3 n4 n5 n6 n1 n2

Rotate the top three cell pairs on the stack bringing cell pair n1 n2 to the top of the stack.

2SWAP CORE

PRONUNCIATION: two-swap

COMPILES TO: ROT (0)

>R (0)

ROT (0)

R> (0)

STACK EFFECTS: n1 n2 n3 n4 — n3 n4 n1 n2

Exchange the top two cell pairs.

4TH# 4TH

COMPILES TO: LITERAL (<4tH version in hexadecimal>)

STACK EFFECTS: — n

Constant containing the 4tH version in hexadecimal.

: CORE

PRONUNCIATION: colon

COMPILES TO: BRANCH (<address of matching ; token>)

SYNTAX: :<space><name>..<space>;

Creates a subroutine defining <name> as equivalent to the following sequence of 4tH word definitions until the next ;.

:NONAME CORE EXT

PRONUNCIATION: colon-no-name

COMPILES TO: LITERAL (<address of next BRANCH>)

BRANCH (<address of matching ; token>)

SYNTAX: :NONAME<space>..<space>;

STACK EFFECTS: — xt

Create an execution token xt and compile the current definition. The execution semantics of xt will be determined by the words compiled into the body of the definition. This definition can be executed later by using xt EXECUTE.

:REDO 4TH

COMPILES TO: BRANCH (<address of matching ; token>)

LITERAL (<original value>) | VARIABLE (<original value>)

SYNTAX: :REDO<space><name><space>..<space>;

Create an subroutine <name> that first pushes the original value of <name> on the stack. The words after <name> determine what the actual execution behavior will be (see: *DOES*>).

:TOKEN 4TH

COMPILES TO: BRANCH (<address of matching ; token>)

SYNTAX: :TOKEN<space><name><space>..<space>;

FORTH: The equivalent in Forth is:

```
:noname ; constant <name>
```

Create an execution token <name> and compile the current definition. The execution semantics of <name> will be determined by the words compiled into the body of the definition. This definition can be executed later by using <name> EXECUTE.

; CORE

PRONUNCIATION: semi-colon

COMPILES TO: EXIT (0)

SYNTAX: See *:*

Terminate a colon definition. At runtime, return to the calling word by popping a token-address from the return stack.

;] COMP.LANG.FORTH

COMPILES TO: EXIT (0)

SYNTAX: See *[:*

FORTH: Several Forths support this word, e.g. gForth, iForth and VFX Forth.

Ends the current nested definition, and resumes compilation to the previous current definition (if any).

< CORE

PRONUNCIATION: less-than

COMPILES TO: < (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is less than n2; otherwise leave a FALSE flag in f.

<# CORE

PRONUNCIATION: less-number-sign

COMPILES TO: <# (0)

FORTH: In Forth a double number is required.

Setup for pictured numeric output formatting in PAD using the words <#, #, #S, SIGN, HOLD, #>.

<> CORE EXT

PRONUNCIATION: not-equals

COMPILES TO: <> (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 does not equal n2; otherwise leave a FALSE flag in f.

<= 4TH

COMPILES TO: > (0)

0= (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is less or equal than n2; otherwise leave a FALSE flag in f.

= CORE

PRONUNCIATION: equals

COMPILES TO: = (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 equals n2; otherwise leave a FALSE flag in f.

> CORE

PRONUNCIATION: greater-than

COMPILES TO: > (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is greater than n2; otherwise leave a FALSE flag in f.

>= 4TH

COMPILES TO: < (0)

0= (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is greater or equal than n2; otherwise leave a FALSE flag in f.

>BODY CORE

PRONUNCIATION: to-body

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

STACK EFFECTS: n — x

FORTH: In Forth, >BODY works with every CREATEd datatype.

n is the ticked value of a VARIABLE, VALUE, DEFER or FILE. >BODY returns its address in the Variable Area.

>FLOAT FLOATING

PRONUNCIATION: to-float

INCLUDE: ansfpio.4th

zenfpio.4th

fpin.4th

STACK EFFECTS: addr n — f

FLOATING: — r

An attempt is made to convert the string specified by addr and n to internal floating-point representation. If the string represents a valid floating-point number in the syntax ”mantissa (with optional exponent)”, its value r and true are returned. If the string does not represent a valid floating-point number only false is returned.

>IN CORE

PRONUNCIATION: to-in

COMPILES TO: LITERAL (<address of >IN>)

STACK EFFECTS: — x

A variable containing the address within the Character Segment from which the next text will be parsed. PARSE uses and moves the value of >IN.

>NUMBER CORE

PRONUNCIATION: to-number

INCLUDE: tonumber.4th

todbl.4th

STACK EFFECTS: n1 a1 n2 — n3 a2 n4

d1 a1 n1 — d2 a2 n4

n3|d2 is the unsigned result of converting the characters within the string specified by a1 n2 into digits, using the number in BASE, and adding each into n1|d1 after multiplying n1|d2 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any + or -, is encountered or the string is entirely converted. a2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. n4 is the number of unconverted characters in the string. An ambiguous condition exists if n3|d2 overflows during the conversion.

>R CORE

PRONUNCIATION: to-r

COMPILES TO: >R (0)

STACK EFFECTS: n —

Remove n from the stack and place it on the return stack. Use should be balanced with R> in the same definition.

>STRING 4TH

COMPILES TO: OVER (0)

PLACE (0)

STACK EFFECTS: a n —

Convert a string identified by address a and length n to a terminated string at address a. Use only on string variables.

? TOOLS

PRONUNCIATION: question

COMPILES TO: @ (0)

. (0)

STACK EFFECTS: x —

Print the value contained in the variable at address x in free format according to the current BASE.

?DO CORE EXT

PRONUNCIATION: question-do

COMPILES TO: ?DO (0)

SYNTAX: ?DO<space>..<space>+LOOP

?DO<space>..<space>LOOP

STACK EFFECTS: n1 n2 —

If n1 is equal to n2, continue execution at LOOP or +LOOP. Otherwise set up loop control parameters with index n2 and limit n1 and continue executing immediately following ?DO. Anything already on the return stack becomes unavailable until the loop control parameters are discarded.

?DUP CORE

PRONUNCIATION: question-dupe

INCLUDE: pickroll.4th

STACK EFFECTS: n — 0 | n n

Duplicate n if it is non-zero.

@ CORE

PRONUNCIATION: fetch

COMPILES TO: @ (0)

STACK EFFECTS: x — n

Leave the contents n of the variable at address x on the stack.

@C CROSS EXT

COMPILES TO: @C (0)

STACK EFFECTS: xt — n | addr

FORTH: In Forth the word @ can also be used to fetch values from the dictionary. Due to 4tHs internal structure this is not possible. This word is officially not a part of ANS-Forth.

Leave the contents n of the parameter field of token address xt on the stack. If n contains an string constant compiled by ," it is copied to the PAD. Its address is returned as addr.

@GOTO 4TH

SYNTAX: @GOTO<space><string>

The remainder of the line is discarded. This word is used to start a 4tH source program from a MS type shell. An alias for \.

ABORT CORE

FORTH: In Forth the behaviour of ABORT is different from QUIT (`-1 THROW`). In 4tH it doesn't really matter which one you use.

An alias for QUIT.

ABORT" CORE

PRONUNCIATION: abort-quote

COMPILES TO: 0BRANCH (<address of QUIT>)

LITERAL (stdout)

USE (0)

." (<address of string constant>)

CR (0)

QUIT (0)

SYNTAX: ABORT"<space><string>"

STACK EFFECTS: n —

FORTH: In Forth the behaviour of ABORT" is different from QUIT (`-2 THROW`).

Remove n from the stack. If any bit of n is not zero, display the string and set the program counter to the end of the program. Effectively quits execution.

ABS CORE

PRONUNCIATION: abs

COMPILES TO: ABS (0)

STACK EFFECTS: n1 — n2

Leave the absolute value of n1 as n2.

ACCEPT CORE

COMPILES TO: ACCEPT (0)

STACK EFFECTS: addr n1 — n2

FORTH: In Forth no null character is appended.

Read n1 characters from the current input device to address addr. If input is read from the terminal CR will terminate the input stream. All other devices will terminate reading when an EOF occurs. In all cases input will end when n1 characters have been read. A null character is added to the end of the input when reading from the keyboard. The number n2 represents the number of characters actually read.

AGAIN CORE EXT

COMPILES TO: BRANCH (<address of the token following BEGIN>)

SYNTAX: BEGIN<space>..<space>AGAIN

At runtime, AGAIN forces execution to return to the corresponding BEGIN. Execution cannot leave this loop. AGAIN is an alias for REPEAT.

AKA 4TH

SYNTAX: AKA<space><word name><space><name>

Create a word <name> with the same compilation and execution semantics as the existing word <word name>. The word <word name> has to be user defined, a built-in constant or a word that compiles to a single token. Flow control words, preprocessor[2] words, defining words and inline macros cannot be 'AKA'ed.

ALIAS 4TH

COMPILES TO: TO (<variable address>)

STACK EFFECTS: xt —

SYNTAX: ALIAS<space><name>

Store xt in the value identified by name. ALIAS is an alias for IS, but does not require a previously defined DEFER.

ALIGN CORE

COMPILER: The instruction pointer is not incremented. In fact, ALIGN is a dummy.

If the dataspace pointer is not aligned, reserve enough space to align it.

ALIGNED CORE

COMPILER: The instruction pointer is not incremented. In fact, ALIGNED is a dummy.

STACK EFFECTS: n — n

n is the first aligned address greater than or equal to n.

ALLOCATE MEMORY

INCLUDE: ansmem.4th

[2] 4tHs internal preprocessor, not the external preprocessor PP4tH. External preprocessor words are not recognized by 4tH at all, so they can't be 'AKA'ed anyway.

memchar.4th

memcell.4th

STACK EFFECTS: n — addr f

Allocate n address units of contiguous data space. The initial content of the allocated space is undefined. If the allocation succeeds, addr is the aligned starting address of the allocated space and f is false. If the operation fails, addr does not represent a valid address and f is true.

AND CORE

COMPILES TO: AND (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical AND of n1 AND n2 as n3.

APP 4TH

COMPILES TO: LITERAL (<application variable>)

STACK EFFECTS: — x

This word returns the variable address x in the Variable Area to an array of application specific variables. If APP equals FIRST no application specific variables have been defined.

APPEND 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method modifier on the stack, signalling that output will be appended. Must be added to another file access modifier. Used in combination with OUTPUT.

ARGN 4TH

COMPILES TO: ARGN (0)

STACK EFFECTS: — n

Returns the number of arguments that have been passed to 4tH (see: *ARGS*).

ARGS 4TH

COMPILES TO: ARGS (0)

STACK EFFECTS: n1 — addr n2

Copies argument n1 to the PAD and leaves address addr and length n2 on the stack (see: *ARGN*).

ARRAY 4TH

SYNTAX: <literal><space>ARRAY<space><name>

COMPILER: The previously compiled literal is taken as an argument for ARRAY. The instruction pointer is decremented, actually deleting the literal.

FORTH: Roughly equivalent to:

```
: ARRAY CREATE CELLS ALLOT ;
```

Allocate <literal> cells of contiguous data space beginning at <name> in the Integer Segment. The initial content of the allocated space is undefined.

ASSERT(4TH

SYNTAX: ASSERT(<space><word>..<word><space>)

FORTH: Similar constructions are available in GForth and Win32For.

Mark the beginning of an assertion. If assertions are disabled all words following upto) are commented out (see: *[ASSERT]* and *)*).

BASE CORE

COMPILES TO: LITERAL (<address of BASE>)

STACK EFFECTS: — x

A variable containing the current number BASE used for input and output.

BEGIN CORE

SYNTAX: BEGIN<space>..<space>AGAIN

BEGIN<space>..<space>WHILE<space>..<space>UNTIL

BEGIN<space>..<space>WHILE<space>..<space>REPEAT

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime begin marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs. Multiple WHILEs may be used.

B/BUF SOURCEFORGE

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: — n

Returns the length of a block.

BIN FILE

INCLUDE: ansfile.4th

STACK EFFECTS: fam1 — fam2

Modify file access method fam1 to additionally select a binary, i.e., not line oriented, file access method, giving access method fam2. Since 4tH does this automatically, BIN is a dummy.

BL CORE

PRONUNCIATION: b-l

COMPILES TO: LITERAL (<ASCII value of space>)

STACK EFFECTS: — c

A constant that leaves the ASCII value for "blank".

BLANK STRING

COMPILES TO: LITERAL (<ASCII value of space)

FILL (0)

STACK EFFECTS: n addr —

If n is greater than zero, store the character value for space in n consecutive character positions beginning at addr.

BLK BLOCK

PRONUNCIATION: b-l-k

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: — x

FORTH: In Forth, a block cannot have the number zero. BLK contains the number of the block being *interpreted*.

x is the address of a cell containing the number of the mass-storage block currently cached. An ambiguous condition exists if a program directly alters the contents of BLK.

BLOCK BLOCK

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: n — addr

Addr is the address of the first character of the block buffer assigned to mass-storage block n. An ambiguous condition exists if u is not an available block number. If block n is already in a block buffer, addr is the address of that block buffer. If block n is not already in memory, unassign the block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage and transfer block n from mass storage into that buffer. a-addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to n.

BOUNDS COMUS

COMPILES TO: OVER (0)

+ (0)

SWAP (0)

STACK EFFECTS: addr n — addr addr+n

Convert a starting value and count into the form required for a DO or ?DO loop.

BUFFER BLOCK

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: n — addr

Addr is the address of the first character of the block buffer assigned to mass-storage block n. An ambiguous condition exists if n is not an available block number. If block n is already in a block buffer, addr is the address of that block buffer. If block n is not already in memory, unassign the block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage. a-addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to n.

BUFFER: CORE EXT

SYNTAX: <literal><space>BUFFER:<space><name>

COMPILER: The previously compiled literal is taken as an argument for BUFFER:. The instruction pointer is decremented, actually deleting the literal.

FORTH: This is a Forth-2012 word.

Allocate <literal> address units of contiguous data space beginning at <name> in the Character Segment. The initial content of the allocated space is undefined.

C! CORE

PRONUNCIATION: c-store

COMPILES TO: C! (0)

STACK EFFECTS: c addr —

Store 8 bits of c at address addr in the Character Segment.

C" 4TH

SYNTAX: C"<space><string>"

COMPILER: All printable characters from the string are added as characters to the String Segment. Nothing is actually compiled.

FORTH: There is a rarely used[3] **CORE EXT** word in ANS-Forth with the same name. This one behaves entirely different.

Reserve space for all the printable characters of the string issued in the String Segment and store those characters in that space.

C, CORE

PRONUNCIATION: c-comma

SYNTAX: <literal><space>C,

COMPILER: The previously compiled literal is added as a character to the String Segment. The instruction pointer is decremented, actually deleting the literal.

FORTH: Forth pops a value from the stack. This is not possible in 4tH.

Reserve space for one character in the String Segment and store char in the space.

C@ CORE

PRONUNCIATION: c-fetch

[3] **A.3.1.3.4 Counted strings** *(abridged)*: "Forth 94 moved toward the consistent use of the 'c-addr u' representation of strings on the stack. The use of the alternate 'address of counted string' stack representation is discouraged. The new word C", added as a porting aid for existing programs, also uses the counted string representation".

COMPILES TO: C@ (0)

STACK EFFECTS: addr — c

Leave the 8 bits contents of Character Segment address addr as c.

CATCH EXCEPTION

COMPILES TO: CATCH (0)

(CATCH) (0)

STACK EFFECTS: xt — n

Push an exception frame on the return stack and execute the execution token xt in such a way that control can be transferred to a point just after CATCH if THROW is executed during the execution of xt (see: *THROW*).

CELL+ CORE

PRONUNCIATION: cell-plus

COMPILES TO: +LITERAL (1)

STACK EFFECTS: x1 — x2

Add the the size of a cell in cells to x1 giving x2.

CELL- COMUS

COMPILES TO: +LITERAL (-1)

STACK EFFECTS: x1 — x2

Subtract the the size of a cell in cells to x1 giving x2.

CELLS CORE

COMPILER: The instruction pointer is not incremented. In fact, CELLS is a dummy.

STACK EFFECTS: n — n

n is the size in cells of n cells.

CFIELD: FACILITY EXT

SYNTAX: CFIELD:<space><name>

COMPILER: The previously compiled literal is taken as an argument for CFIELD: and incremented afterwards. The instruction pointer is left unchanged.

FORTH: This word may only be used in structures that are completely made up out of CHARS. In Forth-2012, there is no such restriction.

The semantics are identical to the execution semantics of the phrase `1 CHARS +FIELD`.

CHAR CORE

PRONUNCIATION: char

COMPILES TO: LITERAL (<ASCII-value of character>)

SYNTAX: CHAR<space><char>

STACK EFFECTS: — c

Compiles the ASCII-value of <char> as a literal. At runtime the value is thrown on the stack.

CHAR+ CORE

PRONUNCIATION: char-plus

COMPILES TO: +LITERAL (1)

STACK EFFECTS: addr1 — addr2

Add the the size of a character in characters to addr1 giving addr2.

CHAR- 4TH

COMPILES TO: +LITERAL (-1)

STACK EFFECTS: addr1 — addr2

Subtract the the size of a character in characters to addr1 giving addr2.

CHAR-BITS 4TH

COMPILES TO: LITERAL (<#bits in a character>)

STACK EFFECTS: — n

n is the number of bits in a character.

CHARS CORE

PRONUNCIATION: chars

COMPILER: The instruction pointer is not incremented. In fact, CHARS is a dummy.

STACK EFFECTS: n — n

FORTH: In 4tH CHARS is a dummy, but it can be used to make a program ANS-compatible.

n is the size in characters of n characters.

CHOP 4TH

COMPILES TO: +LITERAL (-1)

SWAP (0)

+LITERAL (1)

SWAP (0)

STACK EFFECTS: a n — a+1 n-1

Deletes the first character from the string defined by address a and length n.

CIN 4TH

COMPILES TO: ENVIRON (<address of CIN>)

STACK EFFECTS: — n

Identifies the input source.

CLOSE 4TH

COMPILES TO: CLOSE (0)

STACK EFFECTS: h —

CLOSE will close a file or pipe, previously opened by OPEN and release the stream. Depending on the file access method, the terminal will be made the current input-device, otherwise the screen will be made the current output-device.

CLOSE-BLOCKFILE SOURCEFORGE

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: —

Flush and close the current block file.

CLOSE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — f

Close the file identified by handle h. Flag f is the implementation-defined I/O result code.

CMOVE STRING

PRONUNCIATION: c-move

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

FORTH: In Forth there are two words for this operation, CMOVE and CMOVE>. Usage depends on the direction of the move. In 4tH CMOVE is smart, like MOVE.

Move the specified quantity of bytes (n) beginning at address addr1 to addr2.

CMOVE> STRING

PRONUNCIATION: c-move-up

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

An alias for CMOVE (see: *CMOVE*).

COMPARE STRING

INCLUDE: compare.4th

STACK EFFECTS: addr1 n1 addr2 n2 — n3

Compare the string specified by addr1 n1 to the string specified by addr2 n2 . The strings are compared, beginning at the given addresses, character by character, up to the length of the shorter string or until a difference is found. If the two strings are identical, n3 is zero. If the two strings are identical up to the length of the shorter string, n3 is -1 if n1 is less than n2 and 1 otherwise. If the two strings are not identical up to the length of the shorter string, n3 is -1 if the first non-matching character in the string specified by addr1 n1 has a lesser numeric value than the corresponding character in the string specified by addr2 n2 and 1 otherwise.

CONSTANT CORE

SYNTAX: <literal><space>CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for CONSTANT. The instruction pointer is decremented, actually deleting the literal.

FORTH: In Forth, the literal value is popped from the stack. This cannot be done in 4tH.

A defining word used to create word <name>. When <name> is later executed, it will push the value of <literal> on the stack.

COUNT CORE

COMPILES TO: COUNT (0)

STACK EFFECTS: addr1 — addr2 n

FORTH: Programs assuming that the string is a so-called counted string will *not* work. Well-written programs only assume the correct input- and output-parameters.

Leave the Character Segment address addr2 and count n of an ASCIIZ string beginning at Character Segment address addr1. Typically COUNT is followed by TYPE.

COUT 4TH

COMPILES TO: ENVIRON (<address of COUT>)

STACK EFFECTS: — n

Identifies the output source.

CR CORE

PRONUNCIATION: c-r

COMPILES TO: CR (0)

Transmit a carriage return to the selected output-device. The actual sequence sent is OS- and stream-dependant.

CREATE CORE

SYNTAX: CREATE<space><name>

FORTH: In Forth this will create a dictionary header.

Leaves <name> in the symboltable and replace further occurences with LITERAL <xt>. <xt> represents the address in the Code Segment where CREATE was compiled.

CREATE-BLOCKFILE SOURCEFORGE

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: n1 addr n2—

Create a blank block file named in the character string specified by addr and n2, with a size of n1 blocks.

CREATE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n fam — h f

Create the file named in the character string specified by addr and n, and open it with file access method fam. The meaning of values of fam is implementation defined. If a file with the same name already exists, recreate it as an empty file. If the file was successfully created and opened, f is zero, handle h is its identifier, and the file has been positioned to the start of the file. Otherwise, f is the implementation-defined I/O result code and h is undefined.

C| 4TH

SYNTAX: C|<space><string>|

COMPILER: All printable characters from the string are added as characters to the String Segment. Nothing is actually compiled.

Reserve space for all the printable characters of the string issued in the String Segment and store those characters in that space.

D+ DOUBLE

PRONUNCIATION: d-plus

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

Add d1 to d2, giving the sum d3.

D- DOUBLE

PRONUNCIATION: d-minus

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

Subtract d2 from d1, giving the difference d3.

D. DOUBLE

PRONUNCIATION: d-dot

INCLUDE: dbldot.4th

STACK EFFECTS: d —

Display d in free field format.

D.R DOUBLE

PRONUNCIATION: d-dot-r

INCLUDE: dbldot.4th

STACK EFFECTS: d n —

Display d right aligned in a field n characters wide. If the number of characters required to display d is greater than n, all digits are displayed with no leading spaces in a field as wide as necessary.

D0< DOUBLE

PRONUNCIATION: d-zero-less

INCLUDE: ansdbl.4th

STACK EFFECTS: d — f

Flag f is true if and only if d is less than zero.

D0= DOUBLE

PRONUNCIATION: d-zero-equals

INCLUDE: ansdbl.4th

STACK EFFECTS: d — f

Flag f is true if and only if d is equal to zero.

D2* DOUBLE

PRONUNCIATION: d-two-star

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the result of shifting d1 one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

D2/ DOUBLE

PRONUNCIATION: d-two-slash

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the result of shifting d1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

D< DOUBLE

PRONUNCIATION: d-less-than

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — f

Flag f is true if and only if d1 is less than d2.

D= DOUBLE

PRONUNCIATION: d-equals

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — f

Flag f is true if and only if d1 is equal to d2.

D>F FLOATING

PRONUNCIATION: d-to-f

INCLUDE: ansfloat.4th

zentodbl.4th

STACK EFFECTS: d —

FLOATING: — r

r is the floating-point equivalent of d. An ambiguous condition exists if d cannot be precisely represented as a floating-point value.

D>S DOUBLE

PRONUNCIATION: d-to-s

COMPILER: The instruction pointer is not incremented. In fact, D>S is a dummy.

STACK EFFECTS: n — n

Convert the number n to number n with the same numerical value.

DABS DOUBLE

PRONUNCIATION: d-abs

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the absolute value of d1.

DECIMAL CORE

COMPILES TO: RADIX (10)

FORTH: See *HEX*.

Set the numeric conversion BASE for decimal output at runtime.

DEFER CORE EXT

COMPILES TO: LITERAL ((ERROR))

TO (<variable address>)

SYNTAX: DEFER<space><name>

STACK EFFECTS: —

FORTH: This is a Forth-2012 word.

Create a value name which will hold an execution token for a word whose behavior will be determined later and may be varied. The initial value will trigger an error if used before proper assignment.

DEFER! CORE EXT

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

! (0)

STACK EFFECTS: xt x —

FORTH: This is a Forth-2012 word.

Set the vector x to execute xt.

DEFER@ CORE EXT

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

@ (0)

STACK EFFECTS: x — xt

FORTH: This is a Forth-2012 word.

xt is the xt associated with the deferred word corresponding to x.

DELETE-FILE FILE

COMPILES TO: DELETE-FILE (0)

STACK EFFECTS: a n — f

Delete the file named in the character string specified by c-addr u. Return zero on success or true on failure.

DEPTH CORE

COMPILES TO: SP@ (0)

STACK EFFECTS: — n

Returns the number of items on the stack in n, before DEPTH was executed. An alias for SP@.

DMAX DOUBLE

PRONUNCIATION: d-max

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

D3 is the greater of d1 and d2.

DMIN DOUBLE

PRONUNCIATION: d-min

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

D3 is the lesser of d1 and d2.

DNEGATE DOUBLE

PRONUNCIATION: d-negate

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the negation of d1.

DO CORE

COMPILES TO: DO (0)

SYNTAX: DO<space>..<space>+LOOP

DO<space>..<space>LOOP

STACK EFFECTS: n1 n2 —

At runtime DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP or +LOOP the index is altered. Until the new index equals or exceeds the limit, execution loops back to just after DO; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at runtime and may be the result of other operations. Within a loop I will copy the current value of the index on the stack.

DOES> CORE

PRONUNCIATION: does

COMPILES TO: BRANCH (<address of matching ; token>)

LITERAL (<original value>) | VARIABLE (<original value>)

SYNTAX: CREATE|BUFFER:|STRING|CONSTANT|VARIABLE|ARRAY<space>..-<space>DOES><space>..<space>;

FORTH: In ANS-Forth, DOES> is typically combined with CREATE and has no interpretation semantics. The typical use of DOES> in 4tH will lead usually to compilation errors. Some Forth compilers (like gForth) are largely compatible with this 4tH construct.

Replace the execution semantics of the most recent definition with the execution semantics following DOES>. An error condition exists if the most recent definition was not a CREATE, BUFFER:, STRING, CONSTANT, VARIABLE or ARRAY.

DONE 4TH

COMPILES TO: BRANCH (<address of matching DONE, REPEAT or UNTIL token>)

SYNTAX: WHILE<space>..<space>DONE<space>..<space>DONE

FORTH: Equivalent to:

```
: DONE 1 CS-ROLL POSTPONE ELSE 1 CS-ROLL ; IMMEDIATE
```

At runtime DONE executes after the true following WHILE. DONE forces execution to skip over the following false part and resumes execution after the second DONE, which skips execution to just after REPEAT or UNTIL, effectively exiting the structure.

DROP CORE

COMPILES TO: DROP (0)

STACK EFFECTS: n —

Drop the number from the stack.

DU< DOUBLE EXT

PRONUNCIATION: d-u-less

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — f

Flag is true if and only if usigned double d1 is less than unsigned double d2.

DUMP TOOLS

INCLUDE: dump.4th

dumpbase.4th

STACK EFFECTS: addr n —

Display the contents of n consecutive addresses starting at addr.

DUP CORE

PRONUNCIATION: dupe

COMPILES TO: DUP (0)

STACK EFFECTS: n — n n

Duplicate the value on the stack.

ELSE CORE

COMPILES TO: BRANCH (<address of matching THEN token>)

SYNTAX: IF<space>..<space>ELSE<space>..<space>THEN

At runtime ELSE executes after the true following IF. ELSE forces execution to skip over the following false part and resumes execution after the THEN.

EMIT CORE

COMPILES TO: EMIT (0)

STACK EFFECTS: c —

Transmit the ASCII character with code n to the selected output device.

EMPTY-BUFFERS BLOCK EXT

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: —

Unassign all block buffers. Do not transfer the contents of any UPDATEd block buffer to mass storage.

END-STRUCT 4TH

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>END-STRUCT<space><name>

STRUCT<space><literal><space>/FIELD<space>END-STRUCT<space><name>

COMPILER: The previously compiled literal is taken as an argument for END-STRUCT, creating a constant that holds the length of the STRUCT. The instruction pointer is decremented, actually deleting the literal.

FORTH: Similar constructions are available in gForth. +FIELD is part of Forth 2012.

Terminate the definition of a STRUCT. The created structure is an constant that memorizes the size of the structure (see: *+FIELD, /FIELD, STRUCT*).

ENUM 4TH

SYNTAX: <literal><space>ENUM<space><name>

COMPILER: The previously compiled literal is taken as an argument for ENUM and incremented afterwards. The instruction pointer is left unchanged.

FORTH: This word is available in some Forths.

A defining word used to create word <name>. When <name> is later executed, it will push the value of <literal> on the stack.

ENVIRON@ 4TH

COMPILES TO: ENVIRON@ (0)

STACK EFFECTS: addr1 n1 — addr2 n2

addr1 is the address of a character string containing the name of an environment variable and n1 is the string's character count. If successful, it returns address addr2 and count n2 of the contents of the environment variable. On error both address addr2 and count n2 are zero. Note that the contents may be truncated if there is not enough space available in the PAD.

ENVIRONMENT? CORE

PRONUNCIATION: environment-query

INCLUDE: environ.4th

STACK EFFECTS: addr n — -f

addr is the address of a character string and n is the string's character count. The character string should contain a keyword from ANS-Forth environmental queries or the optional word sets to be checked for correspondence with an attribute of the present environment. The system treats the attribute as unknown, the returned flag is false.

EQUATES 4TH

SYNTAX: <variable><space>EQUATES<space><name>

COMPILER: A variable, with an optional literal expression containing an offset, is entered as a new variable in the symbol table.

FORTH: This word is equivalent to CONSTANT in most Forths.

A defining word used to create a static pointer to a variable or an array, which will behave as if it were a normal variable. When <name> is later executed, it will push the address <var> on the stack, so that a fetch or store may access this location.

ERASE CORE EXT

COMPILES TO: LITERAL (0)

FILL (0)

STACK EFFECTS: addr n —

If n is greater than zero, clear all bits in each of n consecutive address units of memory beginning at addr.

ERROR? 4TH

COMPILES TO: LITERAL (<largest negative integer>)

OVER (0)

= (0)

STACK EFFECTS: n — n f

If n equals (ERROR), leave a true flag, otherwise leave a false flag. Determines whether n indicates an error condition. The resulting stack diagram is ANS-Forth compliant.

EVALUATE CORE

INCLUDE: evaluate.4th

STACK EFFECTS: addr n —

FORTH: In Forth, the entire dictionary is available. In 4tH, the only words available are explicitly defined by the program.

Make the string described by addr and n the input buffer and interpret. Other stack effects are due to the words EVALUATEd.

EXECUTE CORE

COMPILES TO: EXECUTE (0)

STACK EFFECTS: xt —

Execute the colon definition whose token-address xt is on the stack. The current token-address is pushed on the returnstack.

EXIT CORE

COMPILES TO: EXIT (0)

When compiled within a colon-definition, terminates execution of that definition at that point. At runtime functionally equivalent to ;.

EXPECT CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: addr n —

Receive a string of at most n-1 characters. The editing functions, if any, that the system performs in order to construct the string of characters are implementation-defined. Input terminates when an implementation-defined line terminator is received or when the string is n-1 characters long. When input terminates the display is maintained in an implementation-defined way. Store the string at addr and its length in SPAN (see *SPAN*).

F! FLOATING

PRONUNCIATION: f-store

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: x —

FLOATING: r —

Store r at address x.

F* FLOATING

PRONUNCIATION: f-star

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Multiply r1 by r2 giving r3.

F** FLOATING EXT

PRONUNCIATION: f-star-star

INCLUDE: falog.4th

zenfalog.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Raise r1 to the power r2, giving the product r3.

F+ FLOATING

PRONUNCIATION: f-plus

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Add r1 to r2 giving the sum r3.

F- FLOATING

PRONUNCIATION: f-minus

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Subtract r2 from r1 giving r3.

F. FLOATING EXT

PRONUNCIATION: f-dot

INCLUDE: ansfpio.4th

zenfpio.4th

fpout.4th

STACK EFFECTS: —

FLOATING: r —

Display, with a trailing space, the top number on the floating-point stack using fixedpoint notation. An ambiguous condition exists if the value of BASE is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

F/ FLOATING

PRONUNCIATION: f-slash

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Divide r1 by r2, giving the quotient r3. An ambiguous condition exists if r2 is zero, or the quotient lies outside of the range of a floating-point number.

F0< FLOATING

PRONUNCIATION: f-zero-less-than

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: — f

FLOATING: r —

Flag f is true if and only if r is less than zero.

F0= FLOATING

PRONUNCIATION: f-zero-equals

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: — f

FLOATING: r —

Flag f is true if and only if r is equal to zero.

F< FLOATING

PRONUNCIATION: f-less-than

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: — f

FLOATING: r1 r2 —

Flag f is true if and only if r1 is less than r2.

F>D FLOATING

PRONUNCIATION: f-to-d

INCLUDE: ansfloat.4th

zentodbl.4th

STACK EFFECTS: — d

FLOATING: r —

Double number d is the double-cell signed-integer equivalent of the integer portion of r. The fractional portion of r is discarded. An ambiguous condition exists if the integer portion of r cannot be precisely represented as a double-cell signed integer.

F>S FLOATING EXT

PRONUNCIATION: f-to-s

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: — n

FLOATING: r —

FORTH: The implementation of `ansfloat.4th` is Forth-2012 compliant.

Number n is the less significant portion of the double word that would be produced by the word F>D applied to the floating-point value r. An ambiguous condition exists if applying F>D to r would result in an ambiguous condition.

F@ FLOATING

PRONUNCIATION: f-fetch

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: x —

FLOATING: — r

Float r is the value stored at address x.

FABS FLOATING EXT

PRONUNCIATION: f-abs

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the absolute value of r1.

FACOS FLOATING EXT

PRONUNCIATION: f-a-cos

INCLUDE: asinacos.4th

zenfasin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the principal radian angle whose cosine is r1. An ambiguous condition exists if | r1 | is greater than one.

FACOSH FLOATING EXT

PRONUNCIATION: f-a-cosh

INCLUDE: fatanh.4th

zenatanh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the floating-point value whose hyperbolic cosine is r1. An ambiguous condition exists if r1 is less than one.

FALIGN FLOATING

PRONUNCIATION: f-align

INCLUDE: ansfloat.4th

STACK EFFECTS: —

FLOATING: —

If the data-space pointer is not float aligned, reserve enough data space to make it so. In 4tH, it is a dummy.

FALIGNED FLOATING

PRONUNCIATION: f-aligned

INCLUDE: ansfloat.4th

STACK EFFECTS: x — x

FLOATING: —

Address x is the first float-aligned address greater than or equal to address x. In 4tH, it is a dummy.

FALOG FLOATING EXT

PRONUNCIATION: f-a-log

INCLUDE: falog.4th

zenfalog.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Raise ten to the power r1, giving r2.

FALSE CORE EXT

COMPILES TO: LITERAL (<false>)

STACK EFFECTS: — -f

Returns a FALSE flag on the stack.

FASIN FLOATING EXT

PRONUNCIATION: f-a-sine

INCLUDE: asinacos.4th

zenfasin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the principal radian angle whose sine is r1. An ambiguous condition exists if | r1 | is greater than one.

FASINH FLOATING EXT

PRONUNCIATION: f-a-cinch

INCLUDE: fatanh.4th

zenatanh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

FORTH: The implementation on `fatanh.4th` is Forth-2012 compliant.

Float r2 is the floating-point value whose hyperbolic sine is r1. An ambiguous condition exists if r1 is less than zero.

FATAN FLOATING EXT

PRONUNCIATION: f-a-tan

INCLUDE: asinacos.4th

zenfasin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the principal radian angle whose tangent is r1.

FATAN2 FLOATING EXT

PRONUNCIATION: f-a-tan-two

INCLUDE: fatan2.4th

zenatan2.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

FORTH: The implementation on `fatan2.4th` is Forth-2012 compliant.

Float r3 is the radian angle whose tangent is r1/r2. An ambiguous condition exists if r1 and r2 are zero.

FATANH FLOATING EXT

PRONUNCIATION: f-a-tan-h

INCLUDE: fatanh.4th

zenatanh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the floating-point value whose hyperbolic tangent is r1. An ambiguous condition exists if r1 is outside the range of -1 to 1.

FCOS FLOATING EXT

PRONUNCIATION: f-cos

INCLUDE: fsinfcos.4th

fsincost.4th

zenfsin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the cosine of the radian angle r1.

FCOSH FLOATING EXT

PRONUNCIATION: f-cosh

INCLUDE: sinhcosh.4th

zenfsinh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the hyperbolic cosine of r1.

FDEPTH FLOATING

PRONUNCIATION: f-depth

INCLUDE: ansfloat.4th

STACK EFFECTS: — n

FLOATING: —

N is the number of values contained on the default separate floating-point stack.

FDROP FLOATING

PRONUNCIATION: f-drop

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: —

FLOATING: r —

Remove r from the floating-point stack.

FDUP FLOATING

PRONUNCIATION: f-dupe

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: —

FLOATING: r — r r

Duplicate r.

FE. FLOATING EXT

PRONUNCIATION: f-e-dot

INCLUDE: ansfpio.4th

fpout.4th

STACK EFFECTS: —

FLOATING: r —

Display, with a trailing space, the top number on the floating-point stack using engineering notation, where the significand is greater than or equal to 1.0 and less than 1000.0 and the decimal exponent is a multiple of three. An ambiguous condition exists if the value of BASE is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

FEXP FLOATING EXT

PRONUNCIATION: f-e-x-p

INCLUDE: fexp.4th

fexpt.4th

zenfexp.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Raise e to the power r1, giving r2.

FEXPM1 FLOATING EXT

PRONUNCIATION: f-e-x-p-m-one

INCLUDE: fexpm1.4th

zenexpm1.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Raise e to the power r1 and subtract one, giving r2.

FIELD: FACILITY EXT

SYNTAX: FIELD:<space><name>

COMPILER: The previously compiled literal is taken as an argument for FIELD: and incremented afterwards. The instruction pointer is left unchanged.

FORTH: This word may only be used in structures that are completely made up out of CELLS. In Forth, there is no such restriction.

The semantics are identical to the execution semantics of the phrase `ALIGNED 1 CELLS +FIELD`.

FILE 4TH

COMPILES TO: LITERAL ((ERROR))

TO (<variable address>)

SYNTAX: FILE<space><name>

STACK EFFECTS: —

Create a value name which will hold a filehandle. The initial value will trigger an error if used before proper assignment.

FILE-POSITION FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — n f

n is the current file position for the file identified by handle h. Flag f is the implementation-defined I/O result code. n is undefined if f is non-zero.

FILE-SIZE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — n f

n is the size, in characters, of the file identified by handle h. Flag f is the implementation-defined I/O result code. This operation does not affect the value returned by FILE-POSITION. n is undefined if f is true.

FILE-STATUS FILE EXT

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 — n2 f

Return the status of the file identified by the character string addr n1. If the file exists, flag f is zero; otherwise flag f is the implementation-defined I/O result code. n2 contains implementation defined information about the file.

FILES 4TH

COMPILES TO: LITERAL (<number of open files>)

STACK_EFFECTS: — n

Returns the maximum number of open streams 4tH can handle. Two of these streams are predefined, STDIN and STDOUT.

FILL CORE

COMPILES TO: FILL (0)

STACK EFFECTS: addr n c —

Fills n bytes in the Character Segment, beginning at address addr, with character c.

FIRST 4TH

COMPILES TO: ENVIRON (<address of FIRST>)

STACK EFFECTS: — x

Leaves the variable address x of the first user-variable. If FIRST is greater than LAST, no user-variables have been defined.

FLN FLOATING EXT

PRONUNCIATION: f-l-n

INCLUDE: flnflog.4th

flnflogb.4th

zenfln.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the natural logarithm of r1. An ambiguous condition exists if r1 is less than or equal to zero.

FLNP1 FLOATING EXT

PRONUNCIATION: f-l-n-p-one

INCLUDE: flnp1.4th

zenflnp1.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the natural logarithm of the quantity r1 plus one. An ambiguous condition exists if r1 is less than or equal to negative one.

FLOAT+ FLOATING

PRONUNCIATION: float-plus

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: x1 — x2

FLOATING: —

Add the size in address units of a floating-point number to address x1, giving address x2.

FLOATS FLOATING

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: n1 — n2

FLOATING: —

Number n2 is the size in address units of n1 floating-point numbers.

FLOG FLOATING EXT

PRONUNCIATION: f-log

INCLUDE: flnflog.4th

flnflogb.4th

zenfln.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the base-ten logarithm of r1. An ambiguous condition exists if r1 is less than or equal to zero.

FLOOR FLOATING

INCLUDE: ansfloat.4th

zenfloor.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Round r1 to an integral value using the ”round toward negative infinity” rule, giving r2.

FLUSH BLOCK

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: —

Perform the function of SAVE-BUFFERS, then unassign the block buffer.

FLUSH-FILE FILE EXT

INCLUDE: ansfile.4th

STACK EFFECTS: h — f

Attempt to force any buffered information written to the file referred to by handle h to be written to mass storage, and the size information for the file to be recorded in the storage directory if changed. If the operation is successful, f is zero. Otherwise, it is an implementation-defined I/O result code.

FM/MOD CORE

PRONUNCIATION: f-m-slash-mod

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 — n2 n3

Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

FMAX FLOATING

PRONUNCIATION: f-max

INCLUDE: ansfloat.4th

zenfmin.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Float r3 is the greater of r1 and r2.

FMIN FLOATING

PRONUNCIATION: f-min

INCLUDE: ansfloat.4th

zenfmin.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r3

Float r3 is the lesser of r1 and r2.

FNEGATE FLOATING

PRONUNCIATION: f-negate

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the negation of r1.

FOVER FLOATING

PRONUNCIATION: f-over

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r1 r2 r1

Place a copy of r1 on top of the floating-point stack.

FREE MEMORY

INCLUDE: ansmem.4th

memchar.4th

memcell.4th

STACK EFFECTS: addr — f

Return the contiguous region of data space indicated by addr to the system for later allocation. addr shall indicate a region of data space that was previously obtained by ALLOCATE or RESIZE. If the operation succeeds, f is false. If the operation fails, f is true.

FROT FLOATING

PRONUNCIATION: f-rot

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: —

FLOATING: r1 r2 r3 — r2 r3 r1

Rotate the top three floating-point stack entries.

FROUND FLOATING

PRONUNCIATION: f-round

INCLUDE: ansfloat.4th

zenround.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Round r1 to an integral value using the ”round to nearest” rule, giving r2.

FS. FLOATING EXT

PRONUNCIATION: f-s-dot

INCLUDE: ansfpio.4th

fpout.4th

STACK EFFECTS: —

FLOATING: r —

Display, with a trailing space, the top number on the floating-point stack in scientific notation. An ambiguous condition exists if the value of BASE is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

FSIN FLOATING EXT

PRONUNCIATION: f-sine

INCLUDE: fsinfcos.4th

fsincost.4th

zenfsin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the sine of the radian angle r1.

FSINCOS FLOATING EXT

PRONUNCIATION: f-sine-cos

INCLUDE: fsinfcos.4th

fsincost.4th

zenfsin.4th

STACK EFFECTS: —

FLOATING: r1 — r2 r3

Float r2 is the sine of the radian angle r1. Float r3 is the cosine of the radian angle r1.

FSINH FLOATING EXT

PRONUNCIATION: f-cinch

INCLUDE: sinhcosh.4th

zenfsinh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the hyperbolic sine of r1.

FSQRT FLOATING EXT

PRONUNCIATION: f-square-root

INCLUDE: ansfloat.4th

zenfsqrt.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the square root of r1. An ambiguous condition exists if r1 is less than zero.

FSWAP FLOATING

PRONUNCIATION: f-swap

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: —

FLOATING: r1 r2 — r2 r1

Exchange the top two floating-point stack items.

FTAN FLOATING EXT

PRONUNCIATION: f-tan

INCLUDE: fsinfcos.4th

fsincost.4th

zenfsin.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the tangent of the radian angle r1. An ambiguous condition exists if cos (r1) is zero.

FTANH FLOATING EXT

PRONUNCIATION: f-tan-h

INCLUDE: sinhcosh.4th

zenfsinh.4th

STACK EFFECTS: —

FLOATING: r1 — r2

Float r2 is the hyperbolic tangent of r1.

FTRUNC FLOATING

INCLUDE: ftrunc.4th

zentrunc.4th

STACK EFFECTS: —

FLOATING: r1 — r2

FORTH: The implementation of `ftrunc.4th` is Forth-2012 compliant.

Round r1 to an integral value using the "round towards zero" rule, giving r2.

F~ FLOATING EXT

PRONUNCIATION: f-proximate

INCLUDE: ansfloat.4th

zenfprox.4th

STACK EFFECTS: — f

FLOATING: r1 r2 r3 —

If r3 is positive, flag is true if the absolute value of (r1 minus r2) is less than r3. If r3 is zero, flag is true if the implementation-dependent encoding of r1 and r2 are exactly identical. If r3 is negative, flag is true if the absolute value of (r1 minus r2) is less than the absolute value of r3 times the sum of the absolute values of r1 and r2.

HERE CORE

COMPILES TO: LITERAL (<token address>)

STACK EFFECTS: — xt

FORTH: Leaves the address of the next available dictionary location. Since 4tH doesn't have a dictionary location, its use is very different.

At runtime, HERE leaves the address xt in the Code Segment where it was compiled.

HEX CORE EXT

COMPILES TO: RADIX (16)

FORTH: In Forth this construction

```
HEX : SOMETIN 16 ;
```

will compile 16 as a hexadecimal number. In 4tH it will simply be compiled and 16 will be compiled as a decimal number. To emulate this construction use

```
[HEX] : SOMETIN 16 ;
```

instead.

Set the numeric conversion BASE for hexadecimal output at runtime.

HI 4TH

COMPILES TO: ENVIRON (<address of HI>)

STACK EFFECTS: — addr

Leaves the address of the last character in the Character Segment.

HIDE 4TH

SYNTAX: HIDE<space><name>

Find <name>, then delete name from the symbol table. Used to create private definitions.

HOLD CORE

COMPILES TO: HOLD (0)

STACK EFFECTS: c —

Used between <# and #> to insert an ASCII character into a pictured numeric output string, e.g. `[HEX] 2E HOLD` will place a decimal point.

HOLDS CORE EXT

INCLUDE: holds.4th

STACK EFFECTS: a n —

FORTH: This is a Forth-2012 word.

Adds a string to the picture numeric output buffer.

I CORE

COMPILES TO: I (0)

STACK EFFECTS: — n

Used with a DO .. LOOP to copy the loop index to the stack. An alias for R.

IF CORE

COMPILES TO: 0BRANCH (<address of matching ELSE|THEN token>)

STACK EFFECTS: f —

SYNTAX: See *ELSE, THEN*

At runtime, IF selects execution based on f. If f is non-zero, execution continues ahead through the true part. If f is zero execution skips till just after ELSE to execute the false part. After each part, execution resumes after THEN.

IMMEDIATE CORE

COMPILER: The instruction pointer is not incremented. In fact, IMMEDIATE is a dummy.

STACK EFFECTS: —

Make the most recent definition an immediate word.

INCLUDE FILE-EXT

SYNTAX: INCLUDE<space><string><space>

COMPILER: The contents of the file are inserted at this position.

FORTH: This is a Forth-2012 word.

An alias for [NEEDS (see: *[NEEDS*).

INPUT 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method on the stack, signalling an operation on an input-device.

INVERT CORE

COMPILES TO: INVERT (0)

STACK EFFECTS: n1 — n2

Leave n1's binary complement as n2. This word is *not* equivalent to 0=.

IS CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: xt —

SYNTAX: IS<space><name>

Store xt in the value identified by name, previously defined by DEFER (see: *DEFER*).

J CORE

COMPILES TO: J (0)

STACK EFFECTS: — n

Used with an embedded DO .. +LOOP to copy the outer loop index to the stack. Copies in fact the third item of the returnstack.

KEY CORE

INCLUDE: key.4th

STACK EFFECTS: — c

FORTH: This implementation is identical to the one in pForth. It requires hitting the <*ENTER*> key, before any character is returned.

Receive one character char, a member of the implementation-defined character set. All standard characters can be received.

KEY? FACILITY

PRONUNCIATION: key-question

INCLUDE: key.4th

STACK EFFECTS: — f

FORTH: This implementation is identical to the one in pForth. KEY? always returns false.

If a character is available, return true. Otherwise, return false.

LAST 4TH

COMPILES TO: ENVIRON (<address of LAST>)

STACK EFFECTS: — x

Leaves the variable address x of the last variable in the Variable Area.

LATEST 4TH

COMPILES TO: The last defined word

Compile the last word that has been added to the symbol table (defined). Is equivalent to RECURSE if used within a colon definition.

LEAVE CORE

COMPILES TO: LEAVE (0)

Force termination of a DO .. +LOOP at the next opportunity by setting the loop index equal to the loop limit. The limit itself remains unchanged, and execution proceeds normally until +LOOP is encountered.

LIST BLOCK EXT

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: n —

Display block n in an implementation-defined format. Store n in SCR.

LO 4TH

COMPILES TO: LITERAL (<TIB+PAD>)

STACK EFFECTS: — addr

Leaves the offset of the first character of the Allocation Area in the Character Segment. If LO is greater than HI, no memory has been allocated.

LOAD BLOCK

INCLUDE: ansblock.4th

multiblk.4th

BEFORE: evaluate.4th

STACK EFFECTS: n —

FORTH: In Forth, the entire dictionary is available. In 4tH, the only words available are explicitly defined by the program.

Save the current input-source specification. Store n in BLK (thus making block n the input source and setting the input buffer to encompass its contents), set >IN to zero, and execute EVALUATE. When the parse area is exhausted, restore the prior input source specification.

LOOP CORE

COMPILES TO: LOOP (<address of matching DO token>)

SYNTAX: DO<space>..<space>+LOOP

Used in the form DO .. LOOP. At runtime, LOOP selectively controls branching back to the corresponding DO based on the loop index and the loop limit. The index is incrementex and compared to the limit. The branch back to DO occurs until the new index is equal to the limit. Upon exiting the loop, the parameters are discarded and execution continues ahead.

LSHIFT CORE

PRONUNCIATION: l-shift

COMPILES TO: SHIFT (0)

STACK EFFECTS: n1 n2 — n3

Performs a logical bit shift on n1. Specifically, SHIFT shifts a number a number of bits, specified in n2, using a logical register shift. An alias for SHIFT.

M* CORE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: n1 n2 — d

d is the signed product of n1 times n2.

M*/ DOUBLE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 n2 — d2

Multiply d1 by n1 producing the triple-cell intermediate result t. Divide t by n2 giving the double-cell quotient d2. An ambiguous condition exists if n2 is zero or negative, or the quotient lies outside of the range of a double-precision signed integer.

M+ DOUBLE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 — d2

Add n1 to d1, giving the sum d2.

MAX CORE

COMPILES TO: MAX (0)

STACK EFFECTS: n1 n2 — n3

Leave n3 as the greater of the two numbers n1 and n2.

MAX-CHAR 4TH

COMPILES TO: LITERAL (<largest unsigned character>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: MAX-CHAR S" MAX-CHAR" ENVIRONMENT? DROP ;
```

Returns the largest unsigned character that 4tH can handle. Usually 255.

MAX-N COMUS

COMPILES TO: LITERAL (<largest positive integer>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: MAX-N S" MAX-N" ENVIRONMENT? DROP ;
```

Returns the largest positive integer that 4tH can handle. Usually 2^31 or 2^63.

MIN CORE

COMPILES TO: MIN (0)

STACK EFFECTS: n1 n2 — n3

Leave n3 as the smaller of the two numbers n1 and n2.

MOD CORE

COMPILES TO: MOD (0)

STACK EFFECTS: n1 n2 — n3

Leave the remainder of n1/n2 with the same sign as n1 in n3.

MOVE CORE

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

Move the specified quantity of bytes (n) beginning at address addr1 to addr2 in the Character Segment.

MS FACILITY EXT

INCLUDE: ansfacil.4th

STACK EFFECTS: n —

FORTH: In Forth, the resolution is significantly higher than between +0 and +1999 ms.

Wait at least u milliseconds.

N>R TOOLS EXT

INCLUDE: ntor.4th

STACK EFFECTS: n1 n2 n3 nx x —

FORTH: This is a Forth-2012 word.

Remove x+1 items from the data stack and store them for later retrieval by NR>. The return stack may be used to store the data. Until this data has been retrieved by NR>: (a) this data will not be overwritten by a subsequent invocation of N>R and (b) a program may not access data placed on the return stack before the invocation of N>R. Be sure to *undercut tail optimization* by applying the [FORCE] directive to *each definition* using this word! (see *[FORCE]*).

NEGATE CORE

COMPILES TO: NEGATE (0)

STACK EFFECTS: n1 — -n1

Leave n1 negated (two's complement).

NIP CORE EXT

COMPILES TO: SWAP (0)

DROP (0)

STACK EFFECTS: n1 n2 — n2

Drop the first item below the top of stack.

NOT COMUS

COMPILES TO: 0= (0)

STACK EFFECTS: n — f

An alias for 0= (see: *0=*).

NR> TOOLS EXT

INCLUDE: ntor.4th

STACK EFFECTS: — n1 n2 n3 nx x

FORTH: This is a Forth-2012 word.

Retrieve the items previously stored by an invocation of N>R. x is the number of items placed on the data stack. It is an ambiguous condition if NR> is used with data not stored by N>R. Be sure to *undercut tail optimization* by applying the [FORCE] directive to *each definition* using this word! (see *[FORCE]*).

NUMBER 4TH

COMPILES TO: NUMBER (0)

STACK EFFECTS: addr n1 — n2

FORTH: Some Forths support this word too, but issue a message on error.

Convert an string at offset addr with length n1 in the Character Segment to number n2. If numeric conversion is not possible (ERROR) is left on the stack.

OCTAL 4TH

COMPILES TO: RADIX (8)

FORTH: See *HEX*.

Set the numeric conversion BASE for octal output at runtime.

OFFSET 4TH

SYNTAX: OFFSET<space><name>

FORTH: Equivalent to:

```
: OFFSET CREATE DOES> SWAP CHARS + C@ ;
```

Leaves <name> in the symboltable and replaces further occurences of <name> with an execution procedure which takes an index from the stack and leaves the character concerned on the stack.

OMIT 4TH

COMPILES TO: OMIT (0)

STACK EFFECTS: c —

Skips all leading delimiters in the Character Segment, using character c as a delimiter.

OPEN 4TH

COMPILES TO: OPEN (0)

STACK EFFECTS: addr n fam — h

OPEN will open the file, which name has been specified by an ASCIIZ string, starting at offset addr in the Character Segment and having length n. Depending on the file access method, the file or pipe will be opened for reading, otherwise for writing. If the file or pipe was succesfully opened it will be connected to a stream and a valid filehandle will be left on the stack. If not, (ERROR) will be left on the stack. Note that OPEN does not connect a stream to a channel (see: *USE*).

OPEN-BLOCKFILE SOURCEFORGE

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: addr n —

Open the block file named in the character string specified by addr n. If successful then set it as the current block file, otherwise abort.

OPEN-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n fam — h f

Open the file named in the character string specified by addr n, with file access method indicated by fam. The meaning of values of fam is implementation defined. If the file is successfully opened, flag f is zero, handle h is its identifier, and the file has been positioned to the start of the file. Otherwise, f is the implementation-defined I/O result code and h is undefined.

OR CORE

COMPILES TO: OR (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical OR in n3 of the numbers n1 and n2.

OUT COMUS

COMPILES TO: LITERAL (<address of OUT>)

STACK EFFECTS: — x

A variable containing the the value that will be returned to the host program.

OUTPUT 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method on the stack, signalling an operation on an output-device.

OVER CORE

COMPILES TO: OVER (0)

STACK EFFECTS: n1 n2 — n1 n2 n1

Copy the second stack value to the top of the stack.

PAD CORE EXT

COMPILES TO: LITERAL (<address of PAD>)

STACK EFFECTS: — addr

Leave the address of the text output buffer.

PARSE CORE EXT

COMPILES TO: PARSE (0)

STACK EFFECTS: c — addr n

Reads a string from the Character Segment, using character c as a delimiter. Leaves the addr/count pair addr n. The resulting string is *not* zero-terminated. If the parse area was empty, the resulting string has a zero length.

PARSE-WORD 4TH

COMPILES TO: DUP (0)

OMIT (0)

PARSE (0)

STACK EFFECTS: c — addr n

FORTH: This is a Forth-2012 word.

Reads a string from the Character Segment, using character c as a delimiter and skipping all leading delimiters. Leaves the addr/count pair addr n. The resulting string is *not* zero-terminated.

PAUSE 4TH

COMPILES TO: PAUSE (0)

STACK_EFFECTS: —

Saves a stackframe, closes all files and quits execution. Leaves the virtual machine in a state where it can resume execution.

PICK CORE EXT

INCLUDE: pickroll.4th

STACK EFFECTS: nu .. n1 n2 u — nu .. n1 n2 nu

Remove u. Copy the nu to the top of the stack.

PIPE 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method modifier on the stack, signalling an operation on a pipe. Must be added to another file access modifier. Used in combination with INPUT and OUTPUT. If an OS does not support pipes, opening a pipe will always fail.

PLACE COMUS

COMPILES TO: PLACE (0)

STACK EFFECTS: addr1 n addr2 —

Copies the string at address addr1 with count n to address addr2.

PRECISION FLOATING EXT

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: — n

FLOATING: —

Return the number of significant digits currently used by F, FE, or FS as n.

QUERY CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: —

Make the user input device the input source. Receive input into the terminal input buffer, replacing any previous contents. Make the result, whose address is returned by TIB, the input buffer. Set >IN to zero.

QUIT CORE

COMPILES TO: QUIT (0)

FORTH: This word has quite another meaning in Forth.

Sets the program counter to the end of the program. Effectively quits execution.

R> CORE

PRONUNCIATION: r-from

COMPILES TO: R> (0)

STACK EFFECTS: — n

Remove the top value from the return stack and leave it on the stack.

R'@ TOOLBELT

COMPILES TO: R> (0)
I (0)
SWAP (0)
>R (0)

STACK EFFECTS: — n

Copy the second return stack item to the stack.

R/O FILE

PRONUNCIATION: r-o

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the read only file access method.

R/W FILE

PRONUNCIATION: r-w

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the read write file access method.

R@ CORE

PRONUNCIATION: r-fetch

COMPILES TO: I (0)

STACK EFFECTS: — n

Copy the top of the return stack to the stack.

READ-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 h — n2 f

Read n1 consecutive characters to addr from the current position of the file identified by handle h. If n1 characters are read without an exception, flag f is zero and n2 is equal to n1. If the end of the file is reached before n1 characters are read, flag f is zero and n2 is the number of characters actually read. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

READ-LINE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 h — n2 f1 f2

Read the next line from the file specified by handle h into memory at the address addr. At most n1 characters are read. Up to two implementation-defined line terminating characters may be read into memory at the end of the line, but are not included in the count n2. The line buffer provided by addr should be at least n1+2 characters long. If the operation succeeded, flag f1 is true and flag f2 is zero. If a line terminator was received before n1 characters were read, then n2 is the number of characters, not including the line terminator, actually read (0 <= n2 <= n1). When n1 = n2 the line terminator has yet to be reached. If the operation is initiated when the value returned by FILE-POSITION is equal to the value returned by FILE-SIZE for the file identified by handle h, flag f1 is false, flag f2 is zero, and n2 is zero. If flag f2 is non-zero, an exception occurred during the operation and f2 is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

RECURSE CORE

COMPILES TO: CALL (<last defined word>)

Compile a call to the current colon-definition inside the current colon-definition. If this word is used outside a colon definition it is undefined.

REFILL CORE EXT FILE EXT

COMPILES TO: REFILL (0)

STACK EFFECTS: — f

Attempt to fill the input buffer from the input source, returning a true flag if successful. When the input source is the user input device, attempt to receive input into the terminal input buffer. When the input source is a text file, attempt to read the next line from the text-input file. If successful, make the result the input buffer, set >IN to zero, and return true. Receipt of a line containing no characters is considered successful. If there is no input available from the current input source, return false.

RENAME-FILE FILE EXT

INCLUDE: ansren.4th

STACK EFFECTS: a1 n1 a2 n2 — f

FORTH: Forth usually does a true rename. 4tH copies the original file to a new file and deletes the old one.

Rename the file named by the character string a1 n1 to the name in the character string a2 n2. It returns true on error and false on success.

REPEAT CORE

COMPILES TO: BRANCH (<address of matching BEGIN>)

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>REPEAT

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime, REPEAT forces an unconditional branch back to just after the corresponding BEGIN. Multiple WHILEs may be used.

REPLACES STRING EXT

INCLUDE: substit.4th

STACK EFFECTS: a1 n2 a2 n2 —

FORTH: This is a Forth-2012 word.

Set the string defined by a1 and n1 as the text to substitute for the substitution named by a2 and n2. If the substitution does not exist it is created. The program may then reuse the buffer a1 n1 without affecting the definition of the substitution. Ambiguous conditions occur as follows:

1. The substitution cannot be created.
2. The name of a substitution contains a delimiter character.

REPOSITION-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: n h — f

Reposition the file identified by handle h to n. Flag f is the implementation-defined I/O result code. An ambiguous condition exists if the file is positioned outside the file boundaries. At the conclusion of the operation, FILE-POSITION returns the value n.

REPRESENT FLOATING

INCLUDE: represnt.4th

STACK EFFECTS: addr n1 — n2 f1 f2

FLOATING: r —

At addr, place the character-string external representation of the significand of the floating-point number r. The size of the buffer identified by addr must be greater than or equal to MAXDIGITS and will *not* be terminated by REPRESENT. Return the decimal-base exponent as n2, the sign as f1 and "valid result" as f2. The character string shall consist of the n1 most significant digits of the significand represented as a decimal fraction with the implied decimal point to the left of the first digit, and the first digit zero only if all digits are zero. The significand is rounded to n1 digits following the "round to nearest" rule; n2 is adjusted, if necessary, to correspond to the rounded magnitude of the significand. If f2 is true then r was in the implementation-defined range of floating-point numbers. If f1 is true then r is negative. An ambiguous condition exists if the value of BASE is not decimal ten.

RESIZE MEMORY

INCLUDE: ansmem.4th

memchar.4th

memcell.4th

STACK EFFECTS: addr1 n — addr2 f

Change the allocation of the contiguous data space starting at the address addr1, previously allocated by ALLOCATE or RESIZE, to n address units. n may be either larger or smaller than the current size of the region. If the operation succeeds, addr2 is the aligned starting address of n address units of allocated memory and f is false. The values contained in the region at addr1 are copied to addr2, up to the minimum size of either of the two regions. If they are the same, the values contained in the region are preserved to the minimum of n or the original size. If addr2 is not the same as addr1, the region of memory at addr1 is returned to the system according to the operation of FREE. If the operation fails, addr2 equals addr1, the region of memory at addr1 is unaffected, and f is true.

RESTORE-INPUT CORE EXT

INCLUDE: evaluate.4th

STACK EFFECTS: n1 n2 a1 n3 h n4 — f

Attempt to restore the input source specification to the state described by n1 through h. Flag is true if the input source specification cannot be so restored.

REWIND 4TH

COMPILES TO: LITERAL (0)

SWAP (0)

SEEK (0)

STACK EFFECTS: h — f

Reposition the file identified by handle h to the beginning of the file. If the operation is successful, FALSE is returned, otherwise TRUE.

ROLL CORE EXT

INCLUDE: pickroll.4th

STACK EFFECTS: nu n1 .. n2 u — n1 .. n2 nu

Remove u. Rotate u+1 items on the top of the stack.

ROT CORE

PRONUNCIATION: rote

COMPILES TO: ROT (0)

STACK EFFECTS: n1 n2 n3 — n2 n3 n1

Rotate the top three values on the stack, bringing the third to the top.

RP@ 4TH

COMPILES TO: RP@ (0)

STACK EFFECTS: — sp

Return the address sp of the stack position of the top of the return stack as it was before RP@ was executed.

RSHIFT CORE

PRONUNCIATION: r-shift

COMPILES TO: NEGATE (0)

SHIFT (0)

STACK EFFECTS: n1 n2 — n3)

Perform a logical right shift of n2 bit-places on n1, giving n2. Put zeroes into the most significant bits vacated by the shift (*depends on implementation*[4]).

S" CORE FILE

PRONUNCIATION: s-quote

COMPILES TO: S" (<address of string constant>)

SYNTAX: S"<space><string>"

STACK EFFECTS: — addr n

Compiles string delimited by " in the String Segment with an execution procedure to move the string to PAD. Leaves the address and the length of the string on the stack.

S>D CORE

PRONUNCIATION: s-to-d

COMPILER: The instruction pointer is not incremented. In fact, S>D is a dummy.

STACK EFFECTS: n — n

Convert the number n to double number n with the same numerical value.

S>F FLOATING EXT

PRONUNCIATION: s-to-f

INCLUDE: ansfloat.4th

zenfloat.4th

STACK EFFECTS: n —

FLOATING: — r

FORTH: The implementation of `ansfloat.4th` is Forth-2012 compliant.

r is the floating-point equivalent of the single-cell value n.

S| 4TH

COMPILES TO: S" (<address of string constant>)

SYNTAX: S|<space><string>|

STACK EFFECTS: — addr n

[4]Some C compilers do an arithmetic shift, leaving the most significant bit set.

Compiles string delimited by | in the String Segment with an execution procedure to move the string to PAD. Leaves the address and the length of the string on the stack.

SAVE-BUFFERS BLOCK

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: —

Transfer the contents of each UPDATEd block buffer to mass storage. Mark the buffer as unmodified.

SAVE-INPUT CORE EXT

INCLUDE: evaluate.4th

STACK EFFECTS: — n1 n2 a1 n3 h n4

n1 through h describe the current state of the input source specification for later use by RESTORE-INPUT.

SCONSTANT 4TH

SYNTAX: S"<space><string>"<space>SCONSTANT<space><name>

COMPILER: The previously compiled string address is taken as an argument for SCONSTANT. The instruction pointer is decremented, actually deleting the string address.

A defining word used to create word <name>. When <name> is later executed, it will push the current address and the length of the string constant on the stack.

SCR BLOCK EXT

PRONUNCIATION: s-c-r

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: — x

x is the address of a cell containing the block number of the block most recently LISTed.

SEARCH STRING

INCLUDE: search.4th

STACK EFFECTS: addr1 n1 addr2 n2 — addr3 n3 f

Search the string specified by addr1 n1 for the string specified by addr2 n2 . If flag is true, a match was found at addr3 with n3 characters remaining. If flag is false there was no match and addr3 is addr1 and n3 is n1.

SEEK 4TH

COMPILES TO: SEEK (0)

STACK EFFECTS: n h — f

Reposition the file identified by handle h to n. If n is positive, TELL returns the value n. If n is negative, the file is repositioned relative to the end of the file. If n equals (ERROR), the file is repositioned to the end of the file. If the operation is successful, FALSE is returned, otherwise TRUE.

SET-PRECISION FLOATING EXT

INCLUDE: ansfloat.4th

zenans.4th

STACK EFFECTS: n —

FLOATING: —

Set the number of significant digits currently used by F, FE, or FS to n.

SHIFT 4TH

COMPILES TO: SHIFT (0)

STACK EFFECTS: n1 n2 — n3

Performs a logical bit shift on n1. Specifically, SHIFT shifts a number a number of bits, specified in n2, using a logical register shift.

SIGN CORE

COMPILES TO: SIGN (0)

STACK EFFECTS: n1 n2 — n2

Stores an ASCII '-' sign just before the converted numeric output string in PAD when n1 is negative. n1 is discarded, but n2 is maintained. Must be used between <# and #>.

SM/REM CORE

PRONUNCIATION: s-m-slash-rem

INCLUDE: mixed.4th

STACK EFFECTS: d n1 — n2 n3

Divide d by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

SMOVE 4TH

COMPILES TO: SMOVE (0)

STACK EFFECTS: x1 x2 n —

FORTH: Equivalent to:

```
: SMOVE CELLS MOVE ;
```

Move the specified quantity of cells (n) beginning at address x1 to x2.

SOURCE CORE

COMPILES TO: LITERAL (<address of TIB variable>)

@ (0)

LITERAL (<address of TIB-size variable>)

@ (0)

STACK EFFECTS: — addr n

addr is the address of, and n is the number of characters in, the currently used TIB.

SOURCE! SOURCEFORGE

COMPILES TO: LITERAL (<address of TIB-size variable>)

! (0)

LITERAL (<address of TIB variable>)

! (0)

STACK EFFECTS: addr n —

FORTH: In Forth, >IN is set to zero. In 4tH this is left up to the application programmer.

Make the string described by c-addr and u the current input buffer. A program is allowed to refill the input buffer without restoring the original input source; upon a refill, the system shall accept the new portion of text to the current refill buffer and make it the input buffer.

SOURCE-ID CORE EXT FILE

PRONUNCIATION: source-i-d

COMPILES TO: ENVIRON (<address of CIN>)

STACK EFFECTS: — n

Identifies the input source.

SP@ 4TH

COMPILES TO: SP@ (0)

STACK EFFECTS: — sp

Return the address sp of the stack position of the top of the stack as it was before SP@ was executed.

SPACE CORE

COMPILES TO: LITERAL (<ASCII value of space>)

EMIT (0)

Transmit an ASCII blank to the current output device.

SPACES CORE

COMPILES TO: SPACES (0)

STACK EFFECTS: n —

Transmit n ASCII blanks to the current output device.

SPAN CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: — x

X is the address of a cell containing the count of characters stored by the last execution of EXPECT (see *EXPECT*).

STACK-CELLS 4TH

COMPILES TO: LITERAL (<number of integers>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: STACK-CELLS S" STACK-CELLS" ENVIRONMENT? DROP
;
```

Returns the number of integers that the Stack Area can contain. Both stacks share the Stack Area.

STDIN 4TH

COMPILES TO: LITERAL (<address of stream>)

STACK EFFECTS: — h

Leaves a filehandle on the stack associated with the standard keyboard input device. This stream cannot be closed.

STDOUT 4TH

COMPILES TO: LITERAL (<address of stream>)

STACK EFFECTS: — h

Leaves a filehandle on the stack associated with the standard screen output device. This stream cannot be closed.

STRING 4TH

SYNTAX: <literal><space>STRING<space><name>

COMPILER: The previously compiled literal is taken as an argument for STRING. The instruction pointer is decremented, actually deleting the literal.

FORTH: This word is 4tH specific. Roughly equivalent to:

```
: STRING CREATE CHARS ALLOT ;
```

Allocate <literal> characters of contiguous data space beginning at <name> in the Character Segment. The initial content of the allocated space is undefined.

STRUCT 4TH

COMPILES TO: LITERAL (0)

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>END-STRUCT<space><name>

STRUCT<space><literal><space>/FIELD<space>END-STRUCT<space><name>

STACK EFFECTS: — n

FORTH: Similar constructions are available in GForth. +FIELD is part of Forth 2012.

A constant, which initiates a STRUCT definition (see: *+FIELD, /FIELD, END-STRUCT*).

SUBSTITUTE STRING EXT

INCLUDE: substit.4th

STACK EFFECTS: a1 n1 a2 n2 – a2 n3 n4

FORTH: This is a Forth-2012 word.

Perform substitution on the string at a1 and n1 placing the result at string a2 and n2, returning a2 and n3, the length of the resulting string. An ambiguous condition occurs if the resulting string will not fit into a2 n2 or if a2 is the same as a1. The return value n4 is positive (0..+n) on success and indicates the number of substitutions made. A negative value for n4 indicates that an error occurred, leaving a2 and n3 undefined. Substitution occurs from the start of a1 and n1 in one pass and is non-recursive. When a substitution name surrounded by '%' delimiters is encountered by SUBSTITUTE, the following occurs:

1. If the name is null, a single delimiter character is substituted, i.e. %% is replaced by %.
2. If the name is a valid substitution name, the leading and trailing delimiter characters and the enclosed substitution name are replaced by the substitution text.
3. If the name is not a valid substitution name, the name with leading and trailing delimiters is passed unchanged to the output.

SWAP CORE

COMPILES TO: SWAP (0)

STACK EFFECTS: n1 n2 — n2 n1

Exchange the top two values on the stack.

SYNC 4TH

COMPILES TO: SYNC (0)

STACK EFFECTS: —

Attempt to force any buffered information written to the device referred to by the output channel to be written.

TABLE 4TH

SYNTAX: TABLE<space><name>

FORTH: Available in some Forths.

Leaves <name> in the symboltable and replace further occurences with LITERAL <xt>. <xt> represents the address in the Code Segment where TABLE was compiled. An alias for CREATE.

TAG 4TH

SYNTAX: TAG<space><name1><space><name2>

FORTH: This word is 4tH specific. Roughly equivalent to:

```
: TAG ' >BODY 1 CELLS - CREATE , DOES> DUP @ -
;
```

Create a CONSTANT <name2> which contains the difference in characters between the current address of the String Segment and previously defined OFFSET <name1>. This constant can be used as an index to <name1> to retrieve any binary strings defined directly after the definition of <name2>.

TELL 4TH

COMPILES TO: TELL (0)

STACK EFFECTS: h — n

n is the current file position for the file identified by handle h.

TH COMUS

COMPILES TO: + (0)

STACK EFFECTS: x1 n — x2

FORTH: This word is not part of ANS-Forth or Forth-79, but can be found in other Forths. It can be very handy when porting 4tH programs. Just define TH as:

```
: TH CELLS + ;
```

When you're using a construction like:

```
VAR 2 TH
```

In both 4tH and Forth the third element will be referenced. The use of TH to reference an element of a string in the Character Segment is allowed in 4tH, but the resulting source cannot be ported to Forth.

Used to reference an element in an array of integers. Will return the address of the n-th element in array x1 as x2. An alias for +.

THEN CORE

SYNTAX: IF<space>..<space>ELSE<space>..<space>THEN

At runtime THEN serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure.

THROW EXCEPTION

COMPILES TO: THROW (0)

STACK EFFECTS: n —

FORTH: The values of THROW are not conforming the ANS-Forth standard.

If n is non-zero, pop the topmost exception frame from the return stack, along with everything beyond that frame. Then adjust the return- and datastacks so they are the same as the depths saved in the exception frame, put n on top of the data stack, and transfer control to a point just after the CATCH that pushed that exception frame (see: *CATCH*).

TIB CORE EXT

PRONUNCIATION: t-i-b

COMPILES TO: LITERAL (<address of Terminal Input Buffer>)

STACK EFFECTS: — addr

FORTH: In Forth this is a variable. However, it is unlikely you'll ever find a program which assigns another value to it.

A constant which leaves the address of the Terminal Input Buffer on the stack.

TIME 4TH

COMPILES TO: TIME (0)

STACK EFFECTS: — n

Returns the number of seconds since January 1st, 1970.

TIME&DATE FACILITY

PRONUNCIATION: time-and-date

INCLUDE: ansfacil.4th

STACK EFFECTS: — n1 n2 n3 n4 n5 n6

Return the current time and date. n1 is the second {0...59}, n2 is the minute {0...59}, n3 is the hour {0...23}, n4 is the day {1...31}, n5 is the month {1...12}, and n6 is the year (e.g., 1991).

TO CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: n —

SYNTAX: TO<space><name>

Store n in the value identified by name.

TRUE CORE EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

FORTH: In ANS-Forth TRUE is represented by a -1 value.

Returns a true flag on the stack.

TUCK CORE EXT

COMPILES TO: SWAP (0)

OVER (0)

STACK EFFECTS: n1 n2 — n2 n1 n2)

Copy the first (top) stack item below the second stack item.

TYPE CORE

COMPILES TO: TYPE (0)

STACK EFFECTS: addr n —

Transmit n characters from addr to the selected output device.

U. CORE

PRONUNCIATION: u-dot

INCLUDE: dbldot.4th

STACK EFFECTS: n —

Display n in free field format as an unsigned number.

U.R CORE EXT

PRONUNCIATION: u-dot-r

INCLUDE: dbldot.4th

STACK EFFECTS: n1 n2 —

Display unsigned number n1 right aligned in a field n2 characters wide. If the number of characters required to display n1 is greater than n2, all digits are displayed with no leading spaces in a field as wide as necessary.

U< CORE

PRONUNCIATION: u-less-than

INCLUDE: ansdbl.4th

STACK EFFECTS: n1 n2 — f

Flag f is true if and only if unsigned number n1 is less than unsigned number n2.

U> CORE EXT

PRONUNCIATION: u-greater-than

INCLUDE: ansdbl.4th

STACK EFFECTS: n1 n2 — f

Flag f is true if and only if unsigned number n1 is greater than unsigned number n2.

UM* CORE

PRONUNCIATION: u-m-star

INCLUDE: mixed.4th

STACK EFFECTS: n1 n2 — d

Multiply n1 by n2, giving the unsigned double-cell product d. All values and arithmetic are unsigned.

UM/MOD CORE

PRONUNCIATION: u-m-slash-mod

INCLUDE: mixed.4th

STACK EFFECTS: d n1 — n2 n3

Divide d by n1, giving the quotient n3 and the remainder n2. All values and arithmetic are unsigned. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell unsigned integer.

UNESCAPE STRING EXT

INCLUDE: substit.4th

STACK EFFECTS: a1 n1 a2 – a2 n2

FORTH: This is a Forth-2012 word.

Replace each '%' character in the input string a1 n1 by two '%' characters. The output is represented by a2 n2. The buffer at a2 must be big enough to hold the unescaped string.

UNLOOP CORE

COMPILES TO: R> (0)

R> (0)

DROP (0)

DROP (0)

STACK EFFECTS: —

Discard the loop-control parameters for the current nesting level. An UNLOOP is required for each nesting level before the definition may be EXITed.

UNTIL CORE

COMPILES TO: 0BRANCH (<address of matching BEGIN>)

STACK EFFECTS: f —

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>UNTIL

FORTH: The optional WHILE word is not supported.

At runtime UNTIL controls the conditional branch back to the corresponding BEGIN. If f is FALSE execution returns to just after BEGIN; if f is TRUE execution continues ahead.

UPDATE BLOCK

INCLUDE: ansblock.4th

multiblk.4th

STACK EFFECTS: —

Mark the current block buffer as modified. An ambiguous condition exists if there is no current block buffer. UPDATE does not immediately cause I/O.

USE 4TH

COMPILES TO: USE (0)

STACK EFFECTS: h —

USE will associate the stream identified by filehandle h with the appropriate input- or output-channel, depending on the file access method used when opening the stream (see: *OPEN*). No streams are closed.

VALUE CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: n —

SYNTAX: <literal><space>VALUE<space><name>

Create a symboltable entry for the value name with an initial value n. At runtime, n will be placed on the stack. In 4tH it is an alias for TO.

VARIABLE CORE

SYNTAX: VARIABLE<space><name>

A defining word used to create variable <name>. When <name> is later executed, it will push the address <var> on the stack, so that a fetch or store may access this location.

VARS 4TH

COMPILES TO: VARS (0)

STACK EFFECTS: — x

This word returns the begin of the variables area.

W/O FILE

PRONUNCIATION: w-o

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the write only file access method.

WHILE CORE

COMPILES TO: 0BRANCH (<address of matching REPEAT token>)

STACK EFFECTS: f —

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>REPEAT

BEGIN<space>..<space>WHILE<space>..<space>UNTIL

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime, WHILE selects conditional execution based on number n. If f is TRUE, WHILE continues execution of the code thru to REPEAT, which branches back to BEGIN. If f is FALSE, execution skips to just after REPEAT, exiting the structure. Multiple WHILEs may be used.

WIDTH 4TH

COMPILES TO: LITERAL (<number of characters>)

STACK EFFECTS: — n

A constant which leaves the maximum number of characters, allowed in a <name> label.

WITHIN CORE EXT

INCLUDE: range.4th

STACK EFFECTS: n1 n2 n3 — f

Perform a comparison of a test value n1 with a lower limit n2 and an upper limit n3 , returning true if either (n2 < n3 and (n2 <= n1 and n1 < n3)) or (n2 > n3 and (n2 <= n1 or n1 < n3)) is true, returning false otherwise.

WORD CORE

INCLUDE: word.4th

STACK EFFECTS: c — addr

Skip leading delimiters and parse characters delimited by c. Addr is the address of the parsed word. If the parse area was empty or contained no characters other than the delimiter, the resulting string has a zero length.

WRITE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n h — f

Write n characters from addr to the file identified by handle h starting at its current position. Flag f is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character written to the file, and FILE-SIZE returns a value greater than or equal to the value returned by FILE-POSITION.

WRITE-LINE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n h — f

Write n characters from addr followed by the implementation-dependent line terminator to the file identified by handle h starting at its current position. Flag f is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character written to the file, and FILE-SIZE returns a value greater than or equal to the value returned by FILE-POSITION.

X-SIZE XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: addr n1 — n2

FORTH: This is a Forth-2012 word.

n2 is the number of chars used to encode the first xchar stored in the string addr n1. To calculate the size of the xchar, only the bytes inside the buffer may be accessed. An ambiguous condition exists if the xchar is incomplete or malformed.

X-WIDTH XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: addr n1 — n2

FORTH: This is a Forth-2012 word.

n2 is the number of monospace ASCII characters that take the same space to display as the the xchar string addr n1; assuming a monospaced display font, i.e. xchar width is always an integer multiple of the width of an ASCII character.

X\STRING- XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: addr n1 — addr1 n2

FORTH: This is a Forth-2012 word.

Search for the penultimate xchar in the string addr1 n1. The string addr1 n2 contains all xchars of addr n1, but the last. Unlike XCHAR-, X\STRING- can be implemented in encodings where xchar boundaries can only reliably detected when scanning in forward direction.

XC-SIZE XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: addr — n

FORTH: This is a Forth-2012 word.

n is the number of pchars used to encode xchar in memory.

XC-WIDTH XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: n1 — n2

FORTH: This is a Forth-2012 word.

n2 is the number of monospace ASCII characters that take the same space to display as the xchar; i.e. xchar width is always an integer multiple of the width of an ASCII char.

XC@+ XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: addr1 — addr2 n

FORTH: This is a Forth-2012 word.

Fetches the xchar at addr1. addr2 points to the first memory location after the retrieved xchar.

XC!+ XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: n addr1 — addr2

FORTH: This is a Forth-2012 word.

Stores the xchar at addr1. addr2 points to the first memory location after the stored xchar.

XC!+? XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: n1 addr1 n2 — addr2 n3 f

FORTH: This is a Forth-2012 word.

Stores the xchar into the string buffer specified by addr1 n2. addr2 n3 is the remaining string buffer. If the xchar did fit into the buffer, flag is true, otherwise flag is false, and addr2 n3 equal addr1 n2. XC!+? is safe for buffer overflows.

XCHAR+ XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: addr1 — addr2

FORTH: This is a Forth-2012 word.

Adds the size of the xchar stored at addr1 to this address, giving addr2.

XCHAR- XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: addr1 — addr2

FORTH: This is a Forth-2012 word.

Goes backward from addr1 until it finds an xchar so that the size of this xchar added to addr2 gives addr1. There is an ambiguous condition when the encoding doesn't permit reliable backward stepping through the text.

XEMIT XCHAR

INCLUDE: xchar.4th

STACK EFFECTS: n —

FORTH: This is a Forth-2012 word.

Prints an xchar on the terminal.

XHOLD XCHAR EXT

INCLUDE: xchar.4th

STACK EFFECTS: n —

FORTH: This is a Forth-2012 word.

Adds an xchar to the picture numeric output buffer.

XOR CORE

PRONUNCIATION: x-or

COMPILES TO: XOR (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical XOR of n1 XOR n2 as n3.

['] CORE

PRONUNCIATION: bracket-tick

COMPILES TO: LITERAL (<tok>)

SYNTAX: ['']<space><name>

STACK EFFECTS: — n | x | xt

FORTH: See '.

Compile the value contents of the symboltable entry identified as symbol <name> as a literal. An alias for '.

[: COMP.LANG.FORTH

COMPILES TO: LITERAL (<address of next BRANCH>)

BRANCH (<address of matching ; token>)

SYNTAX: [:<space>..<space>;]

STACK EFFECTS: — xt

FORTH: Several Forths support this word, e.g. gForth, iForth and VFX Forth.

Suspends compiling to the current definition (if any), starts a new nested definition, and compilation continues with this nested definition.

[=] 4TH

COMPILES TO: LITERAL (<flag>)

SYNTAX: <literal><space><literal><space>[=]

COMPILER: Two previously compiled literals are taken as arguments and a true flag is recompiled when they are equal. The instruction pointer is decremented, actually deleting the literals.

STACK EFFECTS: — f

FORTH: Equivalent to =.

[ABORT] 4TH

FORTH: Roughly equivalent to:

```
[ ABORT ]
```

Compilation is aborted immediately.

[ASSERT] 4TH

Toggles assertions. Assertions are disabled by default (see: *ASSERT(* and *)*).

[BINARY] 4TH

FORTH: Roughly equivalent to:

```
[ 2 BASE ! ]
```

When encountered during compilation it will set the radix to binary. All subsequent literals will be interpreted as binary numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[CHAR] CORE

PRONUNCIATION: bracket-char

COMPILES TO: LITERAL (<ASCII-value of character>)

SYNTAX: [CHAR]<space><char>

STACK EFFECTS: — c

Compiles the ASCII-value of <char> as a literal. At runtime the value is thrown on the stack. An alias for CHAR.

[DECIMAL] 4TH

FORTH: Roughly equivalent to:

```
[ DECIMAL ]
```

When encountered during compilation it will set the radix to decimal. All subsequent literals will be interpreted as decimal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[DEFINED] SEARCH EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

SYNTAX: [DEFINED]<space><name>

FORTH: This is a Forth-2012 word.

If the name is defined, return TRUE, else return FALSE.

[ELSE] TOOLS EXT

PRONUNCIATION: bracket-else

SYNTAX: <literal><space>[IF]<space><word>..[ELSE]<space><word>..[THEN]

Skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of [IF] ... [THEN] and [IF] ... [ELSE] ... [THEN], until the word [THEN] has been parsed and discarded.

[FORCE] 4TH

Disables all optimization for the compilation of one single word.

[HEX] 4TH

FORTH: Roughly equivalent to:

```
[ HEX ]
```

When encountered during compilation it will set the radix to hexadecimal. All subsequent literals will be interpreted as hexadecimal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[IF] TOOLS EXT

PRONUNCIATION: bracket-if

SYNTAX: <literal><space>[IF]<space><word>..[ELSE]<space><word>..[THEN]

COMPILER: The previously compiled literal is taken as an argument for [IF]. The instruction pointer is decremented, actually deleting the literal.

FORTH: Forth pops a value from the stack. This is not possible in 4tH.

If flag is true, do nothing. Otherwise, skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of [IF] ... [THEN] and [IF] ... [ELSE] ... [THEN], until either the word [ELSE] or the word [THEN] has been parsed and discarded.

[IGNORE] 4TH

SYNTAX: [IGNORE]<space><name>

COMPILER: The name is added to the symboltable. Subsequent occurrences of the name will be ignored.

FORTH: Roughly equivalent to:

```
: [ignore] create does> drop ;
```

Recognize name, but do not perform any action or compile any code for it.

[MAX] 4TH

COMPILES TO: LITERAL (<max>)

SYNTAX: <literal><space><literal><space>[MAX]

COMPILER: Two previously compiled literals are taken as arguments, the larger one of the two is recompiled. The instruction pointer is decremented, actually deleting the literals.

STACK EFFECTS: — n

FORTH: Equivalent to MAX.

[NEEDS 4TH

SYNTAX: [NEEDS<space><string>]

COMPILER: The contents of the file are inserted at this position.

Open the file specified by <string> and include its contents at the current position. When the end of the file is reached, close the file and continue compilation. An error condition exists if the named file can not be opened, if an I/O exception occurs reading the file, or if an I/O exception occurs while closing the file.

[NOT] 4TH

COMPILES TO: LITERAL (<flag>)

SYNTAX: <literal><space>[NOT]

COMPILER: A previously compiled literal is taken as an argument and a true flag is recompiled when it is equal to zero. The instruction pointer is decremented, actually deleting the literal.

STACK EFFECTS: — f

FORTH: Equivalent to 0=.

[OCTAL] 4TH

FORTH: Roughly equivalent to:

```
[ 8 BASE ! ]
```

When encountered during compilation it will set the radix to octal. All subsequent literals will be interpreted as octal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[PRAGMA] 4TH

SYNTAX: [PRAGMA]<space><name>

A defining word used to create word <name>. When <name> is later executed, it will push the value TRUE on the stack. Intended to define pragmas for use in an include file.

[SIGN] 4TH

COMPILES TO: LITERAL (<sign>)

SYNTAX: <literal><space>[SIGN]

COMPILER: A previously compiled literal is taken as an argument, and its sign recompiled. The instruction pointer is decremented, actually deleting the literal.

STACK EFFECTS: — -1|0|1

Used to convert a previously compiled literal to a single value, leaving only its sign.

[THEN] TOOLS EXT

PRONUNCIATION: bracket-then

Does nothing. Acts as a marker for [IF] (see: *[IF]*).

[UNDEFINED] SEARCH EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

SYNTAX: [UNDEFINED]<space><name>

FORTH: This is a Forth-2012 word.

If the name is defined, return FALSE, else return TRUE.

\ CORE EXT

PRONUNCIATION: backslash

SYNTAX: \<space><string>

The remainder of the line is discarded. Used for comment.

Chapter 16

Editor manual

16.1 Introduction

Forth organises its mass storage into "screens" of 1024 characters. Forth may have one screen in memory at a time for storing text. The screens are numbered, starting with screen 0.

Each screen is organised as 16 lines with 64 characters. The Forth screens are merely an arrangement of virtual memory and do not correspond to the screen format of the target machine. Due to this format, the use of the comment word '\' is not allowed. Use '(' instead.

16.2 Selecting a screen and input of text

After you've started an editing session, you need to select a screen to edit. The screen is given a number and selected by using:

```
n CLEAR (clear screen n and select for editing).
```

To input new text to screen after CLEAR, the P (put) command is used. Example:

```
0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, 2 OF SELECTED SCREEN.
```

16.3 Line editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text to be found or deleted by a string editing command. Do not confuse this PAD with 4tHs PAD. It is only called that way by convention.

16.4 Line editing commands

n D Delete line n but hold it in PAD. Line 15 becomes free as all statements move up 1 line.

n E Erase line n with blanks.

n I Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.

n H Hold line n at PAD (used by system more often than by user).

n R Replace line n with the text in PAD.

n S Spread at line n. Line n and following lines move down 1 line. Line n becomes blank. Line 15 is lost.

n T Display line n and copy it to PAD.

n P text Put 'text' at line n, overwriting its previous contents.

16.5 Screen editing commands

n LIST List screen n and select it for editing: if screen n is not the current screen, it will request to load from memory.

n CLEAR Clear screen n with blanks and select it for editing.

n INSERT Insert screen n. The current screen n and all screens following it are moved down. The last screen is lost.

n m COPY Copy the contents of screen n to screen m. The original contents of screen m are lost.

FLUSH Used at the end of an editing session to save the current screen to memory.

UNDO Used to reload the current screen again, thus undoing all changes since the last flush (triggered by CLEAR, FLUSH or LIST).

L List the current screen. The cursor line is relisted after the screen listing to show the cursor position.

16.6 Cursor control and string editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided from the user to position the cursor either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

16.7 Commands to position the cursor

n M Move the cursor by n characters and the cursor line. The position of the cursor on its line is shown by a ^ (caret).

n W Wipe n characters to the left of the cursor.

TOP Position the cursor at the start of the screen.

16.8 String editing commands

B Used after F to back up the cursor by the length of the most recent text.

C text Copy in text to the cursor line at the cursor position.

F text Search forward from the current cursor position until string 'text' is found. The cursor is left at the end of the string and the cursor line printed. If the string is not found an error message is given and the cursor repositioned to the top of the screen.

N Find the next occurrence of the string found by an F command

TILL text Delete on the cursor line from the cursor till the end of string text.

X text Find and delete the next occurrence of the string 'text'.

16.9 Saving and exiting

WRITE Saves the current contents of all screens to the block-file. No flushing is done.

WQ Flushes the current screen and saves the current contents of all screens to the block-file.

Q Quits the editor without saving.

EXPORT name Saves the current contents of all screens to the text-file with the name 'name'. No flushing is done.

16.10 Calculator mode

The calculator mode is a simulation of what is known as the "Forth calculator mode". You can use it to try out a host of 4tH words in interactive mode. It also serves nicely as a deskcalculator. You can freely mix editor and calculator commands.

We tried to include as many 4tH words as possible, although we had to modify some due to the limitations imposed by the system. There are eight pre-defined user-variables called "A." though "H.". You can use these variables like any other user-variable.

You cannot declare new variables or make any colon-definitions in interactive mode. If you are unclear how to use the built-in calculator please refer to the Primer and the Glossary. By convention, calculator mode uses "OK" as the prompt. The following table shows you which commands are available:

EDITOR	4TH EQUIVALENT	EDITOR	4TH EQUIVALENT
+	+	@	@
th	th	?	?
-	-	base!	base !
*	*	decimal	decimal
/	/	octal	octal
q	quit	binary	2 base !
quit	quit	.(<*string*>)	.(<*string*>)
bye	quit	mod	mod
.	.	abs	abs
.r	.r	negate	negate
drop	drop	invert	invert
dup	dup	min	min
rot	rot	max	max
swap	swap	or	or
over	over	and	and
A.	variable a. a.	xor	xor
B.	variable b. b.	lshift	lshift
C.	variable c. c.	rshift	rshift
D.	variable d. d.	depth	depth
E.	variable e. e.	cells	cells
F.	variable f. f.	1+	1+
G.	variable g. g.	cell+	cell+
H.	variable h. h.	1-	1-
!	!	cell-	cell-
+!	+!	space	space
(<*string*>)	(<*string*>)	spaces	spaces
cr	cr	2*	2*
time	time	2/	2/
char	char	/mod	/mod
[char]	[char]	*/	*/
emit	emit	*/mod	*/mod

Table 16.2: DC commands

Chapter 17

Shell manual

17.1 Introduction

The *4tsh* shell is a multitasking environment for 4tH. 4tH features cooperative multitasking, which means programs have to relinquish control to the shell using 'PAUSE', otherwise the program will keep in control. The best place to add 'PAUSE' is usually somewhere in a loop. 4tH comes with several example multitasking programs for you to try out. *4tsh* can be used as a command line replacement for *4th*, since you can enable multitasking in the editor.

4tsh is scriptable. Scripts are stored in blockfiles, because block I/O is completed within a single context. If you prefer to use your own editor, you need to convert your script to a blockfile. 4tH comes with a conversion program, called `txt2blk.4th`. Every twelve lines are converted to a block, leaving four additional lines for future modifications. Your lines should be limited to 63 characters or less.

When a script it loaded, the first block is executed automatically. By convention, the first block is block 0. When the execution of a block has completed, the script stops. You can call other blocks by using "LOAD". When the execution of a called block has completed, the execution of the previous block will resume at the point where execution was transferred to the called block. It is recommended to use the first block as an *application load screen*[1], e.g.

```
( 4tsh application load screen)
1 load  ( initialization)
2 load  ( checking conditions)
3 load  ( error handling)
```

An *application load screen* is simply a block that consecutively loads all the blocks that make up your script. You can run an arbitrary number of scripts at startup by issuing them on the command line, e.g.

```
4tsh boot.scr startup.scr tasks.scr
```

When all scripts have finished execution, control will automatically be transferred to the monitor.

[1] See "Thinking Forth", chapter 5.

17.2 Loading and saving

load" s"	Loads HX file **s** from disk and leaves the task number on the stack.
compile" s"	Loads and compiles source file **s** and leaves the task number on the stack.
n save" s"	Saves task number **n** to HX file **s**.
n write" s"	Generates C source file **s** from task number **n**.
n1 n2 see	Decompiles task number **n1** from opcode **n2** on.

17.3 Task management

task	Leaves the task number of the current monitor on the stack.
boot	Starts a new monitor and runs the boot scripts.
pause	Deactivates the monitor for one cycle.
n pauses	Deactivates the monitor for **n** cycles.
n run	Awakes and switches to task number **n**.
n awake	Awakes task number **n**.
n sleep	Deactivates task number **n**, but leaves it in memory.
n kill	Deactivates task number **n** and removes it from memory.
tasks	Lists all tasks.
halt	Kills all tasks and shuts down *4tsh*.

17.4 Scripting

script" s"	Run script **s**. Does only work in interactive mode.
n load	Load and interpret block **n**. Does not work in interactive mode.
:: s	Define label **s**.
goto s	Goto label **s**. Works in interactive mode, but only if **s** resides on the same line.
n if	Execute the words between *if* and the corresponding *then*, but only if **n** is non-zero. Works in interactive mode. If you fail to provide a corresponding *then* you will be prompted to provide it manually.
then	Marker for *if*.
n not	Leaves a non-zero value on the stack if **n** is zero, otherwise zero.
n status	Leaves the status of task number **n** on the stack.
done	Constant holding the termination status returned by *status*.
running	Constant holding the active status returned by *status*.
sleeping	Constant holding the inactive status returned by *status*.

17.5 Stack, I/O and arithmetic

4tsh	4TH EQUIVALENT	*4tsh*	4TH EQUIVALENT
+	+	.	.
-	-	dup	dup
*	*	rot	rot
/	/	over	over
cr	cr	swap	swap
.(	.(	drop	drop
(	(	=	=

Table 17.1: 4tsh commands

Chapter 18

Preprocessor manual

18.1 Introduction

The preprocessor is a tool written in 4tH that has the following features:

- It expands special macro definitions. These macro definitions can contain anything you want;
- It strips whitespace and comments;
- It collects all include files and inserts them into the source;
- It expands [CHAR] and CHAR constructs.

This tool can help you to solve several problems:

- Your 4tH implementation has serious memory restrictions, so certain sources cannot be compiled;
- You have to port Forth programs that use complex jump constructs.
- You have to write Forth compatible programs without the use of `easy.4th`;
- You can simplify development by writing complex programs more easily;
- You need to inline code in a structured way;
- It may serve in some situations as a debugging tool.

Note that the preprocessor is just a tool to automate certain sourcecode manipulations. It doesn't compile anything, although it does perform some syntax checks[1]. Succesfully processed source can be rejected by the compiler for any number of reasons.

Every valid 4tH program is automatically valid 4tH preprocessor source. In addition the preprocessor supports special preprocessor words, which are listed below.

[1] The preprocessor is sometimes even more pedantic than 4tH itself, see section 18.7.

18.2 Macros

A macro starts with the word `:MACRO` and is delimited by a semi-colon like a normal definition. A macro may span several lines and may contain anything you want, including conditional and loop statements. Macros may be nested, but they may *not* contain certain directives.

This ANS Forth definition is impossible to define in 4tH:

```
: STEP 8 POSTPONE LITERAL POSTPONE +LOOP ;
```

In the preprocessor you can define it like this:

```
:MACRO STEP 8 +LOOP ;
```

Macros are expanded by the preprocessor which means that every time it finds ”STEP” it will be replaced by ”8 +LOOP”.

18.2.1 Back quoted strings

You can’t print a semi-colon inside a macro, because it will be interpreted to be the end of the macro. In order to defeat the interpreter you can use back quoted strings. Back quoted strings are printed verbatim, so this will work:

```
:macro extra_stack : dip over swap ` ;` ;
```

You can even define entire strings:

```
:macro nul? dup 0= ` abort" Missing string"` ;
```

Note that back quoted strings are only interpreted *within* macros. Outside macros they are ignored.

18.2.2 Registers

The preprocessor has 4 *global* registers, which can only be used *inside* macro definitions. Take this definition:

```
:MACRO MONTH @1@ @2@ CREATE #1# #2# , DOES> @C . ` ;` ;
MONTH SEPTEMBER 9
```

Will expand to:

```
CREATE SEPTEMBER 9 , DOES> @C . ;
```

The commands `@1@` through `@4@` parse a word and store it in register 1 through 4. The commands `#1#` though `#4#` print the string stored there. Registers retain their values between calls, e.g.:

```
:MACRO BEGIN-STRUCTURE @1@ STRUCT ;
:MACRO END-STRUCTURE END-STRUCT #1# ;

BEGIN-STRUCTURE point
  1 CELLS +FIELD p.x
  1 CELLS +FIELD p.y
END-STRUCTURE
```

Will expand to:

```
STRUCT
  1 CELLS +FIELD p.x
  1 CELLS +FIELD p.y
END-STRUCT point
```

18.2.3 Parsing strings

Strings can be parsed by using the `$1$` through `$4$` commands and store them in register 1 through 4. They *always* have to be followed by a delimiter, e.g.:

```
INCLUDE lib/escape.4th
:MACRO S\" $1$ " ` S" ` >1> |#| >>> " |#| S>ESCAPE ;

S\" \tAnd the winner is:\t\q4tH compiler\q!"
```

Will expand to:

```
S" \tAnd the winner is:\t\q4tH compiler\q!" S>ESCAPE
```

18.2.4 The string stack

If four registers are not enough, there is also a string stack you can use to store strings. This macro will read a string and push it on the string stack:

```
:MACRO BEGIN-STRUCTURE @1@ >1> STRUCT ;
```

This macro will pop the string from the string stack and print it:

```
:MACRO END-STRUCTURE END-STRUCT <2< #2# ;
```

So if you write code like this:

```
BEGIN-STRUCTURE point
  FIELD: p.x
  FIELD: p.y
END-STRUCTURE
```

It will be expanded to this:

```
STRUCT
  FIELD: p.x
  FIELD: p.y
END-STRUCT point
```

If you're really short on registers, you can write a macro that saves the current value on the stack and restores it afterwards. Note that the string stack remains completely intact between calls. Just remember to keep it balanced at all times.

18.2.5 Phony variables

There are six phony variables >#>, <#<, >>>, $#$, _#_ and |#|. >#> works very much like @1@, @2@, etc. in that it parses a string. But instead of assigning it to a register, it is put on the string stack directly. <#< drops the top of the stack, so the sequence >#> <#< simply ignores a string:

```
:macro create @1@ @2@ >#> <#< >#> <#< #2# string #1# ;
create mystring 20 chars allot
```

>>> takes the next name or back quoted string and puts it on the string stack. So this will assign "Hello world!" to the first register:

```
:macro hello >>> ` Hello world¡ <1< ;
```

$#$ is much like the 1 through 4 commands, except it puts the parsed string on the string stack. Just like its cousins, it requires a delimiter. We could rewrite our implementation of "S\"" as follows:

```
include lib/escape.4th
:macro S\" $#$ " ` S" ` |#| >>> " |#| s>escape ;

S\" \tAnd the winner is:\t\q4tH compiler\q!"
```

Then _#_, which simply prints the top of the stack. Finally |#|, which is used to concatenate strings. Normally *before* a string is printed a space or newline is added. |#| takes the top of the string stack and prints it as is. This way you can build macros like this, which prints a string constant:

```
:macro literal: @1@ ` S" ` >1> |#| >>> " |#| ;
literal: Hello
```

Needless to say that phony variables are only recognized *within* macros.

18.2.6 Branching and looping

The preprocessor also offers some simple branching and looping functionality. It may seem unusual and awkward at first, but it does the job. Basically, it exits the macro *without* discarding the flag when the comparison renders false, e.g.:

```
:macro ex1 >>> 0 @if ` This is never printed` ;
:macro example ` Print this` ex1 @drop ;
example
```

Will only print "Print this". @if will render true when the numerical equivalent of the string on the stack is non-zero. However, the flag - either zero or non-zero - will remain on the stack, so we have to drop it one nesting level lower by calling @drop, which is a synonym for <#<. We can also add an "else" clause easily:

```
:macro ex1 >>> 0 @if ` This is never printed` ;
:macro ex2 ex1 @else ` But this is` ;
:macro example ` Print this` ex2 @drop ;
example
```

`@else` is the reverse of `@if` and exits the macro when the numerical equivalent of the string on the stack is non-zero, preserving *and* inverting the flag. The reason for the latter is, that you can *always* use `@else` for the *alternate* branch, no matter if you used `@if`, `@ifnot`, `@while` or `@until`.

`@else` is also known by another name `@ifnot`. Using `@if` and `@else` you can build an entire "case" like construct:

```
:macro chars? >>> chars @while #2# string #1# ;
:macro cells? >>> cells @while #2# array  #1# ;
:macro (create)
  @1@ @2@ @3@ @4@
  >3> chars? @else @drop
  >3> cells? @else table #1# #2# #3# #4# ;
:macro create (create) @drop ;
```

This one in particular mimics the ANS-Forth 'CREATE' word, which is subsequently transformed into typical 4tH constructs, so you can compile code like this:

```
create mystring 20 chars allot
create myarray  40 cells allot
create mytable  5 , 6 , 7 ,
```

`@while` leaves a true flag on the stack when the top two strings are equal. Its counterpart is `@until`, which leaves a true flag on the stack when the top two strings are *not* equal. Using recursion, you can even build loops:

```
:macro line
  @1@ >1> >>> %stop% @until @drop
  ` ," ` >1> |#| >>> ` "` |#| @2@ #2# , line ;
:macro make @1@ table #1# line @drop ;
```

Note you'll have to drop the previous flag with each new iteration if you don't want to unbalance your string stack. It allows you to process code like this:

```
make bla
  terminate 5 negate  6
  virtual   7 endorse 9
%stop%
```

Which will result in a nice little table, ready to compile. Note all loops are based on recursion, so if you nest too deep you may run out of stack space, which will result in an internal error.

18.2.7 Functions

There are several functions you can use within a macro. `@add`, for example, will add the two topmost entries on the string stack and leave the result. You can easily create counted loops with `@add`:

```
:macro hello @if Hello! >>> -1 @add hello ;
:macro hello's >#> hello @drop ;
```

So this will print "Hello!" ten times:

```
hello's 10
```

It is even powerful enough to resolve a CASE..ENDCASE construct:

```
:macro case >>> 0 ;
:macro of over = if drop ;
:macro endof else >>> 1 @add ;
:macro endif @if then >>> -1 @add endif ;
:macro endcase drop endif @drop ;

: .WEEKDAY ( daynum --- )
  CASE
      1 OF ." Sunday"    ENDOF
      2 OF ." Monday"    ENDOF
      3 OF ." Tuesday"   ENDOF
      4 OF ." Wednesday" ENDOF
      5 OF ." Thursday"  ENDOF
      6 OF ." Friday"    ENDOF
      7 OF ." Saturday"  ENDOF
  ENDCASE ;
```

With `@minus` you can negate a number on the stack, so subtractions are possible too. Multiplications can also be performed by using `@mul`. Divisions are supported by the `@divrm` function, which leaves the division *and* modulo on the stack. If you want to evaluate the result of an arithmetic expression, you can use `@sign`. `@sign` returns -1 when the top of the string stack is negative, 1 when it is positive and else zero. It closely resembles the `[SIGN]` directive of 4tH.

`@cell` will leave a true flag on the stack if the top of the stack can be converted to a number. Some stack instructions are provided as well, like `@drop`, `@dup`, `@nip` and `@over`. They work exactly like their 4tH counterparts. `@cr` simply issues a newline.

The most elusive function may be `@eval`. If the top of the string stack contains the name of a user-defined macro, it will be executed. If it doesn't, the top of the stack is simply printed. You can use this function to interpret a string of user-defined macro's or simply print the top of the stack.

You can check the existence of a user-defined macro with `@exist`, which leaves a non-zero value if the top of the string stack is the name of a user-defined macro. Finally, `@match` will compare the two topmost items on the string stack and leave a zero value if they match.

18.3 Invocation (script)

Since the preprocessor is written in 4tH, you can invoke it in the usual way:

```
4th cxq pp4th.4th mysource.4pp mysource.4th
```

You can also use `make`, see section 25.11. Note the preprocessor uses the `DIR4TH` environment variable to locate include files.

18.3.1 Options

The preprocessor supports the following options:

-D *symbol* Define ***symbol*** as a `CONSTANT`, with value ***1***. You may add as many **-D** options as you require.

-D *symbol=value* Define ***symbol*** as a `CONSTANT`, with value ***value***. You may add as many **-D** options as you require.

-v Enter verbose mode and show all files included hierarchically.

18.4 Invocation (executable)

The preprocessor is *also* embedded in a executable, so you can execute, decompile or save the generated file right away. It recognizes `.4pp` files and preprocesses them automatically. All other files are automatically compiled.

When a file is preprocessed a temporary file is created in your *working directory*. Unless you *specifically* indicate this temporary file should be kept, it is deleted. The name of this temporary file is equal to the preprocessor file, but with a `.4pi` extension.

This version of the preprocessor honors the `DIR4TH` environment variable completely. Parameters following the input filename are passed to the compiled program when executing.

18.4.1 Options

The preprocessor supports the following options:

-D *symbol* Define ***symbol*** as a `CONSTANT`, with value ***1***. You may add as many **-D** options as you require.

-D *symbol=value* Define ***symbol*** as a `CONSTANT`, with value ***value***. You may add as many **-D** options as you require.

-o *filename* Specify the name of the output file when preprocessing, generating or saving.

-i Show all files included hierarchically.

-c Load a sourcefile and compile it.

-p Preprocess a 4tH preprocessor file.

-d Decompile a 4tH program.

-g Generate a C sourcefile (*default: out.c*).

-k Keep intermediate file.

-s Save a 4tH program (*default: out.hx*).

-x Execute a 4tH program.

-v Enter verbose mode.

-V Show version and exit.

18.5 Preprocessor commands

\ s	The remainder of the line **s** is discarded. Used for comment.
(s)	Discard comment **s** that is delimited by a right parenthesis. A blank after the leading parenthesis is required.
char c	Replaces character **c** as with its ASCII value.
[char] c	Replaces character **c** as with its ASCII value.
include s	The contents of file **s**, delimited by whitespace, are inserted at this position.
[needs s]	The contents of file **s**, delimited by a right bracket, are inserted at this position.
:macro s ;	Create a macro with the name **s**, delimited by a semi-colon. When **s** is encountered it is replaced by the contents of the macro. A macro may contain any sequence of valid 4tH words, including conditional and loop statements, but no macros or include files.
scrap: s	Delete a macro with the name **s**. After that, **s** will not be expanded anymore and name **s** can be used to define a new macro. Note that the macro space allocated by **s** is *not* freed.
[binary]	Sets radix to binary.
[octal]	Sets radix to octal.
[decimal]	Sets radix to decimal.
[hex]	Sets radix to hexadecimal.

18.6 Error messages

Usage: pp4th infile outfile	Issue a preprocessor file and a source file on the commandline
Macro space exhausted	The combined size of all macro definitions is too large
Unexpected macro	The preprocessor found a `:MACRO` word within a previously started macro definition
Too many macros	There are too many macro definitions in the currently processed file
Duplicate macro	A macro with this name is already defined
Undefined macro	A `SCRAP:` directive was followed by an undefined macro name
Directive not allowed here	You cannot use an `INCLUDE`, `[NEEDS` or `SCRAP:` directive within a macro
Unexpected end of file	A number, character, string, name or other expression was expected in the currently processed file
String too long	The string parsed was too long to be stored in a variable

Missing string	A `$1$`, `$2$`, `$3$`, `$4$`, `$#$` or >>> variable was not followed by a string
Bad number	The top of the string stack is not a valid number
String stack empty	There were no strings left on the string stack
String stack full	There were too many strings stored on the string stack
String stack not empty	There were still items on the string stack after processing the current file
Seek failed	The preprocessor was unable to restore the file status after processing an include file
Cannot open <file>	Could not find a preprocessor file
Include file nested too deep	Too many include files included within other include files
Cannot open include file	Could not find an include file
Cannot read include file	Could not read the include file
Internal error	An internal condition caused the preprocessor to stop, e.g. stack overflow

18.7 Known bugs and limitations

- Some programs use conditional compilation to accommodate large comments. The preprocessor can only assume such sections contain compilable code and will treat it as such. If these comments contain words that are identical to macro names or preprocessor commands the appropriate substitutions will be made, which may in some circumstances raise errors.
- Some programs will span `(` comments across several lines. Although 4tH will accept such comments, it is a violation of the ANS-Forth standard. The preprocessor does *not* support such usage.
- Character literals are expanded under the *current* radix, so take care when using them in a macro.
- Back quoted strings offer *no protection* against macro expansion. If a back quoted string is recognized as a macro name, add a leading or trailing space.
- Although macro definitions may be nested, *extreme* nesting of macros is known to trigger internal errors.
- A runaway recursion may trigger internal or end-of-file errors.
- Severe violations of the ANS-Forth standard may trigger erratic behavior.

18.8 Preprocessor libraries

This table list all libraries, the user words they provide, their field of application and the registers they use. It is not recommended to use any internal words defined in the library nor to define any words starting with an underscore. Some libraries contain words that are intended to be used inside other macros, others provide new keywords you can use in your programs. Note that not all libraries can be combined. Refer to the source for details.

Some libraries clobber one or more registers. If any of these registers contain meaningful values when you use the library, these values will be overwritten. Save them on the stack before using any of these words or refrain from using these registers yourself.

LIBRARY	PROVIDES	APPLICATION	REGISTERS USED
`closures.4pp`	**:: vars end-vars var: this { } set sizeof {: ;}**	Program	–
`ansforth.4pp`	**action-of synonym begin-structure end-structure case of endof endcase**	Program	–
`comus.4pp`	**+to**	Program	–
`double.4pp`	**d% 2variable 2constant**	Program	–
`float.4pp`	**f% fvariable ffield: fconstant**	Program	–
`foos.4pp`	**derived? :: type@ typeof parent@ kindof parentof sizeof class extends end-class virtual: method: end-extends { } this {} :virtual :default :method => virtual ;method <- default static new instance private{ overrides**	Program	–
`rangext.4pp`	**le ge lt gt**	Program	–
`ifdef.4pp`	**@ifdef @ifndef @then**	Program	–
`sbslquot.4pp`	**s\"**	Program	–
`yield.4pp`	**yield grab**	Program	–
`interprt.4pp`	**[ ]**	Macro	–
`standard.4pp`	**@swap @rot @max @min @abs**	Macro	**@swap**: 1; **@rot**: 1, 2

Chapter 19

uBasic manual

19.1 Introduction

uBasic is an integer Basic interpreter in the tradition of Tiny Basic, with which it is largely compatible. It is derived from a program made by Herbert Schildt in the late eighties and published in his book "*C: The complete reference*". This version is entirely written in 4tH, some bugs have been removed and several additional features have been added. In order to execute a program you can invoke it in the usual way:

```
4th cxq ubasic.4th lander.bas
```

Of course, you may opt to create an executable or script suited for your operating system.

19.2 Statements

- Labels *must* be prefixed by an underscore and can contain *any* character, *except* delimiters;
- Labels are internally converted to signed integers and can be treated as such[1];
- Labels are valid line numbers;
- Line numbers are numeric, so *0020* refers to the same label as *20*;
- Line numbers and labels are optional *unless* there is a **GOSUB** or **GOTO** statement referring to them;
- Precedence, from high to low: (,), unary -, ^, *, / , %, +, -, #, =, <, >;
- A compound statement may consist of a single statement;
- Whitespace is ignored;
- uBasic is *not* case sensitive.

[1]This is called "hashing". Two strings may result in same hash value, which is called a "collision". The chance on a collision is rather small, about 0.04%.

CONCEPT	DEFINITION	ABBREVIATION
Line number	Any positive integer	**a**
Label	Any combination of ASCII-7 characters, except delimiters; must be prefixed by an underscore	**a**
Number	Any signed integer or label	**n**
Function	ABS(), RND(), TOS(), POP(), SGN(), NOT(), AND(), XOR(), OR(), SHL(), ORD(), TIME(), FUNC()	**f**
Variable	A to Z	**v**
Local variable	A@ to Z@	**v**
Primitive	Any variable, function or number	**p**
Operator	#, =, <, >, *, /, +, −, ^ or %	**o**
Delimiters	All operators, plus ', ;, : , (,) and ,	**d**
Term	*(p o p)*	**t**
Expression	*(t o t)*	**e**
Array element	*@ (e)*	**v**
String	Any combination of ASCII-7 characters	**s**
Quoted string	Any combination of ASCII-7 characters (except a double quote) between double quotes	**q**
Keyword	IF, PRINT, REM, GOTO, GOSUB, FOR, NEXT, LET, INPUT, RETURN, END, STOP, PUSH, CONTINUE, BREAK, DO, LOOP, UNTIL, WHILE, ELSE, ENDIF, LOCAL, PARAM, PROC, UNLOOP	**k**
Statement	Correct use of a keyword	**st**
Compound statement	*st : st : st*	**c**
Line	*a c*	**ln**

LET **v = e**

This statement assigns the value of the expression to the variable. The keyword **LET** is optional.

GOTO **e**

The **GOTO** statement permits changes in the sequence of program execution. Normally programs are executed in the numerical sequence of the program line numbers, but the next statement to be executed after a **GOTO** has the line number derived by the evaluation of the expression in the **GOTO** statement. Note that this permits you to compute the line number of the next statement on the basis of program parameters during program execution. An error stop occurs if the evaluation of the expression results in a number for which there is no line.

GOSUB **e[,e]**

The **GOSUB** statement is like the **GOTO** statement, except that uBasic remembers the line number of the **GOSUB** statement, so that the next occurrence of a **RETURN** statement will result

in execution proceeding from the statement following the **GOSUB**. Subroutines called by **GOSUB** statements may be nested up to a predefined depth. The label or line number may optionally be followed by a list of comma separated expressions. The results of these expressions are put on the stack.

RETURN(**e**) — The **RETURN** statement transfers execution control to the line following the most recent un**RETURN**ed **GOSUB**. If there is no matching **GOSUB** an error stop occurs. A **RETURN** statement may be followed by an optional expression. The result of this expression will be put on the top of the stack.

LOCAL(**e**) — **LOCAL()** allocates the number of local variables, specified in **e**, in the current scope. Subsequent **LOCAL()** statements within the current scope add additional local variables, but leave existing local variables intact. The scope changes by either issuing a **GOSUB** or **RETURN** statement. Local variables are *not* initialized and naming starts with 'A'. Thus, an initial **LOCAL(3)** defines local variables A@, B@ and C@. A subsequent **LOCAL(1)** within the same scope will create D@.

PARAM(**e**) — **PARAM()** creates local variables, like **LOCAL()**, but also initializes them using the stack. The top of the stack is used to initialize the last created local variable.

IF **e** THEN **c** — The **IF** statement evaluates an expression. If the expression renders non-zero, the entire compound statement is executed; if zero, the associated compound statement is skipped. The keyword **THEN** is optional. If an **IF** statement does not contain a compound statement, multiple line syntax is assumed, requiring the use of the **ENDIF** keyword. A multiple line syntax block may not contain a single line syntax statement.

ELSE — Will execute all statements following it until the keyword **ENDIF** is encountered, if the expression of the corresponding **IF** statement renders zero. Assumes multiple line syntax. The **ELSE** clause is optional.

ENDIF — Terminates an **IF** statement in multiple line syntax.

END — The **END** statement must be the last executable statement in a program. The **END** statement may be used to terminate a program at any time, and there may be as many **END** statements in a program as needed.

REM **s** — The **REM** statement permits comments to be interspersed in the program. Its execution has no effect on program operation, except for the time taken. A simple quote (**'**) can be used as an alias for "**: REM**".

PUSH **e**[, **e**] — Evaluates one or more expressions, separated by commas, and puts the results on the stack. The rightmost expression is top of stack.

PRINT **q**|**e** ,|; — This statement prints the values of the expressions and/or the contents of the quoted strings. Expressions are evaluated and printed as signed numbers; strings are printed as they occur in

the **PRINT** statement, except for the escape characters listed in table 13.8. When the items are separated by commas the printed values are justified in columns of 8 characters wide; when semicolons are used there is no separation between the printed items. Thus, **PRINT** 1,2,3 prints as 1 2 3 and **PRINT** 1;2;3 prints as 123. Commas and semicolons, strings and expressions may be mixed in one **PRINT** statement at will. If a **PRINT** statement ends with a comma or semicolon, uBasic will not terminate the output line so that several **PRINT** statements may print on the same output line, or an output message may be printed on the same line as an input request (see **INPUT**). When the **PRINT** statement does not end with a comma or semicolon the line is terminated.

PROC **e[,e]** — An alias for **GOSUB**.

INPUT **q** ,|; **v** — The quoted string is printed. If the quoted string is omitted, a question mark is prompted. A new line is read in and the input line is scanned for a number. The value thus derived is stored in the variable.

FOR **v** = **e** TO **e** STEP **e** — Sets up a control variable with an initial value (*first expression*), an optional limit (*second expression*), an optional increment (*third expression*) and a looping address referring to the statement after the **FOR** statement. If **STEP** is omitted an increment of 1 is assumed. If **TO** and **STEP** are omitted, an infinite loop is assumed. If the initial value is greater than the limit, execution will continue *after* the matching **NEXT**. See **NEXT**.

NEXT **v** — Increments the control variable and then jumps to the matching **FOR** statement. The variable is optional. Every **FOR** statement must be matched by one single **NEXT** statement. Poor programming style, such as the use of **GOTO** to exit loops or conditional **NEXT** statements, may leave unused frames on the **FOR** stack. See **UNLOOP**.

DO — A control structure to repeat a set of statements an indefinite number of times, usually **WHILE** or **UNTIL** a condition is satisfied. See **LOOP**.

LOOP — Every **DO** statement must be matched by one single **LOOP** statement. Poor programming style, such as the use of **GOTO** to exit loops or conditional **LOOP** statements, may leave unused frames on the **FOR** stack. See **UNLOOP**.

CONTINUE — Resumes iteration of the matching **FOR** loop, incrementing the corresponding control variable where appropriate. If any loop limits are reached, execution will continue after the matching **NEXT**.

BREAK — Terminates the matching **FOR** or **DO** loop unconditionally.

WHILE **e** — Terminates the matching **FOR** or **DO** loop by calling **BREAK** if the expression renders zero.

UNTIL **e** — Terminates the matching **FOR** or **DO** loop by calling **BREAK** if the expression renders non-zero.

UNLOOP — Discard the **FOR** stack frame for the current nesting level. An **UNLOOP** is required for each nesting level before the loop can be exited by executing **GOTO** or **RETURN**.

19.3 Functions

USING **q** — Prints the first number following it according to the quoted format string. The quoted string should contain one or more format specifiers as listed in table 9.1. May only be used within a **PRINT** statement. Returns nothing.

TAB(**e**) — Moves the text cursor to a specified print position. May only be used within a **PRINT** statement. Returns nothing.

CHR(**e**) — Prints the character associated with code **e**. May only be used within a **PRINT** statement. Returns nothing.

RND(**e**) — Returns a random number between 0 and the value of the expression. Thus, B=**RND**(3) assigns either 0, 1 or 2 to B.

ABS(**e**) — Returns the absolute value of the expression.

SGN(**e**) — Returns the signum of the expression. When **e** is less than, equal to and greater than zero, **SGN**(**e**) returns -1, zero and one, respectively.

NOT(**e**) — Returns **e** binary complement.

AND(**e**,**e**) — Returns the bitwise logical AND of **e** AND **e**.

OR(**e**,**e**) — Returns the bitwise logical OR of **e** OR **e**.

XOR(**e**,**e**) — Returns the bitwise logical XOR of **e** XOR **e**.

SHL(**e**,**e**) — Returns the result of a logical bit shift on **e**. It shifts a number of bits left, specified in the second **e**. A negative value performs a right shift.

ORD(**q**) — Returns the value associated with a single character, specified by **q**.

TOS() — Returns the top of the stack.

POP() — Takes the top from the stack and returns its value.

TIME() — Returns the Unix epoch. A value of zero equals 00:00:00 GMT on January 1st, 1970.

FUNC(**e[,(e)]**) — Calls the subroutine at **e**. The label or line number may optionally be followed by a list of comma separated expressions. The results of these expressions are put on the stack. Returns the top of the stack.

19.4 Error messages

An error message takes the form of:

```
A message, B:C
```

Where ***A*** is the error code, ***B*** is the line number where the error occurred and ***C*** the position within that line (*excluding the label itself*). The latter points to the last character of the token which caused the error.

CODE	MESSAGE	DESCRIPTION
0	**OK**	Program terminated succesfully.
1	**NEXT, CONTINUE or BREAK without FOR**	There was one more **NEXT** than there were **FOR**s. **BREAK** or **CONTINUE** were used outside a **FOR** loop (***FOR*** *stack underflow*).
2	**Assignment expected**	A '=' character was expected. May be a superfluous variable.
3	**Too many labels**	Too many labels were defined, decrease the number of labels.
4	**Duplicate label**	A label was defined twice.
5	**Undefined label**	A jump or call to an undefined label was attempted.
6	**Number too big**	The number didn't fit in a signed integer.
7	**RETURN without GOSUB**	There has been one more **RETURN** than there were **GOSUB**s (***GOSUB*** *stack underflow*).
8	**Too many nested FOR loops**	Too many **FOR** statements were embedded in other **FOR** statements (***FOR*** *stack overflow*).
9	**Too many nested GOSUBs**	Too many **GOSUB**s were called (***GOSUB*** *stack overflow*).
A	**ENDIF or NEXT expected**	A **FOR** or **IF** statement was not properly terminated by **NEXT** or **ENDIF**.
B	**Delimiter expected**	A closing parenthesis, comma or double quote was expected.
C	**Nonsense in BASIC**	A statement or expression was not composed correctly.
D	**Expression expected**	An expression was expected.
E	**Not a variable**	The variable name did not consist of a single, alphabetic character, local variable or array element. The subscript of the array element was out of bounds. The local variable didn't exist within the current scope.
F	**Not a function**	The function is unknown or misspelled.
G	**Stack empty**	Too many **POP()**s were issued (*data stack overflow*)
H	**Stack full**	Too many **PUSH**s were issued (*data stack overflow*)
I	**Too many local variables**	Too many local variables were defined.
J	**Out of scope**	An attempt was made to enter an invalid scope.

CODE	MESSAGE	DESCRIPTION
K	**String expected**	A quoted string was expected.
L	**Memory leak detected**	An internal condition caused a memory leak.

19.4.1 System messages

System failure An internal condition caused the interpreter to stop, e.g. division by zero.

File loading error The source file could not be opened.

Out of memory The source file was bigger than available memory.

Chapter 20

ANS Forth statement

Forth, like BASIC, has always suffered from a lack - or may be an abundance of standards. Both languages had many dialects, which were highly incompatible. However, although there was never a generally accepted BASIC standard, a simple BASIC program can be easily converted to almost any existing implementation of the language.

The Forth community had a different approach to the problem. They kept changing the core every few years, so even now it's very hard to find a program which can run on any Forth with little modification. Calling those very different versions a standard didn't really help.

So when the ANSI-standard committee began its work they had a few very though nuts to crack. In our view the ANS-Forth standard is big step forward, but not perfect. It has not fully regained the simplicity we found in the Forth-79 and still has some serious flaws, although most are an inheritance from Forth-83.

We do feel the need for a real Forth standard, so we tried to make 4tH as ANS-Forth (a.k.a. Forth-94) compatible as possible without sacrificing the ease of use that we had in mind when we designed it. About 95% of the CORE wordset is supported.

4tH was built according to the ANS-Forth standard, but with a tiny Forth-79 flavor. Full compliance to the ANS-Forth standard was never an objective. According to the ANS-Forth standard 4tH cannot be an "ANS-Forth System", since the standard does not cover this kind of implementation.

There have been efforts to modernize ANS-Forth, the Forth-200x initiative, which has resulted in Forth-2012. In our view a number of proposals have been accepted which erodate the consistency of Forth as a language standard even further. We tend to handle Forth-2012 proposals pragmatically: if it doesn't violate the basic principles we have stated, we will comply and align.

If you want to know if a certain language construct in Forth-2012 is supported, please consult the documentation.

20.1 ANS-Forth Label

According to the ANS-Forth standard, section 5.2.2, this system is capable of compiling:

ANS Forth Programs

Requiring:

- *the Block word set*
- *the Exception word set*
- *the Memory word set*
- *the String Extensions word set*

Requiring selected words from:

- *the Core Extensions word set*
- *the Block Extensions word set*
- *the Double number word set*
- *the Double number Extensions word set*
- *the Facility Extensions word set*
- *the File-Access word set*
- *the File-Access Extensions word set*
- *the Floating-Point word set*
- *the Floating-Point Extensions word set*
- *the Programming-Tools word set*
- *the Programming-Tools Extensions word set*
- *the String word set*
- *the XCHAR wordset*
- *the XCHAR Extensions word set.*

End of label. Although the ANS-Forth standard (section 4.1) requires documentation to be presented in a prescribed format, 4tH does not comply for the simple reason that due to its architecture it is not considered to be a "ANS Forth System" (sections 3.3, 3.4, 5.1).

Note that due to this special architecture some words are missing from the CORE wordset or behave slightly different, so some "ANS Forth Programs" with the requirements mentioned above may not compile or compile only with modifications.

20.2 Unsupported CORE words

These words are not available in 4tH. Some CORE words are only available in source (ANS-Forth, section 3). You can find them in the 4tH glossary. The behaviour of some 4tH words may differ from the ANS-Forth definition.

ALLOT
FIND
LITERAL
POSTPONE
STATE
[
]

20.3 Supported ANS Forth word sets

The words in the following sections are supported by 4tH; *external words are in italics*, preprocessor words are underlined. Please note that due to 4tHs special architecture some words may behave slightly different, so some "ANS Forth Programs" using these words may need modifications in order to run properly. More words are available in source and can be loaded when required.

20.3.1 Core Extensions word set

#TIB

.(

.R

0<>

0>

2>R

2R>

2R@

:NONAME

<>

?DO

ACTION-OF

AGAIN

CASE

ENDCASE

ENDOF

ERASE

EXPECT

FALSE

HEX

HOLDS

OF

NIP

PAD

PARSE

PARSE-NAME

PICK

QUERY

REFILL

RESTORE-INPUT

ROLL

S\"

SAVE-INPUT
SOURCE-ID
TIB
TO
TRUE
TUCK
U.R
U>
VALUE
WITHIN
****

20.3.2 Block Extensions word set

EMPTY-BUFFERS
LIST
SCR

20.3.3 Double number word set

2CONSTANT
2VARIABLE
D+
D-
D.
D.R
D0<
D0=
D2*
D2/
D<
D=
D>S
DABS
DMAX
DMIN
DNEGATE
M+
M*/

20.3.4 Double number Extensions word set

2ROT
DU<

20.3.5 Facility Extensions word set

+FIELD
BEGIN-STRUCTURE
CFIELD:
END-STRUCTURE
FIELD:
KEY?
MS
TIME&DATE

20.3.6 File-Access word set

(
BIN
CLOSE-FILE
CREATE-FILE
DELETE-FILE
FILE-POSITION
FILE-SIZE
OPEN-FILE
R/O
R/W
READ-FILE
READ-LINE
REPOSITION-FILE
S"
SOURCE-ID
W/O
WRITE-FILE
WRITE-LINE

20.3.7 File-Access Extensions word set

FILE-STATUS
FLUSH-FILE
REFILL
RENAME-FILE

20.3.8 Floating-Point word set

>FLOAT
D>F
F!
F*
F+
F-
F0<
F0=
F<
F>D
F@
FALIGN
FALIGNED
FCONSTANT
FDEPTH
FDROP
FDUP
FLOAT+
FLOATS
FLOOR
FMAX
FMIN
FNEGATE
FOVER
FROT
FROUND
FSWAP
FVARIABLE
REPRESENT

20.3.9 Floating-Point Extensions word set

F*
F.
F>S
FABS
FACOS
FACOSH
FALOG
FASIN

FASINH
FATAN
FATAN2
FATANH
FCOS
FCOSH
FE.
FEXP
FEXPM1
FFIELD:
FLN
FLNP1
FLOG
FS.
FSIN
FSINCOS
FSINH
FSQRT
FTAN
FTANH
F~
PRECISION
S>F
SET-PRECISION

20.3.10 Programming-Tools word set

.S
?
DUMP

20.3.11 Programming-Tools Extensions word set

N>R
NR>
SYNONYM
[ELSE]
[IF]
[THEN]

20.3.12 String word set

-TRAILING
/STRING
BLANK
CMOVE
CMOVE>
COMPARE
SEARCH

20.3.13 XCHAR word set

X-SIZE
XC-SIZE
XC@+
XC!+
XC!+?
XCHAR+
XEMIT

20.3.14 XCHAR Extensions word set

-TRAILING-GARBAGE
X-WIDTH
XHOLD
XC-WIDTH
+X/STRING
X\STRING-
XCHAR-

Chapter 21

Porting guide

21.1 Introduction

4tH is ANS-Forth compatible. That means that 4tH and ANS-Forth share a common word-set, so you can write programs that run on both systems. This guide will show you how you can write portable programs or convert eligible ANS-Forth programs to 4tH with as little effort as possible.

21.2 General guidelines

We have already stated that 4tH and ANS-Forth have much in common, but it is unlikely that you can write a non-trivial program that runs unmodified on both platforms without resorting to conditional compilation, which allows you to "hide" implementation specific code. The word '4TH#' not only holds 4tH's version number, but is also an effective way to differentiate between 4tH and other compilers:

```
[DEFINED] 4TH# [IF]
variable span
: expect 1- accept span ! ;
[THEN]
```

Of course, the opposite works too:

```
[UNDEFINED] 4TH# [IF]
s" easy.4th" included
[THEN]
```

If you have an interactive program you might want to disable the 4tH autostart:

```
[DEFINED] 4TH# [IF] start-program [THEN]
```

Otherwise 'REFILL' will try to get its input from the file instead of the keyboard.

21.3 Differences between 4tH and ANS-Forth

Like any software, 4tH is a compromise. We have to address the requirements of both newbies and power users, which means we have to make choices[1] concerning ANS-Forth compliancy. There are several reasons why 4tH is not completely ANS-Forth compliant:

1. 4tH uses a different architecture which makes it impossible to be ANS-Forth compliant, so some constructions are simply not feasible;
2. Some constructions in ANS-Forth are considered to be illogical, unelegant, bloated, not intuitive, error prone, inefficient or otherwise not acceptable;
3. 4tH maintains a close relationship with C, so it is more logical and efficient to use C-conventions instead of ANS-Forth conventions.

Where possible, we try to minimize the consequences for our users by hiding the differences behind abstractions or other transparent solutions. But sometimes, we simply can't. In this section we will show you which differences there are between 4tH and ANS-Forth and how you can either avoid or resolve them.

21.3.1 Strings

In 4tH, strings are stored in an ASCIIZ format. ANS-Forth uses counted strings. In 4tH there is no such thing as a countbyte, since it uses a terminator. If you limit the use of 'COUNT' only to string variables and constants, and exclusively use 'PLACE' or '+PLACE' you should be fine, since the address/count convention of ANS-Forth is fully supported. Should you resort to low level operations which require a terminator, you might have to define an equivalent word in ANS-Forth to make your program portable.

'S"' does have interpretation semantics, but the string stored at the address 'S"' returns might have a very short lifespan, depending on your ANS-Forth compiler. 4tH has a transparent, circular buffer that protects the string from overwriting, but when you port your program you might not be that lucky. Note that ANS-Forth does not require compilers to provide these facilities.

21.3.2 Double numbers

4tH uses only signed 32 bit cells, but some words in ANS-Forth, like '<#', '#>', 'FILE-SIZE', 'FILE-POSITION' and 'REPOSITION-FILE' require the use of double numbers. You can easily fix this by adding 'S>D', which converts a number to a double number. Its counterpart, 'D>S', is available too. In 4tH these words have no effect.

21.3.3 Booleans

Another nice topic for a flame war is the value of truth. In ANS-Forth the 'TRUE' has the value "-1", which means all bits are set. Which is very clever. You can 'XOR', 'OR', 'AND' and 'INVERT' it with any other value and it will behave as logical value. But "the all bits set" flag has its drawbacks too. Let's see what the ANS-Forth standard says about flags:

[1] You may or may not agree with the choices we made, but you can rest assured we have given them considerable thought.

> "A FALSE flag is a single-cell datum with all bits unset, and a TRUE flag is a single-cell datum with all bits set. While Forth words which test flags accept any non-null bit pattern as true, there exists the concept of the well-formed flag. If an operation whose result is to be used as a flag may produce any bit-mask other than TRUE or FALSE, the recommended discipline is to convert the result to a well-formed flag by means of the Forth word 0<> so that the result of any subsequent logical operations on the flag will be predictable. In addition to the words which move, fetch and store single-cell items, the following words are valid for operations on one or more flag data residing on the data stack: AND OR XOR INVERT"

We highly recommend the discipline of converting a non-zero value to a well-formed flag. But we don't understand why 'INVERT' is a valid way to manipulate a flag. We'll try to explain you why.

Forth traditionally has no specific logical operators. Instead, binary operators were used. This put 'INVERT' (or 'NOT' as it was called in Forth-79) in a difficult position. 'INVERT'ing any non-zero value will result in a non-zero value, except when all bits are set.

That is why '0=' was introduced, a full-fledged logical operator. So why use 'INVERT' when you want to perform a logical operation? Another quote:

> "Since a "char" can store small positive numbers and since the character data type is a sub-range of the unsigned integer data type, C! must store the n least-significant bits of a cell (8 <= n <= bits/cell). Given the enumeration of allowed number representations and their known encodings, "TRUE xx C! xx C@" must leave a stack item with some number of bits set, which will thus will be accepted as non-zero by IF."

This is another problem of using "all bits set" as a true flag: you store a well formed flag in an address unit that should easily be able to handle it and you'll never get it back. A flag is a boolean and can have two values: either true or false. The smallest unit that can hold a boolean is a bit. ANS-Forth programmers are denied that privilege.

But why are some Forth programmers so keen on their "all bits set" flag? Well, you can do neat things with it.

```
: >CHAR DUP 9 > 7 AND + ASCII 0 + ;
```

This will convert a digit to its ASCII representation. True, it is a clever piece of programming, but in our opinion it is bad style. Why? Because you are using a flag as a bitmask, which is a completely different datatype. Although there is no such thing as "data typing" in Forth, this way of programming makes it difficult to understand and maintain a program, which the ANS-Forth standard acknowledges:

> "The discipline of circumscribing meaning which a program may assign to various combinations of bit patterns is sometimes called data typing. Many computer languages impose explicit data typing and have compilers that prevent ill-defined operations. Forth rarely explicitly imposes data-type restrictions. Still, data types implicitly do exist, and discipline is required, particularly if portability of programs is a goal. In Forth, it is incumbent upon the programmer (rather than the compiler) to determine that data are accurately typed."

But there is an even more compelling reason why 4tH returns ”1” as *the* ’TRUE’ value. ”-1” is only valid in a 2-complement context. What it actually means is ”all bits set”. Mathematically it means *nothing*, contrary to ”1”, which is return value associated with the Iverson bracket[2].

That is why 4tH uses "1" as a true flag. Usually, it won’t make much difference. Except when you use ’INVERT’ to invert a flag or intend to make obfuscated programs. If you use ’0=’ instead, you won’t run in any trouble, not even when you port your program to ANS-Forth. Clarity may introduce a little overhead, but in this age of multi-gigaherz machines, who is counting? E.g. you could program “>CHAR” like this:

```
\ convert a flag to a bit mask
: >MASK 0 SWAP IF INVERT THEN ;           ( f -- mask)
                                          \ convert a digit to ASCII
: >CHAR DUP 9 > >MASK 7 AND + ASCII 0 + ; ( n -- c)
```

If you still want to change the true flag, you can by simply changing a `#define` in `cmds_4th.h`:

```
#define F_T ~(0L)
```

But we doubt whether it will be a great benefit to your programming style.

21.3.4 CREATE..DOES>

In both 4tH and ANS-Forth it is possible to change the runtime behavior of variables. E.g. in ANS-Forth, ’CONSTANT’ is usually defined as:

```
: CONSTANT CREATE , DOES> @ ;
10 CONSTANT MY_CONST
MY_CONST . CR
```

Of course there is a predefined word in 4tH that does this, but if you wanted to *mimic* this behavior you would have to define it like this:

```
CREATE MY_CONST 10 , DOES> @C ;           \ Definition of CONSTANT
MY_CONST . CR                             \ Works the same way
```

The point is that the ANS-Forth "CREATE DOES>" construct cannot be ported to 4tH, although all words seem to be supported. A rule of the thumb is that *defining words cannot be used to define new defining words*, like in ANS-Forth. Most errors will be trapped by 4tH’s compiler, though[3].

Just remember that a ’:REDO’ definition can easily be ported to ANS-Forth. If you want to write a portable program, ’:REDO’ is the way to do it.

[2] https://en.wikipedia.org/wiki/Iverson_bracket

[3] Note that the use of ’DOES>’ in 4tH is called ”interpretation” in ANS-Forth, which is *explicitly* forbidden. Some Forth compilers (like gForth) have defined the interpretation semantics of ’DOES>’ and are consequently largely compatible with 4tH’s ’DOES>’.

21.3.5 HERE

Be careful with 'HERE'. 'HERE' looks and acts a lot like the ANS-Forth 'HERE', but since the architecture is different it serves quite another function. When 'HERE' is used for address arithmetic with definitions or arrays of constants, it works right out of the box. If not, it usually doesn't.

21.3.6 2>R and 2R>

These words are included in order to let you put two items on the Return Stack at once, fetch them using '2R@' and return them on the Data Stack safely. However, ANS-Forth assumes a certain order of these two items, so they can be fetched separately by 'R@' and 'R>'. 4tH does not support this specific order. A safe way to emulate the ANS-Forth behavior is to swap the values before you put them on the stack and swap them again when they are retrieved, e.g.:

```
2R> DO
  I over -  3 .R
  over I = IF ." *" ELSE SPACE THEN
  I 2 +  7 MOD 0= IF  CR  THEN
LOOP 2DROP ;
```

Becomes:

```
R> R> SWAP DO
  I over -  3 .R
  over I = IF ." *" ELSE SPACE THEN
  I 2 +  7 MOD 0= IF  CR  THEN
LOOP 2DROP ;
```

Note that the latter is ANS-Forth compliant.

21.3.7 Interpretation and compilation mode

There are several words, which act differently in interpretation and compilation mode. In Forth-79, some of them were "state-smart", which means they adjusted their behavior depending on the mode the system was in. In Forth-83 and subsequently ANS-Forth, they became "dumb" words and counterparts were designed for each mode. Other words lacked interpretation semantics all together.

4tH has got neither a true interpretation mode nor a state. But if you want to port 4tH code to ANS-Forth, this has to be dealt with. In 4tH this porting issue is resolved by several aliases. Some words have an alias since they do not have interpretation semantics in the ANS-Forth standard, but are often used outside colon-definitons in 4tH. This will enable you to make a word that mimics these interpretation semantics.

This table lists all "dumb" words with their counterparts. "Interpretation" means it has to be used outside colon-definitions. "Compilation" means it has to be used inside colon-definitions.

INTERPRETATION	COMPILATION
'	[']
.(	."
CHAR	[CHAR]

Table 21.1: Dumb words

Finally, in ANS-Forth all flowcontrol words (like IF, THEN, BEGIN, WHILE, DO, LOOP) may only be used inside colon-definitions.

21.3.8 BEGIN..WHILE..REPEAT

4tH allows you to use multiple WHILE's in a BEGIN..WHILE..REPEAT construct. ANS-Forth allows that too, but requires an extra 'THEN' for each additional 'WHILE'. In short, this is the 4tH version:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat
```

And this is the ANS-Forth version:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat then
```

To make this work, we have to resort to conditional compilation:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat
[undefined] 4th# [if] then [then]
```

It's not beautiful, but it works. The same applies when 'UNTIL' is used instead of 'REPEAT'. BEGIN..WHILE..AGAIN constructs are not supported by ANS-Forth, so be careful when considering 'AGAIN' too much of an alias of 'REPEAT'. Also bear in mind that DONE..DONE clauses *can* be defined in ANS-Forth, but still require conditional compilation if you resort to multiple 'WHILE's.

21.3.9 CASE..OF..ENDOF..ENDCASE

Many users wonder why this "essential" construct is missing in 4tH. The explanation is pretty simple: it is horrible! First, this construct puts a *heavy* burden on the controlstack, which is pretty shallow in 4tH. Why? Because it is essentially a nested IF..ELSE..THEN construct as you will see later on. Second, these kinds of problems are better handled by a lookup table. 4tH has excellent support for lookup tables, much better than other Forths.

Of course, sometimes you just want to convert a program and don't feel like redesigning it. Fortunately, converting CASE..OF..ENDOF..ENDCASE constructs is pretty straightforward. It only requires four simple steps. Let's examine this simple example:

```
: .WEEKDAY ( daynum --- )
  CASE
      1 OF ." Sunday" ENDOF
      2 OF ." Monday" ENDOF
      3 OF ." Tuesday" ENDOF
      4 OF ." Wednesday" ENDOF
      5 OF ." Thursday" ENDOF
      6 OF ." Friday" ENDOF
      7 OF ." Saturday" ENDOF
  ENDCASE ;
```

Step 1: Replace ENDOF with ELSE and eliminate CASE

```
: .WEEKDAY ( daynum --- )
      1 OF ." Sunday" ELSE
      2 OF ." Monday" ELSE
      3 OF ." Tuesday" ELSE
      4 OF ." Wednesday" ELSE
      5 OF ." Thursday" ELSE
      6 OF ." Friday" ELSE
      7 OF ." Saturday" ELSE
  ENDCASE ;
```

Step 2: Replace OF with OVER = IF DROP

```
: .WEEKDAY ( daynum --- )
      1 OVER = IF DROP ." Sunday" ELSE
      2 OVER = IF DROP ." Monday" ELSE
      3 OVER = IF DROP ." Tuesday" ELSE
      4 OVER = IF DROP ." Wednesday" ELSE
      5 OVER = IF DROP ." Thursday" ELSE
      6 OVER = IF DROP ." Friday" ELSE
      7 OVER = IF DROP ." Saturday" ELSE
  ENDCASE ;
```

Step 3: Replace ENDCASE with DROP

```
: .WEEKDAY ( daynum --- )
      1 OVER = IF DROP ." Sunday" ELSE
      2 OVER = IF DROP ." Monday" ELSE
      3 OVER = IF DROP ." Tuesday" ELSE
      4 OVER = IF DROP ." Wednesday" ELSE
      5 OVER = IF DROP ." Thursday" ELSE
      6 OVER = IF DROP ." Friday" ELSE
      7 OVER = IF DROP ." Saturday" ELSE
  DROP ;
```

Step 4: Add as many THENs as there are ELSEs

```
: .WEEKDAY ( daynum --- )
      1 OVER = IF DROP ." Sunday" ELSE
      2 OVER = IF DROP ." Monday" ELSE
      3 OVER = IF DROP ." Tuesday" ELSE
      4 OVER = IF DROP ." Wednesday" ELSE
      5 OVER = IF DROP ." Thursday" ELSE
      6 OVER = IF DROP ." Friday" ELSE
      7 OVER = IF DROP ." Saturday" ELSE
  DROP THEN THEN THEN THEN THEN THEN THEN ;
```

Done! Ugly? Yes, but that's what a CASE..OF..ENDOF..ENDCASE construct internally looks like! Wrapping it in some slick keywords doesn't change that. If 4tH issues an error message saying that you're nesting too deep, you'll understand why. If you happen to convert a program that contains many CASE..OF..ENDOF..ENDCASE constructs it may be a bit tiresome, but remember you only have to do it *once*. If you don't feel like editing your source manually, you can use the script `case24th.th` to do the job for you. Finally, CASE..OF..ENDOF..ENDCASE is supported by the preprocessor[4] as well.

[4] See section 21.5.

21.3.10 DO..LOOP

It is well-known in the Forth community that DO..LOOP is flawed. There have been several attempts to correct this, but they never got it right. On many occasions it even got worse. But why is DO..LOOP flawed?

'DO' puts the limit and the index on the Return Stack, but it doesn't decide whether the loop is actually entered. So, every loop is executed at least once. After each iteration 'LOOP' decides whether it iterates once more.

In our opinion it would have been better when 'DO' had made that decision (like any other language), but we can still live with that. The real trouble came with DO..+LOOP.

'+LOOP' is a logical extension. Every single language allows you to change the step. But contrary to what one might expect, '+LOOP' doesn't terminate when the loop limit is reached or exceeded, but when the loop index crosses "the boundary between the loop limit minus one and the loop limit".

What does that mean? Well, consider these three loops and try to predict what will be printed. Note: every loop is executed at least once:

```
( 1) 5 0 do i . 1 +loop cr
( 2) 5 0 do i . -1 +loop cr
( 3) -5 0 do i . -1 +loop cr
```

You would probably expect to see:

```
0 1 2 3 4
0
0 -1 -2 -3 -4
```

And that is what you get when you use 4tH. But this is *not* what you will get with ANS-Forth:

```
0 1 2 3 4
0 -1 -2 -3 .. 6 5
0 -1 -2 -3 -4 -5
```

The behaviour of the second loop is caused, because '+LOOP' doesn't take into account that it is counting down. So it iterates until the loop index reaches the loop limit by wrap-around arithmetic.

The behaviour of the third loop is caused by the ANS-Forth definition: the loop index must "cross the boundary between the loop limit minus one and the loop limit". In this case, the boundary is between -5 and -6.

DO..+LOOP didn't behave like this since the beginning of Forth: it was introduced in Forth-83. We preserved the Forth-79 definition as closely as possible, because it is much more intuitive.

Some claim that ?DO..+LOOP will save it. As a matter of fact, it does. But only when the loop index and the loop limit are the same:

```
0 0 ?do i . loop
```

In that case the loop won't be entered. But it still won't save us for loops like this:

```
5 0 do i . -1 +loop cr
```

The authors of gForth claim that a whole host of new DO..LOOP words are the solution. We don't think so:

```
100 -100 do i . i 1+ 2/ negate +loop cr
```

The bottomline is: you can't let two words make the same decision. 4tHs '+LOOP' checks which direction it is going (up or down) and evaluates the loop arguments accordingly. We feel it is the best we can do for you.

Is there no way we can circumvent these problems? Yes, there is. It may not be too elegant or even fast, but it solves the problem. We just emulate C's for():

```
12 >r begin                                \ set up loop index
  r@ 10 <                                  \ check loop limit
while
  r@ .                                     \ access loop index
  r> 2+ >r                                 \ increment loop index
repeat                                     \ next iteration
r> drop cr                                 \ drop loop index
```

Which is "equivalent" to:

```
10 12 ?do i . 2 +loop cr
```

Except that it works as expected. And as an extra bonus it is portable to ANS-Forth. Are these differences between the ANS-Forth and 4tH implementation of DO..LOOP really that important? Not in practice. Nobody really wants a loop that depends on wrap-around arithmetic, and you'll hardly ever see a '+LOOP' with a negative subscript. Everybody wants their programs to be understandable and maintainable, so the DO..LOOPs you'll encounter will usually be well-behaved.

21.3.11 I/O

It is trivial to define the ANS-Forth **FILE** wordset in 4tH, but almost impossible to do the opposite. So if you want to make a portable program use the ANS-Forth **FILE** wordset by including the `ansfile.4th` library file. The reason why 4tH uses a different I/O subsystem is twofold:

1. 4tH's I/O subsystem is far more powerful and elegant. Instead of defining a whole new wordset, 4tH reuses most of the available I/O words, like 'TYPE', 'EMIT', 'ACCEPT' and 'REFILL'[5], which is very Forth-like.
2. 4tH's I/O was initially quite primitive and this was the only way to extend the system without breaking too much code.

This example is taken from gForth, but runs identically on both 4tH and gForth:

```
[defined] 4th# [if]                        \ if this is 4tH, include
include lib/ansfile.4th                    \ ANS Forth FILE wordset
include lib/compare.4th                    \ and the word COMPARE
[then]
```

[5] As a matter of fact, a new set of words have been proposed that allow redirection of these words. 4tH already has that functionality.

```
[undefined] 4th# [if]                  \ if this is not 4tH,
s" lib/easy.4th" included              \ include 4tH compatibility
[then]

0 Value fd-in                          \ input file handle
0 Value fd-out                         \ output file handle
: open-input ( addr u -- )  r/o open-file throw to fd-in ;
: open-output ( addr u -- )  w/o create-file throw to fd-out ;

s" foo.in" open-input                  \ open input file
s" foo.out" open-output                \ open output file

: show 2dup type cr ;                  \ show the line

256 Constant max-line                  \ size of basic buffer
max-line 2 [+] string  line-buffer     \ extend by two bytes:
                                       \ ANS Forth requirement!
: scan-file ( addr u -- )
  begin                                \ read a line
      line-buffer max-line fd-in read-line throw
  while                                \ is it identical?
      >r 2dup line-buffer r> show compare dup
  while                                \ if so, exit loop
      drop                             \ clean up
  repeat                               \ ANS requires an extra
[undefined] 4th# [if] then [then]      \ then after each WHILE
  drop 2drop
;
                                       \ now scan the file
s" The text I search is here" scan-file

fd-in close-file throw                 \ close input file
fd-out close-file throw                \ close output file
```

Since section 11.3.12 of the ANS-Forth standard clearly states that an I/O exception shall not cause a 'THROW', we have to mention that 'FILE-SIZE', 'FILE-STATUS' and 'REPOSITION-FILE' are not entirely ANS-Forth compliant.

21.4 Easy 4tH

4tH programs won't run on ANS-Forth all by itself. You'll usually need several definitions to make them work. In collaboration with Wil Baden we have developed an interface between ANS-Forth and 4tH. It consists of two files, `easy.4th` and `ezneeds.4th`. These library files enable you to run most 4tH programs under ANS-Forth. In order to succesfully compile and run a 4tH program under ANS-Forth it must have been written with ANS-Forth in mind. The rest is simple: just add a couple of lines at the beginning of your 4tH program:

```
[UNDEFINED] 4TH# [IF]
s" easy4th.4th" included
[THEN]
```

That's all! Most of the 4tH words are now known to your very own ANS-Forth compiler. If your compiler already supports 'INCLUDE', you might be tempted to use:

```
[UNDEFINED] 4TH# [IF]
include easy4th.4th
[THEN]
```

This will actually work but since 4tH recognizes and acts on the 'INCLUDE' directive, it will load the interface. A slight memory and CPU penalty is the result.

21.4.1 Enabling the String Space

Optionally, you can define a 'CONSTANT' before including Easy 4tH, which enables support for *arrays of string constants*:

```
<size> constant /STRING-SPACE
```

The parameter "SIZE" represents the size of the String Segment. When you decompile a 4tH program, it will show you exactly how much space is allocated to the String Segment. In order to port a single 4tH program, this is all the information you need!

```
4tH message : No errors at word 1105
Object size : 1106 words
String size : 2539 chars
Variables   : 19 cells
Strings     : 262 chars
Reliable    : Yes
```

In this case the "/STRING-SPACE" must be at least 2539 bytes. So, give or take a few changes, let's say 3072 bytes. We advise you to allocate a little more memory than is strictly necessary. You can also use Easy 4tH to make ANS-Forth understand 4tH. Just type:

```
16384 constant /STRING-SPACE
s" easy4th.4th" included
```

Now you can play around with ANS-Forth using the 4tH language. If you use a lot of string constants you might run out of space, but your ANS-Forth compiler will give you a message when that happens.

Note that - depending on the ANS-Forth compiler you're using - Easy 4tH *may* redefine some words, although it will try to minimize these redefinitions as much as possible.

21.4.2 The structure of Easy 4tH

Easy 4tH may look like a large program, but it isn't. It basically tries to figure out what your compiler supports and what is still left to define. It always prefers the native definition to its own. E.g. if your compiler already supports 'PLACE', Easy 4tH will leave that definition intact and *assume* it has been defined correctly.

- Easy 4tH will start by defining several defining words like 'STRING', 'STRUCT' and 'ARRAY'. Since there is no standard definition for these words, it will overwrite any existing definition.
- After that, Easy 4tH will query the environment and define a 'CONSTANT' when successful. When not, a warning is issued.
- Several 4tH specific compiling words are defined.
- Easy 4tH checks for the presence of several ANS-Forth and COMUS words. If they are not there, they are defined. Warnings are issued where applicable.
- In the next stage the PARSING, CONVERSION and TIME subsystems of 4tH are defined. Warnings are issued where applicable.

- All 4tH words that *cannot* be defined in ANS-Forth are marked as *unsupported*. When used, an error message is issued and compilation aborted.
- `ezneeds.4th` is loaded and '[NEEDS' and 'INCLUDE' are defined if needed.

Note that your own compiler may issue error messages or warnings too, e.g. about redefinitions.

21.5 The preprocessor

The preprocessor[6] (PP4tH) is also a very useful tool to make ANS-Forth compliant programs, especially when double numbers or floating point numbers are involved. The preprocessor supports a lot of ANS-Forth words which are not supported by 4tH like "SYNONYM", "ACTION-OF", etc. It allows you to enter double numbers or floating point constants with ease, provided you *included* the proper preprocessor library, e.g.:

```
f% 234.34e-2
```

Yes, "F%" isn't ANS-Forth either, but a very small library enables your ANS-Forth compiler to compile these numbers *without any additional overhead*, e.g.:

```
[DEFINED] 4TH# [IF]
include 4pp/lib/float.4pp
[ELSE]
s" lib/ezpp4th.4th" included
[THEN]

f% 234.34e-2 f. cr
```

For your convenience, those words are "state smart", so you can use them both inside and outside definitions as this sample session shows you:

```
Gforth 0.6.2, Copyright (C) 1995-2003 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type 'license'
Type 'bye' to exit
[DEFINED] 4TH# [IF]  ok
include 4pp/lib/float.4pp  ok
[ELSE]  ok
s" lib/ezpp4th.4th" included  ok
[THEN]  ok
f% 234.34e-2 f. cr 2.3434
 ok
: test f% 234.34e-2 f. cr ;  ok
test 2.3434
 ok
see test
: test
  2.343400E0 f. cr ; ok
```

Of course, you can write double number or floating point programs without the preprocessor, but they will contain a lot of "noise" words and are neither easy to read nor to maintain. Using the preprocessor really simplifies your life in these circumstances.

[6] See section 18.

21.6 Converting ANS-Forth programs to 4tH

4tH is a subset of ANS-Forth, so it might be difficult to find a program that will run on 4tH without at least *some* rewriting. And there is no guarantee that it will work, because most ANS-Forth programs weren't written with 4tH in mind. We'll list the major pitfalls:

- Programs requiring unsupported words or most words from the **FACILITY**, **FACILITY EXT**, **SEARCH** and **SEARCH EXT** wordsets are generally impossible to port.
- Definitions manipulating the dictionary or the stacks. But 4tH has no dictionary and does not allow direct access to the stacks.
- Definitions that switch between interpretation and compilation mode. 4tH either interprets or compiles; you cannot switch between the two on the fly. User-defined 'IMMEDIATE' words generally don't work.
- Definitions using 'CREATE' and 'DOES>' can be difficult to port. The only way is to do the 'CREATE' part manually and wrap the 'DOES>' part into a ':REDO' definition. Another way to port this kind of code is to resort to the 4tH preprocessor.
- Definitions requiring the **LOCAL** and **LOCAL EXT** wordsets are difficult to port. You'll need to rewrite them extensively by using the `locals.4th` library file.
- Definitions using ANS-Forth enhanced flow control require some rewriting and conditional compilation.
- Programs that assume they may store cells and characters in the same dataspace require some rewriting. Use the `ncoding.4th` library file.

Chapter 22

Errors guide

22.1 How to use this manual

This manual contains all the error messages 4tH can possibly issue. It is organized like this:

MESSAGE: This features the message from `errs_4th.c`, the error-code returned in `ErrNo` and the C-mnemonic.

WORDS: Words that can trigger this error.

EXAMPLE: This features a 4tH one-liner that will trigger the error.

CAUSE: This lists all possible causes of the error.

HINTS: This will give you some directions on how to fix the error.

22.2 Interpreter (exec_4th)

When exiting this function ErrLine will contain the address of the word in the Code Segment where the error occured.

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully executed.

HINTS: Make an error ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the Character Segment or the Integer Segment.

HINTS:

1. Reduce the amount of memory your program allocates and recompile.
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another operating system (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: An unknown token was encountered in the H-code.

HINTS: Contact us, this should never happen.

MESSAGE: **Stack overflow** (*#3 M4SOVFLW*)

WORDS: Any word that pushes items on the Data Stack.

EXAMPLE: `STACK 1+ 0 DO I LOOP`

CAUSE: The Data Stack collided with the Return Stack.

HINTS:

1. Don't push too many elements on the Data Stack.
2. Merge colon-definitions. Reduce the number of nested DO..LOOPs.
3. If you are using recursion, try if you can achieve the same result with a loop.
4. Make sure that your stacks are still balanced when returning from a colon-definition. Don't leave any unused data on the Data Stack. Flow-control words can have unexpected stack effects!

MESSAGE: **Stack empty** (*#4 M4SEMPTY*)

WORDS: Any word that pops items from the Data Stack.

EXAMPLE: `0 SWAP`

CAUSE: The Data Stack did not contain the required number of items to complete the operation.

HINTS:

1. Make sure that your stack is still balanced when returning from a colon-definition.

2. Make sure that the required number of items are on the stack when performing the operation.
3. If the problem occurs within an interpreter driven application, make sure that you check the number of elements are on the stack before allowing the operation.

MESSAGE: **Return stack overflow** (*#5 M4ROVFLW*)

WORDS: Any word that pushes items on the Return Stack; calling a user defined word

EXAMPLE: `: DUMMY DUMMY ; DUMMY`

CAUSE: The Return Stack collided with the Data Stack.

HINTS:

1. Don't push too many elements on the Data Stack.
2. Merge colon-definitions. Reduce the number of nested DO..LOOPs.
3. If you are using recursion, try if you can achieve the same result with a loop.
4. Make sure that your stacks are still balanced when returning from a colon-definition. Don't leave any unused data on the Data Stack. Flow-control words can have unexpected stack effects!

MESSAGE: **Return stack empty** (*#6 M4REMPTY*)

WORDS: Any word that pops items from the Return Stack; returning from a user defined word

EXAMPLE: `R>`

CAUSE: The Return Stack did not contain the required number of items to complete the operation.

HINTS:

1. Balance R> and >R inside your colon-definition. Flow-control words can have unexpected stack effects!
2. Be careful when using R> and >R inside a DO..LOOP.

MESSAGE: **Bad string** (*#7 M4BADSTR*)

WORDS: `ARGS OFFSET`

EXAMPLE: -1 ARGS

CAUSE: There was either no argument on the command line or no binary string constant with this index.

HINTS:

1. Use a valid index for ARGS.
2. Use a valid index for the offset.

MESSAGE: **Bad variable** (*#8 M4BADVAR*)

WORDS: `! @ +! ? SMOVE`

EXAMPLE: `6 ARRAY NAME NAME -5 TH @`

CAUSE: You tried to access a variable or array element, but its address in the Integer Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or stores are made.
2. Use a valid array index or address.
3. Don't transfer any cells to or from a invalid array index or address.
4. Don't try to overwrite read-only system variables.

MESSAGE: **Bad address** (*#9 M4BADADR*)

WORDS: All string handling words

EXAMPLE: `10 STRING BUFFER TIB CHAR- BUFFER /TIB CMOVE`

CAUSE: You tried to access a character, but its address in the Character Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or stores are made.
2. Make sure that the number of elements is correct when you use words like CMOVE, COUNT, FILL.
3. Terminate strings.
4. You exceeded the maximum length of PAD when you defined a string constant using S".
5. You exceeded the maximum length of PAD when you fetched a commandline argument using ARGS.

MESSAGE: **Divide by zero** (*#10 M4DIVBY0*)

WORDS: `/ MOD /MOD */ */MOD`

EXAMPLE: `1 0 / . CR`

CAUSE: You tried to divide by zero.

HINTS: Check the divisor before you use it.

MESSAGE: **Bad token** (*#11 M4BADTOK*)

WORDS: `@C EXECUTE EXIT CATCH`

EXAMPLE: `: DUMMY ; ' DUMMY 5 - DUP @C SWAP EXECUTE`

CAUSE: You tried to jump to a token or access the argument of a token, but its address in the Code Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or jumps are made.
2. Make sure the address you're using is within the Code Segment.
3. Be sure that the name after ' is that of a colon-definition.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: All words performing I/O

EXAMPLE: `OUTPUT FILE 5 . CR`

CAUSE:

1. You tried to read from or write to an unopened file.
2. You tried to USE, SEEK or TELL an unused stream.
3. There was an I/O error when you tried to read from or write to a file.
4. There was an error when you tried to close an open file with CLOSE.
5. There was an error when 4tH tried to close a file after the program terminated.

HINTS:

1. Open a file before you try to read or write to it. Check the value OPEN returns.
2. Make sure the values on the stack are correct when you perform I/O.
3. Make sure the values on the stack are correct when addressing streams.
4. Make sure that there is enough space left on the device you try to write to. Make sure it functions correctly.

MESSAGE: **Assertion failed** (*#15 M4ASSERT*)

WORDS: `)`

EXAMPLE: `[ASSERT] ASSERT( FALSE )`

CAUSE: The top of the stack was FALSE when) executed.

HINTS: Correct the condition) acted upon.

MESSAGE: **Unhandled exception** (*#16 M4THROW*)

WORDS: `THROW`

EXAMPLE: `1 THROW`

CAUSE: A THROW was encountered without a previous call from CATCH. The top of stack contained an error number outside the range of system errors.

HINTS: Make sure that a THROW can only be reached from a previous CATCH.

MESSAGE: **Bad radix** (*#17 M4BADRDX*)

WORDS: `.R . # NUMBER`

EXAMPLE: `1 BASE ! 5 . CR`

CAUSE: The 4tH variable BASE contained a value outside the 2 to 36 range during a conversion.

HINTS: Take care that BASE stays within the 2 to 36 range.

MESSAGE: **Bad stream** (*#18 M4BADDEV*)

WORDS: `USE SEEK TELL CLOSE`

EXAMPLE: `-1 CLOSE`

CAUSE:

1. The filehandle you tried to use was out of range.
2. You may not SEEK, TELL or CLOSE the streams STDIN and STDOUT.
3. You may not SEEK or TELL a pipe.

HINTS:

1. Make sure you use a proper stream when using USE, SEEK, TELL or CLOSE.
2. Check stack manipulations or use a variable or value.

MESSAGE: **Bad pointer** (*#20 M4BADPTR*)

WORDS: `CATCH THROW PAUSE`

EXAMPLE: `: ME R> R> R> DROP -5 >R >R >R 1 THROW ; ' ME CATCH`

CAUSE: The stack pointer THROW or PAUSE tried to use was invalid.

HINTS:

1. Be careful when you manipulate the Return Stack.
2. Contact us, this should never happen.

22.3 Compiler (comp_4th)

When exiting this function ErrLine will contain the address in the Code Segment where the next word would have been compiled if the error hadn't occured. This is logical, since 4tH always reports where the error occured. And all previous words have been succesfully compiled.

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A source was succesfully compiled.

HINTS: Make an error ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the H-code header, the Code Segment, the symbol-table or the control-stack.

HINTS:

1. Compact your source by removing all comment and whitespace or use the preprocessor[1].
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another compiler (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

[1] See chapter 18.

WORDS: All defining words

EXAMPLE: `CR CR 20 STRING`

CAUSE:

1. A word could not be compiled due to lack of space in the Code Segment.
2. A definition could not be compiled due to lack of space in the symbol-table.

HINTS:

1. Trying to make words private by using conditional compilation may trigger this error. Remove the offending HIDE.
2. In certain circumstances, incomplete data declarations may trigger this error. Complete the declaration.
3. Contact us, this should never happen with normal source-code.

MESSAGE: **Bad string** (*#7 M4BADSTR*)

WORDS: `SCONSTANT`

EXAMPLE: `6 SCONSTANT BAD`

CAUSE: The expression, which was compiled right before the word which caused the error, was not a string constant created by S" or S|.

HINTS: Use a string constant created by S" or S|.

MESSAGE: **Bad variable** (*#8 M4BADVAR*)

WORDS: `EQUATES`

EXAMPLE: `6 EQUATES BAD`

CAUSE: The expression, which was compiled right before the word which caused the error, did not compile to a variable.

HINTS: Use a expression, which compiles to a variable.

MESSAGE: **Divide by zero** (*#10 M4DIVBY0*)

WORDS: `/ /CONSTANT`

EXAMPLE: `0 /CONSTANT WRONG 1 WRONG`

CAUSE:

1. You tried to divide by zero in a literal expression.
2. You tried to apply a zero /CONSTANT.

HINTS:

1. Check the divisor before you use it.
2. Don't define a zero /CONSTANT.

MESSAGE: **Wrong type** (*#12 M4NOTYPE*)

WORDS: `TO IS ALIAS DEFER FILE VALUE :REDO DOES> TAG`

EXAMPLE: `0 CONSTANT WRONG 5 TO WRONG`

CAUSE:

1. The name after TO isn't a VALUE.
2. The name after IS isn't a DEFER.
3. The name after DEFER or ALIAS is already defined and not a DEFER.
4. The name after VALUE or FILE is already defined and not a VALUE.
5. You tried to create a :REDO or DOES> definition for an unsupported datatype.
6. The name after TAG isn't an OFFSET.

HINTS:

1. Use TO only in combination with a VALUE.
2. Use IS only in combination with a DEFER.
3. Use a different name.
4. Use a different name.
5. Use a proper datatype when creating a :REDO or DOES> definition.
6. Use TAG only in combination with an OFFSET.

MESSAGE: **Undefined name** (*#13 M4NONAME*)

WORDS: *<name>* `' ['] RECURSE AKA HIDE`

EXAMPLE: `' HELLO ( "hello" is not defined)`

CAUSE:

1. The name which caused the error is not present in the symbol-table.
2. The name is not defined at all.
3. It is not a valid number in the current radix.
4. Underflow or overflow occured during number conversion.

5. RECURSE is used outside a colon definition.
6. The name you used is longer than WIDTH characters.

HINTS:

1. Note that most of the words above only work with names defined inside the program and not with built-in names[2].
2. Usually a typo; correct spelling.
3. Set the appropriate radix by using [BINARY], [OCTAL], [DECIMAL] or [HEX].
4. Don't use numbers less or equal to (ERROR) or greater than MAX-N.
5. Remove the offending RECURSE.
6. Use a shorter name.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: `[NEEDS INCLUDE`

EXAMPLE: `[NEEDS nosuchfile.4th]`

CAUSE:

1. The source file you tried to read doesn't exist.
2. There was an error reading the source file.
3. There was an error when 4tH tried to close the source file.

HINTS:

1. Make sure the file you try to open exists and is in the path. Change your working directory if necessary. Check the *DIR4TH* environment variable.
2. Make sure the device functions correctly.

MESSAGE: **Bad literal** (*#19 M4BADLIT*)

WORDS: All words requiring a literal expression

EXAMPLE: `10 DUP ARRAY NAME`

CAUSE: The expression, which was compiled right before the word which caused the error, did not compile to a literal.

HINTS: Use a literal expression.

[2] AKA *does* offer limited capability in this field.

MESSAGE: **Nesting too deep** (*#21 M4NONEST*)

WORDS: All flow control words and colon definitions

EXAMPLE: `10 0 DO 10 0 DO` *<more flow-control structures>* `LOOP LOOP`

CAUSE: The control-stack, that holds all references to addresses of flow-control structures in the Code Segment, overflowed.

HINTS: Make separate colon-definitions of the flow-control structures that caused the error.

MESSAGE: **No program** (*#22 M4NOPROG*)

WORDS: All words that do *not* compile any tokens

EXAMPLE: `10 ARRAY NAME ( Won't compile)`

CAUSE:

1. The source didn't contain any compilable words.
2. The source was corrupt.
3. A runaway comment or conditional compilation clause.
4. In rare cases use of reserved words as names.

HINTS:

1. Make a program that does something.
2. Make sure that the source actually contains 4tH source- code.
3. Terminate your comments and conditional compilations properly.
4. Don't use any reserved words as names.

MESSAGE: **Incomplete declaration** (*#23 M4NODECL*)

WORDS: All defining words and compiler directives

EXAMPLE: `10 CONSTANT CONSTANT NAME`

CAUSE:

1. Syntax errors; usually a missing name or a literal expression.
2. Incomplete compiler directives or expressions, like a leading comma or a trailing CHAR.
3. An assertion, beginning with ASSERT(, is missing a right parenthesis. Assertions are not enabled at that point.
4. An [IF] is not balanced by a [THEN].

HINTS:

1. Use an appropriate expression or name.
2. Complete compiler directives and expressions.
3. Add a right parenthesis at the end of the expression.
4. Add a [THEN] for each [IF] statement.

MESSAGE: **Unmatched conditional** (*#24 M4NOJUMP*)

WORDS: All flow control words and colon definitions

EXAMPLE: `: WRONG IF DROP BEGIN FALSE LOOP ;`

CAUSE: The flow-control word that caused the error didn't match with the previous flow-control word (BEGIN after IF) or was missing.

HINTS: Use the appropriate flow-control word to terminate a flow-control structure.

MESSAGE: **Unterminated string** (*#25 M4NOSTR*)

WORDS: `." \ ( .( ," S" ABORT" S| ,| [CHAR] CHAR @GOTO [NEEDS INCLUDE [DEFINED] [UNDEFINED]`

EXAMPLE: `." Hello world`

CAUSE:

1. A required delimiter is missing at the end of a string.
2. An internal error occured at the very end of the source.

HINTS:

1. Add the required delimiter at the end of the string.
2. Contact us, this should never happen.

MESSAGE: **Null string** (*#26 M4NULSTR*)

WORDS: *See error #25*

EXAMPLE: ." "

CAUSE:

1. The string between the word and its delimiter did not contain any characters.
2. There was more than one whitespace character between a [DEFINED], [UNDEFINED], [CHAR], CHAR or INCLUDE and the string following it.

HINTS:

1. Use a string that contains at least one single character.
2. Delete all superfluous whitespace characters between [DEFINED], [UNDEFINED], [CHAR], CHAR or INCLUDE and the string following it.

MESSAGE: **Duplicate name** (*#27 M4DUPNAM*)

WORDS: All defining words

EXAMPLE: `: TH CELLS + ;`

CAUSE: The name you used for a definition is already in use by 4tH or your own program.

HINTS: Use a different name.

MESSAGE: **Name too long** (*#28 M4BADNAM*)

WORDS: All defining words

EXAMPLE: `VARIABLE JUSTTOOLONGANAMEFORCOMFORT`

CAUSE: The name you used for a definition is too long.

HINTS: Use a shorter name.

MESSAGE: **Compilation aborted** (*#29 M4CABORT*)

WORDS: `[ABORT]`

EXAMPLE: `[ABORT]`

CAUSE: An [ABORT] directive was encountered during compilation.

HINTS: The original programmer must have had a reason to abort compilation in this particular circumstance. See the program for additional information.

22.4 Loader (load_4th)

Since the loader works with complete segments, the words don't have to do much with fixing an error. Therefore, it reports that nothing has been loaded (word 0) or everything has been loaded (the last word).

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully loaded.

HINTS: Keep up the good work. ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the header, the Code Segment or the String Segment.

HINTS:

1. Reduce the amount of memory your program allocates and recompile.
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another compiler (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. You tried to load a file, that was not an HX-file.
2. You tried to load an HX-file from a previous version of 4tH.
3. You tried to load an HX-file for a different application.
4. You tried to load an HX-file for a different architecture.
5. You tried to load an inconsistent HX-file.

HINTS:

1. Use a proper HX-file.
2. Recompile the source, using the current 4tH compiler.
3. If the source is compatible, you might recompile the source, using your own 4tH compiler.
4. If the source is compatible, you might recompile the source, using your own 4tH compiler.
5. Recompile the source, using your own 4tH compiler.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. The file could not be opened.
2. There was an I/O error while the file was read.
3. The file could not be closed.

HINTS:

1. Use a valid filename.
2. Make sure the device functions correctly.
3. Make sure the device functions correctly.

22.5 Saver (save_4th)

Since the saver works with complete segments, the words don't have to do much with fixing an error. Therefore, it reports that nothing has been saved (word 0) or everything has been saved (the last word).

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully saved.

HINTS: Keep up the good work. ;)

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. The file could not be opened.
2. There was an I/O error while the file was written.
3. The file could not be closed.

HINTS:

1. Make sure you got enough inodes or directory-entries left on the device you want to write to. Use a valid filename.
2. Make sure that there is enough space left on the device you try to write to. Make sure it functions correctly.
3. Make sure the device functions correctly.

Chapter 23

4tH library

23.1 4tH library files

This list contains all the 4tH library files with short descriptions of their functionality. If a file only contains ANS-Forth words, these words are listed.

3dup3rot.4th	Standard stack operators for manipulating three stack items.
4dup4rot.4th	Standard stack operators for manipulating four stack items.
acker.4th	An implementation of the Ackermann function.
acptword.4th	Read a word from a file, regardless of linebreaks.
adler32.4th	A port of the FFL ”Adler 32 checksum” module.
allsheet.4th	Write CSV, XLS, XML, KSP, FODS or HTML spreadsheet files.
ansblock.4th	BLK, UPDATE, CLEAR, EMPTY-BUFFERS, SAVE-BUFFERS, FLUSH, BLOCK, BUFFER, LIST
anscore.4th	2!, 2@, 2OVER, 2ROT
ansdbl.4th	U<, U>, D+, D2*, DNEGATE, D-, D<, DABS, D2/, DU<, D=, D0<, D0=, DMAX, DMIN
ansfacil.4th	TIME&DATE, MS
ansfile.4th	FILE-SIZE, READ-LINE, OPEN-FILE, CLOSE-FILE, FILE-POSITION, REPOSITION-FILE, READ-FILE, WRITE-FILE, FLUSH-FILE, WRITE-LINE, FILE-STATUS, CREATE-FILE, BIN, R/O, W/O, R/W
ansfloat.4th	FLOAT, F2/, F2*, PRECISION, SET-PRECISION, FCLEAR, FDEPTH, FDUP, FDROP, FNEGATE, D>F, F>D, FLOAT+, FLOATS, F@, F!, FSIGN?, F0=, F0<, FABS, FALIGN, FALIGNED, FSWAP, FPICK, FNIP, FROT, F+, F-, F<, F=, FLOOR, FROUND, FMIN, FMAX, F*, F/, F~, FSQRT
ansfpio.4th	F., R., FS., FE., >FLOAT, F.S
ansmem.4th	ALLOCATED, FREE, PARAGRAPH, ALLOCATE, RESIZE

ansquote.4th	An implementation of an S\" like word.
ansren.4th	RENAME-FILE
anstools.4th	.S
arcfour.4th	An implementation of the RC4 encryption algorithm.
argopen.4th	Allows you to easily open files, whose names were listed on the command line.
ascii7.4th	Filtering strings to the ASCII-7 subset.
asciixml.4th	Conversion of XML character entities.
asinacos.4th	FATAN, FASIN, FACOS
back.4th	The TOOLBELT implemention of the backward string tokenizing words -SCAN and BACK.
backtrak.4th	An implementation of the backtracking words by M.L. Gassanenko.
banners.4th	An implementation of the Unix 'banner' functionality.
base64.4th	Base64 encoding and decoding words.
basename.4th	Several POSIX `basename()`, `dirname()` like words.
binomial.4th	An implementation of binomial coefficients and Catalan numbers.
bintree.4th	An implementation of a "binary tree", containing keys and values.
bintrkv.4th	An implementation of an associative array, using a binary tree.
bitarray.4th	An implementation of bit arrays.
bitfield.4th	An implementation of bitfields.
bitosort.4th	An implementation of the bitonic sort algorithm.
bitrevse.4th	A library to reverse bitpatterns.
bits.4th	Several words to manipulate individual bits.
bitset.4th	Several BSD `ffs()`, `fls()` like words.
boxes.4th	Places ASCII boxes around strings.
breakq.4th	Tests for character equality in a string.
bsearch.4th	An implementation of a binary search.
bub2sort.4th	An implementation of the bubble sort algorithm.
bublsort.4th	An implementation of the bubble sort algorithm.
buffer.4th	An implementation of user-defined ring buffers.
cansignd.4th	Prints numbers in canonical signed digit (CSD) representation.
capitals.4th	Converts strings to CaMeL case.
cards.4th	A library which allows you to easily implement card games.
charat.4th	Some `strchr()` like words.

chars.4th	All printable ASCII-7 characters turned into constants.
chi2.4th	Several ”Chi square” related floating point functions.
chmatch.4th	A library with regular expression like words.
choose.4th	Some words that return a random number within a user-defined range.
cir2sort.4th	An implementation of the enhanced circle sort algorithm.
circsort.4th	An implementation of the circle sort algorithm.
coc2sort.4th	An implementation of the optimized cocktail sort algorithm.
cocksort.4th	An implementation of the cocktail sort algorithm.
com2sort.4th	An implementation of the ”comb sort with a different ending” algorithm.
combsort.4th	An implementation of the ”comb sort with a different ending” algorithm.
compare.4th	COMPARE
comus.4th	Some well known COMUS words.
concat.4th	Concatenate multiple character strings.
constant.4th	Several constants.
convert.4th	A template which allows you to easily create simple conversion programs.
country.4th	A table with ISO 3166-1 country codes.
countstr.4th	Support for ANS-Forth counted strings.
cp437htm.4th	Converts CP437 characters to HTML character entities.
crc.4th	An implementation of the CRC16 encoding.
create-w.4th	A library for outputting tables in 4tH format.
csimpson.4th	A library for numerical integration using ”Simpsons composite rule”.
cstring.4th	Extracting characters from the beginning or the end of a string.
csvfrom.4th	Converts a string with embedded double (CSV escaped) quotes.
csv-w.4th	Write CSV files.
ctos.4th	Converts a character to a string.
cyclsort.4th	An implementation of the cycle sort algorithm.
date2xls.4th	Converts Julian dates to/from the Excel-1900 format.
dblbin.4th	Double word binary operations.
dbldiv.4th	Double word divisions.
dbldot.4th	D.R, D., U.R, U.
dblsharp.4th	Double word equivalents of the <#, #>, SIGN, # and #S words.
dbm.4th	A tiny database management system.

dbmidx.4th	An index extension to the tiny DBM.
dbms.4th	Binds indexes to DBM tables and extends CRUD operations.
ddigroot.4th	Calculates OEIS A010888 sequence and additive persistence of double cell numbers.
ddotf.4th	Prints double numbers as floating point numbers.
debug.4th	An implementation of a breakpoint debugger.
digiroot.4th	Calculates OEIS A010888 sequence and additive persistence of single cell numbers.
digit.4th	Converts a numeric character to its numeric equivalent.
discord.4th	A library for calculating the Discordian or Erisian calendar.
drange.4th	A double word equivalent of WITHIN.
dsqrt.4th	Double word square root.
dst.4th	Determines ”daylight saving time” in the US and EU.
dump.4th	DUMP
dumpbase.4th	DUMP
easter.4th	Calculates the date of Easter for a given year.
easy.4th	An ANS-Forth library to port 4tH programs to ANS-Forth.
ellipint.4th	An implementation of the Complete Elliptic Integrals functions.
embed.4th	Converts a string with embedded ASCII codes.
emits.4th	Emits a single character several times.
enclose.4th	Retrieve the string between the delimiters.
enter.4th	Enter a number from the keyboard.
environ.4th	ENVIRONMENT?
erf.4th	A high accuracy implementation of the `erf()` function.
erf1.4th	An alternative implementation of the `erf()` function.
escape.4th	Converts a string with embedded escape codes.
etl.4th	Extract, transform and load library for CSV files.
evaluate.4th	SAVE-INPUT, RESTORE-INPUT, EVALUATE
exceptn.4th	A minimal implementation of the Sourceforge library extention to the ANS EXCEPTION wordset.
expout.4th	Simple floating point exponential output library.
ezneeds.4th	Part of the ANS-Forth library to port 4tH programs to ANS-Forth (`easy.4th`).
ezpp4th.4th	An ANS-Forth library defining the preprocessor prefixes D% and F%.
falog.4th	F**, FALOG

fatan2.4th	FATAN2
fatanh.4th	FASINH, FACOSH, FATANH
fbessel.4th	An implementation of Regular spherical Bessel functions.
fbeta.4th	An implementation of the Complete Beta function.
fbetacdf.4th	An implementation of the Incomplete Beta function.
fbetapdf.4th	An implementation of the Beta Probability density function.
fcartes.4th	Cartesian to polar coordinates conversion.
fcbrt.4th	An implementation of the floating point cube root function.
fdeg2rad.4th	Degrees to radians conversion (and vice versa).
felip.4th	An implementation of the Complete Elliptic Integral function.
fenter.4th	Enter a floating point number from the keyboard.
fequals.4th	An implementation of several extended floating point comparison words.
ferf.4th	An implementation of the `erf()` function.
fexp.4th	FEXP
fexpint.4th	An implementation of the Real Exponential Integral function.
fexpm1.4th	FEXPM1
fexpt.4th	FEXP
ffl-frc.4th	An implementation of words for using fractions.
fgamma.4th	An implementation of the high accuracy Spouge gamma function.
fhaversn.4th	An implementation of the haversine formula.
figures.4th	Prints out numbers in full English words.
files.4th	Several words for querying and manipulating files.
filestat.4th	An interface to the Unix `stat` command.
filter.4th	Filters a character out of a string.
filtring.4th	Filters several characters out of a string.
fixeddot.4th	Prints out numbers in fixed point notation.
fixedfld.4th	A library for reading fixed length field files.
flnflog.4th	FLN, FLOG
flnflogb.4th	FLN, FLOG
flnp1.4th	FLNP1
flocals.4th	A library for implementing local floating point variables.
flogist.4th	An implementation of the logistic function and its first derivative.
floormod.4th	A library for floored integer division.

forwdiff.4th	An implementation of the forward difference function.
fp0.4th	Floating point configuration 0.
fp1.4th	Floating point configuration 1.
fp2.4th	Floating point configuration 2.
fp3.4th	Floating point configuration 3.
fp4.4th	Floating point configuration 4.
fpconst.4th	Several floating point constants.
fpdot.4th	Floating point pictured numeric output operators.
fpin.4th	>FLOAT
fpolar.4th	Polar to Cartesian coordinates conversion.
fpout.4th	FS., FS.R, (FS.), FE., FE.R, (FE.), F., F.R, (F.), G., G.R, (G.)
fpow.4th	Fast initialization of ANS floating point constants with extreme exponents.
fpow10.4th	Fast initialization of ANS floating point base 10 constants with extreme exponents.
fq.4th	An implementation of the ”Q” function.
fracout.4th	Simple fixed-point output with optional thousands separator.
fraction.4th	An implementation of fractional arithmatic words.
frexp.4th	Several `frexp()`, `ldexp()` like words.
fsincost.4th	FSIN, FCOS, FSINCOS, FTAN
fsinfcos.4th	FSIN, FCOS, FSINCOS, FTAN
fsl-util.4th	The utilities for the Forth Scientific Library.
fsm.4th	Finite States Machines, Julian Noble style.
fsprintf.4th	An almost complete implementation of `sprintf()` with floating point support.
fstack.4th	A library for creating extra floating point stacks.
ftrunc.4th	FTRUNC, FCEIL
fvector.4th	Several ANS floating point vector functions.
fwt.4th	A port of the FFL ”fixed width types” module.
fzeta.4th	An implementation of the Riemann zeta function.
gamma.4th	An implementation of the gamma, loggamma and reciprocal gamma functions.
gammaln.4th	An implementation of the Lanczos loggamma function.
gammapq.4th	An implementation of the Incomplete Gamma function.

gauslege.4th	A library for numerical integration using a 10-point Gauss–Legendre quadrature.
gauss.4th	An implementation of the Gaussian (normal) probablity functions.
gbanner.4th	Graphic character output, part of Portable Bitmap graphics suite.
gbk2utf8.4th	Converts GBK/2 encoded strings to UTF8.
gc.4th	A simple ”mark-and-release” garbage collector.
gcdlcd.4th	An implementation of the least and greatest common denominator functions.
gcol2gry.4th	Color to grayscale conversion, part of the Portable Bitmap graphics suite.
getdate.4th	A library for detecting and converting several date formats.
getenv.4th	A `getenv()` like word.
getopts.4th	Some `getopt()` like words.
gplot3d.4th	A library for plotting and generating 3D graphics images.
gmkiss.4th	A high performance random number generator.
gmskiss.4th	An even higher performance random number generator.
gno2sort.4th	An implementation of an optimized gnome sort algorithm.
gnomsort.4th	An implementation of the gnome sort algorithm.
graphics.4th	An implementation of Portable Bitmap graphics functions.
gshrink.4th	Halves the dimensions of an image, part of the Portable Bitmap graphics suite.
gtkipc.4th	An alternative interface to GTK-server.
gtkserv.4th	An interface to GTK-server.
gturtle.4th	An implementation of Turtle graphics, part of the Portable Bitmap graphics suite.
gview.4th	Show an image in ASCII, part of the Portable Bitmap graphics suite.
hamming.4th	An implementation of the Hamming (7,4) linear error-correcting codes.
harshad.4th	Identifies Harshad numbers.
hash.4th	An implementation of several well known hash functions.
hashbuck.4th	An implementation of hash tables with buckets.
hashkey.4th	An implementation of hash tables with binary search.
hashtabl.4th	An implementation of hash tables without buckets.
headings.4th	An implementation of ”boxing the compass”.
hea2sort.4th	An implementation of the heap sort algorithm.
heapsort.4th	An implementation of the heap sort algorithm.

hiorder.4th	Several high order words.
hms2secs.4th	Decodes time strings.
holds.4th	HOLDS
horner.4th	An implementation for evaluation of a polynomial by the Horner method.
html.4th	Create simple HTML 4.01 Transitional formatted files.
html-w.4th	Writes tables to HTML 4.01 Transitional formatted files.
huntjoin.4th	The TOOLBELT implemention of the string replacing words HUNT and JOIN.
i8859htm.4th	Converts ISO-8859 characters to HTML character entities.
identify.4th	Converts a string to an identifier.
idsystem.4th	Differentiating between Microsoft and POSIX OSes.
info.4th	A library for making Unix `man` like information systems.
inifile.4th	A library for interpreting INI files.
inimanag.4th	A library for creating and modifying INI files.
ins2sort.4th	An implementation of the insert sort algorithm.
instsort.4th	An implementation of the insert sort algorithm.
interprt.4th	A library for making simple Forth-like interpreters.
intpfile.4th	A library for interpreting Forth-like scripts from file.
intrsort.4th	An implementation of the intro sort algorithm.
isdate.4th	Checks whether a date is valid.
ismpsort.4th	An implementation of the improves simple sort algorithm.
isprime.4th	Checks whether a number is a prime number.
isqrt.4th	A very fast integer square root function.
istype.4th	Several `isalnum()`, `isalpha()`, `isascii()`, `isdigit()` like words.
json-w.4th	Write JSON ECMA-404 files.
kaprekar.4th	Checks whether a number is a Kaprekar number.
key.4th	KEY, KEY?
koksp-w.4th	Write KOffice KSpread XML 1.x files.
kpre.4th	An enhanced implementation of the Regular Expressions engine by Brian W. Kernighan and Rob Pike.
latex.4th	Create simple LaTeX files.
leading.4th	Strips leading spaces from a string.
leaf.4th	An implementation of key/value pairs using a linked list.

legacy.4th	An implementation of the legacy word CONVERT.
license.4th	Prints the GPLv2 license.
lists.4th	An implementation of LISP-like list functions.
locals.4th	A library for implementing local variables.
logger.4th	Write Linux-like log-files.
logtime.4th	Format the current time and date as in Linux log-files.
longjday.4th	Prints a date in a long or short format.
luhn.4th	Performs the Luhn test, which is used by some credit card companies to distinguish valid credit card numbers.
lz77.4th	An implementation of the 1977 Ziv-Lempel file compression algorithm.
m3todate.4th	Converts a large number of ”DD-MMM-YY(YY)” strings to day, month, year format.
math.4th	Several integer math words.
mbinoml.4th	An implementation of binomial coefficients and Catalan numbers.
md5.4th	A port of the FFL ”MD5 checksum” module.
memcell.4th	ALLOCATED, FREE, ALLOCATE, RESIZE
memchar.4th	ALLOCATED, FREE, ALLOCATE, RESIZE
menu.4th	A library for creating simple menus.
mergsort.4th	An implementation of the merge sort algorithm.
mersenne.4th	An implementation of the Mersenne Twister.
message.4th	A more complete implementation of the Sourceforge library extention to the ANS EXCEPTION wordset.
mif-prng.4th	An implementation of the Philips/NXP Mifare Crypto-1 randomizer.
mixed.4th	UM/MOD, FM/MOD, SM/REM, UM*, M*, M*/, M+, MU*, MU/MOD
mon3date.4th	Formats a date in the DD-MMM-YYYY notation.
morse.4th	Words for encoding and decoding Morse signs.
msxls2-w.4th	Write MS-Excel 2.1 files.
msxmls-w.4th	Write MS-Excel XML 2003 files.
multiblk.4th	BLK, UPDATE, CLEAR, EMPTY-BUFFERS, SAVE-BUFFERS, FLUSH, BLOCK, BUFFER, LIST
multifac.4th	A library for calculating multi factorials.
ncoding.4th	A library for storing numbers in character buffers.
norm-pdf.4th	An implementation of several Probability Density functions.
ntor.4th	Saves or retrieves several values to or from the return stack.

obsolete.4th	#TIB, EXPECT, QUERY
odevsort.4th	An implementation of the ”odd even” merge sort algorithm.
oofods-w.4th	Write OpenOffice FODS files.
opgftran.4th	Infix to postfix formula translation.
ordidate.4th	An implementation of several ”ordinal date” and week functions.
oyelsort.4th	An implementation of the Oyelami sort algorithm.
padding.4th	Pads strings and prints padded strings.
palette.4th	Several Portable Bitmap palette manipulation words.
palindrm.4th	Checks whether a string is a palindrome.
pancsort.4th	An implementation of the pancake sort algorithm.
pangram.4th	Checks whether a string is a pangram.
parsexml.4th	A library which allows easy parsing of XML and HTML files.
parsing.4th	Several enhanced parsing words.
parsname.4th	An implementation of PARSE-NAME, which allows parsing regardless of whitespace character.
pcylfun.4th	An implementation of parabolic cylinder functions U and V, plus related confluent hypergeometric functions.
perfect.4th	Checks whether a number is a perfect number.
permcomb.4th	An implementation of permutations and combinations.
perrin.4th	An implementation of the Perrin sequence.
phone.4th	A library for parsing and standardizing phone numbers.
pickroll.4th	PICK, ROLL, ?DUP
picture.4th	An implementation of pictured numeric output words.
placelne.4th	Parse a buffer and copy the resulting line to a string variable.
place-n.4th	Copy a string several times to a string variable.
plecuyer.4th	An implementation of the Pierre LEcuyer randomizer.
printf.4th	A tiny implementation of `printf()`.
prio-q.4th	An implementation of a priority queue.
prng.4th	Leaves a pseudo random number in the range (0,1) on the FP stack.
polys.4th	An implementation of Chebyshev, Bessel, Hermite, Laguerre and Legendre polynomials.
prefixno.4th	Formats prefixed numbers.
print.4th	Printing of strings with automatic linebreaks.
qsort.4th	An implementation of the `qsort()` function.

quiz.4th	A template for creating and evaluating simple quiz programs.
quotes.4th	Removes leading and trailing quotes from a string.
ramdisk.4th	An implementation of a simple RAM disk.
ran4.4th	An implementation of the RAN4 random number generator.
randbin.4th	Generates random binary patterns.
randlcg.4th	A collection of linear congruential random number generators.
randlcg2.4th	A collection of linear congruential random number generators.
randmwcp.4th	An implementation of a ”Multiply With Carry” random number generator.
random.4th	Simple linear congruential random number generators.
randoom.4th	An exact copy of the ”Doom” random number generator.
range.4th	WITHIN, BETWEEN
rcheck.4th	Checks if the Return Stack is balanced.
rdepth.4th	DEPTH and .S like words for the return stack.
refills.4th	A library to store file position markers on a stack.
replace.4th	Several words for replacing and deleting substrings within strings.
represnt.4th	REPRESENT
reverse.4th	An implementation of the `strrev()` function.
rot13.4th	An implementation of the ROT13 function.
row.4th	A library for searching lookup tables.
rtf.4th	Create simple RTF 1.x formatted files.
runiform.4th	An implementation of the Uniform randomizer.
savefile.4th	SAVE-INPUT and RESTORE-INPUT like words for files.
say.4th	An interface to the Festival library.
sbreak.4th	An implementation of the `strpbrk()` function.
scanfile.4th	A library for quickly skipping to a predefined position in a text file.
scanskip.4th	An implementation of the COMUS string tokenizing words SKIP, SCAN and SPLIT.
scomma.4th	Add thousands separator to a decimal numeric string.
search.4th	SEARCH
sel2sort.4th	An implementation of the selection sort algorithm.
selcsort.4th	An implementation of the selection sort algorithm.
semiprim.4th	A library for testing semi-prime numbers.
sha1.4th	A port of the FFL ”SHA-1 checksum” module.

sha256.4th	A port of the FFL ”SHA-2 checksum” module.
shelsort.4th	An implementation of the Shell sort algorithm.
shuffle.4th	A library for shuffling character and numeric arrays.
simp38.4th	A library for numerical integration using ”Simpsons 3/8 rule”.
simpson.4th	A library for numerical integration using ”Simpsons rule”.
simpsort.4th	An implementation of the simple sort algorithm.
sinhcosh.4th	FSINH, FCOSH, FTANH
slice.4th	Several string slicing words.
slowsort.4th	An implementation of the slow sort algorithm.
soundex.4th	An implementation of the SOUNDEX function.
speak.4th	An interface to the eSpeak library.
spelldis.4th	An implementation of a simple ”spelling distance” algorithm.
split.4th	The TOOLBELT implemention of the string splitting words /SPLIT and -SPLIT.
splitext.4th	Splits a string at a given substring.
sprintf.4th	A tiny implementation of `sprintf()`.
srtstack.4th	Sorts the Data Stack.
stack.4th	An implementation of user-defined stacks.
startend.4th	The TOOLBELT implemention of the string comparison words STARTS? and ENDS?.
statist.4th	Several floating point statistical functions.
stemleaf.4th	A library for plotting stem-leaf diagrams.
stester.4th	A small library by Josh Grams for testing Forth words.
stmstack.4th	An implementation of a single string stack with error detection.
stoosort.4th	An implementation of the stooge sort algorithm.
strbuf.4th	A library for creating and managing string buffers.
strcount.4th	Counts the number of certain substrings in a string.
strstack.4th	An implementation of a string stack using a circular buffer.
stsstack.4th	An implementation of a true string stack.
substit.4th	REPLACES, SUBSTITUTE, UNESCAPE
sumdivs.4th	Calculates the sum of all positive divisors, a.k.a. divisor function.
swapping.4th	Exchanges one character in a string for another.
system.4th	An implementation of the `system()` function.
tabs.4th	Converts tabs in strings to spaces and vice versa.

taylor.4th	A library for creating approximation functions using Taylor series.
tea.4th	An implementation of the TEA encryption algorithm.
tean.4th	An implementation of the TEA New encryption algorithm.
termansi.4th	Several ANSI Terminal words.
threevl.4th	An implementation of ”three value logic”.
throw.4th	Several THROW constants.
throwstr.4th	Converts a THROW code to a string.
th-word.4th	The TOOLBELT implemention of the substring extraction words TH-WORD and TH-WORD-BACK.
time.4th	Several time and date related words.
timer.4th	Words for measuring execution times.
toarabic.4th	Converts Roman numerals to Arabic numerals.
todate.4th	Recognizes, parses and converts date strings.
todbl.4th	>NUMBER
totriple.4th	Converts a string to a triple length number.
tokenize.4th	A library for tokenizing strings.
tonumber.4th	>NUMBER
toolbelt.4th	Some well known TOOLBELT words.
toolfile.4th	Some TOOLBELT inspired, contextless I/O words.
toroman.4th	Converts Arabic numerals to Roman numerals.
totime.4th	A library for printing time stamps.
translat.4th	An implementation of the PHP `strtr()` function.
tre.4th	An implementation of a tiny Regular Expressions engine.
trim.4th	The TOOLBELT implemention of the string leading and trailing whitespace stripping words.
triple.4th	A library of arithmetic words for triple length numbers.
triplus.4th	Triple word equivalents of D+, D2/ and D2*.
trisharp.4th	Triple word equivalents of the <#, #>, SIGN, # and #S words.
tst.4th	A port of the FFL test module.
tt800.4th	An implementation of the TT800 random number generator.
ttester.4th	A library for testing Forth words.
ulcase.4th	Converts strings and characters from upper case to lower case and vice versa.
unicdgbk.4th	Converts Chinese Unicode characters to GBK/2.

unitconv.4th	Converts imperial units to SI units and vice versa.
userpad.4th	An implementation of a user-defined circular string buffer.
ustodate.4th	Converts a large number of "Month Day Year" strings to day, month, year format.
utf8.4th	An implementation of the UTF-8 codec.
utf8gbk2.4th	An UTF-8 to GBK/2 characterset conversion.
utf8type.4th	An implementation of TYPE with built-in ISO-8859 to UTF8 conversion.
version.4th	A word returning the 4tH version as a string.
vmem.4th	Allocates variables, strings and arrays in blockfiles.
well512.4th	An implementation of the WELL512 random number generator.
westhtml.4th	A CP437 and ISO-8859 Western characterset HTML conversion.
wichill.4th	An implementation of the Wichman Hill randomizer.
wildcard.4th	A word for comparing strings using wildcards.
wildcrdr.4th	A recursive word for comparing strings using wildcards.
word.4th	WORD
xchar.4th	X-SIZE, XC-SIZE, XC@+, XC!+, XC!+?, XCHAR+, XEMIT, -TRAILING-GARBAGE, X-WIDTH, XHOLD, XC-WIDTH, +X/STRING , X\STRING- , XCHAR-
yesno.4th	A library with vectors for inverting words.
yesorno.4th	Get a 'yes' or 'no' answer from the console and return a flag.
yield.4th	An implementation of generator-type coroutines.
zenans.4th	FROT, FOVER, FDUP, FSWAP, FDROP, F@, F!, FLOATS, PRECISION, SET-PRECISION
zenatan2.4th	FATAN2
zenatanh.4th	FASINH, FACOSH, FATANH
zencart.4th	Cartesian to polar coordinates conversion.
zendegrd.4th	Degrees to radians conversion (and vice versa).
zenexpm1.4th	FEXPM1
zenfalog.4th	F**, FALOG
zenfasin.4th	FATAN, FASIN, FACOS
zenfcbrt.4th	An implementation of the floating point cube root function.
zenferf.4th	An implementation of the `erf()` function.
zenfexp.4th	FEXP
zenfln.4th	FLN, FLOG

zenflnp1.4th	FLNP1
zenfloat.4th	FLOAT, FLOAT+, F+, FNEGATE, F-, FABS, F*, F/, F0=, F0<, F<, F=, D>F
zenfloor.4th	FLOOR
zenfmin.4th	FMIN, FMAX
zenfpio.4th	>FLOAT, F.
zenfprox.4th	F~
zenfsin.4th	FSIN, FCOS, FTAN, FSINCOS
zenfsinh.4th	FSINH, FCOSH, FTANH
zenfsqrt.4th	FSQRT, F2/
zengamma.4th	An implementation of the Gamma function by the Mortici method.
zengamm2.4th	A slower, but more accurate implementation of the Gamma function by the Ramanujan method.
zenhaver.4th	An implementation of the haversine formula.
zenhornr.4th	An implementation for evaluation of a polynomial by the Horner method.
zenpolar.4th	Polar to Cartesian coordinates conversion.
zenround.4th	FROUND
zentaylr.4th	A library for creating approximation functions using Taylor series.
zentodbl.4th	F>D
zentrunc.4th	FTRUNC, FCEIL
zxrandom.4th	An exact copy of the Sinclair ZX Spectrum floating point random number generator.

23.2 Library dependencies

Here all mixed, double and floating point library files and their dependencies are listed. If only *one* of the listed files is needed resolve a dependency, they are printed in *italics*.

Library	Family	Depends on
ansdbl.4th		anscore.4th constant.4th
ansfloat.4th	ANS	mixed.4th
ansfpio.4th	ANS	ansfloat.4th dblsharp.4th
asinacos.4th	ANS	fpconst.4th taylor.4th
chi2.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
csimpson.4th	ANS	ansfloat.4th
dblbin.4th		ansdbl.4th

Library	**Family**	**Depends on**
dbldiv.4th		mixed.4th
dbldot.4th		dblsharp.4th
dblsharp.4th		mixed.4th
ddigroot.4th		mixed.4th
dsqrt.4th		mixed.4th
ellipint.4th	ANS	flnflog.4th horner.4th
erf.4th	ANS	fexp.4th fequals.4th
erf1.4th	ANS	fexp.4th
erf1.4th	Zen	zenans.4th zenfexp.4th
expout.4th	ANS/Zen	fpdot.4th
falog.4th	ANS	fexp.4th flnflog.4th
fatan2.4th	ANS	asinacos.4th
fatanh.4th	ANS	flnflog.4th
fbessel.4th	ANS	ansfloat.4th
fbeta.4th	ANS	gammaln.4th fexp.4th
fbetacdf.4th	ANS	gammaln.4th fexp.4th
fbetapdf.4th	ANS	fbeta.4th falog.4th
fcartes.4th	ANS	asinacos.4th
fcbrt.4th	ANS	ansfloat.4th
fdeg2rad.4th	ANS/Zen	fpconst.4th
felip.4th	ANS	ansfloat.4th fpconst.4th
fequals.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
fenter.4th	ANS/Zen	*ansfpio.4th* *zenfpio.4th* *fpin.4th*
ferf.4th	ANS	fpconst.4th taylor.4th
fexp.4th	ANS	fpconst.4th fpow.4th
fexpt.4th	ANS	fpconst.4th taylor.4th
fexpint.4th	ANS	fexp.4th flnflog.4th
fexpm1.4th	ANS	fexp.4th
fgamma.4th	ANS	falog.4th
fhaversn.4th	ANS	fsinfcos.4th asinacos.4th fdeg2rad.4th
flnflog.4th	ANS	frexp.4th *ansfpio.4th* *fpin.4th*
flnflogb.4th	ANS	ansfloat.4th fpconst.4th

Library	Family	Depends on
		fpow10.4th
flnp1.4th	ANS	flnflog.4th
flocals.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
flogist.4th	ANS	fexp.4th
forwdiff.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
fp0.4th	Zen	zenfloat.4th zenfpio.4th
fp1.4th	Zen	zenfloat.4th zenfpio.4th zenans.4th
fp2.4th	Zen	zenfloat.4th zenans.4th fpin.4th fpout.4th
fp3.4th	ANS	ansfloat.4th ansfpio.4th
fp4.4th	ANS	ansfloat.4th fpin.4th fpout.4th
fpdot.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
fpconst.4th	ANS/Zen	*ansfpio.4th* *zenfpio.4th* *(zenans.4th)* *fpin.4th*
fpin.4th	ANS/Zen	tonumber.4th *ansfloat.4th* *zenans.4th*
fpolar.4th	ANS	fsinfcos.4th
fpout.4th	ANS/Zen	range.4th represnt.4th
fpow.4th	ANS	ansfloat.4th
fpow10.4th	ANS/Zen	*ansfloat.4th* *zenfloat.4th*
fq.4th	ANS	ferf.4th
fracout.4th	ANS/Zen	fpdot.4th
frexp.4th	ANS	ansfloat.4th
fsinfcos.4th	ANS	fpconst.4th
fsincost.4th	ANS	fpconst.4th taylor.4th
fsprintf.4th	ANS/Zen	fpout.4th
fsl-util.4th	ANS	*ansfpio.4th* *fpout.4th*
fstack.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
ftrunc.4th	ANS	ansfloat.4th
fvector.4th	ANS	ansfloat.4th
fzeta.4th	ANS	falog.4th
gamma.4th	ANS	fequals.4th fexp.4th

Library	**Family**	**Depends on**
		flnflog.4th fsinfcos.4th horner.4th
gammaln.4th	ANS	flnflog.4th fpow10.4th
gammapq.4th	ANS	gammaln.4th fexp.4th
gauslege.4th	ANS	fsinfcos.4th
gauss.4th	ANS	horner.4th
horner.4th	ANS	fsl-util.4th
mbinoml.4th		mixed.4th
mersenne.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
mixed.4th		ansdbl.4th
norm-pdf.4th	ANS	fexp.4th
pcylfun.4th	ANS	falog.4th fpow10.4th gamma.4th
permcomb.4th		mixed.4th
plecuyer.4th	ANS/Zen	*ansfloat.4th* *zenfloat.4th*
polys.4th	ANS	ansfloat.4th
prng.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
ran4.4th		mixed.4th dblbin.4th
randlcg.4th		mixed.4th
represnt.4th	ANS/Zen	dbldot.4th *ansfloat.4th* *zenans.4th*
runiform.4th	ANS/Zen	*ansfloat.4th* *zenfloat.4th*
simps38.4th	ANS	ansfloat.4th
simpson.4th	ANS	ansfloat.4th
sinhcosh.4th	ANS	fexpm1.4th
statist.4th	ANS	falog.4th
statist.4th	Zen	zenfsqrt.4th zenans.4th zenfalog.4th
taylor.4th	ANS	ansfloat.4th
todbl.4th		digit.4th mixed.4th
tonumber.4th		digit.4th
tt800.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
wichill.4th	ANS/Zen	*ansfloat.4th* *zenans.4th*
zenans.4th	Zen	zenround.4th zentodbl.4th
zenatan2.4th	Zen	zenfasin.4th
zenatanh.4th	Zen	zenfln.4th

Library	Family	Depends on
		zenfsqrt.4th
zencart.4th	Zen	zenfasin.4th zenfsqrt.4th
zendegrd.4th	Zen	zenfloat.4th
zenexpm1.4th	Zen	zenfexp.4th
zenfalog.4th	Zen	zenfexp.4th zenfln.4th
zenfasin.4th	Zen	zenfsqrt.4th zentaylr.4th
zenfcbrt.4th	Zen	zenfsqrt.4th
zenferf.4th	Zen	zentaylr.4th
zenfexp.4th	Zen	zentaylr.4th zentrunc.4th
zenfln.4th	Zen	zenfloat.4th
zenflnp1.4th	Zen	zenfln.4th
zenfloat.4th	Zen	mixed.4th
zenfloor.4th	Zen	zenfloat.4th
zenfmin.4th	Zen	zenfloat.4th
zenfpio.4th	Zen	(none)
zenfprox.4th	Zen	zenfloat.4th
zenfsin.4th	Zen	zenfloor.4th zentaylr.4th
zenfsinh.4th	Zen	zenexpm1.4th
zenfsqrt.4th	Zen	zenfloat.4th
zengamma.4th	Zen	zenfsqrt.4th zenfalog.4th
zengamm2.4th	Zen	zenfalog.4th
zenhaver.4th	Zen	zenfsin.4th zenfasin.4th zenfsqrt.4th zendegrd.4th
zenhornr.4th	Zen	zenfloat.4th
zenpolar.4th	Zen	zenfsin.4th zenround.4th
zenround.4th	Zen	zentrunc.4th
zentaylr.4th	Zen	zenfloat.4th
zentodbl.4th	Zen	zenfloat.4th
zentrunc.4th	Zen	zenfloor.4th
zxrandom.4th	ANS/Zen	*ansfloat.4th* *zenfloat.4th*

- The grey ovals depict library file dependencies that are not automatically resolved.
- Dotted lines indicate that only one of the pictured dependencies needs to be resolved.

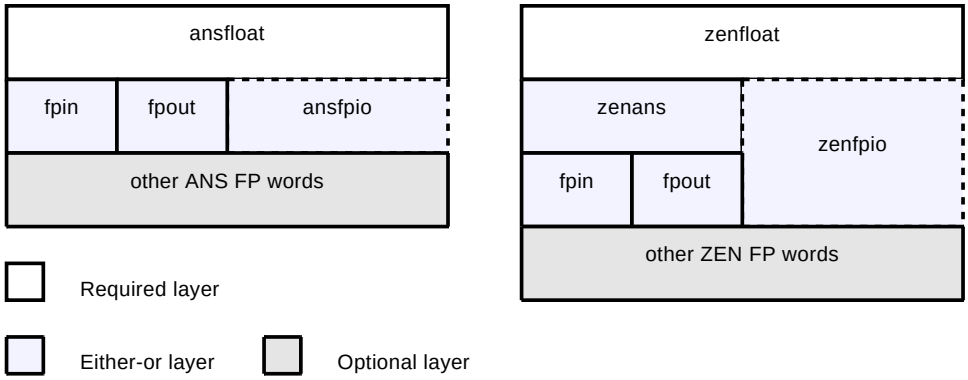

Figure 23.1: Basic FP architecture

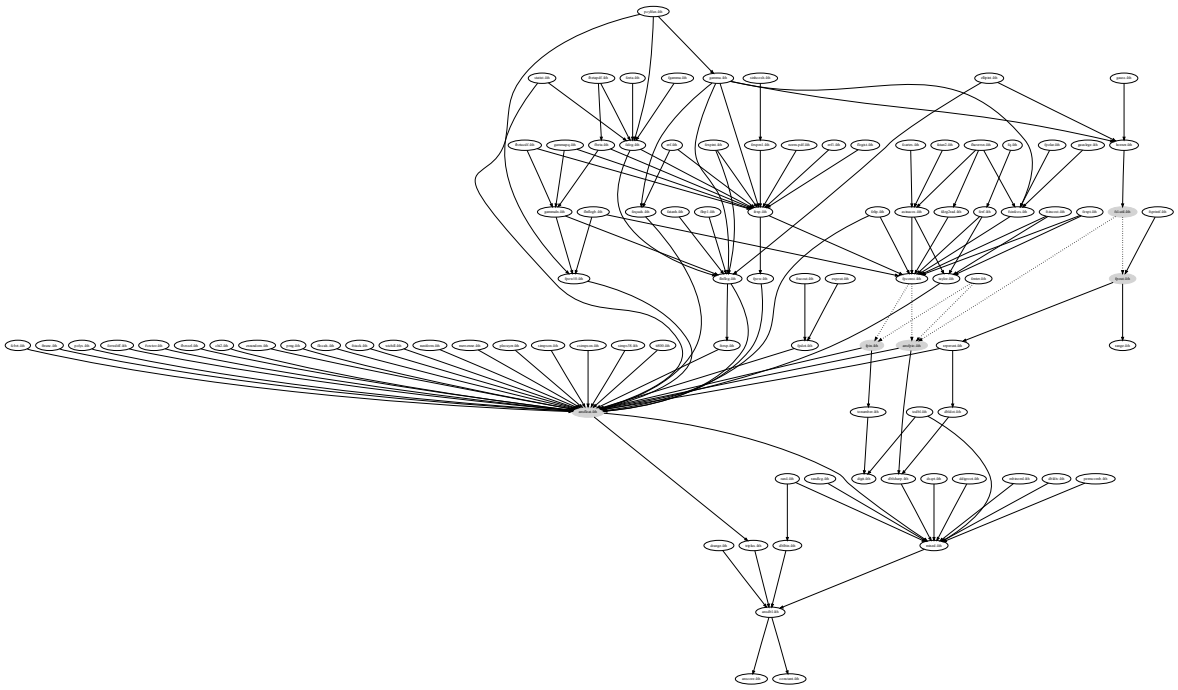

Figure 23.2: Double, mixed and floating point word dependencies (ANS)

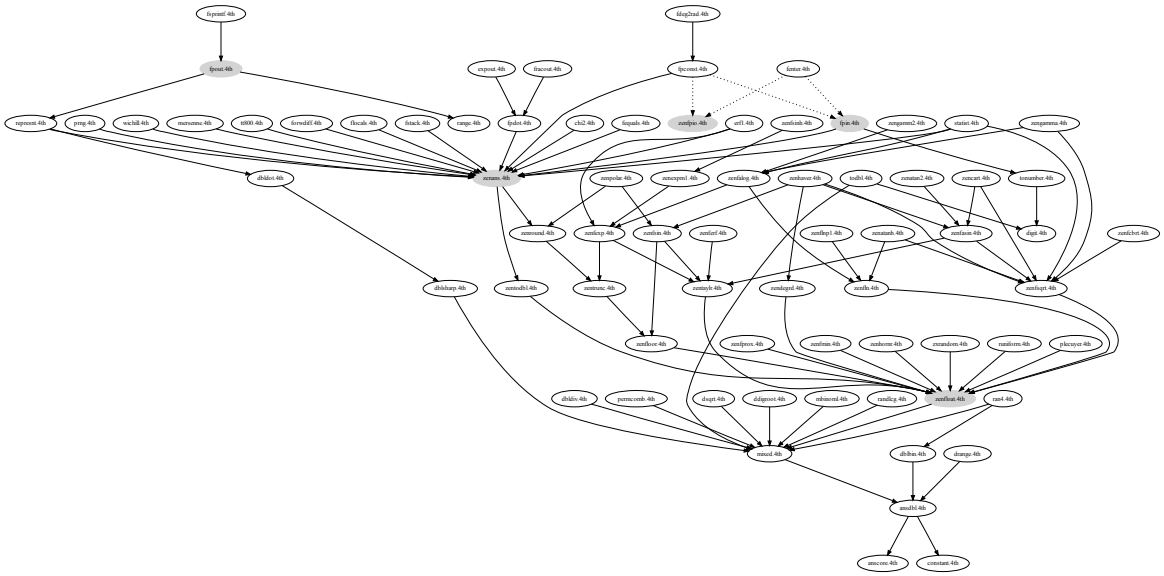

Figure 23.3: Double, mixed and floating point word dependencies (Zen)

Chapter 24

Change log

24.1 What's new in version 3.62.4

Words

- The words 'MAX-CHAR' and 'CHAR-BITS' were added.

Functionality

- The words 'MAX-CHAR' and 'CHAR-BITS' were added.
- The library files now support RTF generation, triple cell extended precision maths, numerical integration, canonical signed digit representation and a high speed integer square root.
- New implementations of the gamma function were added.
- A 3D library was added to the graphics libraries.

Bugfixes

- An obscure bug in '+LOOP' optimization was fixed.
- A name collision between `math.4th` and `fconstant.4th` was fixed.
- Some LaTeX related issues were fixed in `latex.4th` and `html.4th`.

Developer

- The words 'MAX-CHAR' and 'CHAR-BITS' were added.
- The library files now support RTF generation, triple cell extended precision maths, numerical integration, canonical signed digit representation and a high speed integer square root.
- New implementations of the gamma function were added.
- A 3D library was added to the graphics libraries.

Documentation

- Compatibility with Forth-20xx is now explicitly listed. The section ”ANS Forth statement” was modified accordingly.
- New sections on writing RTF files, 3D plotting and numerical integration were added to the manual.
- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.62.3 programs to V3.62.4 shouldn't be any problem. Most source files will compile correctly without modification and executables will run unmodified. There are two things to consider:

Library reorganization

Some LaTeX related issues were fixed in `latex.4th` and `html.4th`. Consequently, their behavior was slightly changed. The `UseTOC` pragma isn't required anymore, because you can now supply it as a flag to `%texTitle` and `%htmlTitle`. No other changes are required.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'MAX-CHAR' and 'CHAR-BITS'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[1]) ANS-Forth standard, except for porting purposes.

24.2 What's new in version 3.62.3

Words

- The word ',""' has been added.

Functionality

- Null-string constants can be compiled.
- ”>FLOAT” of the library `zenfpio.4th` will now accept most floating point formats.
- The library files now support regular expressions, bit reversal, CREATE and JSON output, database indexes, local variables (both cell and floating point), floating point user stacks, several new randomizers and sorting routines.
- 4tH executable can now call external editors .

[1] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Bugfixes

- A bug in `dsqrt.4th` was fixed.

Developer

- Null-string constants can be compiled.
- ">FLOAT" of the library `zenfpio.4th` will now accept most floating point formats.
- The library files now support regular expressions, bit reversal, CREATE and JSON output, database indexes, local variables (both cell and floating point), floating point user stacks, several new randomizers and sorting routines.
- 4tH executable can now call external editors.
- The library `wildcard.4th` was rewritten.
- The library `dbmsort.4th` was replaced by `dbmidx.4th`.
- The optimizer has been enhanced by optimizing 'NEGATE' more consistently.

Documentation

- All documentation now reflects the functionality of the current version.
- The section on database sorting was replaced by database indexing.
- Sections on local variables, tabular formats and regular expressions were added.

Hints

Porting your V3.62.2 programs to V3.62.3 shouldn't be any problem. Most source files will compile correctly without modification and executables will run unmodified. There are four things to consider:

Android version

if you're using Android v3.x or higher, 4tH is now installed in ATEs `app_HOME` folder. If you're upgrading you may have to remove older 4tH executables in the `shared_prefs` folder and repeat the post-installation procedure.

Library reorganization

If you used `wildcard.4th` in your programs you have to make a few changes to make them work properly. The easiest way is to define this word at the beginning of your program:

```
: wild-match 2swap match-wild ;
```

After that your program should compile and run correctly.

Databases

If you used `dbmsort.4th` in your programs you have to make a few changes to make them work properly. This library is completely replaced by `dbmidx.4th`, which supports indexing. Table 24.1 will show you which changes to make.

Although we regret the significant changes you may have to make, please note that there are significant advantages to indexes. Most importantly, you can update, save and load indexes, so you may never need to have to scan and sort tables again. Furthermore, you can use several indexes at the same time. See section 13.32 for more details.

SORT (old)	INDEX (new)	COMMENT
8192 constant /sort-index include lib/heapsort.4th include lib/dbmsort.4th	include lib/shelsort.4th include lib/dbmidx.4th	You can choose any algorithm from table 12.4. The size will be determined later on.
' heapsort is db.sorting		This is not required anymore.
db.scan db.key Firstname db.sort	idx.create 8192 {char} db.key Firstname abort" Can't make index" to Pers.First	A sorted index is created with this single statement. Note the size is defined here. It returns an address to the index and a true flag on error. Creating an index is done only *once*.
0 db.sorted @ db.goto	Pers.First 0 idx.goto	You can't directly retrieve rowid's anymore. On the other hand, you rarely need to.
db.norows 0 do i db.sorted @ db.goto db.buffer -> Firstname count type cr loop	Pers.First dup idx.first begin dup idx.error 0= while db.buffer -> Firstname count type cr dup idx.next repeat idx.clear	This a typical retrieval loop, moving through the entire index from begin to end. It will now be exited by detecting the end of the index. In this example, the error is cleared.
db.buffer -> db.sortkey count type cr	Pers.First idx.key count type cr	Both print the current value of the key field in the active record.
	Pers.First idx.close abort" Can't close index"	An index has to be explicitly closed.

Table 24.1: Database index conversion

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved word is ',"'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[2]) ANS-Forth standard, except for porting purposes.

24.3 What's new in version 3.62.2

Words

- The words '[*]', '[+]', '[/]' and '[NEGATE]' have been removed.
- The words 'C"', 'C|', '[:' and ';]' have been added.

Functionality

- Support for quotations was added.
- The library files now support coroutines, HTML generation, Adler32, MD5, SHA-1 and SHA-256 checksum routines and several new randomizers.
- ANS Forth conversion and library specific support in the preprocessor were moved to external preprocessor libraries.
- The object orientation method definition syntax has been cleaned up.
- The closures syntax has been cleaned up.
- Most preprocessor libraries don't use any registers anymore.
- Symbol definition and verbose options were added to the preprocessor.
- The preprocessor now supports `_#_`, `$#$`, `@dup`, `@over`, `@nip`, `@cr` and the new `@cell` function.
- The preprocessor has been vastly enhanced, so you can now execute, save or generate a C source without calling `4th` separately.

Bugfixes

- A bug in `parsing.4th` was fixed.
- A malformed call to `calloc()` was fixed in `comp_4th()`.
- Some murky code in `comp_4th.c` was cleaned up.

[2] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Developer

- Support for quotations was added.
- The library files now support coroutines, HTML generation, Adler32, MD5, SHA-1 and SHA-256 checksum routines and several new randomizers.
- ANS Forth conversion and library specific support in the preprocessor were moved to external preprocessor libraries.
- The object orientation method definition syntax has been cleaned up.
- The closures syntax has been cleaned up.
- Most preprocessor libraries don't use any registers anymore.
- Symbol definition and verbose options were added to the preprocessor.
- The preprocessor now supports `_#_`, `$#$`, `@dup`, `@over`, `@nip`, `@cr` and the new `@cell` function.
- The preprocessor has been vastly enhanced by adding a C wrapper (`pp4th.c`), so you can now execute, save or generate a C source without calling `4th` separately.

Documentation

- All documentation now reflects the functionality of the current version.
- A section on coroutines was added.
- A section on generating HTML files was added.

Hints

Porting your V3.62.1 programs to V3.62.2 shouldn't be any problem. Most source files will compile correctly without modification and executables will run unmodified. There are eight things to consider:

Preprocessor

The previous incarnations of the preprocessor were just a straight compile of the 4tH source. Now a C wrapper has been added that vastly expands the capabilities of the preprocessor. It also means that the invocation of the preprocessor has changed from:

```
pp4th infile.4pp outfile.4th
```

To:

```
pp4th -o outfile.4th infile.4pp
```

If you use any scripts or Makefiles that use the preprocessor, you have to change them accordingly.

Obsolete compiler directives

The compiler directives '[*]', '[/]', '[+]' and '[NEGATE]' were obsolete since V3.62.0. They have been removed. Replace them with '*', '/', '+' and 'NEGATE'.

3DUP

It moved from `comus.4th` to `3dup3rot.4th`. Change your include files accordingly.

LaTeX

Some changes were made to `latex.4th` in order to support compatibility with `html.4th`. If this is important to you, we advise you to use `%print` when adding plain text. Furthermore, the optional package "url" was exchanged for "hyperref", so `URL:` is not supported anymore. You can use `%link` instead, but you will have to rewrite your code. If you used the following words in your code, you either have to define them yourself or use an alternative word with a *different* stack diagram:

Old	*Definition*	*Alternative*
`means:`	`: means: ." \item[" type ." ] " cr ;`	`%describe`
`item:`	`: item: ." \item " ;`	`%item`

No other changes are required.

FOOS

The FOOS syntax has been significantly improved and is now much more consistent. This table will show you the differences between the old and the new syntax:

Old FOOS syntax	*New FOOS syntax*
`method: name`	`virtual: name`
`static: name`	`method: name`
`:method .. ; defines name`	`:virtual name .. ;method`
`:static name .. ;`	`:method name .. ;method`
`:static name1 .. ; overrides name2`	`:default name2 .. ;method`
`new name1 name2`	`instance name1 name2`

Invoking methods has been slightly changed as well:

Method	*Invocation*	*Shorthand*
static	`object -> method`	(none)
virtual	`object ->` **`virtual`** `method`	`object` **`=>`** `method`
default	`object ->` **`default`** `method`	`object` **`<-`** `method`

Closures

The closures syntax has been significantly improved and is now much more consistent. This table will show you the differences between the old and the new syntax:

Old closures syntax	*New closures syntax*
`fields`	`vars`
`field:` *name*	`var:` *name*
`end-fields`	`end-vars`
`new` *name1 name2*	`set` *name1 name2*

Note the previous version of closures hardly *had* any syntax, so conversion is not straightforward. Still, since the design *itself* hasn't changed the original code should still run. Most of the new syntax has been derived from FOOS, so you should feel at home pretty quickly. The run time definition is enclosed between "{:" and ";}" pairs. All closures are statically defined now.

Preprocessor

Several built-in macros have been moved to a library. If your program uses any of the keywords listed in "Provides", you have to include the library listed in "Include".

INCLUDE	PROVIDES
`double.4pp`	**d% 2variable 2constant**
`float.4pp`	**f% fvariable ffield: fconstant**
`ansforth.4pp`	**action-of synonym** **begin-structure end-structure** **case of endof endcase**
`sbslquot.4pp`	**s\"**

No other changes are required.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'C"', 'Cl', '[:' and ';]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[3]) ANS-Forth standard, except for porting purposes.

24.4 What's new in version 3.62.1

Words

- The words ':TOKEN' and 'EQUATES' have been added.

[3] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- Named execution tokens can be defined.
- Static variable and array pointers can be defined.
- The library files now support virtual memory, simple garbage collection, RfC 4180 compliant parsing, SEDOL numbers, Chi square calculations, runtime macros and several new sorting routines.
- Several new date- and time parsing functions were added to the library.
- Several beta- and gamma related functions were added to the floating point libraries.
- The preprocessor now supports division and modulo with the new `@divrm` function.

Bugfixes

- Several obscure bugs in the peephole optimizer were fixed.
- Address checking was added to the 'TO' and 'VALUE' tokens.
- A documented bug in `SkipSource()` was fixed.
- A bug in `DoCloseParen()` was fixed.
- A bug in `bsearch.4th` was fixed.
- Some bugs in `parsing.4th` were fixed.
- `parse-csv` in `parsing.4th` is now mostly RfC 4180 compliant.

Developer

- The library files now support virtual memory, simple garbage collection, RfC 4180 compliant parsing, SEDOL numbers, Chi square calculations, runtime macros and several new sorting routines.
- Several new date- and time parsing functions were added to the library.
- Several beta- and gamma related functions were added to the floating point libraries.
- The new library file `multiblk.4th` is a multi buffer implementation of the BLOCK wordset. It is a drop-in replacement for `ansblock.4th`.
- A new peephole optimizer was added to the compiler, optimizing '1 +LOOP' expressions.
- Named execution tokens can be defined.
- Static variable and array pointers can be defined.
- The file `null.4th` was integrated in `constant.4th`.
- The preprocessor now supports division and modulo with the new `@divrm` function.

Documentation

- All documentation now reflects the functionality of the current version.
- The sections ”Random numbers” and ”Sorting” were moved.
- Sections on static variable pointers, execution tokens, garbage collection and virtual memory were added.

Hints

Porting your V3.62.0 programs to V3.62.1 shouldn't be any problem. All source files will compile correctly without modification and executables will run unmodified. There are three things to consider:

Obsolete compiler directives

Last warning: the compiler directives '[*]', '[/]', '[+]' and '[NEGATE]' have become obsolete since V3.62.0. It is recommended to change your sources accordingly by replacing them with '*', '/', '+' and 'NEGATE'. Note these language elements *will be removed* in a future version.

NULL

The constant 'NULL' is now part of `constant.4th`. The file `null.4th` has been removed. Please change your sources accordingly.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are ':TOKEN' and 'EQUATES'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[4]) ANS-Forth standard, except for porting purposes.

This release was time-sponsored by Ordina N.V.
http://www.ordina.com

24.5 What's new in version 3.62.0

Words

- The words 'SMOVE', 'LATEST', '*CONSTANT' and '/CONSTANT' have been added.

[4] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- The library files now support logfiles, automated date parsing, LZ77 file compression and CSV file creation.
- The compiler directives '[*]', '[/]', '[+]' and '[NEGATE]' are no longer required. They can be replaced by '*', '/', '+' and 'NEGATE'.
- The latest defined word can be compiled anonymously.
- Object orientation now supports static methods.

Bugfixes

- A "division by zero" error in '[/]' was fixed.
- A small error in the Linux `Makefile` was fixed.
- A warning in `4th.c` with some GCC compilers was fixed.
- Some patches were applied to facilitate the Raspberry Pi port.
- Some bugs in `chars.4th` and `base64.4th` were fixed.
- 'BUFFER' in `ansblock.4th` is now ANS-Forth compliant.
- 'CREATE-FILE' in `ansfile.4th` is now ANS-Forth compliant.

Developer

- The library files now support logfiles, automated date parsing, LZ77 file compression and CSV file creation.
- The latest defined word can be compiled anonymously.
- Object orientation now supports static methods.
- The library file `ansblock.4th` is now largely ANS-Forth compliant. The Sourceforge "*Block reserved extension words*" were added.
- New peephole optimizers were added to the compiler, so the compiler directives '[*]', '[/]', '[+]' and '[NEGATE]' are no longer required.
- `exec_4th()` can be optimized for GCC by using the compiler switch `-DUSEGCCGOTO`.
- The word 'SMOVE' was added, making the library `cellmove.4th` superfluous.

Documentation

- All documentation now reflects the functionality of the current version.
- A section on GCC specific optimizations was added.
- A "change log" was added.

Hints

Porting your V3.61.5 programs to V3.62.0 shouldn't be any problem. All source files will compile correctly without modification. There are four things to consider:

SMOVE

Since the word 'SMOVE' is now a native word, the library `cellmove.4th` has become superfluous. However, you will notice that most source files will continue to compile fine. Still, it is recommended to remove any references to this former library.

Obsolete compiler directives

The compiler directives '[*]', '[/]', '[+]' and '[NEGATE]' have become obsolete. It is recommended to change your sources accordingly by replacing them with '*', '/', '+' and 'NEGATE'. Note obsolete language elements *may be removed* in future versions.

Block files

The former block file interface has been replaced with the Sourceforge version. If you used "USE-BLOCK" or "C/SCR" in any of your programs, you will have to replace them with "OPEN-BLOCKFILE" and "B/BUF", respectively. "BUFFER" and "BLOCK" were aliases. If you have used "BUFFER" anywhere in your program, you should check whether it continues to run properly.

In general, you should check *any* program that uses the ANS BLOCK wordset, since the new version has more solid error trapping, which may require additional exception handling.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'SMOVE', 'LATEST', '*CONSTANT' and '/CONSTANT'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[5]) ANS-Forth standard, except for porting purposes.

24.6 What's new in version 3.61.5

Words

- The word 'CFIELD:' has been added.

[5] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- The library files now support FCEIL, KEY, KEY?, SUBSTITUTE, REPLACES, UNESCAPE, counted strings, ”Q” function, Catalan numbers, Riemann zeta function, a spell distance function, several new graphics functions and `frexp()`, `ldexp()` like words.
- A new implementation of the ANS MEMORY wordset was added, allowing for concurrent string- and cell heaps.
- New preprocessor words were added like `FFIELD:`, `@eval`, `@mul`, `@minus` and `@sign`.
- New preprocessor libraries add support for object orientated programming and closures.

Bugfixes

- The speed and accuracy of the FSIN, FCOS, FTAN, FEXP, FLN and FLOG words was improved.
- Some minor fixes were applied to VECTOR and EXECUTE.
- A bug in ”PARSE” was fixed.
- Stack usage of the preprocessor was significantly reduced.

Developer

- The library files now support FCEIL, KEY, KEY?, SUBSTITUTE, REPLACES, UNESCAPE, counted strings, ”Q” function, Catalan numbers, Riemann zeta function, a spell distance function, several new graphics functions and `frexp()`, `ldexp()` like words.
- A new implementation of the ANS MEMORY wordset was added, allowing for concurrent string- and cell heaps.
- The library files `back.4th`, `scanskip.4th`, `startend.4th`, `tokenize.4th` and `trim.4th` were rewritten.
- The library files `fsinfcos.4th`, `fexp.4th` and `flnflog.4th` were rewritten.
- The library file `dproduct.4th` was replaced by `fvector.4th`.
- New preprocessor words were added like `FFIELD:`, `@eval`, `@mul`, `@minus` and `@sign`. Stack usage was significantly reduced.
- New preprocessor libraries add support for object orientated programming and closures.
- New peephole optimizers were added to the compiler.
- The message ”Name too long” was added.

Documentation

- All documentation now reflects the functionality of the current version.
- A chapter on preprocessor libraries was added.
- A section on floating point functions was added.

Hints

Porting your V3.61.4 programs to V3.61.5 shouldn't be any problem. All executables will run correctly without recompilation. Most source files will need no modifications. There are two things to consider:

Library reorganization

- The library file `dproduct.4th` was replaced by an ANS floating point version, called `fvector.4th`. If you have programs that use this library, you either have to integrate the old library or rewrite your program for ANS floating point.
- The words /SPLIT and -SPLIT were taken out of `back.4th` and placed in library file of their own. If you have programs that use these words, you have to add `split.4th` to the include files.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved word is 'CFIELD:'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[6]) ANS-Forth standard, except for porting purposes.

24.7 What's new in version 3.61.4

Words

- The words '[ELSE]' and '[IGNORE]' have been added.

Functionality

- Alternative code after a failed condition can be evaluated during compilation without repeating the inverted expression.
- Words without compilation semantics can be defined.
- The library files now support stem-leaf plots, finite difference, compass boxing, HTML and XML character entities, Hamming (7,4) codes and Base64 conversion.

[6] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

- The library files now support FEXPM1 and FLNP1 for both ANS and Zen floating point.
- The preprocessor was enhanced. Macros can be undefined with the 'SCRAP:' keyword. Simple flow control was added.
- PAD size is increased to 768 characters to make pictured numeric output 64 bit compliant.

Bugfixes

- Several minor bugs in selected library members have been fixed.
- The speed and accuracy of the FSINH, FCOSH and FTANH words was improved.
- The accuracy of the FASIN, FACOS and FATAN words was significantly improved.
- The speed of several other ANS floating point words was significantly improved.
- The preprocessor does now check for undefined or duplicate macros.
- A bug in 'OFFSET' was fixed.

Developer

- Alternative code after a failed condition can be evaluated during compilation without repeating the inverted expression.
- Words without compilation semantics can be defined.
- The library files now support stem-leaf plots, finite difference, compass boxing, HTML and XML character entities, Hamming (7,4) codes and Base64 conversion.
- The library files now support FEXPM1 and FLNP1 for both ANS and Zen floating point.
- The library file `stsstack.4th` was rewritten.
- The preprocessor was enhanced. Macros can be undefined with the 'SCRAP:' keyword. Simple flow control was added.
- PAD size is increased to 768 characters to make pictured numeric output 64 bit compliant.

Documentation

- All documentation now reflects the functionality of the current version.
- A section on XML and HTML conversion was added.

Hints

Porting your V3.61.3 programs to V3.61.4 shouldn't be any problem. Most of them will only need recompilation. Source files will need no or just minor modifications. There are two things to consider:

Library reorganization

The `stsstack.4th` library used to be initialized like this:

```
1024 constant /mystack
/mystack string mystack
mystack string-stack
```

That has now become:

```
1024 constant /mystack
/mystack string mystack
mystack /mystack string-stack
```

No other changes are required.

Preprocessor

The following preprocessor variables are not supported anymore: `?#?` and `===`. If you used them anywhere in your preprocessor files you'll have to replace them or use the following macro definitions (assuming macro variable 4 is not used):

```
:macro ?#? @match <4< #4# ;
:macro === ?#? [not] ;
```

Note the definition of `===` is slightly different, so it *cannot* be used as a "drop in" replacement. Given the following example:

```
:macro create
  @1@ @2@ @3@ >#> <#<
  >3> === chars [if] #2# string #1# [then]
  >3> === cells [if] #2# array  #1# [then]
;
```

Then all you have to do is change it to this:

```
:macro ?#? @match <4< #4# ;
:macro === ?#? [not] ;

:macro create
  @1@ @2@ @3@ >#> <#<
  >3> >>> chars === [if] #2# string #1# [then]
  >3> >>> cells === [if] #2# array  #1# [then]
;
```

No other changes are required. Note the new preprocessor has some new and powerful features so it may be worth to rewrite your macros.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[ELSE]' and '[IGNORE]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[7]) ANS-Forth standard, except for porting purposes.

[7] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

24.8 What's new in version 3.61.3

Words

- The words '[PRAGMA]', '/FIELD' and '[MAX]' have been added.
- Renamed ':THIS' to ':REDO'.
- The word 'DOES>' has been changed.

Functionality

- Unions can be defined.
- The library files now support Chebyshev, Bessel, Hermite, Laguerre and Legendre polynomials, RAM disks, menus, bitfields, bit arrays, several new sorting algorithms, Morse signs and 3VL.
- The preprocessor was enhanced, supporting S\", token concatenation, string comparison, variable assignment and smart double number/FP literals.
- 'DOES>' can also be used to change the execution semantics of basic datatypes.

Bugfixes

- A bug concerning shared libraries in the Linux `Makefile` was fixed.
- Overflow and underflow errors in number conversion were fixed.
- A bug in the "IF" statement of *4tsh* was fixed.
- The accuracy of the "FERF" words was significantly improved.

Developer

- Unions can be defined.
- The library files now support Chebyshev, Bessel, Hermite, Laguerre and Legendre polynomials, RAM disks, menus, bitfields, bit arrays, several new sorting algorithms, Morse signs and 3VL.
- The preprocessor was enhanced, supporting S\", token concatenation, string comparison, variable assignment and smart double number/FP literals.
- The support function `str2cell()` was added.
- 'DOES>' can also be used to change the execution semantics of basic datatypes.
- Instead of defining constants to control conditional compilation in include files, pragmas are now supported.

Documentation

- All documentation now reflects the functionality of the current version.
- Sections on menus, unions, bit arrays and bitfields were added.
- The sections on structures, sorting, random numbers, changing behavior of data and adding your own libraries were expanded.

Hints

Porting your V3.61.2 programs to V3.61.3 shouldn't be any problem. Most executables will run correctly without recompilation. Source files will need no or just minor modifications. There are three things to consider:

:THIS..DOES>

The easiest way to convert this expression to the new syntax is to remove 'DOES>' and rename ':THIS' to ':REDO', e.g.:

```
create myconstant max-n ,
:this myconstant does> @c ;
```

Becomes:

```
create myconstant max-n ,
:redo myconstant @c ;
```

If you fail to remove 'DOES>' a compiler error is issued. No other changes are required.

Preprocessor

The following preprocessor variables are not supported anymore: `%1%`, `%2%`, `%3%`, `%4%`, `@#@` and `#;#`. If you used them anywhere in your preprocessor files you'll have to replace them or use the following macro definitions:

```
:macro %1% ` S" ` >1> |#| >>> " |#| ;
:macro %2% ` S" ` >2> |#| >>> " |#| ;
:macro %3% ` S" ` >3> |#| >>> " |#| ;
:macro %4% ` S" ` >4> |#| >>> " |#| ;
:macro @#@ >#> <#< ;
:macro #;# ` ;` ;
```

Just include them at the beginning of your file and you should be fine. Note that these definitions are also expanded *outside* macros, so don't use preprocessor variables as names inside your program.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are ':REDO', '[PRAGMA]', '/FIELD' and '[MAX]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[8]) ANS-Forth standard, except for porting purposes.

This release is dedicated to Dennis Ritchie, without whom none of this would be here.

24.9 What's new in version 3.61.2

Words

- The words 'DONE', 'TAG', 'REWIND' and '>STRING' have been added.

Functionality

- Specific code can be executed after a failed 'WHILE' just before exiting the loop.
- Binary strings can be tagged individually.
- The library files now support UTF-8 to GBK/2 conversion (and vice versa), infix formula translation, string pattern matching, finite state machines, statistical functions and enhanced command line parsing.
- The library files now support the full range of ANS Forth floating point words for ZEN float.
- The preprocessor now supports string parsing with custom delimiters, macros within macros and a string stack.

Bugfixes

- An annoying bug in the ANS float library file was fixed.
- Several minor bugs were fixed in assorted library files.

Developer

- Specific code can be executed after a failed 'WHILE' just before exiting the loop.
- Binary strings can be tagged individually.
- The library files now support UTF-8 to GBK/2 conversion (and vice versa), infix formula translation, string pattern matching, finite state machines, statistical functions and enhanced command line parsing.

[8] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

- The library files now support the full range of ANS Forth floating point words for ZEN float.
- The library files `dblmul.4th`, `mixed.4th`, `anscore.4th`, `files.4th`, `2rotover.4th` and `pickroll.4th` were reorganized.
- Several double and floating point words were added.
- The message "Wrong type" was added.
- The preprocessor now supports string parsing with custom delimiters, macros within macros and a string stack.
- Several Makefiles were changed on request by packagers.

Documentation

- All documentation now reflects the functionality of the current version.
- Sections on Chinese character conversion, string pattern matching, statistical functions, finite state machines, infix formula translation, command line parsing and Makefiles were added.
- The sections on binary strings and complex control structures were expanded.

Hints

Porting your V3.61.1 programs to V3.61.2 shouldn't be any problem. Most executables will run correctly without recompilation. Source files will need no or just minor modifications. There are three things to consider:

Wrong type

Some 'THROW' signals have changed, although all standard ANS signals have remained the same. If you explictly evaluate system signals in your program, you may want to recompile it before use. Evaluating system signals by *number* instead of by *symbol* (`throw.4th`) is considered bad practice.

Library reorganization

Some words have been moved to another library file, so you might have to change your includes according to the following table:

Word	v3.61.1	v3.61.2
2OVER	`2rotover.4th`	`anscore.4th`
2ROT	`2rotover.4th`	`anscore.4th`
?DUP	`anscore.4th`	`pickroll.4th`
D*	`dblmul.4th`	`mixed.4th`
UD*	`dblmul.4th`	`mixed.4th`
REWIND	`files.4th`	- *builtin* -

Although "PICK" and "ROLL" are fully compatible with the previous versions, the current, recursive implementation is more compact and also includes "?DUP". In some circumstances certain programs may cease to function properly. In that case you can either:

- Rewrite the program, so ”PICK” or ”ROLL” are no longer required;
- Use the previous version of `pickroll.4th`.

The use of ”PICK”, ”ROLL” and ”?DUP” in newly created programs is discouraged.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are ’DONE’, ’TAG’, ’REWIND’ and ’>STRING’.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[9]) ANS-Forth standard, except for porting purposes.

24.10 What’s new in version 3.61.1

Words

- The word ’SEEK’ has been changed.
- The words ’2NIP’ and ’[FORCE]’ have been added.

Functionality

- The library files now support a tiny database management system.
- The library files now support double number multiplication and division.
- The library files now support LaTeX file generation.
- The library files now support the XCHAR wordset.
- ’SEEK’ can now be used to reposition the file to the end of the file.

Bugfixes

- Corrupted HX files are now handled properly.
- 4tsh now terminates gracefully when it cannot load the monitor.
- ZEN float is now 64bit compatible.
- If a file cannot be closed and throws an exception, it is labeled as such to prevent a crash.

[9] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Developer

- The library files now support a tiny database management system.
- The library files now support double number multiplication and division.
- The library files now support LaTeX file generation.
- The library files now support the XCHAR wordset.
- 'SEEK' can now be used to reposition the file to the end of the file.
- A peephole optimizer was added.
- Automated the include file generation.

Documentation

- All documentation now reflects the functionality of the current version.
- Sections on debugging, databases, database sorting and LaTeX file generation were added.
- The section on optimization was partly rewritten.

Hints

Porting your V3.61.0 programs to V3.61.1 shouldn't be any problem. All executables will run without recompilation. Source files will need no or just minor modifications. There is one thing to consider:

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '2NIP' and '[FORCE]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[10]) ANS-Forth standard, except for porting purposes.

24.11 What's new in version 3.61.0

Words

- The words 'DELETE-FILE' and 'ENVIRON@' have been added.

[10] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- The library files now support writing OpenOffice, KOffice and Microsoft XML spreadsheets.
- The library files now support several different sorting algorithms.
- The library files now support Portable Bitmap graphics and Turtle graphics.
- The library files now support the creation of floating point interpreters.
- The library files now support GTK-server.
- The library files now support ANS Forth compatible floating point I/O for ZEN float.
- Files can be deleted and environment variables can be queried.
- The maximum symbol length is increased to 23 characters.

Bugfixes

- Several bugs in the preprocessor PP4tH were fixed.
- Several minor bugs were fixed in assorted library files.
- A TurboC warning concerning certain `cgen_4th()` generated files was fixed.

Developer

- The library files now support writing OpenOffice, KOffice and Microsoft XML spreadsheets.
- The library files now support several different sorting algorithms.
- The library files now support Portable Bitmap graphics and Turtle graphics.
- The library files now support the creation of floating point interpreters.
- The library files now support GTK-server.
- The library files now support ANS Forth compatible floating point I/O for ZEN float.
- The maximum symbol length is increased to 23 characters.
- `toPAD()` in `exec_4th()` will truncate any string that would overwrite the HOLD area.

Documentation

- All documentation now reflects the functionality of the current version.
- A list of contributors has been added.
- Sections on writing spreadsheet files, using GTK-server, file deletion, graphics and environment variables were added to the Primer.
- A section on packing several words in a single token was added to the Development Guide.

Hints

Porting your V3.60.1 programs to V3.61.0 shouldn't be any problem. Most of them will only need recompilation. There are two things to consider:

Library reorganization

The `xlswrite.4th` library was renamed to `msxls2-w.4th`. The "XLSopen" word now returns a flag. This means if you have written any programs that use that library, you will have to rewrite them slightly:

```
include lib/xlswrite.4th
s" mysheet.xls" XLSopen
23 XLS.
```

Has to be changed into:

```
include lib/msxls2-w.4th
s" mysheet.xls" XLSopen abort" Can't open file"
23 XLS.
```

If you have used the ZEN floating point wordset in your programs you will have to add an extra include file. Just add `zenfpio.4th` to the list of include files, right after `zenfloat.4th`, e.g.:

```
include lib/zenfloat.4th
include lib/zenfsin.4th
```

Becomes:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
include lib/zenfsin.4th
```

The reason for this change is that a set of fully ANS compatible floating point I/O words has been made available. The file `zentoflt.4th` has been replaced by `fpin.4th`. If you use ANS compatible floating point I/O, you *cannot* use standard ZEN floating point I/O (`zenfpio.4th`) as well, e.g.:

```
include lib/zenfloat.4th
include lib/zentoflt.4th
```

Has to be changed it into:

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
```

Note the use of `zenans.4th` is *mandatory* with the use of ANS compatible floating point I/O. If you still experience problems, you might want to consult chapter 23.2 to resolve more complex dependency issues.

The word "SIZE" from `files.4th` behaves slightly different now. Instead of -1 it now returns '(ERROR)' on error, e.g.:

```
include lib/files.4th
s" doesntex.ist" size dup
-1 = abort" Cannot determine file size"
```

Becomes:

```
include lib/files.4th
s" doesntex.ist" size error?
abort" Cannot determine file size"
```

Which is not only more 4tH-like, but also much easier to understand.

If you have used '>DOUBLE', which is defined in `todbl.4th`, in one of your programs, e.g.:

```
include lib/todbl.4th
0. s" -1234567890" >double
```

Then change it into:

```
include lib/todbl.4th
0. s" -1234567890" >dnumber
```

This is a tricky one, because your program will compile fine, but won't run properly. The new '>DOUBLE' has a behavior similar to '>FLOAT'. 'S>DOUBLE' will continue to function properly, as will the double number version of '>NUMBER'.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'DELETE-FILE' and 'ENVIRON@'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[11]) ANS-Forth standard, except for porting purposes.

24.12 What's new in version 3.60.1

Words

- The word 'AKA' has been changed.
- The word 'FIELD:' has been added.

[11] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- The library files now support new TEA and ARC4 encryptions.
- The library files now support cell memory allocations.
- The library files now support several new string words, including Soundex conversion.
- 'AKA' now also allows aliasing of most built-in 4tH words.
- The preprocessor now supports variables, block files, BEGIN-STRUCTURE, END-STRUCTURE and SYNONYM.

Bugfixes

- The trigonometric words of the integer math library will now work beyond the 0 - $\frac{\Pi}{2}$ range.
- `cmove()` was made more solid.

Developer

- BSD pipes API is now supported.
- The RANGE() macro was added to `exec_4th()`.
- The performance of 'REFILL', 'FILL', 'TYPE' and 'ACCEPT' was enhanced.
- The library files now support new TEA and ARC4 encryptions.
- The library files now support cell memory allocations.
- The library files now support several new string words, including Soundex conversion.
- 'AKA' now also allows aliasing of most built-in 4tH words.
- The preprocessor now supports variables, block files, BEGIN-STRUCTURE, END-STRUCTURE and SYNONYM.
- Compilation options of `ansmem.4th`, `compare.4th` and `interprt.4th` were added or changed.

Documentation

- All documentation now reflects the functionality of the current version.
- A list of all 4tH library files and their descriptions was added.
- A section on card games was added to the tutorial.

Hints

Porting your V3.60.0 programs to V3.60.1 shouldn't be any problem. All executables will run without recompilation. Source files will need no or just minor modifications. There are three things to consider:

COMPARE

Before V3.60.1 case sensitivity was toggled by changing the boolean value of the `ignorecase` constant. This is now done by simply *defining* the `casesensitive` constant.

INTERPRET

Before V3.60.1 the interpretation of numbers was toggled by changing the boolean value of the `ignorenumbers` constant. This is now done by simply *defining* the `ignorenumbers` constant.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved word is 'FIELD:'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[12]) ANS-Forth standard, except for porting purposes.

24.13 What's new in version 3.60.0

Words

- The words 'RANDOM' and 'MAX-RAND' have been removed. The word 'RSHIFT' has been changed.

Functionality

- The library files now support several random number generators.
- The library files now support simple hashtables and associative arrays.
- The library files now support `strpbrk()` and `strchr()` like words.
- 'RSHIFT' now performs a logical shift.

Bugfixes

- None.

Developer

- Version numbering was changed.
- The library files now support several random number generators.
- The library files now support simple hashtables and associative arrays.

[12] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

- The library files now support `strpbrk()` and `strchr()` like words.
- 'RSHIFT' now performs a logical shift.
- The compiler now supports tail call optimization.
- `strtoc()` is now called `strtocell()`.

Documentation

- All documentation now reflects the functionality of the current version.
- A section on optimization and a chapter on uBasic were added.

Hints

Porting your V3.5d release 3 programs to V3.60.0 shouldn't be any problem. Most of them will only need recompilation. There are two things to consider:

Random numbers

Every source that uses the 'RANDOM' or 'MAX-RAND' word has to be changed slightly, e.g.:

```
6 random * max-rand 1 [+] / 1+ . cr
```

Now becomes:

```
include lib/random.4th
randomize
6 random * max-rand 1 [+] / 1+ . cr
```

Note it is best to include "RANDOMIZE" into any initialization word you already defined. Programs using "CHOOSE"[13] are not affected.

Right shift

Programs *depending* on an arithmetic shift when using 'RSHIFT' should be changed accordingly. Note that although most C compilers perform an arithmetic shift on signed numbers, this is *not* standardized behavior[14].

However, if your 4tH compiler's '2/' performs an arithmetic shift, 'RSHIFT' may be replaced by a series of '2/' calls, e.g.:

```
: arshift 0 ?do 2/ loop ;
```

But that doesn't mean it will work on another platform or even another 4tH compiler on the same platform compiled with a different C-compiler.

[13] Which is defined in `choose.4th`.
[14] See: http://en.wikipedia.org/wiki/Arithmetic_shift

24.14 What's new in version 3.5d, release 3

Words

- The words '[/]' and '[SIGN]' have been added.

Functionality

- The preprocessor was expanded and now takes the DIR4TH environment variable into account.
- The library files now support ANS Forth compatible versions of all floating point input and output words.

Bugfixes

- None.

Developer

- The library files now support ANS Forth compatible versions of all floating point input and output words.
- The library file `getenv.4th` was rewritten.
- The library file `row.4th` was changed.

Documentation

- All documentation now reflects the functionality of the current version.
- A chapter on library dependencies was added.

Hints

Porting your V3.5d release 2 programs to V3.5d release 3 isn't a problem. All executables will run without recompilation. Source files will need no or just minor modifications. There are two things to consider:

Library reorganization

If you have used the ANS floating point wordset in your programs you will have to add an extra include file. Just add `ansfpio.4th` to the list of include files, right after `ansfloat.4th`, e.g.:

```
include lib/ansfloat.4th
include lib/fsinfcos.4th
```

Becomes:

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fsinfcos.4th
```

The reason for this change is that a set of fully ANS compatible floating point I/O words has been added. If you still experience problems, you might want to consult chapter 23.2 to resolve more complex dependency issues.

The `row.4th` library member was awkward to use and syntactically not very clean. This has been fixed. The following snippet:

```
:this keyword does>
  ['] skey= is key= 2 row
  if cell+ @c execute else drop type then
;
```

Can now be expressed like this:

```
:this keyword does>
  2 string-key row
  if cell+ @c execute else drop type then
;
```

Numeric keys can be searched by using the `num-key` word. Both `num-key` and `string-key` are execution tokens, so you can still replace them with your own versions. The words `key=`, `nkey=` and `skey=` are now private and cannot be called directly anymore.

The `getenv.4th` library member has been enhanced. Before it was up to you to decide which OS your program was intended to support, e.g.:

```
256 env-buffer
/env-buffer string env-buffer
s" $PATH" env-buffer /env-buffer getenv
```

The `getenv.4th` library member now determines automatically at runtime which OS is used. You only have to remove any prefixes or postfixes, e.g.:

```
256 env-buffer
/env-buffer string env-buffer
s" PATH" env-buffer /env-buffer getenv
```

No other changes are required.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[/]' and '[SIGN]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[15]) ANS-Forth standard, except for porting purposes.

[15] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

24.15 What's new in version 3.5d, release 2

Words

- None.

Functionality

- A preprocessor was added to the toolchain.
- The library version is shown during *4th* startup.

Bugfixes

- A few bugs in the ANS floating point library were fixed.

Developer

- Another floating point library called ZEN float was added.
- REPRESENT is now supported.
- A preprocessor was added to the toolchain.
- The `patch4th.4th` script allows more complex modifications.
- 4tH can be compiled as a shared library under Linux.
- *4tsh* was almost completely rewritten.

Documentation

- All documentation now reflects the functionality of the current version.
- A chapter on the preprocessor was added.

Hints

Porting your V3.5d programs to release 2 isn't a problem. All executables will run without recompilation. All sources will compile properly without modification.

This release is dedicated to my father.

24.16 What's new in version 3.5d

Words

- The words 'BUFFER:' and 'ERROR?' have been added.
- The word 'OPEN' has been changed.
- The word 'AS' has been dropped.

Functionality

- The editor can now save text files.

Bugfixes

- A few bugs in the floating point library were fixed.
- The circular buffer library was rewritten.

Developer

- The library files now support a subset of the Forth Scientific Library.
- Some recent Forth200x submissions were added to the library.
- The license was changed from LGPL v2+ to LGPL v3.

Documentation

- All documentation now reflects the functionality of the current version.
- The Development Guide was expanded.

Hints

Porting your V3.5c release 3 programs to V3.5d shouldn't be any problem. Most of them will only need recompilation. There are three things to consider:

Changed words

Previously, 'OPEN' returned zero on error. This was not according to the 4tH standard, which uses '(ERROR)' to signal a problem. Although there were sound reasons at the time to deviate from the standard (see section 24.25), this deficiency has been corrected in the current version.

In order to facilitate the transition, the word 'ERROR?' has been added. It converts the 4tH convention to the ANS-Forth convention by adding an additional error flag. Consequently, 'ERROR?' can also be used for other 4tH words that already return '(ERROR)', but its use is *not* required and existing programs will continue to work correctly.

This is a typcal file opening construction in version 3.5c:

```
s" Hello.txt" input open
dup 0= abort" Cannot open file"
```

You can easily convert it to version 3.5d with 'ERROR?':

```
s" Hello.txt" input open
error? abort" Cannot open file"
```

Note that the stackdiagrams of both constructions are exactly the same.

Dropped words

The word 'AS' has been dropped since it seems very unlikely that this syntactic sugar will ever be used again. Just replace 'AS' with 'TO'.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'ERROR?' and 'BUFFER:'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[16]) ANS-Forth standard, except for porting purposes.

24.17 What's new in version 3.5c, release 3

Words

- None.

Functionality

- None.

Bugfixes

- A few bugs in the mixed numbers library were fixed.

Developer

- The library files now support most of the FLOAT and FLOAT EXT wordsets.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5c release 2 programs to release 3 isn't a problem. All executables will run without recompilation. All sources will compile properly without modification.

24.18 What's new in version 3.5c, release 2

Words

- None.

[16] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- A default 4tH directory can be defined by setting an environment variable.
- Support for creating custom 4tH implementations.
- *4tsh* is scriptable now.

Bugfixes

- None.

Developer

- The library files concerning ANS Core Extensions, table searching and interpretation have been rewritten or replaced.
- A superfluous `#define` was removed from `4th.h`.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5c programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly.

Library reorganization

Some words have been moved to another library file, so you might have to change your includes according to the following table:

Word	v3.5c	v3.5c, release 2
WITHIN	`anscext.4th`	`ranges.4th`
BETWEEN	`comus.4th`	`ranges.4th`
SAVE-INPUT	`anscext.4th`	`evaluate.4th`
RESTORE-INPUT	`anscext.4th`	`evaluate.4th`

Interpreter

The inclusion of `interprt.4th` has to be done at the *very beginning* of the program, like all other include files. "NotFound" now always uses the same stack diagram: it leaves the address/count string on the stack that could not be interpreted. "NotFound" is now a deferred word with default behaviour, so defining it is optional. Either remove the definition or change it from, e.g.:

```
: NotFound type ."  is not defined" cr ;
```

To:

```
:noname type ."  is not defined" cr ; is NotFound
```

The ”dictionary” table used to be mandatory. Change it from e.g.:

```
create dictionary
```

To:

```
create wordlist
```

After you've *completely* defined the table add this line:

```
wordlist to dictionary
```

Your program should compile and run correctly now.

Table search

Both `find.4th` and `lookup.4th` have been superseded by `row.4th`. Since ”ROW” works slightly different, you might have to do some rewriting. Please consult the primer if you're unsure how. If you're not willing to do that, there are two options:

1. Use the `find.4th` and `lookup.4th` from a previous version of 4th;
2. Use the following definitions:

```
: find
  ['] skey= is key= >r row
  if nip nip r> cells + @c true
  else r> drop drop false
  then
;

: lookup
  ['] nkey= is key= >r row
  if nip r> cells + @c true
  else r> drop drop false
  then
;
```

Reserved words

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[17]) ANS-Forth standard, except for porting purposes.

24.19 What's new in version 3.5c

Words

- The words 'C,' and 'OFFSET' have been added.

[17] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- Binary string constants can be defined.

Bugfixes

- None.

Developer

- MakeSymbol() has been added to comp_4th().

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5b, release 2 programs to V3.5c shouldn't be any problem. Most of them will only need recompilation. There is one thing to consider:

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'C,' and 'OFFSET'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[18]) ANS-Forth standard, except for porting purposes.

24.20 What's new in version 3.5b, release 2

Words

- Renamed 'FIELD' to '+FIELD'. The words '[NEGATE]', 'CHOP' and '/STRING' have been added.

Functionality

- None.

Bugfixes

- The word '->' allocated slightly more memory than needed. This has been fixed.

[18] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Developer

- The function hgen_4th() has been removed from the API.
- The library files have been updated and expanded.

Documentation

- All documentation now reflects the functionality of the current version.
- A section on the 4tH shell (*4tsh*) has been added.

Hints

Porting your V3.5b programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Renamed words

If you used 'FIELD' in your programs, you'll have to replace it by '+FIELD'. No other changes are necessary.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[NEGATE]', 'CHOP' and '/STRING'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[19]) ANS-Forth standard, except for porting purposes.

24.21 What's new in version 3.5b

Words

- The words '.|' and 'SYNC' have been added.

Functionality

- Output buffers can be flushed.

Bugfixes

- None.

[19] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Developer

- The CODE() and NEXT macros have been added to allow easy modification of exec_4th().
- The library files now support most of the CORE and DOUBLE wordsets.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5a release 2 programs to V3.5b shouldn't be any problem. Most of them will only need recompilation. There is one thing to consider:

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '.l' and 'SYNC'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[20]) ANS-Forth standard, except for porting purposes.

24.22 What's new in version 3.5a, release 2

Words

- Renamed 'SLEEP' to 'PAUSE'. The word 'FILES' has been added.

Functionality

- None.

Bugfixes

- A bad mode string disabled pipes in the Unix version. This has been fixed.

Developer

- None.

Documentation

- All documentation now reflects the functionality of the current version.

[20] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Hints

Porting your V3.5a programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Renamed words

If you used 'SLEEP' in your programs, you'll have to replace it by 'PAUSE'. No other changes are necessary.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved word is 'FILES'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[21]) ANS-Forth standard, except for porting purposes.

24.23 What's new in version 3.5a

Words

- The words 'WORD', '"', 'TOKEN', 'COPY', 'TEXT' and 'WAIT' have been discarded.
- The words 'NUMBER', 'ARGS', 'IS', 'REPEAT', 'AGAIN' and 'UNTIL' have been changed.
- Renamed '@" to '@C', 'SKIP' to 'OMIT' and 'RESULT' to 'OUT'.
- The words '@GOTO', '+CONSTANT', 'SOURCE-ID', 'CIN, 'COUT', 'PARSE-WORD', 'IMMEDIATE', 'NOT', 'INCLUDE', '[UNDEFINED]', '4TH#', 'SLEEP', ',"', '2DUP', '2DROP', '2SWAP', '2>R', '2R>', 'S|', ',|', '+PLACE', '-ROT', 'BOUNDS', '2R@', 'R'@', 'UNLOOP', 'SOURCE', 'SOURCE!', 'DEFER@', 'DEFER!', '>BODY', 'SCONSTANT', ':THIS', 'DOES>', 'STRUCT', 'END-STRUCT', '->', 'FIELD', 'ENUM', 'SEEK', 'TELL', 'AKA', 'ALIAS' and 'HIDE' have been added.

Functionality

- The execution of a 4tH program can be suspended.
- A suspended 4tH program can be saved and reloaded.
- A 4tH program can be embedded in a MS batch file.
- User defined words can be made private.
- User defined words can be aliased.

[21] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

- User defined terminal input buffers are supported.
- Complete, ANS-Forth compatible redesign of all string handling words.
- Multiple WHILEs are supported with REPEAT, AGAIN and UNTIL.
- Support for structures and enumerations has been added.
- Files can now be opened in read/write mode.
- File pointers can be interrogated and repositioned.
- Limited DOES> support has been added.
- More ANS-Forth, COMUS and TOOLKIT words have been added.

Bugfixes

- Several small bugs in the editor were fixed.
- A small bug in 'FILL' was fixed.
- A bug in `hgen_4th.c` that caused SEGFAULT was fixed.
- A security vulnerability in `4th.c` was fixed.

Developer

- Several changes in `exec_4th()`, `comp_4th()`, `save_4th()` and `load_4th()` to support suspension.
- The function `inst_4th()` has been renamed to `fetch_4th()`.
- The function `store_4th()` has been added.
- The Hcode structure has been expanded with the members CellSeg, UnitSeg and Offset.
- PAD has been converted to a circular buffer for temporary strings.
- Most of the string handling and all file functions in `exec_4th()` have been rewritten.
- The entire virtual machine was rewritten and its performance significantly improved.
- All internal 4tH variables are now located in a hidden area of the Integer Segment.
- The performance of 'MOVE' has been significantly improved.
- The library files have been updated and significantly expanded.

Documentation

- All documentation now reflects the functionality of the current version.
- There is now one single manual.

Hints

Porting your V3.3d release 2 programs to V3.5a may require some effort. In previous versions, string support was quite a mess (IMHO), requiring awkward words like 'COPY'. Some words returned or expected an address, others an address/count pair. With version 3.5a string support was completely redesigned. Consequently, source files using strings or arrays of string constants will have to be partially rewritten in order to make them compile and run properly. There are several things to consider:

Strings

There has been a conversion to the format recommended by the ANS-Forth standard. *All* strings are now represented by an address/count pair, with the exception of string variables and string addresses returned by '@C'. For this purpose, 'WORD' has been replaced by 'PARSE-WORD'. 'NUMBER' and 'ARGS' now return an address/count pair. Parsed strings are no longer copied to PAD, but remain in TIB and are *not* zero-terminated. However, since parsed strings are now represented by an address/count pair this should not be a problem.

Most programs we examined used constructions like this:

```
[char] ; word count type
s" 567" drop number
1 args count my_variable place
```

Those can easily be converted to:

```
[char] ; parse-word type
s" 567" number
1 args my_variable place
```

As a rule of the thumb, we advise you to use 'COUNT' *only* on string variables and string addresses returned by '@C'. You might find after a while, that these are the *only* situations where 'COUNT' is actually required. In all other situations, you use the count on the stack. Special operators like '2DUP', '2DROP' and '2SWAP' have been added to make manipulation of address/count pairs easier.

Please note that 'OPEN' already required an address/count pair, but simply discarded the count. In version 3.5a the count is *required*. If you didn't program properly, this might cause errors now. Well designed programs will continue to function properly.

We advise against the use of 'MOVE' or 'CMOVE' for moving strings. Most of these constructions will continue to work, but some may fail. In any case, they are not portable. Use 'PLACE' and '+PLACE' wherever you can.

PAD

PAD has been converted into a circular string buffer. Because some routines directly interface with their C counterparts, temporary zero-terminated strings are stored in PAD. When the buffer overflows it wraps around, overwriting whatever is there. Some previously correctly running programs may corrupt the PAD this way. If this happens, you can solve this by storing the overwritten string into a string variable. The reason for all this is that this now works:

```
s" This is not overwritten" s" By this string" compare
```

Number representations are not clobbered unless you use extremely long number formats.

Arrays of string constants

Consider this construction:

```
16 string weekday

create weekdays
  " Monday" ,
  " Tuesday" ,
  " Wednesday" ,
  " Thursday" ,
  " Friday" ,
  " Saturday" ,
  " Sunday" ,

weekdays 4 th @' weekday copy count type cr
```

'@" returns an address in the String Segment. 'COPY' is the only word in pre-3.5a versions that can access the String Segment. It copies the string from the String Segment to an address in the Character Segment and returns that address . In version 3.5a and up, '@" has been replaced by '@C'. '@C' is a lot smarter. It copies the zero-terminated string from the String Segment to PAD and returns that address. A 'COUNT' is still needed, but since all temporary strings in PAD are zero-terminated, this can safely be done:

```
create weekdays
  ," Monday"
  ," Tuesday"
  ," Wednesday"
  ," Thursday"
  ," Friday"
  ," Saturday"
  ," Sunday"

weekdays 4 th @c count type cr
```

Note that the string variable is no longer needed and the resulting code is much cleaner! String constants are now declared by a simple ',"'. '"' has been discarded. '@C' also works for integer constants and behaves like '@".

Deferred words

Deferred words are now fully COMUS compatible. You have to change your programs only slightly:

```
defer my-vector

: do-nothing ;
' do-nothing is my-vector
my-vector execute
```

Just remove the 'EXECUTE':

```
defer my-vector

: do-nothing ;
' do-nothing is my-vector
my-vector
```

Please note that 'IS' is no longer an alias for 'TO'. If you have used illegal constructions like that, you'll have to correct them.

Library files

Note the library files have been revised and expanded. Some words have been renamed or placed into another file. Note that the `toolbelt.4th` and `comus.4th` library files are primarily intented for porting purposes. The `easy.4th` library file is intended to port 4tH programs to other Forth compilers. Note that this only works for ANS-Forth compliant programs.

Dropped words

'WAIT' has been dropped and replaced by 'MS'. You'll have to 'INCLUDE' the library file `ansfacil.4th` in order to use it. Note that this implementation is *very crude* and may vary between 0 and +1999 milliseconds[22].

'TOKEN' has been replaced by 'PARSE', which returns an address/count pair. 'WORD' has been replaced by 'PARSE-WORD', which returns an address/count pair. 'COPY' has been incorporated into '@C'.

'TEXT' has been dropped. If you treat a file as a text file, it will be handled as a text file. Just remove 'TEXT':

```
s" textfile.txt" input text + open
```

So now it reads:

```
s" textfile.txt" input open
```

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '@C', 'OMIT', 'OUT', '@GOTO', '+CONSTANT', 'SOURCE-ID', 'CIN, 'COUT', 'PARSE-WORD', 'IMMEDIATE', 'NOT', 'INCLUDE', '[UNDEFINED]', '4TH#', 'SLEEP', ',"', '2DUP', '2DROP', '2SWAP', '2>R', '2R>', 'S|', ',|', '+PLACE', '-ROT', 'BOUNDS', '2R@', 'R'@', 'UNLOOP', 'SOURCE', 'SOURCE!', 'DEFER@', 'DEFER!', '>BODY', 'SCONSTANT', ':THIS', 'DOES>', 'STRUCT', 'END-STRUCT', '->', 'FIELD', 'ENUM', 'SEEK', 'TELL', 'AKA', 'ALIAS' and 'HIDE'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[23]) ANS-Forth standard, except for porting purposes.

24.24 What's new in version 3.3d, release 2

Words

- The word 'C"' has been discarded. The words '[NEEDS' and '[DEFINED]' have been added.

[22] The "Forth Programmers Handbook" states that 'MS' should be *at least* the duration plus *twice* the resolution of the system (which is one second in 4tH).

[23] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Functionality

- Source files can be included at compile time.
- The existence of words in the dictionary can be checked at compile time.
- More COMUS words have been added.
- The 4tH program allows you to enter parameters in the menu.
- The Linux module 'binfmt_misc' is supported.

Bugfixes

- None.

Developer

- Function `open_4th()` has been rewritten.
- The parser in `comp_4th()` has been changed significantly and is now much more transparant.
- There is an extra option in the menu of `4th.c`

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.3d programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Dropped words

If you used 'C"' in your programs, you'll have to replace it by "". No other changes are necessary.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[NEEDS' and '[DEFINED]'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list or ANS-Forth standard, except for porting purposes.

24.25 What's new in version 3.3d

Words

- The words 'FILE' and 'TTY' have been discarded. The words 'FILE', 'AS', 'USE', 'DEFER', 'IS', 'STDIN' and 'STDOUT' have been added. The words 'INPUT', 'OUTPUT', 'OPEN' and 'CLOSE' have been changed.

Functionality

- Multiple files can be opened concurrently.
- More COMUS words have been added.

Bugfixes

- A segment violation was caused in 4th.c when an invalid sequence of commands was issued. This has been fixed.
- Better errorhandling when a pipe cannot be opened.

Developer

- The file support in function `exec_4th()` has been rewritten.
- Added `DoInitValue()`.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.3c programs to V3.3d shouldn't be any problem. Most of them will only require recompilation, except when files are manipulated. There are three things to consider:

Using files

The new 4tH file handling module adds the concepts of streams and channels. You have two channels, an input channel and an output channel. In (standard) 4tH you have eight streams (you can increase this when you compile 4tH), two are already taken by the system (stdin and stdout). At startup the stdin stream is connected to the input channel and the stdout stream is connected to the output channel.

You can open additional streams by using the 'OPEN' word:

```
OPEN (a n fmod -- handle)
```

E.g.

```
s" ls" input pipe + open
```

This is not a significant deviation from V3.3c in which 'OPEN' returned only a flag. You can still interpret the handle as a flag since 'OPEN' returns zero when it failed.

To use the handle you only have to connect it to the appropriate channel. In V3.3c this was done by using:

```
input file
```

In V3.3d, you use the word 'USE'. 'USE' takes a handle and connects the stream to the appropriate channel.

```
file ls
s" ls" input pipe + open dup as ls
0= abort" Cannot open pipe"

ls use
```

In V3.3c you had to close a file by closing the channel, while the stream was still connected:

```
s" ls" input pipe + open
0= abort" Cannot open pipe"

input file
input close
```

In V3.3d you have to close the stream:

```
file ls

s" ls" input pipe + open dup as ls
0= abort" Cannot open pipe"

ls use
ls close
```

The default stream is reconnected to the channel, *even* if another stream was currently connected to that channel. We give you an example how 4tH now handles files in respect to the previous version:

VERSION 3.3C

```
s" hello.txt" output text + open
0= abort" Cannot open file"
output file
." Hello world" cr
output close
```

VERSION 3.3D

```
file hello
s" hello.txt" output text + open dup as hello
0= abort" Cannot open file"
hello use
." Hello world" cr
hello close
```

I hope you can appreciate the extended possibilities of 4tH and the way we tried to minimize breaking existing code.

Using 'INPUT' and 'OUTPUT'

Two new constants have been added to 4tH: 'STDIN' and 'STDOUT'. Before you could use 'INPUT' and 'OUTPUT' as follows:

```
input  file
output tty
```

Using 'INPUT' and 'OUTPUT' this way is *depreciated* and should be replaced by:

```
stdin  use
stdout use
```

Please use 'INPUT' and 'OUTPUT' *only* as flags for OPEN.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'AS', 'USE', 'DEFER', 'IS', 'STDIN' and 'STDOUT'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

24.26 What's new in version 3.3c

Words

- The word '+UNDER' has been discarded. The words 'PIPE', 'PLACE', 'TOKEN', 'SKIP', 'PARSE', '/CELL', '/CHAR', 'ABORT"', '[ABORT]' and '[=]' have been added.

Functionality

- A complete mini-IDE has been added.
- Parsing has been enhanced significantly.
- The Unix version now supports pipes.
- More CORE words implemented.
- Some environmental dependancies can be checked at compiletime.

Bugfixes

- Reentry of several 4tH functions was seriously flawed, most notoriously in 'comp_4th()'. This has been fixed.

Developer

- Several new functions have been added, most significantly in the area of C source generation.
- The loading of sourcefiles is now done by open_4th(); fload() can still be used, but is no longer supported.
- The function save_4th() has been optimized. HX files are up to 50% smaller compared to those created by previous versions.
- The function dump_4th() has two extra arguments, allowing partial decompilation.
- The file support in function exec_4th() has been rewritten and now supports popen() and pclose().
- The demonstration program 4th.c has been completely rewritten.

Documentation

- All documentation now reflects the functionality of the current version.
- A document describing a sample session in 4tH interactive mode has been added.
- Several documents have been merged.

Hints

Porting your V3.3a programs to V3.3c shouldn't be any problem. Most of them will only require recompilation. There are two things to consider:

Programs using '+UNDER'

Which is no longer supported. If you have such programs, just add this definition at the top:

```
: +UNDER ROT + SWAP ;
```

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'PIPE', 'PLACE', 'TOKEN', 'PARSE', 'SKIP', '/CELL', '/CHAR', 'ABORT"', '[ABORT]', and '[=]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

24.27 What's new in version 3.3a

Words

- The words 'APPEND', 'TEXT', 'S"', '[*]', '[+]', '[NOT]' and '#!' have been added.
- The word 'OPEN' has been changed.

Functionality

- An output file can now be opened in "append" and "text" mode.
- A 4tH program can now be run from the shell.
- More CORE words implemented.
- Compiletime calculation is possible now.

Bugfixes

- When reallocation of the segments during compilation fails, resources are freed.
- When memory allocation of the header during the loading of an HX file fails, the file is closed.

Developer

- Dropped the EasyC syntax.
- Added the proper 'int main()' declarations.
- Modern prototypes, local include files and no stricmp() function are now the default behaviour.
- Dropped stricmp() from the distribution and added MatchName() to comp_4th().
- Added CompileString().

Documentation

- All documentation now reflects the functionality of the current version.
- The Developers Guide has been enhanced.

Hints

Porting your V3.2e programs to V3.3a shouldn't be any problem. Most of them will only require recompilation. There are two things to consider.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '#!', 'S"', 'APPEND', '[*]', '[+]', '[NOT]' and 'TEXT'.

Changed words

The word 'OPEN' now takes an extra value from the stack. If you used an construction like this:

```
64 string filename
" myfile.dat" filename copy
input open
```

Change it to this:

```
s" myfile.dat" input open
```

If you used a construction like this:

```
refill drop
bl word
input open
```

Change it to this:

```
 refill drop
 bl word count
 input open
```

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

24.28 What's new in version 3.2e

Words

- The words 'I", 'R', 'QUERY', 'ENDIF', 'END', 'MINUS', 'NOT', 'ASCII', '2+' and '2-' have been discarded.
- The words ':NONAME', '?DO', 'BLANK', 'ERASE', 'CMOVE>', 'NIP', 'TUCK', '+UNDER' 'REFILL', 'D>S', 'RSHIFT', 'CATCH' and 'MAX-RAND' have been added.
- Renamed '-TRAIL' to '-TRAILING', 'STACK' to 'STACK-CELLS', '#PAD' to '/PAD', '#TIB' to '/TIB' and 'LIMIT' to 'MAX-N'.

Functionality

- The Character Segment is now unsigned, so no more negative characters.
- Vectored execution has been enhanced.
- Better implementation of 'RECURSE'.
- Better ANS-Forth compatibility by adding some commonly used words.
- Compatibility with Forth-79 has been dropped.

Bugfixes

- `load_4th()` closes the file when memory allocation failed.
- An error in `GetImmediate()`, `GetConstant()` and `GetWord()` has been fixed.
- `DoRecurse()` can now detect the use of 'RECURSE' outside a colon definition.

Developer

- Complete redesign of the parser. The whole parser now consists of the functions: `ParseText()`, `ParseString()` and `ParseDirective()`. Inline macros are supported.
- `MoveString()` does not require any arguments anymore.
- A textmode has been added to `accept()`.
- Removed and added several tokens.
- The names of all internal words are now pointers instead of sized arrays, which means name can have any length now.

Documentation

- All documentation now reflects the functionality of the current version.
- The Developers Guide has been enhanced.
- The Porting Guide has been enhanced.

Hints

Porting your V3.1d programs to V3.2e shouldn't be any problem. There are three things to consider.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are ':NONAME', '-TRAILING', '?DO', 'BLANK', 'ERASE', 'CMOVE>', 'NIP', 'TUCK', '+UNDER', 'REFILL', 'D>S', 'RSHIFT', 'CATCH', '/PAD', '/TIB', 'STACK-CELLS', 'MAX-N' and 'MAX-RAND'. Most likely you have used these names for compatibility purposes, e.g.:

```
: rshift negate shift ;
```

In that case you can simply remove these definitions.

Dropped words

The Forth-79 words 'R', '2-', '2+', 'QUERY', 'ENDIF', 'END', 'MINUS', 'NOT', 'ASCII' and 'I" are no longer supported. '-TRAIL', '#PAD', '#TIB', 'LIMIT' and 'STACK' have been renamed. If you have programs that use these words then either modify them or add the following definitions:

```
: 2+ 2 + ;
: 2- 2 - ;
: i' r> r> r> dup >r rot rot >r >r ;
: r r> r> swap over >r >r ;
: query input tty refill drop ;
: minus negate ;
: not invert ;
: -trail -trailing ;
: #pad /pad ;
: #tib /tib ;
: limit max-n ;
: stack stack-cells ;
```

Unfortunately, you still have to replace the following words by their ANS-Forth equivalent, since there is no colon definition available for them:

CHANGE:	TO:
ASCII	CHAR, [CHAR]
END	AGAIN
ENDIF	THEN

Table 24.2: Forth-79 to ANS conversion

Unsigned characters

If a character with an ASCII value greater than 127 was fetched from the Character Segment, it was converted to a negative value. 'C@' will now return a positive value. This means that you can remove patches like these:

```
: c@' c@ dup 0< if 256 + then ;
```

On the other hand, if you have programs that rely on this negative value (e.g. by storing "-1" in a character), then you have to modify them.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

24.29 What's new in version 3.1d

Words

- The words 'AT' and 'ALLOT' have been discarded
- The words 'ARRAY', 'TABLE', '.(', 'ABORT', 'S>D', '"', 'RECURSE', '[IF]', '[THEN]', 'ARGS' and 'ARGN' have been added.

Functionality

- Better ANS-Forth compatibility (thank you, Wil Baden)
- Commandline arguments are now supported
- Nested assertions are now supported
- Conditional compilation is now supported.

Bugfixes

- ASCII bug has been fixed
- Several bugs in `ParseText()` and `ParseStrings()` have been fixed.

Developer

- Added `SkipSource()`,`DecodeSymbol()` and `DecodeLiteral()`
- Added two more arguments to `exec_4th()`
- Added two more tokens to `cmds_4th.h`
- Moved `<limits.h>` to `4th.h`
- Added an extra compilation option "LOCAL_H" for those who cannot access `/usr/include`.

Documentation

- All documentation now reflects the functionality of the current version
- A 'Porting Guide' has been added
- A 'What's New' bulletin has been added
- The 'Developers Guide' has been enhanced
- The 'Primer' has been enhanced.

Hints

Porting your V3.1c programs to V3.1d shouldn't be any problem. There are five things to consider.

AT

'AT' has been discarded. Simply replace all occurences of 'AT' by 'STRING'. If you used the 'CHARS' keyword, you can leave right there since it doesn't have any effect, except when you are porting your program to Forth.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'ARRAY', 'TABLE', '.(', 'ABORT', 'S>D', '"', 'RECURSE', '[IF]', '[THEN]', 'ARGS' and 'ARGN'.

Using 'VALUE' with 'ALLOT'

If you ALLOTted any space to a VALUE, you should rewrite your code. Note that this is bad practice anyway. Example:

```
10 value room 10 cells allot \ allotting space to a VALUE
20 to room                   \ changing ROOM
5 ' room first + 4 th !      \ accessing allotted space
```

Change this to:

```
11 array room                \ define an ARRAY
10 room 0 th !               \ init 1st element of ROOM
20 room 0 th !               \ change 1st element
 5 room 4 th !               \ accessing allotted space
```

Using 'VARIABLE' with 'ALLOT'

This should be common practice to define cell arrays. However, as Wil Baden pointed out, this is not a common practice in ANS-Forth. Therefore, the word 'ARRAY' has been added, which can easily be implemented in both Forth-79 and ANS-Forth. All the programs using the old syntax have to be modified, though. Example:

```
variable room 15 cells allot
```

Change this to:

```
16 array room
```

Special care must be taken of arrays that are sized using a constant, e.g. when the same constant is used to check the range. Example:

```
15 constant size
variable room size cells allot

: room?                      \ is it a valid variable?
  dup                        ( n n)
  size not and               ( n f)
  if                         \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
```

Change this to:

```
16 constant size
size array room

: room?                      \ is it a valid variable?
  dup                        ( n n)
  size 1- not and            ( n f)
  if                         \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
```

ANS-Forth compatibility

Sometimes it proved to be impossible to port a program to ANS-Forth since some constructions could not be implemented. There is no such thing as a 'state' in 4tH, which means that compilation- and interpretation semantics are completely the same, e.g.

```
c" This is a string" value addr
." String address has been stored in ADDR" cr
```

This is perfectly valid in 4tH, but cannot be ported in any way to ANS-Forth. With the new version, you can write:

```
" This is a string" value addr
.( String address has been stored in ADDR) cr
```

So if you have a 4tH program which you wanted to port to ANS-Forth, but couldn't, study the Porting Guide and try again. Note that no change is required if you do not intend to port your program to Forth. Apart from the modifications already mentioned you do not have to change a single line.

Our apologies for any inconvenience caused. It is certainly not our policy to change the syntax with every single version, but we found the arguments in favor of this change so strong that we didn't see any other way.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

Part IV

Development guide

Chapter 25

Compiling the source

25.1 Introduction

4tH is primarily designed as a powerful and easy to use toolkit for developers. You can use it "as is" and you've got a very flexible calculation-engine. You can tailor it to a specific application and push it even further. Or you just might want to use one of the safest and easiest Forth-alike environments ever created. These are all valid reasons and I will try to address them all.

First, I will show you how to compile the example applications. They are far from useless. In fact, you will have created a complete programming environment. Second, I will show you how to create the 4tH library and how to use it. Third, I will explain to you how the compiler works and how you can make simple additions to 4tH.

If you find any errors in this document please contact me by sending email to "The.Beez.-speaks@gmail.com". You would be helping a lot of future 4tH users.

25.2 Recommended and preferred compilers

4tH is written in ANSI-C and K&R C and should be portable to any platform that supports such a compiler. All memory-models are supported, although the usual restrictions apply. Of course, it is impossible to test every single compiler on the market, but there are a number of compilers that are known to work. Preferred compilers are Open Source and are available for a number of platforms. When properly installed, the entire compilation can be performed in two simple steps:

```
make
```

Then login as root and enter:

```
make install
```

That's all. If the installation fails in the last stage, try:

```
make mostlyinstall
```

COMPILER	URL	PLATFORM	LABEL
GCC 2.95 msvcrt	http://downloads.activestate.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip	Win32	Recommended
Cygwin	http://www.cygwin.com/	Win32	Preferred
MinGW	http://sourceforge.net/projects/mingw/	Win32	Preferred
Pelles C	http://www.smorgasbordet.com/pellesc/	Win32	Recommended
LCC	http://www.cs.virginia.edu/~lcc-win32/	Win32	Recommended
TCC	http://fabrice.bellard.free.fr/tcc/	Win32	Recommended
Turbo C V2.01	http://bdn.borland.com/article/images/20841/tc201.zip	DOS	Recommended
DJGPP	http://www.delorie.com/djgpp/	DOS	Preferred

Table 25.1: List of compilers

Any documentation, library files or example programs must be installed manually. Recommended compilers are free (as in beer) and are known to work. It's up to you to figure out the correct installation procedure. Unlisted compilers may or may not work.

There is no such list for OS/X or Linux. Those platforms usually already come with an Open Source GCC compiler.

25.3 Compiling 4th

First copy all files to any directory you like. Now make the latter directory your current directory. The following commands are applicable to Linux, OS/X and other unices.

If you aren't using an ANSI-C compiler add the "-DARCHAIC" switch. If you are working on a Unix platform add the "-DUNIX" switch. If it still fails to compile, add the "-DBSD" switch as well. If it *still* fails to compile, add the "-DNOPIPES" switch. You won't have pipe support of course, but it will work. Finally, the "-DNOUNLINK" switch may be your last resort if you're using a pure C89 or C99 ANSI-C compiler without any POSIX support.

If you add the include-files to your own `/usr/include` directory, use the "-DUSRLIB4TH" switch. *Note that this isn't a recommended practice*.

If your compiler features a `stricmp()` function you want to use instead of 4tHs builtin `MatchName()`, use the "-DSTRICMP" switch. *These optional switches will be referred to in the following examples as "$(CFLAGS)".*

Unfortunately, not all C-compilers are created equal, so you have to check the documentation that came with your compiler. You have to check for three things:

1. First, the 4tH-toolkit assumes that all chars are signed. I know there are a few compilers out there, that assume that chars are unsigned (like the RS/6000 and GNU compilers). In most cases switches are available to correct that. Some K&R compilers do not support the type "void". If so, you might have to add "#define void" to 4th.h.

2. Second, compilers on non-Unix platforms might use different or additional switches, like those that determine the memory-model.

3. Third, this documentation assumes you call your compiler with "cc". Borland compilers are called with "bcc" or "tcc". Watcom compilers are called with "wcc". GNU compilers are called with "gcc". Microsoft compilers are called with "cl" for some obscure reason (C Language?).

The 4th program is a do-all compiler. It compiles the source, executes it and if you made a programming error it will decompile whatever it could compile. You can also load existing objects or save new objects. It takes at least two arguments, which are the command string and the 4tH-program. In Linux, OS/X or other unices, entering:

```
make
su
make install
```

should do the trick. Most MS-DOS compilers contain a version of 'make', but you probably have to recreate the makefile. Otherwise, compile it with:

```
cc $(CFLAGS) -O -o 4th 4th.c open_4th.c comp_4th.c exec_4th.c
dump_4th.c load_4th.c save_4th.c errs_4th.c name_4th.c free_4th.c
cgen_4th.c str2cell.c
```

If your compiler does not support a command line interface, you have to include these files:

```
free_4th.c
errs_4th.c
name_4th.c
dump_4th.c
exec_4th.c
load_4th.c
save_4th.c
comp_4th.c
open_4th.c
cgen_4th.c
str2cell.c
4th.c
```

25.4 Compiling the library

If you want to add the 4tH compiler to your own programs, I strongly advise you to create a library. As a matter of fact, I will assume that you have created the library later on. The library uses only a handful of functions, so it is feasible to create the library manually. However, we advise you to use the `Makefile`[1]. Carefully check all macros. In Linux, OS/X or other unices, entering:

```
make
su
make install
```

should do the trick. Most MS-DOS compilers contain a version of `make`, but you probably have to recreate the `Makefile`. The library consists of 11 functions:

```
comp_4th()
exec_4th()
dump_4th()
free_4th()
save_4th()
load_4th()
errs_4th[]
name_4th[]
open_4th()
cgen_4th()
str2cell()
```

[1] See section 25.5.

You can compile each function manually by issuing the command:

```
cc $(CFLAGS) -O -c <function>.c
```

You'll end up with 11 objectfiles. You can add these functions to a library by issuing:

```
ar r lib4th.a <function>.o
```

The resulting library must be moved to the /usr/lib directory. When using MS-DOS we strongly advise you to create a library for each memory-model, like "4ths.lib" for a small memory-model library, "4thl.lib" for a large memory-model library, etc. When you compile a function for a particular memory-model you'll have to add the memorymodel switch. Then create a library by issuing this command for each function:

```
lib 4th<model>.lib + <function>.obj
```

Whether you use a makefile or create the library manually, you should end up with a working 4tH library. If you are unable to make a working makefile, write a script- or batchfile. If you have to recreate the library, it will save you a lot of work.

25.5 Choosing Makefiles

There are several different Makefiles for different platforms. The default of the generic package is Linux. If that one doesn't work for you, you can resort to the following variants:

FILENAME	DESCRIPTION
Makefile	The default Makefile for your package
Makefile.LIN	A Makefile for the Linux platform, will work for most distributions and flavors
Makefile.UNX	A very vanilla Makefile, most likely to work on a host of different *nix platforms
Makefile.BSD	A vanilla Makefile, targetted at BSD-like platforms like HP-UX, NeXT, etc.
Makefile.COH	A Makefile for the Coherent 4.2 platform
Makefile.OSX	A Makefile for the Apple OS/X platform
Makefile.NAN	A Makefile to cross compile for the Linux Ben Nanonote platform
Makefile.W32	A Makefile (for cross compilation) using the MinGW compiler for the MS-Windows 32-bit platform
Makefile.DOS	A Makefile (for cross compilation) using the DJGPP compiler for the MS-DOS platform

The easiest and most reliable way to activate the `Makefile` required is to delete or rename the orginal and rename the `Makefile` of your choice to `Makefile`. Then issue:

```
make
```

And most likely you'll get a functioning executable. If it doesn't work read the `Makefile` and the `README` file of your package for more details. Don't forget to inform us if something doesn't work as advertised or if you have any comments or experiences to share.

25.6 Shared library

If you're working with 32-bit Linux, you can create shared libraries very easily. Just build the libraries with this switch:

```
make SHARED=1
```

If you get a message, you have to run `ldconfig`. Simply login as root and continue the installation as follows:

```
su
make SHARED=1 libinstall
```

This will install the shared library. Now start `ldconfig`:

```
su
ldconfig
```

Now try again. It should work:

```
su
make SHARED=1 install
```

Or, if installation fails in one of the last stages:

```
su
make SHARED=1 mostlyinstall
```

If you want to compile against the shared library, you don't have to a thing. Just type:

```
gcc -s -Wall -fsigned-char <program>.c -o <program> -l4th
```

This procedure might work on other GNU platforms too, but this has not been tested.

25.7 64-bit platforms

Although 4tH will work perfectly well on a 64-bit platform there are some disadvantages:

- HX files generated by this compiler are not portable to 32-bit platforms
- Some 4tH library files may not work properly without some modifications.

A quick fix is to change the size of a cell to a *four byte datatype*. The following procedure will usually work. Open `4th.h` and change these lines:

```
#define CELL_MIN LONG_MIN
#define CELL_MAX LONG_MAX

typedef long cell;
```

To this:

```
#define CELL_MIN INT_MIN
#define CELL_MAX INT_MAX

typedef int cell;
```

Save `4th.h` and compile as described in the previous sections. If you want a full 64-bit 4tH compiler, be aware that:

- You cannot compile 4tH as a shared library
- You have to regenerate the include files manually, unless you're working with Linux.

Linux *automatically* recreates the include files each time you perform a compile. If you're working with a GNU toolset, you may try the Linux `Makefile`. If that doesn't work or isn't an option in your particular situation you'll have to perform the procedure listed in section 25.8.

25.8 Regenerating the include files

If you compile the source on an incompatible platform, you will find the editor won't work. That's not much of a problem, generating the correct source for your platform only takes a few seconds. Just proceed as follows.

First, generate 4tH as usual and install it. The editor still won't work, but that is not important right now. Now go to the examples directory of your 4tH distribution and type:

```
make
```

After compilation has finished, move `editor.h`, `zeditor.h`, `teditor.h`, `mon.h` and `pp4th.h` to the C source directory. Now compile 4tH again:

```
make clean
make
```

Run 4tH and check if the editor loads properly:

```
./4th
```

You should see something like this:

```
4tH library V3.62 - Copyright 1994,2012 J.L. Bezemer

(S)creen file: new.scr
(O)bject file: out

(E)dit   (C)ompile   (R)un    (A)rguments

(Q)uit   (G)enerate  (B)uild  (D)ecompile

>e
Cannot open file OK
q

(S)creen file: new.scr
(O)bject file: out
```

```
(E)dit   (C)ompile   (R)un    (A)rguments

(Q)uit   (G)enerate  (B)uild  (D)ecompile

>q
```

If it doesn't, try another C compiler. Yours may not be compatible[2]. If it does, you can safely proceed with the installation:

```
su
make install
```

25.9 Optimizations

It really depends on your compiler. Some compilers allow optimizations up to level 3 (`-O3`), others won't even produce a usable compilant with any optimization enabled. It can even depend on the platform you're compiling for. It is hard to give a general recommendation, but try compiling 4tH without any optimization first, then crank optimization gradually up until the compilant doesn't work properly anymore or optimization doesn't give you any more speed or size advantages. You may have to do some benchmarking to find this out[3]. There is also a program that allows you to test the virtual machine. All listed preferred compilers allow the highest optimization level. The optimization level can be set manually or can be adjusted in the `Makefile`. Consult your compiler documentation for details.

25.10 GCC specific optimizations

Some people may wonder why 4tH doesn't come with `-fomit-frame-pointer` enabled. Well, first because the use of this compilation option is highly debatable[4] and second, because it may have detrimental effects on some platforms. However, if you *insist* on using it, you can - although we never found any convincing advantages in either size or speed.

What *does* give you a 15 - 25% speed boost is the use of `-DUSEGCCGOTO`. This compilation option enables a GCC-specific language extension, which makes the VM a whole lot faster. It requires an extra file, named `gcc_exec.h`. This file is generated by a 4tH program, `makegcch.4th`. If you ever want to make extensions to 4tH and you want the best performance under GCC, you will have to regenerate it, since it is *not* regenerated automatically. Any working 4tH will do:

```
4th cxq makegcch.4th cmds_4th.h gcc_exec.h
```

If you want to use any of these options, you will have to edit the `CFLAGS` section of your `Makefile` accordingly, e.g.:

```
CFLAGS= -DUNIX -DUSEGCCGOTO -fsigned-char -Wall -O3 -s
```

See section 25.5 for details.

[2] See table 25.1 for a list of compatible compilers.
[3] 4tH comes with a wide selection of benchmarking programs.
[4] http://timetobleed.com/gcc-optimization-flag-makes-your-64bit-binary-fatter-and-slower/

25.11 Using the library

Before we dive into the depths of the API, we will tell you how to compile a program that uses the 4tH library. If you are using an advanced MS-DOS or MS-Windows compiler this may or may not apply to you. In that case we advise you to check your documentation. If you are using a plain vanilla compiler this will usually work. Compile the program with:

```
cc -c <program>.c
```

When you use a non-Unix system, link the objectfile with the startup code and libraries to create an executable program. The <model>.obj is the startup code for C-programs and <model>.lib is the runtime-library.

```
link <model>.obj <program.obj>, <program>„ 4th<model>.lib <model>.lib
```

Unix developers just give the command:

```
cc <program>.c -o <program> -l4th
```

If you happen to use the GNU Compiler Collection (gcc), just issue:

```
gcc -s -Wall -fsigned-char <program>.c -o <program> -l4th
```

After you've built the library you can also issue these commands to (re)compile 4th. If you happen to use a GNU based platform, you can configure `make` to work with 4tH. You just have to create a small `Makefile` in your current working directory[5]:

```
# GNU Make - implicit rules for 4tH
# Copyright 2006, 2009 Hans Bezemer
%.c : %.4th
        4th cgq $< $@
%.c : %.hx
        4th lgq $< $@
%.4th : %.4pp
        pp4th -o $@ $<
CC=gcc
CFLAGS=-fsigned-char -Wall -O3 -s
LDLIBS=-l4th
LDFLAGS=-s
```

Note this is only an example, so you may have to change it for your system. Now copy `4th.h` to the directory where the 4tH source is located and you're done:

```
$ make examples/eliza
4th cgq examples/eliza.4th examples/eliza.c
gcc -fsigned-char -Wall -O3 -s -c examples/eliza.c -o examples/eliza.o
gcc -s examples/eliza.o -l4th -o examples/eliza
rm examples/eliza.c examples/eliza.o
$ examples/eliza
HI! I'M ELIZA. WHAT'S YOUR PROBLEM?
>
```

[5]That is the directory where you are when you call 4tH and is usually pointed to by the environment variable `DIR4TH`.

Isn't that easy? You can also use this procedure to turn 4tH utilities like the preprocessor into full fledged programs. In fact, the `Makefile` takes the preprocessor into account if required:

```
$ make teonw
pp4th -o teonw.4th teonw.4pp
4th cgq teonw.4th teonw.c
gcc -fsigned-char -Wall -O3 -s -c -o teonw.o teonw.c
gcc -s teonw.o -l4th -o teonw
rm teonw.4th teonw.c teonw.o
```

Chapter 26

Using the 4tH API

26.1 Introduction

One of the design requirements of the 4tH library was that it had to be very easy to use. We've seen many APIs that were impossible to use and put most of the burden on the developer.

4tH takes a different direction. We've designed an API that almost exactly matches the tasks you want to perform. Want to compile? Compile. Want to decompile? Decompile. Want to save? Save. Just like that. No difficult to understand datatypes, no initialization, no garbage collection, no checks.

The only error, you, the developer can make is fill up memory. Virtually all other errors are caught by the API. E.g. 4tH will refuse to save or execute a 4tH-program when compilation failed. Of course, if you manipulate 4tHs datastructures directly you can still bring it to its knees, but I assume that is not what you want.

What most people tend to forget is that 4tH is foremost a *library*. The executables that come with it are just examples:

pp4th A development system much like ordinary command line based compilers like `gcc`.

4th A development system modeled after early versions of Turbo Pascal.

4tsh A multitasking shell that turns 4tH into a kind of operating system.

But a 4tH based C program can be much less elaborate as we will show you in the next chapter.

26.2 A sample program

There are ten API-functions.

That's really all! So if you want to compile a 4tH source, save it to disk, execute it and finally discard it, you've virtually written the program.

H-code is nothing but a pointer to a structure. Even if you thought you never worked with structures before, it's as easy as working with files. We'll show you.

When you want to use files you first have to include `stdio.h`, like:

API	FUNCTION
comp_4th()	Loads and compiles a 4tH source to an H-code object
exec_4th()	Executes an H-code object
dump_4th()	Decompiles an H-code object
free_4th()	Frees an H-code object from memory
save_4th()	Saves an H-code object to an HX-file on disk
store_4th()	Saves an H-code object to an HX-file in memory
load_4th()	Loads an HX-file from disk and installs H-code
fetch_4th()	Loads an HX-file from memory and installs H-code
open_4th()	Creates a small 4tH program that loads a 4tH source
cgen_4th()	Generates a standalone C source with embedded H-code

Table 26.1: API functions

```
#include <stdio.h>
```

If you want to use a single file for output, you've got to declare a file-pointer like:

```
FILE* Outfile;
```

Before you can use a file, you have to open it:

```
Outfile = fopen ("filename.ext", "w");
```

If the file cannot be opened, `fopen()` returns a NULL-pointer. You have to check that before you can safely use the file:

```
if (Outfile == NULL) printf ("Unable to open file");
else fprintf (Outfile, "This is written to disk");
```

Finally, when you're done, you have to close the file:

```
fclose (Outfile);
```

Working with the 4tH library is very similar. When you want to use the 4tH library you write:

```
#include "4th.h"
```

When you use an H-code object (which is a compiled 4tH program), you declare a H-code pointer:

```
Hcode* Program;
```

Before you can use the H-code pointer, we first got to compile a 4tH source. 4tH source is simply a `malloc()`ed ASCIIZ string. That means that anything you can convert to a string stored in dynamic memory can be used as 4tH source. That includes constant strings, environment variables, lines read from a file, etc.

In this example we use the `strdup()` function to convert a constant string. We know that not all runtime-libraries contain such a function, so if you want give it a try, here is the source:

```
char *strdup (char* str)
{
      char *p;
      p = calloc (strlen (str) + 1, sizeof (char));
      if (p) strcpy (p, str);
      return (p);
}
```

Now we can all create a 4tH source and compile it. This one will create the famous "Hello world" program:

```
Program = comp_4th (strdup (".\" Hello world\" cr"));
```

Without any checking you can try to execute it:

```
exec_4th (Program, 0, NULL, 0);
```

The "0" means we do not want to transfer any variables or constants to the execution environment, but we'll get to that later on. You could make a second call to `exec_4th()` and see the program execute twice. The H-code is still in memory. In order to free it from memory, we have to end our program with:

```
free_4th (Program);
```

We are finished now. The full program looks like this:

```
#include "4th.h"
#include <stdlib.h>

int main(int argc, char** argv)
{
      Hcode* Program;
      Program = comp_4th (strdup (".\" Hello world\" cr"));
      exec_4th (Program, 0, NULL, 0);
      free_4th (Program);
      return (EXIT_SUCCESS);
}
```

Now compile the C-program and execute it. You should see that it prints "Hello world" on the screen. Now, that wasn't too hard, was it?

26.3 A first look at open_4th()

You probably don't want to compile constant strings. Most certainly you want to create a source-file and compile it. If you come to think of it, that could be the hardest part when you want to make your own 4tH-compiler.

Don't worry. We've created a function that handles just that: `open_4th()`. The function creates a tiny 4tH program that tells `comp_4th()` which file to load. `open_4th()` just wants to know which file to load.

If an error occurs (which is very rare), `open_4th()` returns a NULL pointer. If `open_4th()` is successful it returns a char-pointer to the program in memory. You can feed the return-value of `open_4th()` directly to `comp_4th()`. So, we've got to declare a char-pointer for the return-value of `open_4th()` and call the function:

```
char* source;

source = open_4th ("venture.4th");
```

No need to open or close files, `comp_4th()` will take care of that. You're one step closer to the creation of your own compiler.

26.4 A closer look at H-code

H-code is not just a simple pointer to some simple structure. In fact, it is more complex than a file-pointer. It is comprised of several parts.

1. First, the header. The header contains all information about the actual program, e.g. the number of variables, the size of allocated space, its status. The mere existence of an H-code pointer doesn't mean you actually got a program you can execute.
2. Second, the Code Segment. This contains the actual program. If you don't have a Code Segment, there is nothing to execute since you have no program. The Code Segment is an array of words. A word consists of a token and an argument. Every token matches a piece of compiled C in the interpreter. We'll get to that later on.
3. Third, the String Segment. This segment only contains constant strings, defined by e.g. S" , ," , .(and .".
4. Fourth, the Integer Segment. This segment contains the stacks and all writable integer data. It is only present when a program is sleeping (or hibernating, if you prefer).
5. Fifth, the Character Segment. This segment contains all writable character data. It is only present when a program is sleeping.

The first three parts are read-only to the 4tH-programmer. If you are smart, you consider them to be read-only too. There is no need whatsoever to change anything here. The API knows best.

26.5 A closer look at HX-code

First of all, you have to understand how numbers are stored in HX-code. There are four different kinds of numbers in HX-code:

1. Constants
2. Tiny numbers
3. Short numbers
4. Long numbers

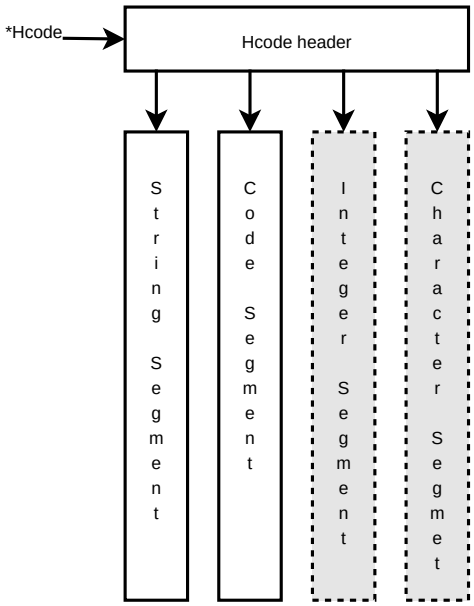

Figure 26.1: Hcode structure

BIT	MNEMONIC	VALUE	SIGNIFICANCE
0	HCSIGN	1	Negative number
1	HCBYTE	2	Tiny number
2	HCSHRT	4	Short number
3	HCZERO	8	0 or CELL_MIN
4	HCONE	16	1 or -1

Table 26.2: HX type-byte encoding

All numbers are preceded by a type-byte, which depicts the kind of number that is stored. The type-byte encoding is listed in table 26.2.

When `HCZERO` or `HCONE` are set, there won't be any additional bytes. If the type-byte contains 17 (which is 16 + 1) we're actually looking at -1. If the type-byte contains 8, we're looking at zero. The first three bytes of any HX-file depict the length in bytes of a tiny number, a short number and a long number. Typically they contain 1, 2 and 4. That means if bit 1 is set, the type-byte contains a 2, which means only a single byte will follow. If bit 2 is set, two bytes will follow and finally, if *no* bit is set four bytes will follow.

The bytes following it will always depict a small-endian, positive number. Bit 0 is valid in any combination and signals a *negative* number. Since `CELL_MIN` can differ depending on the platform it is a constant, depicted by both bit 0 and bit 3 set, which is 1 + 8 (9). HX-code contains several sections:

- The header
- The Code Segment
- The String Segment
- The Character Segment *(optionally)*
- The Integer Segment *(optionally)*
- The checksum

First, we have the header. The header contains certain information so 4tH can establish its compatibility and create the required environment. The header contains the following elements:

- The length of a tiny number *(in bytes)*
- The length of a short number *(in bytes)*
- The length of a long number *(in bytes)*
- The largest positive number a cell can contain (`CELL_MAX`)

- The version of 4tH *(in hexadecimal)*
- The application byte *(usually 0)*
- The size of the Code Segment *(in elements)*
- The size of the String Segment *(in characters)*
- The offset of the Variable Area *(>0 for sleeping VMs)*
- The size of the Variable Area *(in cells)*
- The size of the Character Area *(in characters)*

The numbers are stored in the format, we described above. The Code Segment requires some explanation. If a 4tH token does not require any arguments (e.g. 'CR') no arguments are stored. If it does (e.g. 'LITERAL') the actual value is stored right after it in the number format we have described above.

The Integer Segment and Character Segment are *only* stored when the HX-code contains a sleeping VM. When saved, the Integer Segment is completely contained in the HX-code, stack and all. All numbers are stored in the now famil1ar format. HX-code contains a sleeping VM when the offset in the header is greater than 0. The offset is computed by adding the number of 4tH system variables (both writable and non-writable) to the number of C-variables that have been passed to `exec_4th()`.

The checksum is the way 4tH checks the integrity of HX-code. Every time 4tH writes a byte of HX-code, it is XORed with the previous byte. The byte resulting from the sum of these operations is written last[1]. When 4tH reads back the HX-code the same operation is performed, hence the values should be the same or some kind of corruption has crept in. If that happens, 4tH will reject the HX-code read.

Let's take a look at a real life example:

```
: hello ." Hello world!" cr ;
hello
```

When decompiled, it looks like this:

```
4tH message: No errors at word 5
Object size: 5 words
String size: 13 chars
Variables  : 0 cells
Strings    : 0 chars
Reliable   : Yes
[   0] branch    (3)
[   1] ."        (0)    Hello world!
[   2] cr        (0)
[   3] exit      (0)
[   4] call      (0)
```

And finally, this is the resulting HX-code:

01	Size of a tiny number
02	Size of a short number
04	Size of a long number

[1] The same algorithm was used for the Sinclair ZX Spectrum tape handling routines.

00	Start of a long number
FF	1st byte long number
FF	2nd byte long number
FF	3rd byte long number
7F	*Long number:* 7FFFFFFFh
04	Start of short number
62	1st byte of short number
03	*Short number:* 0362h
08	*Constant:* 00h
02	Start of tiny number
05	*Tiny number:* 05h
02	Start of tiny number
0D	*Tiny number:* 0Dh
08	*Constant:* 00h
08	*Constant:* 00h
08	*Constant:* 00h
1D	BRANCH
02	Start of tiny number
03	*Tiny number:* 03h
05	."
08	*Constant:* 00h
02	CR
41	EXIT
47	CALL
08	*Constant:* 00h
48	H
65	e
6C	l
6C	l
6F	o
20	<space>
77	w
6F	o

72	r
6C	l
64	d
21	!
00	<null>
D6	Checksum

Why this elaborate scheme? Well, for two reasons. First, it allows HX-code to be portable across a wide range of platforms[2]. Second, it is *very* compact. If we would write out all numbers in full, most space would be occupied by zeros. That would be a waste, especially when embedding HX-files. The speed penalty is negligible.

26.6 A first look at comp_4th()

The function `comp_4th()` is one of the most complex and most important functions of the API. It takes a source and compiles that to H-code. If you want to save the source, copy it or reload it, since `comp_4th()` consumes it entirely. That is why source has to be allocated in dynamic memory. On the other hand, it explains why compilation needs very little memory.

If `comp_4th()` can't compile anything, it returns just the header, containing all the information on what went wrong and where. If there is not enough memory to allocate even the header it returns a NULL-pointer. If it could compile part of the code it returns what it could compile. But that might not be everything and you need it all to get a program you can execute.

Lucky for you, but the API can determine whether `comp_4th()` returned an executable H-code or not. If `exec_4th()` doesn't execute the program, it just couldn't.

The `comp_4th()` function is just as smart. It can survive the NULL pointer you feed it. When you read on, you'll find out that the `comp_4th()` function is a toolkit by itself with lots of tools to create your own 4tH words.

If we extend the example we started with `open_4th()`, we could continue like this. In order to make an executable H-code we feed source to `comp_4th()` and get H-code in return. We just need a pointer to store its address:

```
Hcode *Program;
```

So the entire 4tH-compiler now looks like:

```
char *source;
Hcode *Program;

source = open_4th ("venture.4th");
Program = comp_4th (source);
```

We assume you want to execute the program you've just compiled, so we'll continue with the interpreter-function called `exec_4th()`.

[2]As long as the 7-bit ASCII characterset is supported. EBCDIC HX-files will not be portable for obvious reasons.

26.7 A first look at exec_4th()

The `exec_4th()` function is essentially very simple. It contains small pieces of C that can be matched with the tokens in the Code Segment. The `exec_4th()` function executes these small pieces of C until there are no more words left to execute or an error occurs.

Additionally, it creates two new segments, which are discarded when `exec_4th()` terminates without hibernation. Each segment is essentially an array of a specific datatype. The sizes of these segments are specified in the header of the H-code.

The Character Segment contains characters. First, the Terminal Input Buffer, abbreviated to TIB. When you execute REFILL, this is the place where the string you typed is stored. Second, the PAD. This is the place where `exec_4th()` stores temporary strings. Finally the Allocation Area where strings, defined by STRING are allocated.

The Integer Segment contains signed 32 bit integers. First, the Stack Area. It contains both the return stack and the data stack. The return stack grows downward and the data stack grows upward. Second, the System Variable Area. The System Variable Area contains three variables, HANDLER, HERE and HLD. They can only be accessed by 4tH itself. Third, the Variable Area. The Variable Area contains four basic types of variables.

First, the environment variables. In standard 4tH there are five environment variables, HI, FIRST, LAST, CIN and COUT. They are all read-only. Second, the predefined variables. In standard 4tH there are five predefined variables, BASE, >IN, OUT and the variable pair SOURCE. Later on, we'll teach you how to add your own. Third, the application variables. These are copies of C-variables or constants. Fourth, the user-defined variables. These are defined by the application-programmer with VARIABLE, VALUE, DEFER, FILE or ARRAY.

The `exec_4th()` function takes an H-code pointer and returns the current value of the variable OUT when it exits. When an error occurs, `exec_4th()` returns always the largest negative 32 bit integer, which is also the default value of OUT. In order to give you maximum control, you can also transfer string-arrays and C-variables or constants to `exec_4th()`.

C-variables have to be of type 'cell'. This type is predefined in the `4th.h` headerfile. So you can just write:

```
#include "4th.h"
cell february = 29;
```

You can mix constants or variables as you like. Just add the appropriate cast. You also have to declare the number of variables or constants you transfer (in this case 12). Let's say you transfer the number of days of each month to `exec_4th()`:

```
#include "4th.h"
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell february = 29;
      cell Result;
      Hcode *Program;
      char *source;

      source = open_4th ("months.4th");
      Program = comp_4th (source);
      Result = exec_4th (Program, 0, NULL, 12, (cell) 31,
      february, (cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell)
```

```
        31, (cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell) 31);
        return (EXIT_SUCCESS);
}
```

The application-programmer can use the 'APP' variable to access all months:

```
 12 0 do app i th ." days: " ? cr loop
```

Later, we'll learn you how to assign names to those application-dependant variables. Note that although 'Result' contains the return-value of the `months.4th` program, the C-program does absolutely nothing with it. In the next example we'll show you how you can use this value.

Take this C-program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
        cell Result;
        Hcode *Program;
        char *source;

        source = open_4th ("calc.4th");
        Program = comp_4th (source);
        Result = exec_4th (Program, 0, NULL, 2, (cell) 5, (cell) 7);
        printf ("Result: %ld\n", (long) Result);
        return (EXIT_SUCCESS);
}
```

This program compiles `calc.4th` and transfers '5' and '7' to `exec_4th()`. What is returned in 'Result' depends on what `calc.4th` does. Let's take a look at `calc.4th`:

```
app 0 th @ app 1 th @ + out !
```

It fetches both variables, adds them and assigns the sum to OUT. Thus, when `exec_4th()` terminates it returns the value of OUT. This value is assigned to 'Result'. Then, 'Result' is cast to 'long' and displayed:

```
Result: 12
```

We haven't used the other two arguments of `exec_4th()` yet. These are used to pass string constants to your 4tH application. Most of you will use it to pass commandline arguments to 4tH.

Let's say we want to pass these arguments to 4tH in C-style. So "0 ARGS" is the name of our 4tH-program. That means we have to skip the name of the C-program itself. The name of our 4tH-program in in argv [1]. That also means we have to decrement argc, before passing it to `exec_4th()`:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
```

```
        Hcode *Program;
        char *source;

        if (argc > 0) {
              source = open_4th (argv [1]);
              Program = comp_4th (source);
              (void) exec_4th (Program, argc - 1, argv + 1, 0);
              return (EXIT_SUCCESS);
        }
        return (EXIT_FAILURE);
}
```

Note that argv [1] is not the same as argv + 1! When we call our C-program with:

```
main args.4th hello here I am
```

argc will be 6. That means `exec_4th()` will get a stringarray with five strings (since we discarded one). We can access these strings by:

```
argn 0> if argn 0 do i args count type cr loop then
```

Which would print:

```
args.4th
hello
here
I
am
```

You do not *have* to pass argc and argv; other string arrays of the same type (`**char` or `*char[]`) are okay too. Just don't forget to pass the correct number of elements in the string array.

So now you know how your H-code program can communicate with your C-program. You can transfer any number of variables to the 4tH-environment and retrieve the result. You can even pass commandline arguments to the 4tH-environment and use them inside your application. In the next section you will be introduced to some other interesting properties of the 4tH-environment.

26.8 A first look at free_4th()

Let's take a look at this program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
        cell Result;
        Hcode *Program;
        char *source;

        source = open_4th ("calc.4th");
        Program = comp_4th (source);
        Result = exec_4th (Program, 0, NULL, 2, (cell) 5, (cell) 7);
        printf ("Result: %ld\n", (long) Result);
        Result = exec_4th (Program, 0, NULL, 2, (cell) 12, (cell) 9);
        printf ("Result: %ld\n", (long) Result);
        return (EXIT_SUCCESS);
}
```

Essentially, it is the very same program we've seen in the previous section. But here the `calc.4th` program is executed twice with different arguments. Can that be done without recompiling the program? Yes! The pointer 'Program' is still valid and points to the very same 4tH program in memory. You can execute it any number of times without recompiling. You can go even further:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      char *source;
      Hcode *Multiply;
      Hcode *Subtract;

      source = open_4th ("multiply.4th");
      Multiply = comp_4th (source);
      source = open_4th ("subtract.4th");
      Subtract = comp_4th (source);

      Result = exec_4th (Multiply, 0, NULL, 2, (cell) 7, (cell) 5);
      printf ("Result: %ld\n", (long) Result);
      Result = exec_4th (Subtract, 0, NULL, 2, (cell) 7, (cell) 5);
      printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCESS);
}
```

The file `multiply.4th` contains:

```
app 0 th @ app 1 th @ * out !
```

The file `subtract.4th` contains:

```
app 0 th @ app 1 th @ - out !
```

So executing this C-program will give the following result:

```
Result: 35
Result: 2
```

And yes, both programs can be re-executed any number of times. But what if some 4tH program has served his purpose? Does it have to remain in memory all the time? No. Since it is located in dynamic memory it can be freed. Not with `free()`, since H-code is too complex to be served with a simple function like `free()`. But a special function is included in the API, which serves the same purpose:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      char *source;
      Hcode *Multiply;
      Hcode *Subtract;
```

```
        source = open_4th ("multiply.4th");
        Multiply = comp_4th (source);
        source = open_4th ("subtract.4th");
        Subtract = comp_4th (source);

        Result = exec_4th (Multiply, 0, NULL, 2, (cell) 7, (cell) 5);
        printf ("Result: %ld\n", (long) Result);

        free_4th (Multiply);

        Result = exec_4th (Subtract, 0, NULL, 2, (cell) 7, (cell) 5);
        printf ("Result: %ld\n", (long) Result);
        free_4th (Subtract);
        return (EXIT_SUCCESS);
}
```

The function `free_4th()` takes an Hcode-pointer and frees all resources. There is really nothing more to it. Remember that `free_4th()` doesn't alter the pointer itself. It may still contain a value, but of course using that value is asking for trouble. The API checks quite a few things by itself, but that doesn't mean you can start to write sloppy programs!

26.9 A first look at save_4th()

We've already seen that we can compile a 4tH-program and keep it in memory for as long as we want. We can also discard it if we don't need it anymore. But what if we want to reuse the compilant later? Or if we want to distribute 4tH programs without revealing our source-code?

You can do that easily. 4tH uses another main format which not only enables you to load compiled programs, but also run them on a multitude of platforms. It is called the 'Hcode eXecutable' (HX-file) and it is fully portable across all platforms 4tH supports.

Saving a program is very easy too. You don't even have to open or close files. Here is a very simple compiler:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
        char *source;
        Hcode *Program;

        if (argc == 3) {
                source = open_4th (argv [1]);
                Program = comp_4th (source);
                save_4th (Program, argv [2]);

                free_4th (Program);
                return (EXIT_SUCCESS);
        }
        return (EXIT_FAILURE);
}
```

You just declare the input and the output file on the commandline and when no errors occur an HX-file is saved to disk. The `save_4th()` function takes the Hcode pointer and the filename you want to save it to. Note `save_4th()` supports hibernation too; just feed it a sleeping virtual machine. That's all!

26.10 A first look at load_4th()

But you don't just save compiled programs. You want to be able to reuse them too. There is a special function that reads an HX-file and restores the H-code to its original form. This API-function is easy to use too. Just feed it the name of the file and it returns a pointer to the H-code. This is the listing of a simple HX-execute:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      Hcode *Program;
      cell Result;

      if (argc == 2) {
            Program = load_4th (argv [1]);
            Result = exec_4th (Program, 0, NULL, 0);
            printf ("Result: %ld\n", (long) Result);
            free_4th (Program);
            return (EXIT_SUCCESS);
      }
      return (EXIT_FAILURE);
}
```

You just declare the HX-file on the commandline and when no errors occur it is executed. Finally, it displays the result of the program.

26.11 A first look at error-trapping

If you are a professional programmer you might appreciate the ease of use of the 4tH toolkit, but you have the feeling you don't have any control. If that were the case, we would feel the same way. In fact, you have all the control you'll ever need. In the background, 4tH keeps track of everything that is happening.

We've already discussed the header of the H-code. All status-information is stored here. And it's all available. May be you'll find it a little more complex and intimidating, but you can easily master it. Let's take a look at this piece of code:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));
```

This piece of code tries to compile the classic "Hello world" program. But did it compile? If `comp_4th()` returned a NULL-pointer, you know there was not enough memory. But like any other compiler, there are a million other things that can go wrong. Although other API functions will refuse unreliable H-code, sometimes we want to check it ourselves and take alternative action if necessary.

All information regarding the status is saved in the header. But if `comp_4th()` returns a NULL-pointer there is no header. So we have to check that first:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL) printf ("Not enough memory\n");
```

If the program enters the 'else' clause we know that a header exists. Now we need to check the status. Did an error occur? There are two members in the header we can check. First, 'ErrNo' which contains an error-code. If 'ErrNo' contains '0', there were no errors:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      if (Program->ErrNo == 0)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("There were errors\n");
}
```

Note that the member 'ErrNo' is closely linked to the H-code. That is hardly surprising since it is part of the H-code! But we still don't know which error occurred.

Fortunately, there is a predefined array of error-messages we can use. It is called `errs_4th[]` and you can use it without declaring it explicitly, since `4th.h` takes care of that. If you have correctly built the library it will automatically be linked in:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      if (Program->ErrNo == 0)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("Error: %s\n", errs_4th [Program->ErrNo]);
}
```

Of course, checking error-codes by the number is not ideal from a maintenance point of view. In `4th.h` you'll find a lot of `#define`-s describing these errors. The mnemonic for 'no errors' is 'M4NOERRS', so we can slightly alter our program to:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
     printf ("Not enough memory\n");
else {
      if (Program->ErrNo == M4NOERRS)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("Error: %s\n", errs_4th [Program->ErrNo]);
}
```

In the next section we'll show how you can help the 4tH-programmer to pinpoint his errors even more precisely.

26.12 A first look at dump_4th()

Although 4tH can tell you what error you made and where you made it, you may find it pretty hard to locate it anyway. That is because 4tH makes a reference to the compilant instead of the source.

That is because 4tH preprocesses the source and never looks further ahead than one single word, so a reference to the source wouldn't help you much anyway. That is the bad news.

The good news is that the instructions 4tH uses internally are virtually identical to the ones you used in your source. If you decompile the program you should still be able to recognize your source. The function `dump_4th()` is essentially a decompiler. Let us show you a small part of a program by Leo Brodie we converted to 4tH:

```
VARIABLE SPAN
: EXPECT ACCEPT SPAN ! ;

16 CONSTANT #NAME
 8 CONSTANT #EYES
16 CONSTANT #ME               ( length of fields )

#NAME STRING NAME
#EYES STRING EYES
#ME   STRING ME               ( calculate values )

...
```

If you decompile the entire program you will get a listing, which consists of two parts. First the header:

```
4tH message       : No errors at word 80
Object size       : 81 words
String size       : 208 chars
Variables         : 1 cells
Strings           : 40 chars
Reliable          : Yes
```

First it will present the current status of this Hcode program. The words are numbered and we begin counting at zero. This means this program is okay, since word 80 is the very last word. We can derive that information from the second field that lists that there are 81 words, numbered from 0 to 80. The third field tells us there are 208 characters stored in the String Segment.

The next two fields tell us something about the runtime-environment. The total number of strings we defined take up 40 bytes and we defined one single variable. Finally, 4tH tells us this piece of Hcode is reliable. That means it can be saved to disk or executed. If it had told us the Hcode was *not* reliable, we could still have decompiled it. Otherwise it could get very hard to pinpoint an error. Next is the decompiled program itself:

```
[ 0] branch       (4)
[ 1] accept       (0)
[ 2] variable     (0)
[ 3] !            (0)
[ 4] exit         (0)
...
```

As you will see, you can still tell what this program is all about. Since 4tH has no dictionary, but uses a symbol-table, all lexical references are gone. There is no indication that the first

word was ever called 'EXPECT' or that variable #0 was named 'SPAN'. In fact, if you would name them differently, it would still compile to the very same Hcode.

The bracketed numbers are the 'addresses' of the words. Then it prints the name of the compiled token. Finally the argument part of the word is printed within parentheses.

Not all tokens have arguments. 'ACCEPT' and 'EXIT' don't need one. They either take their arguments from the stack or don't have any. But 'LITERAL' and 'BRANCH' do need them. 'LITERAL' needs the value of the number it has to throw on the stack and 'BRANCH' needs the address it has to branch to (in fact, it branches to the next token after the indicated address). The interpreter "knows" which token has a valid argument and which ones it can ignore.

But you surely want to know how you can integrate this decompiler into your own programs. Like all other functions, it needs an Hcode-pointer. It also needs a device where you can send the report to.

To give you maximum flexibility we used an open stream, so you can use the screen, a printer or a file. A disadvantage is you have to open and close the file yourself when applicable.

We also gave you the opportunity to do a partial listing. You can tell `dump_4th()` what range you want to decompile. These parameters are protected too. If you feed `dump_4th()` an invalid range it will try to figure out what range is most applicable. This allows you to do a full listing with minimum effort by issuing a range from word 0 to word -1.

A sample application may look like this:

```
FILE *ErrFile;
Hcode *Program;

/* other code */

if ((ErrFile = fopen ("error.lst", "w")) == NULL)
      printf ("Cannot open file\n");
else {
        dump_4th (Program, ErrFile, 0, -1);
        if (fclose (ErrFile)
            printf ("Error closing file\n");
}
```

If you want to print it to screen, you can use either 'stdout' or 'stderr'. Note that 'stderr' cannot be redirected easily under MS-DOS, so we'll use 'stdout' here:

```
Hcode *Program;

/* other code */

dump_4th (Program, stdout, 0, -1);
```

You can always provide a report when compiling or you can use error-checking to decide whether you execute or save Hcode or print a report. This is the listing of a complete compiler:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
```

```
{
      char *source;
      Hcode *Program;

      if (argc == 2) {
            source = open_4th (argv [1]);
            Program = comp_4th (source);

            if (Program == NULL)
                printf ("Not enough memory\n");
            else {
                  if (Program->ErrNo == M4NOERRS)
                     (void) exec_4th (Program, argc - 1, argv + 1, 0);
                  else dump_4th (Program, stdout, 0, -1);

                  free_4th (Program);
                  return (EXIT_SUCCESS);
            }
      }
      return (EXIT_FAILURE);
}
```

This program loads a source, compiles it and executes the resulting Hcode if no errors occurred. It tells you when there is not enough memory and provides a decompiler-listing on screen when a programming error was made. Pretty neat, huh?

26.13 A first look at cgen_4th()

`cgen_4th()` allows you to create native standalone programs with minimal effort. It is a lot like `save_4th()`. All you need is a Hcode object in memory and an open stream. The stream will allow you to send the C program `cgen_4th()` generates to screen, file or a printer. A sample application could look like this:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      Hcode *Program;
      cell Result;
      FILE *CFile;

      if (argc == 2) {
            if ((CFile = fopen ("myfile.c", "w")) != NULL) {
                  Program = load_4th (argv [1]);
                  cgen_4th (Program, CFile);
                  free_4th (Program);
                  fclose (CFile);
                  return (EXIT_SUCCESS);
            }
      }
      return (EXIT_FAILURE);
}
```

This program will load an HX file and send a complete C source to `myfile.c`. You can compile it by issuing[3]:

```
cc myfile.c exec_4th.c errs_4th.c str2cell.c -o myfile
```

[3]See section 25.11.

or, if you have the 4tH library installed by:

```
cc myfile.c -o myfile -l4th
```

Just like any other 4tH related C program. You will have a native executable for your platform and nobody will ever know it's actually a 4tH program. But there are even more ways to use embedded 4tH as we will see.

26.14 Converting HX-files

With the 4tH program `bin2h.4th` you can convert HX-files to portable C-source. This opens a whole new range of applications. `cgen_4th()` is a quick way to create standalone programs, but `bin2h.4th` allows you to embed highly compacted HX code into your C program.

This is particularly useful when memory is tight, because you can load the HX code when it is actually needed and discard it afterwards. Furthermore, you can have several HX code snippets inside your C program, which is not possible when using `cgen_4th()`.

On the downside, HX code is a little more difficult to handle. `bin2h.4th` just creates the embedded code, not an entire program. You will have to write that yourself. Furthermore, in order to use `bin2h.4th` you need to create an HX file first. Still, using `bin2h.4th` is dead easy:

```
4th cxq bin2h.4th HelloWorld myprog.hx myprog.h
```

You might have noticed that the only thing you have to provide is the name of the HX file (`myprog.hx`), the name of the include file (`myprog.h`) and the name of the variable in which the embedded HX code is stored (`HelloWorld`).

26.15 A first look at fetch_4th()

A typical `bin2h.4th` generated includefile looks like this:

```
static unit EmbeddedHX [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x62', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xc3'
};
```

Since it contains compiled code there is no need for functions like `comp_4th()`. However, HX code can not be fed to `exec_4th()` directly. It has to be loaded first. The function `load_4th()` does this automatically. There is a function in the 4tH API to load bytecode from memory, called `fetch_4th()`. Just pass the HX code pointer to it:

```
Hcode* Program;
Program = fetch_4th (HelloWorld);
```

Now the bytecode is installed and can be executed by `exec_4th()` in the usual way:

```
(void) exec_4th (Program, 0, NULL, 0);
```

In this case, it will simply print "Hello world!" to your screen.

26.16 A first look at store_4th()

And what if you don't want to store bytecode on disk, but in memory? Well, you can do that too. 4tH provides the function `store_4th()`, which takes a Hcode pointer, a pointer to a buffer and the size of that buffer. It is very easy to use. Just create a sufficiently large buffer, either dynamic or static, and load the Hcode. No need to worry about buffer overflow, when properly used `store_4th()` will prevent such mishaps. It will even return the amount of bytes it has written:

```
#include "4th.h"
#include <stdlib.h>

int main (int argc, char **argv)
{
  Hcode *Object;                        /* Hcode object */
  unit   Buffer [1024];                 /* memory allocated to bytecode */
  char  *Program;                       /* 4tH sourcecode */
  size_t MySize;                        /* number of bytes written to memory */

  Program = open_4th ("hello.4th");     /* create sourcecode */
  Object  = comp_4th (Program);         /* compile and save the sourcecode */
  MySize  = store_4th (Object, Buffer, sizeof (Buffer));
  printf ("%s, %ld bytes used\n", errs_4th [Object->ErrNo], MySize);
  free_4th (Object);                    /* destroy the Hcode object */

  Object  = fetch_4th (Buffer);         /* read the bytecode from 'Buffer' */
  exec_4th (Object, 0, NULL, 0);        /* execute the Hcode object */
  free_4th (Object);                    /* destroy the Hcode object again */
  return (EXIT_SUCCESS);                /* signal 'everything ok' */
}
```

This program compiles some 4tH source, saves the HX code in a buffer and then discards the original Hcode. Finally, the HX code is reloaded, run and discarded again. Note that `store_4th()` supports hibernation, like its close brother `save_4th()`. When this program is run it will display:

```
No errors, 36 bytes used
Hello world!
```

Needless to say that you can do very neat things with this function, like paging programs in and out very quickly using *very* little memory, storing multiple programs in a single buffer, etc. Use your imagination!

26.17 Examples of embedded HX code

The include-files `bin2h.4th` generates contain global variables. You can either integrate them in your sourcecode or include them, e.g:

```
#include <stdlib.h>
#include "4th.h"

#include "hello.h"
```

or:

```
#include <stdlib.h>
#include "4th.h"

static unit EmbeddedHX [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x62', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xc3'
};
```

It's really up to you. You can install and uninstall HX code as often as you want. You can also have multiple instances of the HX code in memory if you need to. E.g. this is perfectly valid:

```
#include <stdlib.h>
#include "4th.h"

static unit EmbeddedHX [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x62', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xc3'
};

int main (int argc, char** argv)
{
      Hcode*      Instance1;
      Hcode*      Instance2;
                       /* load two instances of HX code */
      Instance1 = fetch_4th (HelloWorld);
      Instance2 = fetch_4th (HelloWorld);
                       /* execute both instances */
      (void) exec_4th (Instance1, 0, NULL, 0);
      (void) exec_4th (Instance2, 0, NULL, 0);
                       /* free first instance */
      free_4th (Instance1);
                       /* execute and free second instance */
      (void) exec_4th (Instance2, 0, NULL, 0);
      free_4th (Instance2);
                       /* reinstall first instance and execute */
      Instance1 = fetch_4th (HelloWorld);
      (void) exec_4th (Instance1, 0, NULL, 0);
      free_4th (Instance1);

      return (EXIT_SUCCESS);
}
```

The combination of different pieces of HX code is possible too. This code contains two pieces of HX code. The first one adds up two numbers, the second one divides two numbers. Both return the result of the calculation to the variable "Result":

```
#include <stdlib.h>
#include <stdio.h>
#include "4th.h"

/* app dup @ swap cell+ @ + out ! */

static unit Addition [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x62', '\x03', '\x08', '\x02', '\x09', '\x08', '\x08', '\x08', '\x08',
  '\x32', '\x02', '\x0a', '\x11', '\x07', '\x10', '\x33', '\x10', '\x07',
  '\x0b', '\x32', '\x02', '\x07', '\x08', '\xcd' };

/* app dup @ swap cell+ @ / out ! */
```

```
static unit Division [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x62', '\x03', '\x08', '\x02', '\x09', '\x08', '\x08', '\x08', '\x08',
  '\x32', '\x02', '\x0a', '\x11', '\x07', '\x10', '\x33', '\x10', '\x07',
  '\x0e', '\x32', '\x02', '\x07', '\x08', '\xc8'
};

int main (int argc, char** argv)
{
      Hcode*      Instance;
      cell        Result;
                        /* load addition HX code */
      Instance = fetch_4th (Addition);
                        /* execute: add 5 to 7 */
      Result = exec_4th (Instance, 0, NULL, 2, (cell) 5, (cell) 7);
                        /* free instance */
      free_4th (Instance);

                        /* load division HX code */
      Instance = fetch_4th (Division);
                        /* execute: div Result by 6 */
      Result = exec_4th (Instance, 0, NULL, 2, Result, (cell) 6);
                        /* free instance */
      free_4th (Instance);
                        /* print Result and exit */
      printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCESS);
}
```

There are no restrictions whatsoever to the use of the rest of the 4tH API, since `fetch_4th()` returns an ordinary Hcode pointer. For instance, you can still use `load_4th()` to load additional HX-files. Happy embedding!

26.18 Suspended execution

People are wondering how they can enable hibernation[4]. Well, you can't. Only a 4tH programmer can do that by using the word 'PAUSE'. Normally, 4tH closes all files, releases the runtime environment and exits. When 'PAUSE' is encountered, 4tH creates a stackframe, closes all files and exits. *All* API functions recognize a dormant VM and act accordingly, so there is not much you can do. You can recognize a dormant VM by examining the "Offset" member of the Hcode structure. If is non-zero, you got a dormant VM at your hands:

```
Object->Offset
```

Still, there is a lot you can do with a dormant VM as long as you have created special provisions in your 4tH program. Take this very simple interpreter:

```
include lib/interprt.4th

\ The words supported by the interpreter
: bye ." ZZzzzz.." cr pause ." Waky, waky!" cr ;
: test ." Test successfully executed!" cr ;
: _+ + ;
: _- - ;
: _* * ;
: _/ / ;
: _.( [char] ) parse type ;
```

[4]Also referred to as 'hibernation', 'sleeping VMs' or 'dormant VMs'.

```
: _cr cr ;
: _. . ;

\ The dictionary of the interpreter
create wordlist
  ," bye"      ' bye ,
  ," test"     ' test ,
  ," +"        ' _+ ,
  ," -"        ' _- ,
  ," *"        ' _* ,
  ," /"        ' _/ ,
  ," .("       ' _.( ,
  ," cr"       ' _cr ,
  ," ."        ' _. ,
  NULL ,

wordlist to dictionary

\ The interpreter itself
: go ['] interpret catch if ." Oops!" cr then ;
: prompt ." OK" cr refill drop go ;
: script 1 args tib place 0 >in ! go bye ;
: run begin argn 2 = if script else prompt then again ;

run
```

Note you can only *suspend* this program. When you provide a commandline argument, it will interpret it as a script and execute it. That is neat! Now take a look at this C program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

#define HX "tiny.hx"

/*
This function starts an interactive session
*/

void Prompt (Hcode *Program)
{
  puts ("Keyboard control enabled..\n");
  (void) exec_4th (Program, 0, NULL, 0);
  puts ("\nHost control enabled..");
}


/*
This function builds an argument list and starts a script
*/

void Script (Hcode *Program, char *script)
{
  char *(Args [3]);                    /* mimics argv[][] */

  Args [0] = HX;                       /* the program name */
  Args [1] = script;                   /* the script itself */
  Args [2] = NULL;                     /* the list terminator */
  puts ("Script control enabled..\n");
  (void) exec_4th (Program, 2, Args, 0);
  puts ("\nHost control enabled..");
}


/*
Host program, which calls the shots
```

```
*/

int main (int argc, char** argv)
{
  Hcode *Program;
  Program = load_4th (HX);
  if (Program)                          /* if loading was successful */
     {
       Prompt (Program);                /* interactive session */
       Script (Program, ".( This is a test) cr test test test");
       Prompt (Program);                /* interactive session */
       Script (Program, ".( Leaving an item on the stack) cr 23 45 +");
       Prompt (Program);                /* interactive session */

       puts ("Host shutting down..");
       free_4th (Program);              /* free resources */
     }

  return (EXIT_SUCCESS);
}
```

There is a function that supplies arguments and executes a script and a function that does not supply arguments and enters interactive mode. In `main()` we call these functions alternately. It is a pretty mean program! You can allow a user to do what he wants and when he relinguishes control, you can execute whatever you want from your C program. It is so mean that even when the user enters something like:

```
11 13 * 4 + bye .( I still gotta do this!) cr
```

The part after 'BYE' will *still* be executed before the interpreter starts executing a script. Whatever is on the stack *stays* on the stack, no matter if it was left there by a script or an interactive session. Now *that* is powerful! But there are more neat things you can do with suspended execution. You can also use it to read or write 4tH data. That may seem a bit tricky at first, but as a matter of fact it is very easy. Take a look at this example:

```
10 constant NUMCELLS                    \ array size
NUMCELLS array  Xarray                  \ array to be exported
32 string Xstring                       \ string to be exported

: export out ! pause ;                  \ save literal and sleep

: run
  Xarray export                         \ export Xarray
  Xarray 10 bounds do i ? loop cr       \ show array contents
  s" This comes straight from 4tH!" Xstring place
  Xstring export                        \ export Xstring
  ." Famous last words.." cr            \ final message
;

run
```

You probably remember that the contents of OUT are returned by `exec_4th()`, so what 'EXPORT' actually does is saving an address of a variable before returning control to the C program. 'EXPORT' is called almost immediately in this example. Obviously something has been done, since the contents of 'XARRAY' are dumped. Then a string variable is initialized and 'EXPORT' is called again. Finally a message is printed. Doesn't that make you wonder what the C program does?

```
#include "4th.h"
#include "cmds_4th.h"
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MK_CP(a,b) ((a)->CellSeg + STACKSIZ + SYS4TH + (b))
#define MK_UP(a,b) ((a)->UnitSeg + (b))

#define NUMCELLS 10
#define HX "bulk.hx"

int main (int argc, char** argv)
{
  cell   Pointer;
  cell   Carray [] = { 0L, 10L, 20L, 30L, 40L, 50L, 60L, 70L, 80L, 90L };
  Hcode *Program;
  cell  *p;
  int    x;

  Program = load_4th (HX);
  if (Program)                          /* if loading was successful */
     {                                  /* fill the 4tH array */
        Pointer = exec_4th (Program, 0, NULL, 0);
        p = MK_CP (Program, Pointer);
        for (x = 0; x < NUMCELLS; x++, p++) *p = Carray [x];
                                        /* show array and setup string */
        Pointer = exec_4th (Program, 0, NULL, 0);
        puts (MK_UP (Program, Pointer));
                                        /* show famous last words */
        Pointer = exec_4th (Program, 0, NULL, 0);
        free_4th (Program);             /* free hcode */
     }

  return (EXIT_SUCCESS);
}
```

The first thing you notice are the two macros, `MK_CP()` and `MK_UP()`. They have been defined to create pointers to the Integer Segment and the Character Segment. It is very easy: just call 4tH from C and export your variable of choice with 'EXPORT'. In this program, the value is stored in 'Pointer'. Note that it is *essential* that you know *exactly* what has been returned: a CELL or a UNIT.

In this example, `exec_4th()` first returns a cell, so we have to call `MK_CP()` to convert it to a pointer to a cell. After that we can transfer the contents of a C array to 4tH. The second time `exec_4th()` returns a unit, so we'll have to call `MK_UP()` to convert it to a pointer to an unit. After that we can use the pointer to print the string. Then `exec_4tH()` is called for the last time and a string is printed. Note that you don't have to let 4tH finish. You can call `free_4th()` any moment you want to.

That is still not enough for you? You'd like to see something even fancier? Well, what do you think about this tiny cooperative multitasker:

```
#include "4th.h"
#include <stdlib.h>

#define MAX_TASK 16

Hcode *Processes [MAX_TASK];           /* process slots */

/*
This routine adds a task to the process space. It returns the PID
if successful, CELL_MIN if not.
*/

cell task_4th (Hcode **Process, Hcode *Task)
```

```
{
  cell x;

  if (Task)                             /* if there is a task */
    for (x = 0; x < MAX_TASK; x++)      /* search all process slots */
      if (Process [x] == NULL)          /* if empty slot found */
        {
          Process [x] = Task;           /* add the task */
          return (x);                   /* and return success */
        }

  return (CELL_MIN);                    /* if not, return failure */
}


/*
This routine searches the process space for a given task. If found, it is
executed and the return value returned to the calling process. If not found,
it signals termination.
*/

cell wake_4th (Hcode **Process, cell task)
{
  cell x, y;

  for (x = task; x < MAX_TASK; x++)     /* search slots beginning with task */
    if (Process [x])                    /* if it is an active process */
      {                                 /* execute it */
        y = exec_4th (Process [x], 0, NULL, 1, x);
                                        /* if the process has terminated */
        if (Process [x]->Offset == 0)
          {                             /* free the process */
            free_4th (Process [x]);     /* and set the pointer to NULL */
            Process [x] = NULL;
          }
                                        /* return next process number */
        return (y == CELL_MIN ? ++x : y);
      }
                                        /* signal no active processes found */
  return (task == 0 ? CELL_MIN : MAX_TASK);
}


/*
This is the true multitasker. It keeps on looping through the processes until
it receives a kill signal from wake_4th().
*/

void multi_4th (Hcode **Process)
{
  cell pid = 0L;                        /* process id */

  while (pid >= 0L)                     /* seach all process slots */
    {
      pid = wake_4th (Process, pid);    /* now wake this process */
      if (pid >= MAX_TASK) pid = 0;     /* and loop around */
    }
}


/*
Host program, which calls the shots
*/

int main (int argc, char** argv)
{
  cell x;
```

```
                                        /* set all slots to NULL */
  for (x = 0; x < MAX_TASK; x++) Processes [x] = NULL;
                                        /* now load two processes */
  printf ("Process %ld installed\n", task_4th (Processes, load_4th ("1.hx")));
  printf ("Process %ld installed\n", task_4th (Processes, load_4th ("2.hx")));

  multi_4th (Processes);                /* start the multitasker */
  return (EXIT_SUCCESS);                /* return success */
}
```

Since `load_4th()` returns a NULL pointer when it doesn't succeed, it is safe to pass it to `task_4th()`. Even if programs aren't suited to do any multitasking, you can use this program: they will just be executed consecutively. `task_4th()` will add a program to the process list. `wake_4th()` will try to wake up a program beginning with the PID which is passed to it. Note that the PID is passed to the 4tH program, so it can be queried. If 'OUT' contains a valid PID, it will be the next process that the program will try to awaken. When a process terminates, it is taken from the process list. When all processes have terminated, `wake_4th()` returns CELL_MIN. Note that a program can terminate the multitasker by returning a non-zero value. That's as fancy as it gets, folks!

The String Segment and the Code Segment are *read-only* for a reason. Although you *can* access them, we advise you *not to attempt it*. The same goes for the Stack Area and the System Area. The interfaces we've provided here are a relatively safe way to exchange information between 4tH and C, but if you make any errors your program might crash. Therefore, it is a good idea to add exception handlers to critical sections of your 4tH program.

26.19 Useful variables

We've already seen that `dump_4th()` can provide you with a lot of information about Hcode. If you need this information, you don't have to call `dump_4th()`. The `dump_4th()` function simply uses the information that is already available. This small program shows you how to obtain it:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      printf ("Error#       : %u\n", Program->ErrNo);
      printf ("Error at word: %d\n", Program->ErrLine);
      printf ("Object size  : %d\n", Program->CodeSiz);
      printf ("String size  : %u\n", Program->StringSiz);
      printf ("Var. offset  : %u\n", Program->Offset);
      printf ("Variables    : %u\n", Program->Variables);
      printf ("Strings      : %u\n", Program->Strings);
      printf ("Reliable     : %s\n", Program->Reliable ? "Yes" : "No");
}
```

The labels are kept the same as in `dump_4th()`, so if you need more information, read that section again.

Chapter 27

Modifying 4tH

27.1 Introduction

A good scripting language must be easy extensible. We will cover the most common extensions. You will acquire in-depth knowledge of the inner workings of 4tH. All of 4tHs functions are a toolkit in itself and can be put to your own use (especially `comp_4th()`, which is 4tHs most complex function).

As we proceed, there will be more files to edit and the modifications will get more complex. Be sure you mastered the previous extensions, before you get on with the more elaborate ones. A good knowledge of C is required for most operations.

You will also note that several modules, e.g. the parser and the peephole optimizer, are not covered by this manual. First, because by design they are usually not affected by any changes to the language itself and second, they are quite complex - and consequently better left alone unless you *profoundly* understand what you're doing.

27.2 Understanding 4tHs versioning

4tHs version is made up of three components:

1. *Major version*; this indicates the 4tH architecture number. It is incremented only if a new architecture is applied or when the minor version reaches "99".
2. *Minor version*; this indicates the 4tH bytecode number. It is incremented only if the bytecode changes or when a significant feature is added to 4tH. It is reset after an architecture change.
3. *Release*; this indicates the release number. It is incremented with every release within a bytecode version and reset after a bytecode change.

The combination of *major version* and *minor version* is called the *bytecode version*. As long as the bytecode version remains the same, the bytecode is guaranteed to work with all releases within a bytecode version. A bytecode change happens in the following cases:

- A bytecode is added or removed;
- The bytecode numbering (order) is changed;

- The behavior of a bytecode changes significantly (e.g. different input- or output parameters).

If you *change* the bytecode and *fail* to change the bytecode versioning you're in fact *disabling* one of the major security mechanisms of 4tH, since it is unable to determine the compatibility of the bytecode. You may end up with a *very* unstable 4tH and some hard to track bugs, so that is definitely *not* what you want. Versioning is controlled by two `#defines` in `cmds_4th.h`:

```
/* header */
#define Version4th   0x362
#define App4th       0L
```

`Version4th` is the bytecode version. Usually, you want to leave that one alone in order to see on which version of 4tH your modified 4tH is based. For that reason `App4th` was added. Standard 4tH *always* comes with `App4th` set to zero. `App4th` sets the application byte and is also part of an HX file[1]. If the application byte of an HX file differs from the one of the virtual machine, the HX code is rejected. Note that the files generated by `cgen_4th()` do not have *any* protection mechanism[2], so the best thing is to regenerate them with every compile.

In short, if you want to experiment with your own modified 4tH, *be sure* to change the application byte when applicable.

27.3 A closer look at comp_4th()

As we already know, `comp_4th()` compiles source to H-code. First of all, we need to have a source. This is a simple character array, which is pointed to by "Source". Then an H-code header is created.

The header is initialized by `InitObject()`, which calls `ParseText()` to get the initial size of the Code Segment. The initial size of the Code Segment is stored in the header-member "CodeSiz".

`ParseText()` calls two other functions: `ParseDirectives()`, which picks out all directives and calls `ParseStrings()` if need be, which parses the source for string-arguments.

If `ParseStrings()` encounters a '[NEEDS' or 'INCLUDE' directive, it will call `DoNeeds()`, which will create enough room for the file to be included and read the actual file. If `DoNeeds()` fails, it will set the "ErrNo" member of the header accordingly and exit. `MakeRoom()` just moves the last part of the source to the end of the reallocated space. Since all variables are adjusted accordingly, `ParseText()` will pick up parsing after the '[NEEDS' or 'INCLUDE' directive. It never knows the difference.

After parsing `ParseText()` returns the initial size of the Code Segment (the number of words). The function `ParseText()` sets three important variables:

VARIABLE	CONTENT
SourceStrings	The number of source-words
SourceWords	The size of the Code Segment
SourceSymbols	The size of the symbol-table

Table 27.1: comp_4th() variables

[1] See section 26.5.
[2] see section 26.13.

It is imperative to know these numbers since the 4tH-environment has to size its resources. No resizing is required until all compilation is done. All allocated resources should be large enough to contain the resulting compilant, so extending these resources should never be necessary. In the end, they are only shrunk to their actual sizes. Now we can start to allocate the compiler-resources:

- Code Segment
- Symboltable
- Controlstack

This is done by simply calling `AllocResource()`. Note that it is not necessary to allocate the String Segment. Strings remain in memory already allocated to the source and are just shifted to the front.

Then compilation can begin. First, the variable "Cursor" is set to the beginning of the source. Then every call to `GetNextWord()` sets the variable "CurrentWord" to point to the next source-word. If there are no more source-words, "CurrentWord" is set to NULL and compilation is terminated.

Now compilation really gets off. It is important to know that not all words are created equal. There are five kinds of words:

- Immediate words
- Words
- Constants
- Symbols
- Numbers

So, there are five distinct functions which handle these words:

- Immediate words are compiled by `GetImmediate()`
- Words are compiled by `GetWord()`
- Constants are compiled by `GetConstant()`
- Symbols are compiled by `GetSymbol()`
- Numbers are compiled by `comp_4th()`

If the first four functions fail there is one more chance that this is a valid source-word: it might be a number. So, the source-word is converted to a number in the current radix. If this works, the number is compiled. If it doesn't, it isn't a valid source-word and the member "ErrNo" is set.

Compiling is done by a single function called `CompileWord()`. You just provide the token and its argument and `CompileWord()` takes care of the rest.

When all compiling is done, we can discard the symboltable and the controlstack. This is done by calling the function `FreeResource()`. It is called by `ReallocSegs()`, which shrinks the Code Segment and the String Segment to their actual sizes, and `AbortCompile()`, which shuts the compiler down in case of an error.

If an error occurred before any compiling took place `AbortCompile()` also discards the Code Segment and the String Segment, thus returning a bare H-code header. As you already know, the error-code is stored in the member "ErrNo" of the H-code header.

If the error occurred after words have been compiled, this partial compilant is not discarded. Instead the member "Reliable" is set to FALSE, indicating that the compilant cannot be run or saved. It can be decompiled, thus enabling the user to track the error.

In the next sections we will take a closer look at the three main tables in `comp_4th()`, which contain all of 4tHs built-in words.

27.4 Adding a constant

Adding a constant is very easy. You only have to update a single table in `comp_4th()`. The table with constants, which is embedded in `GetConstant()` has four members:

1. Length byte
2. Name
3. Type
4. Value

The length-byte is used to quickly scan the table. All words are skipped until it reaches the constants with the same length. Then it starts to compare the names.

What happens then depends on the argument "mode". When it equals W_EXEC, the constant is compiled (a literal with the value as argument). Otherwise, only the index in the table is returned, pointing to the constant we just found. This enables mere searches in the table. When a name isn't found at all, it returns MISSING.

Say, we want to add a constant called "TWENTY". At least, we know its name: "TWENTY". The name "TWENTY" is six characters long. And of course, we want to compile the number "20" each time it is referenced in the source. To make `comp_4th()` compile a number, we need the token LITERAL. The four members are:

1. Length byte = 6
2. Name = "TWENTY"
3. Type = LITERAL
4. Value = 20

Since every constant is a signed 32 bits number, we add the modifier 'L' to the "20". So the complete line we have to add to the table reads:

```
{ 6, "TWENTY", LITERAL, 20L },
```

Now we have to insert this line into the table. Note that constants with the same length have to be grouped together:

```
{ 5, "INPUT", LITERAL, F4_READ },
{ 6, "OUTPUT", LITERAL, F4_WRITE },
{ 6, "APPEND", LITERAL, F4_APPND },
{ 7, "(ERROR)", LITERAL, CELL_MIN },
```

We decide to put our constant behind "APPEND". Of course, we could have put it behind "INPUT" or "OUTPUT" as well:

```
{ 5, "INPUT", LITERAL, F4_READ },
{ 6, "OUTPUT", LITERAL, F4_WRITE },
{ 6, "APPEND", LITERAL, F4_APPND },
{ 6, "TWENTY", LITERAL, 20L },
{ 7, "(ERROR)", LITERAL, CELL_MIN },
```

Now recompile 4tH and run this simple program:

```
twenty . cr
```

It should compile without errors and print "20". That's all there is to it! Not to difficult to begin with, huh?

27.5 Adding a word

Now for something a little more difficult. Let's say we want to implement 'NIP'. Of course, 'NIP' is already available, but if you compile this program:

```
1 2 nip
```

You will see that it actually compiles to:

```
[0]   literal     (1)
[1]   literal     (2)
[2]   swap        (0)
[3]   drop        (0)
```

So, 'NIP' is actually expanded to 'SWAP' and 'DROP'. That is because 'NIP' can be defined as:

```
: nip swap drop ;       ( n1 n2 -- n2)
```

It removes the number under the top of stack. We call this an inline macro, which we will discuss later on. If you really want to try out 'NIP' in the following example, you have to remove a line from `ImmedList[]`:

```
{ 3, 0, 1, "NIP", "", DoNip },
```

Your compiler might complain about unused functions, but it will work.

Inline macros are not the only way to add 'NIP' to 4tH. We can also implement 'NIP' as a word. Most words can be found in the function `GetWord()`. There is a single table, which is laid out like the table of constants we encountered in the previous section, so:

1. Length byte = 3
2. Name = "NIP"

But instead of the value of the constant, we have to add something else. That something is called the token. The tokens are defined in `cmds_4th.h`. Let's have a look:

```
#define PAUSE    100
#define VECTOR   101
#define ENVIRON  102
#define PLITERAL 103
#define FSEEK    104
#define FTELL    105

/* ranges */
#define LastWord4th  FTELL
#define LastMsg4th   M4CABORT
```

Well, in fact you can place a new token anywhere, but then you have to renumber all other tokens. The easiest way is to place it after the last token you defined, which in this case is "FTELL". The token "FTELL" has the number "105". Well, the next number is "106" and that is the number "NIP" is going to get. So we add:

```
#define PAUSE    100
#define VECTOR   101
#define ENVIRON  102
#define PLITERAL 103
#define FSEEK    104
#define FTELL    105
#define NIP      106

/* ranges */
#define LastWord4th  FTELL
#define LastMsg4th   M4CABORT
```

What part don't you understand? But we're not ready yet. If you look at the #define below the entry we just made, you will see that it says that the last word in 4tH is "FTELL". That is incorrect now. We've just added "NIP". Fixing it is very easy. Just change the line to:

```
#define LastWord4tH     NIP
```

We're done now with `cmds_4th.h`. Now we have change `comp_4th.c`, so that the compiler can recognize and compile "NIP". Now we can complete the entry for `GetWord()`:

1. Length byte = 3
2. Name = "NIP"
3. Token = NIP

So the complete entry reads:

```
{ 3, "NIP", NIP },
```

And like we did with our constant "TWENTY", we have to give it a place inside the table between all other 3-letter words. We decided that pasting "NIP" between "USE" and "SEEK" would be a good idea (but there are plenty of other places too):

```
{ 3, "HEX", HEX },
{ 3, "USE", USE },
{ 3, "NIP", NIP },
{ 4, "SEEK", FSEEK },
{ 4, "TELL", FTELL },
{ 3, "HEX", HEX },
```

Are we done now? No. The compiler will recognize and compile "NIP", but what does it do? That behavior will have to be defined in `exec_4th.c`, but we'll discuss that in the next section.

27.6 A closer look at exec_4th()

Since 4tH has a segmented structure, there are special words for each segment, e.g. "C!" for the Character Segment and "!" for the Variable Area. But when one wants a Virtual Machine that checks every access, the parameters of these words need to be checked.

There are very few words that access the Variable Area, so these are checked within the code for the token itself. Others, like the ones accessing the data stack, are used more often. So, special functions were created that allow or check access to those areas. There are thirteen functions you should know about. They form the basic API.

FUNCTION/MACRO	DESCRIPTION
DPOP	Gets an item from the data stack
DPUSH (cell)	Puts an item on the data stack
DFREE (cell)	Checks amount of free space on the data stack
DSIZE (cell)	Checks the number of items on the data stack
DS (cell)	Random acces item on data stack
RPOP	Gets an item from the return stack
RPUSH (cell)	Puts an item on the return stack
RFREE (cell)	Checks amount of free space on the return stack
RSIZE (cell)	Checks the number of items on the return stack
RS (cell)	Random acces item on return stack
RANGE (cell, cell)	Checks whether the range addr/count is within the bounds of the Character Segment
unit fetch (cell)	Gets a character from the Character Segment
void store (cell, unit)	Puts a character in the Character Segment
cell toPAD (char*)	Puts a string in the PAD
char* toCstring (cell, cell)	Puts a addr/count string in the PAD

Table 27.2: exec_4th() basic API

We strongly recommend you use these functions when accessing any of these segments. But you'll probably need more than functions to create your code. What about variables?

Well, of course there is a host of variables you can use, but there are three variables that are used more frequently. They are called "a", "b" and "c" and are of type cell. Now, what did NIP do?

```
NIP   ( n1 n2 -- n2)
```

That means the first cell is taken from the data stack and saved, then the second cell is taken from the data stack and dropped and finally the first cell is replaced on the data stack. Note there need to be at least two items on the stack to make it work. Since there will be one item less on the stack after the operation, we don't need to check whether there is enough space. So this will do the trick:

```
DSIZE (2);
a = DPOP;
DDROP;
DPUSH (a);
```

There is even a faster way to do it. This implementation uses the `DS()` macro which allows direct access to the datastack:

```
DSIZE (2);
DS (2) = DS (1);
DDROP;
```

`DS(1)` equals ”Top of stack” and `DS(2)` equals ”Second of stack”. Note that the `DS()` macro *doesn’t adjust the stack pointer*. That is where DDROP comes in. It discards the superfluous item on the stack. Now we add a label and a "break" (which is necessary in a `switch()`):

```
case (NIP): DSIZE (2);
            DS (2) = DS (1);
            DDROP;
            break;
```

Although this is slightly more difficult than the previous API, this implementation is much, much faster. So, now we’re finally done! We can compile the whole thing and test our new command:

```
 2 3 NIP . cr
```

Can we? No, there is one more thing we have to do. Your source will compile OK and execute OK, but it is nowhere to be found when we decompile it. Does that mean we have to edit `dump_4th()`? Wrong again, but we will see that in the following section.

Before we leave, we’ll take a look at another word, OVER. This once differs from NIP, since it leaves more items than it consumes. If we wouldn’t take any precautions, we might run into serious trouble. So, what does OVER do?

```
OVER   ( n1 n2 -- n1 n2 n1)
```

’OVER’ requires two items and leaves three items on the stack. That means we need 3 - 2 = 1 extra item on the stack:

```
DSIZE (2);
DFREE (1);
a = DS (2);
DPUSH (a);
```

Note we *must* use `DPUSH()` to adjust the stack pointer. Of course, there are macros that manipulate the return stack in a similar way. One tip: *be careful with nesting macros*, since you might not get what you want.

Strings are quite another ballgame. In 4tH these are usually address/count strings and consequently incompatible with C, since they are not terminated. Strings in the String Segment are not accessible by a 4tH programmer at all. In order to solve these problems, the PAD was created. The PAD is essentially a circular string buffer where temporary strings are stored. The address and count are usually returned to the 4tH programmer enabling him to manipulate these strings. At the same time you can be assured that they are terminated and C-compatible.

There are two API functions which allow you to access the PAD, `toPAD()` and `toCstring()`. The first one allows you to copy a C string to the PAD. It puts the address on the stack and returns the count. The second one allows you to copy an address/count string to the PAD. It returns a pointer to a C string. In `exec_4th()` there is a C string pointer which you can use, `p`. A little example: suppose there is an address/count string on the stack which you need to access:

```
DSIZE (2);
a = DPOP;
b = DPOP;
p = toCstring (b, a);
```

Note that the count resides on the top of the stack and is popped first. The address comes after that. You can use `toPAD()` to copy any C string to the PAD, e.g.

```
DFREE (2);
DPUSH (toPAD ("Hello world!"));
```

Note that `toPAD()` already left the address on the stack, so you only need to push the count it returns.

27.7 A first look at name_4th()

May be `name_4th.c` looks like a function, but it definitely isn't. It is a global array that is as global as globals can get. You can refer to it in any program that uses the 4tH library without defining it first. It contains the names of all the tokens, so when we decompile, the tokens get a readable name:

```
#include "4th.h"
#include "cmds_4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      puts (name_4th [SWAP]);
      return (EXIT_SUCCESS);
}
```

This program will print "swap". Yes, you can use the token-code as an index to this array. We can't make it any easier than that! Now you understand that we can't do the same thing for "NIP", since we didn't add that one. In fact, any program trying it will either print garbage or crash. Let's take a look at the last few lines of `name_4th[]`:

```
  "environ", "+literal",  "seek",    "tell"
};
```

Yeah, we've seen that before. All entries are ordered in the same way as in `cmds_4th.h`. Since "NIP" was added at the end of `cmds_4th.h`, we have to do the very same thing here. Don't forget the comma after "TELL"

```
  "environ", "+literal",  "seek",    "tell",      "nip"
};
```

Now we are finally done. We can recompile the library (and the compiler) and we can use "NIP" like any other 4tH word.

27.8 Extending the compiler

So far we've only added words that are directly compiled into a token/argument pair. If that was the way all 4tH words worked, we would never have branches, variables or other things that make up a language. In fact, you would never construct this complex architecture, since there are easier ways to achieve the same functionality.

The secret lies in the last major function of `comp_4th()` we have discussed, which is called `GetImmediate()`. When you look at it for the first time, it looks quite like `GetConstant()` and `GetWord()`. The associated table `ImmedList[]` has a length-byte and a name but instead of a token or a value the last field is a pointer to a function.

You'll find these functions above `GetImmediate()` and they all start with `Do..()`. In fact, they are the icing on the cake. They make 4tH a compiler, since they allow non-linear compiling. That includes:

- Branching
- Comments
- Allocation of variables
- String handling
- Assertions
- Constants

If there is something you want to do that you cannot define in a single token/argument, this is the place where you have to be. But before you are starting to make new functions, note there are many functions that you can use.

E.g. when you want to compile a word, you don't have to bother yourself with error-checking or other 4tH-internals. Just make a call to `CompileWord()`:

```
CompileWord (NIP, 0);
```

Extending the compiler is quite easy. We will illustrate that by using a very simple example. We have some compiler-words that handle the radix at compile time. They are:

```
[BINARY]
[OCTAL]
[DECIMAL]
[HEX]
```

Now we want to add a new one called "[SEXTAL]", which sets the radix to six. The radix at compile-time is handled by a single variable called "Base". First we have to make a function, which sets "Base". We call it `DoSextal()`. Note that this function cannot receive or return any values, like all `Do..()` functions:

```
#ifndef ARCHAIC
      static void DoSextal (void)
#else
      static void DoSextal ()
#endif

{
      Base = 6;
}
```

BRANCHING	
MarkLink()	Adds a link to the controlstack
MakeLink()	Makes a back-link
PairLink()	Retrieves a link from the controlstack
CompileMark()	Compiles a token and adds a link to the controlstack
SYMBOLTABLE	
AddSymbol()	Adds a symbol to the symboltable
MakeSymbol()	Adds the current word as a symbol to the symboltable
GetSymbol()	Retrieves an entry from the symboltable
PARSING	
GetNextWord()	Gets next word in the source
DecodeSymbol()	Decodes a symbol from source
DecodeLiteral()	Gets a compiled literal expression
DecodeOperand()	Gets a compiled expression
DecodeWord()	Gets a name from source
DecodeName()	Gets a previously delared name from source
SkipSource()	Discards all source between two labels
COMPILING	
CompileWord()	Compiles a token and its argument
InlineWords()	Compiles a sequence of tokens without arguments
STRINGS	
MoveString()	Moves a string inside the String Segment
CompileString()	Compiles a token and its associated string

Table 27.3: comp_4th() basic API

Just place it anywhere before `ImmedList[]` and you're safe. Now we have to make it work. Like the other `Get..()` functions, there is a table called `ImmedList[]` which drives this behaviour, so let's get started:

1. Length byte = 8
2. Symboltable entries = 0[3]
3. Additional tokens = -1[4]
4. Delimiter = ""[5]
5. Name = "[SEXTAL]"
6. Function = DoSextal

Now let's update that table:

```
{ 7,  0, -1, "ALIGNED",     "",    DoDummy },
{ 7,  0,  0, "RECURSE",     "",    DoRecurse },
{ 8,  0, -1, "[SEXTAL]",    "",    DoSextal },
{ 8,  1, -3, "CONSTANT",    "",    DoConstant },
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
```

You're done now! Recompile the program and "[SEXTAL]" has become a part of 4tH. Note that not all words within `ImmedList[]` can be defined that easily. If special tokens are required, you might have to edit other files as well.

[3]See section 27.22
[4]See section 27.15
[5]See section 27.17

27.9 Making aliases

Sometimes you need two different names that do the same thing. Well-known examples are "CHAR" and "[CHAR]" or "I" and "R@". How can this be done?

In fact, it is very simple. Think about it. 4tHs vocabulary is stored in tables. These tables link a name with some kind of behavior. So we have to make two different names that are linked to the same thing. Take a look at this excerpt of `ImmedList[]`:

```
{ 5,  0, -1, "ALIGN",       "",    DoDummy },
{ 5,  0, -1, "CELLS",       "",    DoDummy },
{ 5,  0, -1, "CHARS",       "",    DoDummy },
{ 5,  1, -2, "TABLE",       "",    DoCreate },
{ 6,  1, -2, "CREATE",      "",    DoCreate },
{ 6,  0, -1, "[THEN]",      "",    DoDummy },
```

You might have noticed a few aliases.

WORD	COMPILED BY
[THEN], ALIGN, CELLS, CHARS	DoDummy()
TABLE, CREATE	DoCreate()

Table 27.4: Examples of aliases

That means that if you write your 4tH program you can choose between "TABLE" and "CREATE". It doesn't matter, it will compile to the same thing. But we have to add to that some aliases are created because in Forth they *do* have different meanings, like "[CHAR]" and "CHAR". Read the glossary for details.

We will show you another example. This one comes from `GetWord()`:

```
{ 5, "CHAR+", INC },
{ 5, "CHAR-", DEC },
{ 5, "CELL+", INC },
{ 5, "CELL-", DEC },
```

You see that "CHAR+" and "CELL+" do two very different things. At least in Forth. Within 4tH the smallest addressunit is always an element of that particular segment, thus one. So in 4tH these words are aliases and will compile to the very same code.

27.10 Giving a name to an application variable

We already learned that you can transfer variables to 4tH:

```
Result = exec_4th (Program, 0, NULL, 12, (cell) 31, february,
(cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell) 31, (cell) 31,
(cell) 30, (cell) 31, (cell) 30, (cell) 31);
```

These kind of variables are called "application variables". Of course, you don't *have* to use the same variables every time you call `exec_4th()`, but if you do it may be a good idea to give them a significant name. That makes it a lot easier for a 4tH application programmer to reference your variables. Like everything in 4tH, that is very easy too.

If we take a look at `cmds_4th.h` you will see a C-constant named "VAR4TH". This constant has two functions. First, it shows how many internal 4tH variables there are. Second, it is an index to the first application variable, so 'APP' is defined as "VAR4TH". That means that:

```
app 0 th
```

Is the very first application variable and:

```
app 1 th
```

Is the second application variable. You can do the same. Let's say you have three application variables, which contain the document-number, the page-number and the line-number. You'd like to call them "&DOC", "&PAGE" and "&LINE". The ampersands are not really necessary, but we add them in order to identify the application specific words. To make it work you have to call `exec_4th()` by:

```
Result = exec_4th (Program, 0, NULL, 3, (cell) Doc, (cell) Page,
(cell) Line);
```

Now these are the mappings.

C VARIABLE	4TH EXPRESSION
Doc	app 0 th
Page	app 1 th
Line	app 2 th

Table 27.5: Mapping between 4tH and C variables

Now all we have to do is add constants that are equivalent to these addresses. As we've seen before, we can do that by modifying `comp_4th()`. That is `GetConstant()` to be exact:

```
{ 4, "&DOC",  LITERAL, VAR4TH+0 },
{ 5, "&PAGE", LITERAL, VAR4TH+1 },
{ 5, "&LINE", LITERAL, VAR4TH+2 },
```

That's all! You can now refer to these variables with their proper names.

C VARIABLE	4TH EXPRESSION	4TH VARIABLE
Doc	app 0 th	&doc
Page	app 1 th	&page
Line	app 2 th	&line

Table 27.6: Mapping between 4tH and C variable names

Note that if you use this technique you are bound to calling `exec_4th()` with these arguments in this order! Failure to do so may cause unpredictable results (but no crashes of course).

27.11 Adding new variables

In standard 4tH there are five environment variables, HI, FIRST, LAST, CIN and COUT. There are also five predefined variables, '>IN', 'BASE', 'OUT and the variable pair 'SOURCE'. These variables are initialized by `exec_4th()`, so their initial value should be known by then.

Adding new variables is not difficult. We're going to make a variable that contains 4tHs release number, called "VERSION". First take a look at `cmds_4th.h`. It contains a #define called "VAR4TH":

```
/* variables and environs */
#define SYS4TH   3
#define VAR4TH  10
#define ENV4TH   5
```

Now remember that number behind "VAR4TH". You will need it later. Then increment it:

```
/* variables and environs */
#define SYS4TH   3
#define VAR4TH  11
#define ENV4TH   5
```

If you would have preferred to make 'VERSION' an environment, you should also have incremented "ENV4TH". But we assume you'll allow the variable to be overwritten. Now add a symbolic value for the variable. Just append it to the list and increment the number:

```
#define VBASE    5
#define VIN      6
#define VOUT     7
#define VTIB     8
#define VTIBS    9
#define VVERS    10
```

Or if you prefer it to make an environment variable, add it to the environment variable list:

```
#define VHI      0
#define VFIRST   1
#define VLAST    2
#define VCIN     3
#define VCOUT    4
#define VVERS    5

#define VBASE    6
#define VIN      7
#define VOUT     8
#define VTIB     9
#define VTIBS   10
```

That's all. Now save `cmds_4th.h` and load `comp_4th()` in your editor. This stage is very much like adding a name to an application variable. We simply define a constant that contains the address of our new internal variable. You will remember how we add a constant. Right, we add an entry to the `GetConstant()` table:

```
{ 7, "VERSION", LITERAL, VVERS },
```

Making it an environment variable is very easy too: just replace the 'LITERAL' token by an 'ENVIRON' token:

```
{ 7, "VERSION", ENVIRON, VVERS },
```

All we need to do now is to initialize the variable in `exec_4th()`. Since it is a variable, it resides in the Variable Area of the Integer Segment. The Integer Segment is just a large array of unsigned longs.

The pointer "Stack" points to the beginning of the Integer Segment, which is also the beginning of the Stack Area. The pointer "Vars" points to the area that is assigned to 4tHs variables. Our constant "VERSION" is an index to that array, so the expression "Vars [VVERS]" is a valid reference to our "VERSION" variable.

However, this indexed way of referencing is slower than a pointer. Therefore, we have created pointers that reference these frequently used variables:

```
cell        *In;                /* equivalent of forth >IN */
cell        *Result;            /* return value for apps */

Base        = &(Vars [VBASE]);  /* assign pointer to BASE */
In          = &(Vars [VIN]);    /* assign pointer to >IN */
```

You might have noticed the absence of "Base". Well, since it is referenced elsewhere as well, this is a global variable. But don't worry, there is no need to reference "VERSION" globally. So, we need to define a pointer to a cell, assign it to "Vars [3]" and initialize it:

```
Vars [VVERS] = Version4th;      /* initialize it */
```

That is all! Any questions? Where does "Version4th" come from? It is defined in `cmds_4th.h`. Anybody else? Next subject, please.

27.12 Resizing the 4tH environment

You might come up with a situation that the stack isn't big enough. Or that you want to give your programmers deeper nesting. Or that 512 characters isn't just good enough for temporary storage.

Relax! All these things can be changed with very little effort. And after that, you just need to recompile 4tH like we've done before.

There is a single file you need to edit, `cmds_4th.h`. You will find several easy to change #defines there.

```
/* compiler */
#define LINKSIZ      64
#define SYMLEN       16

/* interpreter */
#define STACKSIZ     512
#define TIBSIZ       256
#define PADSIZ       512
#define DOTSIZ        64
#define MAXDEVS        8
#define PIPEWAIT 102400L
```

You already know "VAR4TH", since we discussed that one earlier in this document. Right, it determines the number of internal variables! "LINKSIZ" determines the nesting depth. Nesting depth has to do with the number of nested branches, e.g.

```
IF
      IF
            IF
            THEN
      THEN
THEN
```

Each 'IF' puts its address and a reference (I'm an IF) on the flow control stack. Each 'THEN' takes an entry off the flow control stack and takes the appropriate action. So, in the current version of 4tH you can nest upto 64 consecutive conditionals, before you get an error. You may increase or decrease that number.

"SYMLEN" is the maximum length of any name you define, e.g. a colon-definition, a constant, a variable. The default is 16, which is enough to define a name like "multiplications".

You can define a longer name, but only the first fifteen characters will be significant. You can increase the maximum number of significant characters, but beware: this can take up a lot of memory!

"STACKSIZ" is the combined size of both data and return stack. This size will do for most applications, since it allows you a combination of high usage of the data stack and low usage of the return stack or vice versa. You might encounter a situation where recursion forces you to resize the Stack Area. Decreasing is possible too, of course, but at your own risk.

"TIBSIZ" is the size of the Terminal Input Buffer used by 'REFILL'. If you need 'REFILL' to accept longer lines than 256 characters and you don't want to allocate your own buffer, resize it.

"PADSIZ" is the size of the scratch PAD, used to store temporary strings. A part of the PAD is reserved to numbers. The size of this area is determined by "DOTSIZ". The rest of PAD (PADSIZ - DOTSIZ) is a circular string buffer. A bigger PAD will allow you to store longer temporary strings that survive longer before getting overwritten.

"MAXDEVS" is the maximum number of I/O devices that 4tH can manage. Note that two of them (STDIN, STDOUT) are already in use, so you can open up to six additional devices concurrently. Finally, " PIPEWAIT" is discussed in detail in the next section.

You will find there are other defines here too. Please, do *not* change them. That just doesn't work. In fact, 4tH just won't work properly anymore.

27.13 Tuning pipe failure detection

Pipes in 4tH are opened by the `popen()` function. This has one big disadvantage. Although `popen()` is able to detect a failed `fork()`, it is unable to detect whether the program was successfully started or not. E.g. if the program cannot be found in the path the pipe fails, although `popen()` has already reported it was successful. In some cases this can have serious consequences.

After careful study we decided to monitor the process for a while and then report success or failure. The default value works very well on most modern systems, but with some systems it may be neccesary to adjust it. This is the case when you experience one of the following symptoms:

- Opening a pipe is slow; there is a long delay before 4tH reports the pipe is successfully opened.
- 4tH reports that the pipe was successfully opened, but most of the time this was not the case.

In that case, you have to adjust a #define in "cmds_4th.h". That is a lot easier than you might think. We've developed a small program to do that. It should be portable across most Unixes:

```
#include <stdio.h>
#include <limits.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>

long TimeBadPipe (void)
```

```
{
  FILE *p;                              /* filepointer to pipe */
  long  x;                              /* simple counter */
  int   s = 0;                          /* status of child */

  p = popen ("nosuchprogram", "r");     /* perform a normal popen() */
  for (x = 0; x < INT_MAX && s == 0; x++) waitpid (-1, &s, WNOHANG);
  pclose (p);                           /* close the pipe */
  return (x);                           /* return the count */
}


int main (int argc, char **argv)
{
  int x;                                /* simple counter */
  int now;                              /* return of TimeBadPipe() */
  int sofar = 0;                        /* highest count */
  int total = 0;                        /* total of all counts */
                                        /* warn the user */
  puts ("Doing 1000 iterations, wait..");
  puts ("(This is going to be messy..)");
  sleep (5);                            /* allow him to read the message */

  for (x = 0; x < 1000; x++) {
    now = TimeBadPipe ();               /* time a bad pipe */
    if (now > sofar) sofar = now;       /* adjust sofar */
    total += now;                       /* add to total */
  }
                                        /* show the results */
  printf ("\nAverage : %d\n", total / 1000);
  printf ("Maximum : %d\n", sofar);
  return (EXIT_SUCCESS);
}
```

If you run it, it will print something like this:

```
Doing 1000 iterations, wait..
(This is going to be messy..)
sh: nosuchprogram: command not found
sh: nosuchprogram: command not found
...
sh: nosuchprogram: command not found
sh: nosuchprogram: command not found
Average : 8685
Maximum : 41235
```

This means that on average the process had to be checked 8685 times, but at no occasion a process died after it had been checked 41235 times. Run it several times, so you will get a good impression of how your particular system behaves. Ignore extremely high and extremely low values. Then take the highest value that pops up several times and change ”cmds_4th.h” appropriately:

```
#define PIPEWAIT 49152L
```

Note we rounded the value a little (we’re a binary kind of guy) and it doesn’t have to be exact. Now you can safely use pipes on your system and the result returned will assure you that the pipe was actually successful opened and ready for use.

If this still doesn’t work, you may have to adjust this #define manually: decrease it if opening a pipe is slow, increase it if 4tH incorrectly reports a successfully opened pipe. If you use MS-DOS, just forget all this. We don’t provide any pipes there.

27.14 Adding new error messages

Adding new messages is quite simple. It requires not much more than adding a #define and adding a string. You might have noticed that every 4tH message has a mnemonic. Although this is not required, it makes it much easier to read and thus maintain your code.

This mnemonic is no longer than eight characters, all uppercase and begins with "M4" (which stands for Message 4tH). Let's say you've added the ANS-Forth floating point wordset and you want to add the error message "Floating point exception". We'll do it the easy way and just append the message at the end of the table.

First we have to come up with a mnemonic. We decide to use "M4FLOATE". Now we start up our favorite editor and load `4th.h`. Then we look for the table with error mnemonics:

```
#define M4NOSTR  24
#define M4NULSTR 25
#define M4DUPNAM 26
#define M4CABORT 27
```

Now we simply add "M4FLOATE" to the end of the table. Since the last message had code 27, we give our message code 28:

```
#define M4NOSTR  24
#define M4NULSTR 25
#define M4DUPNAM 26
#define M4CABORT 27
#define M4FLOATE 28
```

We can now save `4th.h`. Now we have to add the message itself. That is done by adding it to `errs_4th.c`. That file just contains an array of messages. Note that the messages are listed in order of their codes:

```
  "Unterminated string",
  "Null string",
  "Duplicate name",
  "Compilation aborted"
};
```

If you change that order, your compiler might display the right errorcode, but the wrong error message. Since our mnemonic comes last, our message comes last:

```
  "Unterminated string",
  "Null string",
  "Duplicate name",
  "Compilation aborted"
  "Floating point exception"
};
```

Don't forget adding a comma after the last message! If you don't your compiler will certainly complain about that. Are we done now. No, not quite yet. The 'THROW' routine wants to know which codes are exceptions generated by the system and which one are generated by the user. Why? Because user exceptions do not have messages attached to them! We can change that in `cmds_4th.h`.

```
/* ranges */
#define LastWord4th     FTELL
#define LastMsg4th      M4CABORT
```

Now it still points to the "duplicate name" error. We simply change "LastMsg4th" to our mnemonic:

```
/* ranges */
#define LastWord4th     FTELL
#define LastMsg4th      M4FLOATE
```

We're done now! Note that this final step is not necessary when you *insert* messages. Instead, you will have to renumber the table in `4th.h`. No two mnemonics may ever share the same error code, remember that! If you don't keep the mnemonics, the errorcodes and the messages properly synchronized you may get some pretty strange error messages. Which is less than helpful.

27.15 Sizing the Code Segment

By default 4tH assumes a 1:1 relationship between a word in source and a compiled word (in the Code Segment). When `ParseText()` is called it will count the number of words in the source. This number is later used to size the initial Code Segment. This 1:1 relationship is not so strange as it may seem at first, e.g.:

```
BL DROP
```

Will compile to:

```
[0] literal     (32)
[1] drop        (0)
```

Two words in source, two compiled words. But there are exceptions too,

e.g.:

```
BL ,
```

Will compile to:

```
[0] ,           (32)
```

That is because ',' does not compile to anything, but changes the previously compiled literal to an constant array element. Note that ',' is an immediate word. In fact, all exceptions to this 1:1 relationship rule are immediate words! The vast majority of 4tH words obey this 'one on one' rule:

- All numbers and constants compile to literals
- All ordinary words compile to a word without argument
- All symbols compile to a word with argument

In a previous chapter we've created an immediate word called "[SEXTAL]". When you take a closer look, you will see that it just changes the base; it doesn't compile to anything. Still, without the proper argument 4tH assumes it will compile a token and reserves space in the Code Segment.

Can you prevent this? Yes, you can. There is a member in the table of `ImmedList[]` which allows you to signal 4tH that it shouldn't reserve space in the Code Segment for "[SEXTAL]". The first field indicates the length of the keyword, the third indicates the correction 4tH should make to the sizing of the Code Segment when this keyword is encountered, the fourth is the keyword itself and the last one is the C function that compiles the word. The second and the fifth field will be discussed later.

```
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
{ 8,  0, -1, "[ASSERT]",    "",    DoAssert },
{ 8,  0, -1, "[BINARY]",    "",    DoBinary },
{ 9,  0, -1, "[DECIMAL]",   "",    DoDecimal },
```

Now, the correction we want to make is that 4tH should allocate one word *less* in the Code Segment, since "[SEXTAL]" does not compile to anything. One less means "-1". We can now change the table accordingly:

```
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
{ 8,  0, -1, "[ASSERT]",    "",    DoAssert },
{ 8,  0, -1, "[BINARY]",    "",    DoBinary },
{ 8,  0, -1, "[SEXTAL]",    "",    DoSextal },
{ 9,  0, -1, "[DECIMAL]",   "",    DoDecimal },
```

That's all. We'll give to a few more examples.

E.g. 'VARIABLE' does not compile to anything either; it just reserves space in the Variable Area. But 'VARIABLE' always comes with a name, which doesn't compile to anything either. So we should decrease the the number of words in the Code Segment by two!

'CONSTANT' not only requires a name, but consumes a previously compiled literal as well. *Initially* this literal allocates space in the Code Segment, but it is gone after 'CONSTANT' has been compiled. So we decrease the number of words in the Code Segment by three!

'VALUE' is even more complicated. You can write something like:

```
10 value ten
```

But this will compile to:

```
[0]   literal     (10)
[0]   to          (0)
```

'VALUE' does *not* consume the previously compiled literal! But the name does *not* compile to anything. 'VALUE' takes a value from the Data Stack at run time, while 'CONSTANT', 'STRING' and 'ARRAY' take a previously compiled literal at compile time. If you don't believe us, check the glossary.

'VALUE' itself compiles to something, the literal is undisturbed, only the name vanishes. That means only one word less, thus "-1".

27.16 Adding inline macros

We've already seen how we can add new words to 4tH. We add a token and write the runtime. But this approach has a few disadvantages. First, the number of tokens is limited. You can use them, but once you run out of them, that's it. Second, writing a runtime is a little complex if you only have limited knowledge of C.

You can add as many inline macros as you want. From a user point of view there is not much difference between an ordinary 4tH word and an inline macro. The word is recognized by the compiler and works as expected.

Inline macros are simply sequences of existing tokens. As we've seen before, 'NIP' is implemented as an inline macro, so this source:

```
1 2 nip drop
```

Will compile to:

```
[0]   literal     (1)
[1]   literal     (2)
[2]   swap        (0)
[3]   drop        (0)
[4]   drop        (0)
```

There are disadvantages to inline macros as well. Every time you use 'NIP' it will expand to two words, so your Hcode will become a little bigger. We recommend to limit inline macros to three words. Second, an implementation using inline macros will make the compiler less compact compared to an implementation using tokens.

On the other hand, you only need to change the compiler when you use an inline macro. No changes to the interpreter or the decompiler will be necessary. Existing HX files will still run, although some 4tH sources will need modification.

Let's go to business. How can we implement an inline macro. Let's take 'NIP'. First we have to make an entry in the `ImmedList[]` table.

```
{ 2,  1, -1, "TO",          "",    DoValue },
{ 2,  1, -1, "IS",          "",    DoIs },
{ 2,  1, -1, "AS",          "",    DoValue },
{ 2,  0,  0, "IF",          "",    DoIf },
{ 2,  0,  0, "DO",          "",    DoDo },
{ 2,  0,  0, "->",          "",    DoDummy },
{ 3,  0,  1, "2R>",         "",    DoTwoRGet },
{ 3,  0,  1, "2>R",         "",    DoTwoRPut },
{ 3,  0,  3, "2R@",         "",    DoTwoRCopy },
```

Since 'NIP' is defined by:

```
: nip swap drop ;
```

It will compile to:

```
swap  (0)
drop  (0)
```

Which is one token *more* than the parser would expect. So the value of the fourth field is "1". But this will only reserve space for 'NIP'. The word won't be recognized by the compiler yet. In order to do that we have to make a word that compiles the tokens for 'NIP'. You can only do that with an "immediate" word:

```
#ifndef ARCHAIC
      static void DoNip (void)
#else
      static void DoNip ()
#endif

{
      CompileWord (SWAP, 0L);
      CompileWord (DROP, 0L);
}
```

This will compile 'SWAP' and 'DROP' into the compilant. Now we have to make an entry in `ImmedList[]` to link this function to the name "NIP":

```
{ 2,  1, -1, "TO",          "",    DoValue },
{ 2,  1, -1, "IS",          "",    DoIs },
{ 2,  1, -1, "AS",          "",    DoValue },
{ 2,  0,  0, "IF",          "",    DoIf },
{ 2,  0,  0, "DO",          "",    DoDo },
{ 2,  0,  0, "->",          "",    DoDummy },
{ 3,  0,  1, "2R>",         "",    DoTwoRGet },
{ 3,  0,  1, "2>R",         "",    DoTwoRPut },
{ 3,  0,  3, "2R@",         "",    DoTwoRCopy },
{ 3,  0,  1, "NIP",         "",    DoNip },
```

That's all! Since 'NIP' uses existing tokens, the compiler can handle it all by itself. There is no need to write runtime code or define tokens. All the burden is put on the compiler.

27.17 Adding string words

We've already seen that some words in 4tH have a name attached to them, like 'STRING', 'CREATE' or 'VARIABLE'. Since all these names are delimited by whitespace (like any other 4tH word), there is no need for special code.

Some words have special strings attached to them, like '."', '(' or '\'. These string are not delimited by whitespace, so they need special treatment. It's even more complex: each word has a different delimiter. '."' is delimited by '"', '(' is delimited by ')', and '\' is delimited by an end-of-line marker.

In this section we're going to explain how we added '."', since it's the most complex string word. Other string words like '(' are handled by the compiler only.

The first step to adding a string word is letting the compiler know, what delimiter is used. We do that by modifying `ImmedList[]`:

```
{ 2,  0, -2, "#!",          EOL,   DoComment },
{ 2,  0, -1, ",\"",         "\"",  DoCommaQuote },
{ 2,  0, -1, ",|",          "|",   DoCommaQuote },
{ 2,  0, -1, ".\"",         "\"",  DoDotQuote },
{ 2,  0, -1, ".(",          ")",   DoDotQuote },
{ 2,  0,  1, ">=",          "",    DoGreaterEqual },
{ 2,  0,  1, "<=",          "",    DoLessEqual },
{ 2,  0, -1, "S\"",         "\"",  DoSQuote },
{ 2,  0, -1, "S|",          "|",   DoSQuote },
```

The fifth field tells the parser whether this word requires a special delimiter. Yes, '."' is delimited by '"', so we enter a quote in the fifth field. If you enter an empty string, the parser assumes the word isn't a string word at all.

Now, what will the parser do when it encounters '."'? It will find an entry in `ImmedList[]` for '."'. It will see that this is a string word. Then it will make a call to `ParseString()` to find the delimiter and flag everything in between as a word. Which means that if you call `GetNextWord()`, you will get the entire string and not just the next word, e.g.:

```
: hello ." Hello world" cr ;
```

Will be parsed as:

```
GetNextWord ():   :
GetNextWord ():   hello
GetNextWord ():   ."
GetNextWord ():   Hello world
GetNextWord ():   cr
GetNextWord ():   ;
```

You will notice that there are some words that are delimited by whitespace, e.g. 'CHAR'. Why is that? Isn't every word delimited by whitespace? Yes, it is. But note that every word, which is parsed by `ParseText()` is also checked by `ParseDirectives()`. This expression would cause problems:

```
CHAR (
```

After 'CHAR' is parsed by `ParseText()`, '(' follows and is recognized by `ParseDirectives()` to be the start of a comment. To prevent this, we let the string following 'CHAR' be parsed by `ParseString()`. By the time `ParseText()` regains control, the character following 'CHAR' is already parsed and can cause no more problems. In short, if an expression like:

word (

or

word \

is valid, let it be parsed by `ParseString()` by making an entry in the delimiter field of the `ImmedList[]` table. If not, don't.

Next, we have to develop a word that compiles '."'. Now, how can we compile '."'? First, we have to get the string and move it to the String Segment. We can do that by calling `GetNextWord()` manually, but then we have to check for NULL-pointers. It is much easier to call `DecodeWord()` which sets the ErrNo member automatically when an error occurs.

`DecodeWord()` takes one argument, which is the error code it should set the ErrNo member to. It returns TRUE if `GetNextWord()` was called successfully. "CurrentWord" now points to the string after '."'.

Then we have to move the string to the String Segment. `MoveString()` does just that. It expects "CurrentWord" to point to the string that has to be moved. It returns a number. We'll need that when we design the runtime code.

There is no token or combination of tokens for printing strings. So this one will need a token of its own. We'll call it "PRINT" for the time being. Now, we got all components.

- The string can be parsed
- We can move it to the String Segment
- We can compile a token and an argument

This is the code for `DoDotQuote()`:

```
#ifndef ARCHAIC
  static void DoDotQuote (void)
#else
  static void DoDotQuote ()
#endif
{
  CompileString (PRINT);
}
```

We aren't done yet. We still have to link the string '."' to this routine by adding an entry to `ImmedList[]`:

```
{ 2,  0, -1, ",\"",         "\"",  DoCommaQuote },
{ 2,  0, -1, ",|",          "|",   DoCommaQuote },
{ 2,  0, -1, ".\"",         "\"",  DoDotQuote },
{ 2,  0, -1, ".(",          ")",   DoDotQuote },
```

Now we can save `comp_4th()` and get on with the next file. Remember, we still got to add a token. As you will probably know, we do that in `cmds_4th.h`:

```
#define NOOP       0
#define CELLD      0
#define EXECUTE    1
#define CR         2
#define SPACES     3
#define EMIT       4
#define PRINT      5
#define DOT        6
#define FETCH      7
```

And of course, we have to add a name to `name_4th.c`, so it can be decompiled properly:

```
char *name_4th [] = {
  ",",       "execute",   "cr",      "spaces",   "emit",    ".\"",
```

Are we done yet? Not by a long shot. We have created a word with an argument, which is the offset of the string in the String Segment. That requires some special techniques. But we'll go into that in the next section.

27.18 Adding words with arguments

The very first thing you have to do is to make sure that your code can be saved and loaded again. Words that only consist of a token are saved without the argument. That reduces the size of the HX file. If you want to save the argument you have to add a line to both `load_4th()` and `save_4th()`:

```
case (LITERAL):
case (PRINT):
case (BRANCH):
case (BRANCH0):
```

Now we have to add code to `exec_4th()` in order to execute '."'. The first problem we encounter is: how do we access the argument? Accessing an argument is quite an expression:

```
Object->CodeSeg [Object->ErrLine].Value
```

In which:

OBJECT = Hcode pointer

CODESEG = Member of Hcode, pointing to the Code Segment

ERRLINE = Member of Hcode, pointing to the current word

VALUE = Member of word, holding the argument

In plain English it means: give me the argument of the currently executed word in the Code Segment. But we can also make our lives a lot easier by using this macro:

```
OPERAND
```

But this is only half the problem. How can we access the String Segment where the string-constant is stored? We made a pretty table on that subject.

DATATYPE	EXPRESSION	TYPE
String	Object->StringSeg [*{cell}*]	char
String	Object->StringSeg + *{cell}*	char*
Character	Object->UnitSeg [*{cell}*]	char
Character	Object->UnitSeg + *{cell}*	char*
Variable	Vars [Object->Offset + *{cell}*]	cell
Code	Object->CodeSeg [*{cell}*]	dict

Table 27.7: Accessing 4tH data from C

Most of the time it is more convenient to use functions to access those segments instead of addressing them directly:

AREA	FETCH	STORE
Data Stack	DPOP	DPUSH (value)
Return Stack	RPOP	RPUSH (value)
Character Segment	fetch (location)	store (location, char)

Table 27.8: exec_4th() data access API

You might consider using other functions too if certain datatypes are accessed more frequently.

Back to '."'. We have to access the String Segment for this one. Since all output is channelled through `emit()`, we have to convert the string to "units", which are unsigned characters. We could use the expression "Object->StringSeg [{arg}]", but that would be slower than pointer access on most systems. We decide to use "p", which is a temporary string-pointer:

```
case (PRINT):   for (p = Object->StringSeg + (unsigned) OPERAND; *p; p++)
                     emit ((unit) *p);
                break;
```

We assign "p" to a pointer to the string `(Object->StringSeg + {cell})`. We check for null-characters `(*p)`. If it is not a null-character, we 'EMIT' it `(emit (*p))` and advance the pointer `(p++)` before entering the loop again.

Now we are done. Let's do something more complicated now, like adding conditionals.

27.19 Packing several words into one token

Depending on the versiob, you got about 150 free tokens. Some developers think that equals the number of words they can add. But that's not true. You can pack several words into one token - if you know how. Note every token has an argument. You can use that one to distinguish between several words. Here is how. Let's assume you've defined the token `MULTI` in `cmds_4th.h`. In that token, we're gonna pack 'SWAP', 'DROP' and 'OVER'. First we have to change the compiler accordingly. Go to the `ConstList []` table and add these entries:

```
{ 4, "SWAP", MULTI, 0L },
{ 4, "DROP", MULTI, 1L },
{ 4, "OVER", MULTI, 2L },
```

That wasn't too hard, wasn't it? Now we have to add `MULTI` to `exec_4th()`:

```
CODE (MULTI)    switch (OPERAND)
                  {                   /* this is SWAP */
                    case (0L): DSIZE (2); a = DS (1);
                               DS (1) = DS (2); DS (2) = a;
                               break; /* this is DROP */
                    case (1L): DSIZE (1); DDROP; break;
                    case (2L): DSIZE (2); DFREE (1); a = DS (2);
                               DPUSH (a); break;
                    default  : throw (M4BADOBJ); break;
                  }                   /* error if unknown */
                NEXT;
```

So what have we done here? First we have set up an entry for `MULTI`. Then, depending on the value of the argument, we decide whether we have to execute 'SWAP', 'DROP' or 'OVER'. If the argument of `MULTI` has another value, we throw an error, since this should not have happened.

Of course, you could use the full range of a cell, packing *millions* of words into a single token. But there are several disadvantages to that approach. First, the runtime has to make another decision, which takes up valuable execution time. Second, it isn't that clear when you decompile the program what MULTI actually does. Remember that `dump_4th()` only displays the name "multi" and the associated argument. That's it! Third, the parameter takes extra space, making your HX executable a bit larger.

Needless to say, but you shouldn't implement this example in 4tH verbatim. It's - well - just an example. In real life, you would have to add the appropriate entries to `name_4th.c`, `save_4th.c` and `load_4th.c` as described in section 27.18.

27.20 Adding conditionals

Basically, there are ten branch-instructions in 4tH:

1. **BRANCH**, which unconditionally branches to an address in the Code Segment.
2. **0BRANCH**, which branches to an address if the Code Segment if the top of the Data Stack is zero.
3. **CALL**, which unconditionally branches to an address in the Code Segment, throwing its origin on the Return Stack.
4. **EXIT**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Return Stack.
5. **VECTOR**, which unconditionally branches to an address in the Code Segment, which is taken from the contents of a variable, throwing its origin on the Return Stack.
6. **EXECUTE**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Data Stack, throwing its origin on the Return Stack.
7. **CATCH**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Data Stack, throwing the data stack pointer, the previous handler and its origin on the Return Stack.
8. **LOOP**, which branches to an address in the Code Segment if the top of the Return Stack is less than value below it.
9. **+LOOP**, which branches to an address in the Code Segment if the top of the Return Stack plus the top of the Data Stack is not equal to the value below the top of the Return Stack.
10. **?DO**, which branches to an address in the Code Segment if the top of the Return Stack plus the top of the Data Stack is equal to the value below the top of the Return Stack.

The first four are the most common and the most useful ones. Together, they control your entire program. But how do they know where to branch to? There is no instruction like 'BRANCH'. And where is 'DO'?

Well, 'DO' doesn't do any branching. It just puts the loop parameters on the Return stack. And as for 'BRANCH', this is how it works:

IF something ELSE other thing THEN

This is an expression we are very familiar with. We pronounce it as:

"If TOS is non-zero then something is executed."

This is not entirely true. In fact, it is:

"If TOS is zero then branch after 'ELSE'"

Which in effect results in the execution of "something". But when "something" has executed, it has to branch after the "other thing". Unconditionally, that is. 'THEN' does nothing, except serve as a marker for the branch. It doesn't have to compile to anything.

So this little piece to code will compile to:

```
[0]   0branch     (2)
[1]   ...
[2]   branch      (3)
[3]   ...
[4]   ...
```

You see that 'IF' compiles to a '0BRANCH' instruction, 'ELSE' to a 'BRANCH' instruction and 'THEN' to nothing! If you have a closer look you might assume that 4tH will branch to instruction [2] and then branch directly to instruction [3]. This is not quite what was intended.

What a 'BRANCH' instruction actually does is setting the instruction counter to a specific value. Then, like after every instruction, the instruction counter is incremented. Why make exceptions? That only slows the interpreter down. Let's take a look at this piece of code:

```
10 dup if 1+ else . then cr
```

This will compile into:

```
[0]   literal     (10)
[1]   dup         (0)
[2]   0branch     (4)
[3]   1+          (0)
[4]   branch      (5)
[5]   .           (0)
[6]   cr          (0)
```

Now how does this execute. We will show you by giving the value of the instruction pointer before execution, after execution and after the automatic increment.

INSTRUCTION#	INSTRUCTION	BEFORE	AFTER	INCREMENT
[0]	literal	[0]	[0]	[1]
[1]	dup	[1]	[1]	[2]
[2]	0branch	[2]	[2]	[3]
[3]	1+	[3]	[3]	[4]
[4]	branch	[4]	[5]	[6]
[6]	cr	[6]	[6]	[7]

Table 27.9: Example execution plan

You see that '0BRANCH' has no effect when the top of the Data Stack is non-zero. And while 'BRANCH' sets the instruction pointer to "5", it will resume execution at location "6".

If you compile this little piece of code by hand and start compiling from the beginning, you will also see that you can't fill in the destination until it has been compiled. So how does 4tH do that?

4tH has a small stack (Control Stack) where it stores these addresses. So when it encounters an 'IF' or 'ELSE' or 'THEN' instruction, it stores its current address there. What would have happened during the compile of the previous program:

- 'IF' is encountered. It compiles a '0BRANCH' instruction and stores '2' on the Control Stack.
- 'ELSE' is encountered. It compiles a 'BRANCH' instruction, takes '2' from the Control Stack and changes the argument of the '0BRANCH' word to its own address, which is '4'. It stores '4' on the Control Stack again.
- 'THEN' is encountered. It takes '4' from the Control Stack and changes the argument of the 'BRANCH' word to address of the last compiled word, which is '5'.

But this example was correct. What would have happened if we had written something like:

```
5 0 : test begin 1 dup if ; while . then dup dup repeat
```

To prevent the compiler from accepting these kind of constructions, a reference is added. This reference tells the compiler what conditional was put on the stack. If the reference isn't correct, the compiler will throw an exception. There are five predefined references, but you may add your own.

There are three functions which handle conditionals:

FUNCTION	DESCRIPTION
MarkLink	Throws an address on the stack
PairLink	Gets an address from the stack
MakeLink	Makes a "backlink"

Table 27.10: Branch resolving API

They all take a reference as argument. All address calculation and errorchecking is done by these functions. Let's get to business and retrace our steps when we added 'BEGIN.. WHILE.. REPEAT'.

All conditionals are "immediate" words. So they have to be added to `ImmedList[]`. That also means, that each word has its own function. Let's design the one for 'BEGIN'. 'BEGIN' is just a marker, which means we have little more to do than to save the address on the stack:

```
#ifndef ARCHAIC
      static void DoBegin (void)
#else
      static void DoBegin ()
#endif

{
      MarkLink (R_BEGIN);
}
```

Yes, that's all. Just call `MakeLink()` with the proper reference! Just make sure, you've compiled everything you wanted to compile. Jumps resolved by `MarkLink()` will in effect always continue from the word that will be compiled next, although it *seems* they jump to the word compiled last.

Now we have to make 'WHILE'. 'WHILE' executes a piece of code when the top of the DataStack is non-zero. Which means it jumps to 'REPEAT' when the top of the DataStack is zero. Sounds like a '0BRANCH' instruction to us. Note, that 'BEGIN' doesn't play any part whatsoever here!

Because the address it has to jump to isn't known yet (we haven't encountered 'REPEAT', we can only compile the '0BRANCH' instruction with an arbitrary address. But 'REPEAT' will have to know the address in order to make a backlink, so we have to throw it on the stack:

```
#ifndef ARCHAIC
      static void DoWhile (void)
#else
      static void DoWhile ()
#endif

{
  CompileMark (BRANCH0, R_WHILE);
}
```

Note that `CompileMark()` is equivalent to:

```
CompileWord (BRANCH0, 0L);
MarkLink (R_WHILE);
```

Now we come to 'REPEAT'. It has a lot of things to do. First, it has to compile a 'BRANCH' instruction in order to get back to 'BEGIN'. Second, it will have to resolve the backlink from 'WHILE'.

In a way, 'REPEAT' has an advantage over 'WHILE'. It doesn't have to compile an arbitrary address, since it is already on the control stack. It has been provided by 'BEGIN'. It can retrieve that address by calling `PairLink()` with the proper reference:

```
CompileWord (BRANCH, PairLink (R_BEGIN));
```

But there is a problem. `MakeLink()` should always make a link to the last compiled word. And we can't compile the 'BRANCH' instruction first, because of the "WHILE" reference on the top of the controlstack!

So we have to resolve the 'WHILE' backlink first. For that purpose, `MakeLink()` has an extra argument. Usually, `MakeLink()` is called with "LASTW", which means it will jump to the word compiled last. Then the instruction counter will be incremented and the interpreter will continue from there.

In order to compile 'REPEAT', we have to make a backlink that points to the word compiled *next*. So, this statement is inserted before `CompileWord()`:

```
MakeLink (R_WHILE, NEXTW);
CompileWord (BRANCH, PairLink (R_BEGIN));
```

And what if we had made a "BEGIN..AGAIN" or a "BEGIN..UNTIL" loop? Well, in any case we would have to branch back to 'BEGIN'. 'UNTIL' conditionally and 'AGAIN' unconditionally. The address of 'BEGIN' would have already been on the control stack, so a single statement could have taken care of it:

```
CompileWord (0BRANCH, PairLink (R_BEGIN));
CompileWord (BRANCH, PairLink (R_BEGIN));
```

Actually, since 4tH supports multiple 'WHILE's the problem is a little more complex. 'REPEAT' must resolve all 'WHILE's on the control stack before it can even think of compiling a 'BRANCH' instruction. We've already seen that the only difference between an 'AGAIN' and an 'UNTIL' is the branch instruction which is compiled. So in 4tH 'REPEAT', 'UNTIL' and 'AGAIN' are handled in a similar way:

```
#ifndef ARCHAIC
  static void CompileAgain (unit AgainToken)
#else
  static void CompileAgain (AgainToken) unit AgainToken;
#endif
{
  while ((ToCS > 0) && (ControlStack [ToCS - 1].Mark == R_WHILE))
        MakeLink (R_WHILE, NEXTW);
  CompileWord (AgainToken, PairLink (R_BEGIN));
}
```

As long as the control stack is not empty and there is a 'WHILE' reference on top of the control stack, backlinks are made. Finally, the branch instruction is compiled, which jumps back to 'BEGIN'. 'REPEAT' can now be reduced to:

```
CompileAgain (BRANCH);
```

Note that a colon definition also uses the control stack. This reference is resolved by ';', which compiles 'EXIT' and creates a backlink. The 'BRANCH' instruction will prevent the interpreter from entering the definition. At the same time, ':' creates a symbol. We'll go into that in the next section.

If you want to create your own branch instructions, you'll have to define their behaviour in `exec_4th()`. If the argument of the token contains the address you want to jump to in the end, you'll have to define it like this:

```
JUMP (OPERAND);
```

That is pretty easy. This macro changes the Program Counter, which is part of the Hcode header:

```
Object->ErrLine
```

Of course, we've defined a macro for that:

```
PROGCOUNT
```

If the address you want to jump to is issued by the user, you probably want to check whether it is a valid execution token. Just use the macro `XT()`:

```
DSIZE (1);
a = DPOP;
XT (a);
JUMP (a);
```

Consequently, leaving the current Program Counter value on the return stack is pretty easy too:

```
RFREE (1);
RPUSH (PROGCOUNT);
```

I think that covers it all, don't you?

27.21 Extending the I/O subsystem

The 4tH I/O system is entirely built upon the buffered C streams[6] concept. That means every device that can be assigned to a `FILE*` and accessed through `fgetc()` and `fputc()` can be integrated into the 4tH I/O system. If can open a device with `fopen()` and close it with `fclose()`, you're done, it's already supported. If not, you have to design two functions that take the same parameters and return the same values as `fopen()` and `fclose()`. If you can't you can probably still use those devices within 4tH, but you can't integrate them into the I/O system.

If you have defined these functions, you'll have to make changes to `OpenStream()` in `exec_4th()`. 'Mode' is the sum of all file access methods, e.g.

```
s" ls" input pipe + open
```

This definition contains two file access methods, INPUT and PIPE. You can find these values in cmds_4th.h. INPUT equals 1 and PIPE equals 8. That makes 9 and that is what ends up in the 'Mode' parameter of `OpenStream()`. However, if one would allow all possible combinations of all file access methods some would surely make little sense. That is why 'Mode' is filtered by `Mapping[]`. You will find that element 9 of `Mapping[]`[7] contains the value 5. That number corresponds to element 5 in `Modelist[]`, which lists the correct file access method (which is still 9, of course) and the mode parameter for `fopen()`. `OpenStream()` continues by initializing the members of the `Stream[]` structure.

MEMBER	FUNCTION
Mode	Uniform file access method
Device	FILE pointer to opened device
Connect	Function pointer to `fopen()` like function
Disconnect	Function pointer to `fclose()` like function

Table 27.11: Members of Stream[] structure

When this is done, it uses the `Connect()` member to open the device. After that, everything is completely transparent to the programmer.

If you want to add a new device, you probably want to signal which type of device you're using. In order to do that, you must first add a `#define` to the 'file modes' section in cmds_4th.h. Each new file access method has exactly twice the value of the previous one, which means that the first one you define would have to be 16. Then you have to figure out which modes are actually supported. Can your device be opened in read-write mode? Does appending make sense? You add those 'ideal' states to `ModeList[]`. Then map all possible combinations of all file access methods to the 'ideal' states in `ModeList[]` using `Mapping[]` conversion table. Every additional file access method *doubles* the size of `Mapping[]`, so beware!

MACRO	FUNCTION	THROW
`DEV(n)`	Aborts if *n* is not a device	`M4BADDEV`
`UDEV(n)`	Aborts if *n* is not opened by the user	`M4BADDEV`
`ODEV(n)`	Aborts if *n* is not opened	`M4IOERR`
`SDEV(n)`	Aborts if *n* is a pipe	`M4BADDEV`

Table 27.12: Device status macros

[6] See: http://www.aquaphoenix.com/ref/gnu_c_library/libc_118.html
[7] We start counting at 0.

If you want to check a device in `exec_4th()`, there are several macros you can use. They all take the value returned by `OpenStream()` as parameter.

27.22 Using the symbol table

The symboltable is a way to dynamically add words to the vocabulary of 4tH. All other words are hard-coded into the compiler. If you want to add any, you have change the entire compiler and make a new executable. We've seen that before.

Without the symboltable there wouldn't be any strings, tables, variables or even colon-definitions. May be you think that such a powerful feature must be hard to work with. No, it isn't!

The only thing you have to tell the symboltable is "hey, if that word comes along, compile this token and this argument into the object". That's all. There are three functions that control the symbol-table:

FUNCTION	DESCRIPTION
AddSymbol()	Adds a symbol to the symboltable
MakeSymbol()	Makes a symbol of the current word
GetSymbol()	Searches the symboltable
SearchDictionary()	Searches the entire dictionary

Table 27.13: Symboltable API

You can forget about the `GetSymbol()`. You will hardly ever need it. Let's see how it works. We'll continue with ':':

```
if (DecodeWord (M4NODECL))
   {
     AddSymbol (CALL, Object->ErrLine, CurrentWord);
     CompileMark (BRANCH, R_COLON);
   }
```

First it uses `DecodeWord()` to set "CurrentWord" to the next word in the source, which is the name of the definition. Then it adds a symbol to the symboltable. Hey, shouldn't we compile a word first?

No. The member "ErrLine" of an Hcode header always points to the next available word in the Code Segment. That is the place where we will compile our 'BRANCH' instruction. When the instruction pointer is set to that location, it will be automatically incremented and get *inside* the definition. So that is okay.

The token we'll use to branch inside that definition is not 'BRANCH' or '0BRANCH', but 'CALL'. 'CALL' throws the address of its own location on the Return Stack. When the definition is done, 'EXIT' takes that address off the Return Stack and jumps backs.

`AddSymbol()` adds an entry to the symboltable. No, you don't need to check the symboltable when you add a symbol. `AddSymbol()` does that for you and sets the member "ErrNo" when needed.

Finally, a 'BRANCH' token is compiled with a dummy argument. It will be solved later with a backlink, marked by `MarkLink()`. Now, how does it actually work? We'll give you an example. Take this small program:

```
: hello's 0 do ." Hello " loop cr ;
20 hello's 10 hello's
```

When the ':' is reached by the compiler, it hasn't compiled a thing, so "ErrLine" still points to the first word in the Code Segment (0). `DecodeWord()` is called, so "CurrentWord" points to "HELLO'S". Then a symbol is added to the symboltable by calling `AddSymbol()`. The entry looks like this:

HELLO'S -> CALL (0)

That means that every time the name "HELLO'S" is found in the source, the word "CALL (0)" will be compiled. See for yourself:

```
[0]   branch      (6)
[1]   literal     (0)
[2]   do          (0)
[3]   ."          (0)
[4]   loop        (2)
[5]   cr          (0)
[6]   exit        (0)
[7]   literal     (20)
[8]   call        (0)
[9]   literal     (10)
[10]  call        (0)
```

Basically, that is all you need to know about the symboltable. Yes, you can search it yourself, but why should you? It is done automatically for you. But if you really want to know: you do it by calling `GetSymbol()`.

All you need is the name of the symbol you're looking for and what you want the compiler to do when i finds it. E.g. if you were looking for "HELLO'S" and didn't want to compile it, you'd have to write:

```
int x = GetSymbol ("HELLO\'S", W_SEARCH);
```

`GetSymbol()` returns the index of "HELLO'S" in the symboltable. You can use this index to access the symboltable, called "SymTable":

```
printf ("%d, %ld, %s\n", (int) SymTable [x].Token,
       (long) SymTable [x].Value, SymTable [x].Name);
```

Which prints the token, the argument and the name of the symbol. If the name isn't listed `GetSymbol()` returns "MISSING":

```
if ((x = GetSymbol ("HELLO\'S", W_SEARCH)) == MISSING)
     printf ("Not found\n");
```

If you search the symboltable in order to compile a word, you only have to tell:

```
int x = GetSymbol ("HELLO\'S", W_EXEC);
```

This will not only return the index, but compile the word as well. Note that this function can only search the symboltable. It cannot look for other words. These words have their own function, but basically work the same:

CLASS	FUNCTION
Immediate words	GetImmediate()
Simple words	GetWord()
Constants	GetConstant()

Table 27.14: Table search API

They will return an index as well, but that will be of little use since the tables they search are private and cannot be accessed outside the function. `SearchDictionary()` combines all these functions (including `GetSymbol()`) but will only return a boolean to indicate that that the word was found. It is the most common way to access these lower level functions.

If you decide to add your own words that use the symboltable, you have to make an entry in `ImmedList[]`. Let's say you want to add a word, which defines a floating point number, e.g.:

```
float fp_number
```

Now we have to let 4tH know that for each "FLOAT" a symboltable entry has to be reserved:

```
{ 5,  1,  -2, "FLOAT",       "",    DoFloat },
```

Yes, that is where that famous second field is for! It tells 4tH how many symboltable entries it has to reserve for a specific immediate word.

27.23 Using variables and datatypes

We're slowly entering the area where extensions are becoming projects on its own. You should be able to make the most common extensions yourself now. What we have ahead is just for the interested reader of someone who want to add a completely new wordset.

We're going to explain you how 4tH handles strings and other datatypes. Variables (any variable!) are not created during compilation. That means no space is reserved. 4tH only monitors how much space has been allocated to each datatype. This information is saved in the header.

At the moment there are only two basic datatypes: characters (Character Segment) and 32 bit signed integers (Integer Segment). You'll find the size of these segments in the Hcode members "Variables" and "Strings". The Character Segment and Integer Segment are created when a Hcode program is executed and discarded when the Hcode program is terminated.

So the only thing the compiler has to do is keep track of the sizes of the segments and assign pointers to variables. This is quite easy. When an Hcode header is initialized by `InitObject()`, both "Variables" and "Strings" are set to zero. Then it parses this declaration:

```
variable one
```

As a consequence, `DoVariable()` is called:

```
if (DecodeWord (M4NODECL))
    AddSymbol (VARIABLE, Object->Variables++, CurrentWord);
```

It calls `DecodeWord()`, so "CurrentWord" now points to "ONE". "Variables" is still zero. If `DecodeWord()` was called successfully, it just adds a symbol by calling `AddSymbol()`, which looks like this:

ONE -> VARIABLE (0)

After that "Variables" is incremented, so it now holds the value "1". It doesn't matter, what comes next: "ONE" will always compile to "VARIABLE(0)". The next variable will compile to "VARIABLE(1)". Unless it is an array:

```
10 array list
```

The "10" is compiled as a literal. Then the compiler encounters "ARRAY", so `DoArray()` is called:

```
cell val = DecodeSymbol ();

if (! Object->ErrNo) {
      AddSymbol (VARIABLE, (cell) Object->Variables, CurrentWord);
      Object->Variables += (unsigned) val;
}
```

First it calls `DecodeSymbol()`, which does two things:

1. It calls `DecodeWord()`, so "CurrentWord" now points to "LIST".
2. It removes the previously compiled literal (by decrementing the member "ErrLine") and returns it.

Now "val" holds the value "10". If no error occurred, `DoArray()` will call `AddSymbol()`. There an entry is created that looks like this:

LIST -> VARIABLE (1)

So every time the name "LIST" is encountered a 'VARIABLE' token will be compiled with argument "1". Finally, the number of variables is incremented by "val", so the member "Variables" now holds "11". This means that 10 variables have been added, which is correct.

It works about the same for 'STRING', only we compile a literal value here. Why? Because the system areas in the Character Segment are fixed. In the Variable Area there are also the application variables and 4tH cannot know at compile time how many there will be at runtime.

We could have placed the variables right after the system variables, but that would have made it much more difficult to add names to your application variables. But now we have to resolve what 'VARIABLE' has to do at runtime. So we have to edit `exec_4th()`.

Well, the only thing it has to do is calculate its address in the Variable Area. There is a member in the header that holds the offset of the user variables inside the Variable Area. The only thing we have to do is to add the operand to it and push the result:

```
DPUSH (Object->Offset + OPERAND);
```

Since the next word takes the address of the Data Stack there is no real difference with a literal. The changes you want to use the address of a variable as a literal expression are quite remote.

There are two macros that check the status of a variable. `VAR(n)` checks whether *n* is a variable at all. `UVAR (n)` checks whether *n* is a writable variable. When any of these macros fail, `M4BADVAR` will be thrown. *n* is the value that VARIABLE leaves on the stack.

Of course, if you want to add an entirely new datatype, you have lots of work to do, but you can use the same tools as we have used. We have to stress that you use a separate segment for each datatype. That keeps 4tH simple and it won't take more memory than other implementations.

Note that if you want to create constants for a certain datatype you have to work out a scheme to load and save them. If this scheme depends on a certain, non-portable encoding, you won't be able to use the resulting .HX files on different platforms.

27.24 Other tools

We have known assertions since version 3.1c and conditional compilation since version 3.1d. Both conditionally skip source between to markers. And they can be nested. That sounds like quite a challenge, but it isn't. In fact, there is only one simple routine that handles it.

If we encounter a situation where source has to be skipped, we just call `SkipSource()`. In case of conditional compilation, the source that we have to skip is between the markers '[IF]' and '[THEN]'.

First, we call `DecodeLiteral()`. This function gets the argument part of the previously compiled literal and removes that literal from the compilant (actually, the member "ErrLine" is decremented, so it will be overwritten):

```
cell val = DecodeLiteral ();
if (! Object->ErrNo) if (! val) SkipSource ("[IF]", "[THEN]");
```

Then `SkipSource()` is called with the argument "[IF]" and "[THEN]". It will handle everything, including any nested markers. Note that "CurrentWord" still has to point to the "[IF]" that triggered the action.

27.25 Patching 4tH

We're getting at the end of the story here. There is one topic left we want to discuss with you.

It is a drag when you have made some nice extensions to 4tH and you have to reapply them each time a new version of 4tH is released. However, there is a solution. 4tH comes with a program called `patch4th.4th` which can help you. The only thing you have to do is to create a 4tH patch file. It consists of six parts, which *all have to appear in the order presented* to you here. If a section is not applicable, leave it blank.

27.25.1 Tokens

The first part are the tokens or the instructions of the virtual machine, if you prefer. Every entry consists of three fields, delimited by a tilde[8]:

1. The first field is the C constant of the token, as it appears in `cmds_4th.h` (*see* 27.5);
2. The second field indicates whether the token needs a parameter (*see* 27.18);
3. The third field is the mnemonic, as used in `name_4th.c` (*see* 27.7).

So a sample entry might look like:

```
[tokens]
NIP~no~"nip"
```

To terminate this section, add an empty line.

27.25.2 Words

The second part are the words you actually use in a 4tH program. As you will know by now, a word can compile to zero or more tokens. Every entry consists of eight fields, delimited by a tilde:

1. The first field is the name of the word as you will use it in a 4tH program;
2. The second field is the token it will compile to;
3. The third field contains the type of word, *constant*, *immediate* or *word*;
4. The fourth field is the fixed parameter of a *constant*;
5. The fifth field is the number of symbol entries it will need;
6. The sixth field is the source correction that will be applied;
7. The seventh field is the delimiter it uses;
8. The eighth field is the C function which handles the *immediate* word.

A simple *word* requires fields 1, 2 and 3. A *constant* requires fields 1, 2, 3 and 4. An *immediate* word requires 1, 3, 5, 6, 7 and 8. If a field is not applicable for a certain type it will not matter what you enter there. See section 27.3 for more information. A sample entry might look like:

```
[words]
BIRTHDAY~LITERAL~constant~19600902L~0~0~~
BINARY~RADIX~constant~2L~0~0~~
NIP~NIP~word~~0~0~~
[SEXTAL]~~immediate~~0~-1~""~DoSextal
```

To terminate this section, add an empty line.

[8] A tilde is rarely used by 4tH, so that seemed a good choice. If you prefer another delimiter, you have to change the source of `patch4th.4th`.

27.25.3 The virtual machine

These *sections are copied verbatim* - including indententation - into `exec_4th.c`. The first section are the additional `#include` directives you might need. These are located in the `[vm.include]` section:

```
[vm.include]
#include <sys/stat.h>
```

You terminate this section by directly continuing with the next section, `[vm.globals]`. This contains any global variables you use in the virtual machine, e.g. `Sleeping`. They will appear right before the prototype of the `throw()` function:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
```

You terminate this section by directly continuing with the next section, `[vm.io]`. This contains the I/O functions `emit()` and `Accept()` if you want to replace the ones that `exec_4th()` provides. If you want to keep the standard I/O functions, leave it *completely blank, not even an empty line*, e.g.:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
[vm.io]
[vm.support]
```

You terminate this section by directly continuing with the next section, `[vm.support]`. This contains any support functions for the virtual machine, e.g. `OpenStream()`. They will appear right before the main `exec_4th()` function:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
[vm.io]
[vm.support]
/*
Custom support functions
*/
```

You terminate this section by directly continuing with the next section, `[vm.vars]`. This contains any additional local variables of the `exec_4th()` function, e.g. `VarMax`, which will be added to the local variable list:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
[vm.io]
[vm.support]
/*
Custom support functions
*/
[vm.vars]
  time_t   MyLocal;
```

You terminate this section by directly continuing with the next section, `[vm.extension]`. This contains the actual C code which will be copied into the main loop of `exec_4th()`:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
[vm.io]
[vm.support]
/*
Custom support functions
*/
[vm.vars]
  time_t   MyLocal;
[vm.extension]
        CODE (NIP)      DSIZE (2);
                        DS (2) = DS (1);
                        DDROP;
                        NEXT;
```

You terminate this section by directly continuing with the next section.

27.25.4 Immediate words

This section contains the C functions that are executed when immediate words are compiled (*see* 27.3 and 27.8). They will be inserted verbatim just before `ImmedList[]`:

```
[vm.include]
#include <sys/stat.h>
[vm.globals]
static unsigned MyGlobal;
[vm.io]
[vm.support]
/*
Custom support functions
*/
[vm.vars]
  time_t   MyLocal;
[vm.extension]
        CODE (NIP)      DSIZE (2);
                        DS (2) = DS (1);
                        DDROP;
                        NEXT;
[immediate.words]
#ifndef ARCHAIC
      static void DoSextal (void)
#else
      static void DoSextal ()
#endif
{
      Base = 6;
}
```

You don't have to explicitly terminate this section.

27.25.5 Applying the patches

Make a subdirectory, copy the original `cmds_4th.h`, `comp_4th.c`, `exec_4th.c`, `name_4th.c`, `save_4th.c` and `load_4th.c` sources into it and rename them to

`.txt`. They will serve as templates for your custom 4tH sources. In this example we will assume your custom patchfile and the compiled `patch4th.4th` are also located there, but that is not required. When you make a directory listing you will see the following files:

```
cmds_4th.txt
comp_4th.txt
exec_4th.txt
name_4th.txt
save_4th.txt
load_4th.txt
mypatch.txt
patch4th.hx
```

Now run it:

```
4th lxq patch4th.hx mypatch.txt
```

When everything is alright, you will see the following messages:

```
Opening mypatch.txt
.. 1 tokens read
.. 4 words read
Processing cmds_4th.txt
.. done
Processing save_4th.txt
.. done
Processing load_4th.txt
.. done
Processing name_4th.txt
.. done
Processing exec_4th.txt
.. done
Processing comp_4th.txt
.. done
Closing mypatch.txt
.. done
```

When you list the directory again, you will see that new `cmds_4th.h`, `comp_4th.c`, `exec_4th.c`, `name_4th.c`, `save_4th.c` and `load_4th.c` sources have been created.

27.25.6 Error messages

Usage: patch4th patch-file	Issue a patchfile on the commandline
Bad boolean	”yes” or ”no” was expected in this field
Bad datatype	”word”, ”constant” or ”immediate” was expected in this field
Bad number	A number was expected in this field
Cannot find [tokens]	A `[tokens]` section was expected in the patchfile
Cannot find [words]	A `[words]` section was expected in the patchfile
Cannot find [vm.include]	A `[vm.include]` section was expected in the patchfile

Cannot find [vm.support]	A `[vm.support]` section was expected in the patch-file
Too many tokens	Too many tokens were defined in the patchfile
Cannot find /* ranges */	Corrupted `cmds_4th.txt` file
Cannot find NOOP token	Corrupted `cmds_4th.txt` file
Cannot find LastWord4th	Corrupted `cmds_4th.txt` file
Cannot open <file>.h|c	Could not create a source file
Cannot open <file>.txt	Could not find a template file

DOCUMENT ENDS HERE

Copyright 1997, 2017 J.L. Bezemer