

Emacs Voice Commander

User manual

VERSION 1.2

Hans van Dam, April 2000

contents

<u>CONTENTS</u>	<u>1</u>
<u>INTRODUCTION</u>	<u>1</u>
<u>GENERAL CONSIDERATIONS</u>	<u>3</u>
SYNCHRONIZING NATSPEAK MODES AND EMACS MODES.....	3
CREATING EMACS FUNCTIONS VS. NATSPEAK MACROS	3
CREATING EMACS ALIASES VS. NATSPEAK MACROS	3
STRIPPING OFF MODE SPECIFIC PREFIXES.....	4
SEPARATING GENERATED COMMANDS FROM HANDMADE DRAGON MACROS	4
FEEDBACK.....	4
RELEASE NOTES.....	4
<u>USING EMACS VOICE COMMANDER</u>	<u>6</u>
SETTING UP EMACS VOICE COMMANDER.....	6
RUNNING THE GENERATOR.....	6
THE FRAMEWORK	6
CALLING EMACS COMMANDS.....	7
NEW COMMANDS	8
ALIASES.....	8
SINGLE ALIASES	8
MULTIPLE ALIASES	9
BOOKMARKS.....	9
MOVING TO STANDARD BUFFERS	10
MISCELLANEOUS HARD CODED MACROS.....	10
ELSE FUNCTIONALITY.....	11
ADDING EMACS FUNCTIONALITY	11
TRAINING COMMANDS THAT ARE NOT RECOGNIZED.....	12
TRAINING YOUR SELF FOR USING THE FRAMEWORK.....	12
MY PREFERENCES	12
<u>IMPLEMENTATION</u>	<u>13</u>
DISCUSSION OF THE FILES IN THE PACKAGE	13
GLOBAL.DVC GENERATION.....	14
COMMAND_DOC.RTF GENERATION.....	15
<u>THINGS TO DO</u>	<u>16</u>

EMACS WINDOW CONFIGURATION SUPPORT	16
SAPI IMPLEMENTATION	16
MODE AWARENESS OF PROGRAMMING MODES.....	16

Introduction

Emacs is an editor very well-equipped for speech recognition. It is armed with highly advanced navigation methods, it can operate in different modes completely tailored to the task at hand, it has an advanced documentation system and can be customized in detail to your specific needs. The latter is very important for interconnecting Emacs and NaturallySpeaking properly. One of its most important properties for speech recognition is that it is completely character based. Contemporary speech recognition systems are better suited for sending character strings to applications than for controlling GUIs. Besides it being well equipped for speech recognition there are many more reasons to convert to Emacs.

Another important question is why to use NaturallySpeaking professional instead of another speech recognition product. The main reason lies in its commands and control macro system. Macros in NaturallySpeaking professional can be defined per application and per window within that application. So-called sentence macros allow for variable parts (lists) within the macro calling structure (like 'go up <number1-9>'). Macro actions are defined by a scripting language, which offers reasonable flexibility. The choice for NaturallySpeaking professional, however, to which this implementation is severely bound, is also its main disadvantage. I hope this package will prove its conceptual power and lead to a high level SAPI implementation. An SAPI implementation could easily be ported from one speech recognition system to another and hopefully from one platform to another. An SAPI implementation would presumably be written in C/C++ and therefore offer much more flexibility and genericity. It would also remove the need to buy NaturallySpeaking professional, which is very expensive. See chapter "things to do" for details. Other contemporary speech recognition products don't offer all of these ingredients yet at the time of writing. ViaVoice has all ingredients except the scripting language, VoiceXpress has no custom sentence macros and scripting language and FreeSpeech98 has no macros at all. These product features, however, can easily change over time.

This is not the first time a set of speech recognition macros is defined for Emacs. Most of these are made for DragonDictate, some for NaturallySpeaking. They basically consist of a long unorganized list of macros that send key strings that sometimes even correspond to Emacs key-bindings. The names of these macros are often determined by the preferences of their creator. This package improves the previous approaches on the following points:

- Emacs command names are used instead of key-bindings. Key-bindings can be customized to the users preferences, so it is better not to use them in NaturallySpeaking macros. Using Emacs command names also avoids the question of finding the right name for an action. Commands can be pronounced as they are defined in Emacs and can easily be learned from books about Emacs that have nothing to do with speech recognition.
- Different Emacs modes have different commands that are valid in their context. This command recognition context is reflected within NaturallySpeaking. This provides for the same command having different meanings in different 'contexts' ('modes' in Emacs).
- Emacs itself is perfectly capable of defining aliases for its commands. It is better to have alias definitions inside Emacs itself instead of in NaturallySpeaking, both for reasons of portability to other speech recognition products as for platform independence. Furthermore the definitions of these aliases can easily be retrieved using Emacs, so it'll be better documented.
- Comparatively little NaturallySpeaking macros are needed because most Emacs commands or their aliases are incorporated in lists (corresponding to the variable parts of sentence macros).
- This package is not just a NaturallySpeaking macro file but a program written in Emacs Lisp that generates a NaturallySpeaking macro file from the information available in Emacs. This not only means that all Emacs commands are speech enabled, but also that any functionality you add to Emacs can be speech enabled by simply running the generator again. This is a big advantage, because it implicitly speech enables the enormous amount of Emacs functionality available on the Internet. The fact that many

people already have a set of NaturallySpeaking macros for Emacs, or don't want to write Emacs Lisp code is also taken into account. As I'll explain below functionality written in NaturallySpeaking's macro language can easily coexist with the generated functionality.

General considerations

Synchronizing NatSpeak modes and Emacs modes

Emacs has a separate major mode for every type of buffer content (e.g. a c-file, a Perl-file, an ordinary text document or a directory). This mode determines how Emacs behaves and which commands are valid in its context. Every major mode has its own set of key bindings. Since we use speech recognition instead of keys it is quite obvious that every major mode should have its own set of voice commands.

NaturallySpeaking determines its command recognition context by means of the title of the active (sub)window. If Emacs were a Microsoft Windows MDI program any buffer would have its own window. Emacs, however, has its own specific MDI structure, which has nothing to do with Microsoft Windows frames. Therefore to synchronize NaturallySpeaking's recognition contexts with Emacs major modes, the current major mode has to be reflected in the frame caption (window title).

Fortunately Emacs is highly customizable, so I have written code that does that. The function `set-mode-in-frame-title` in `../evc/frame-title.el` puts the name of the major-modes of the current buffer and the active minor-modes in the frame title. The function `set-mode-in-frame-title` is hooked into `post-command-idle-hook` which is triggered every time the focus moves to another buffer (as the name says it's actually triggered every time Emacs is idle).

Macros in NaturallySpeaking (and also in ViaVoice and VoiceXpress) are organized per application, and per sub window (title) within that application. A set of macros is identified by a string and is enabled when that string is present in the frame title of the active window. As I've indicated before the current mode is now reflected as a part of the window title of Emacs. So we can create a separate command set for every mode.

Creating Emacs functions vs. NatSpeak macros

As already stated in the introduction it is preferable to leave as much functionality as possible within Emacs. In the ideal case the only thing NaturallySpeaking does is send key strings to Emacs that invoke single Emacs commands. An Emacs command like `find-file` can be called by sending the keystrokes `{Alt+x}find file{Enter}` to Emacs. Another example is `{Alt+x}other window{Enter}`. The words in Emacs commands are normally separated by a "-" instead of a " ". Emacs however converts a " " to a "-" in this context, and a " " leads to better pronunciation models (using hyphens even leads commands to be completely unrecognizable by NaturallySpeaking).

Basically the two exemplary commands shown above can easily be implemented by one NaturallySpeaking macro and a list containing the space-separated Emacs commands. The action of that macro is simply: `'SendKeys "{Alt+x}" + _arg1 + "{Enter}"`.

This type of framework leads to little functional code in NaturallySpeaking scripting language which makes it much easier to port the concept to for instance an SAPI implementation. It is also little work to add extra commands this way.

Creating Emacs aliases vs. NatSpeak macros

Often you won't be satisfied with the way a command should be pronounced. For instance the command `delete-other-windows` might be used so often that you want a shortcut or alias for it (in fact I already did). It is quite easy to create an alias in the form of a NaturallySpeaking macro (for instance with title 'this') in the global command recognition context, that sends a keystroke sequence like `{Alt+x}delete-other-windows{Enter}` to Emacs. This would lead (again) to a large set of NaturallySpeaking macros which is not preferable because it is a lot of work to create the code in the first place, and it is even harder to maintain (it is also nonportable and not retrievable using Emacs' documentation system).

The better approach is to create aliases for Emacs commands within Emacs itself. These aliases

will be prefixed by their command recognition context to prevent name clashes (no prefixed for global Emacs aliases). The alias-definitions are added to a special alias definition file. For instance the alias for delete-other-windows ('this') is added to the alias definition file as "(defalias 'this 'delete-other-windows)". 'this' will also be added to the NaturallySpeaking list "commands" which is used by the macro '<commands>' (in global recognition context) which contains the code 'SendKeys "{Alt+x}" + _arg1 + "{Enter}"'. This alias creation process is taken care of by two macros.

Stripping off mode specific prefixes

Many mode specific commands in Emacs have prefixes indicating to what mode the command applies (e.g. dired-unmark, dired-mark, info-other-window and info-split). When issuing dired-unmark in dired mode we'd preferably just say "unmark". We know we're in dired mode, so there's no point in pronouncing the prefix of the Emacs command. This is where we need Dragon scripting language. In the list "commands" (in the dired-mode recognition context) we just include the string "unmark". In the dired recognition context we include a macro "<commands>". In the action part of the macro we simply prepend "dired-" to the string "unmark" ('{Alt+x}dired-" + _arg1 + "{Enter}").

In some cases stripping off mode specific prefixes leads to name clashes with global commands. For instance stripping off dired- from dired-find-file leads to "find-file", which is ambiguous because it is also a global command. In these cases the mode specific prefix should still be pronounced to activate the command. In the macro "<commands>" (in dired context) dired- is not prepended to a list item when that list item starts with "dired".

Separating generated commands from handmade Dragon macros

Because many people still have Dragon NaturallySpeaking macros for Emacs they'd like to preserve I've accommodated space for them. I'll use a slightly different command recognition context (i.e. frame title sub string) for generated commands than for handmade macros in Dragon scripting language:

- global generated Emacs commands are in a context activated when the frame title contains the sub string "emacs".
- global handmade macros are in a context activated when the frame title contains the sub string "emacs@"
- mode specific generated Emacs commands are in a context activated when the frame title contains the sub string "<ModeName>" (e.g. "dired", "info" or "c")
- mode specific handmade macros are in a context activated when the frame title contains the sub string "@<ModeName>" (e.g. "@dired", "@info" or "@c")

As you can see I use the word "macro" for NaturallySpeaking macros and "command" for Emacs interactive commands to avoid confusion.

Feedback

If you have feedback about this package, please don't hesitate to send me emails (hans_van_dam@hetnet.nl). This can help me improve it, and more generally I'd like to know what you think about it.

Release notes

The upgrade from version 1.0 to version 1.1 offers the following extra features:

- Also enables invoking commands when in the minibuffer (enables recursiveness for the minibuffer).
- Bulk creation of aliases. E.g. you can now create aliases for all commands containing "kbd" at once, changing the component "kbd" to "keyboard". See section Aliases on page 8 for details.

- Fixes a bug concerning alias creation.
 - Slightly better tuned mode-awareness: the file mode-list.el is updated.
- The upgrade from version 1.1 to version 1.2 offers the following extra features:
- support for Peter Milliken's ELSE package.
 - The availability of my personal removables.txt and aliases.el

Using Emacs voice Commander

Setting up Emacs voice Commander

To set up Emacs voice Commander all you have to do (after extracting Emacs_voice_Commander.zip) is add the contents of the file ".../evc/.emacs" to your own .emacs-file, replacing the "." by the path to which you extracted Emacs_voice_Commander.zip. Emacs voice Commander needs Dragon NaturallySpeaking professional (3.5/4.0) and Emacs 20.3.1 or higher (at least I haven't tested it for lower versions).

Running the generator

To run the generator for the first time you'll have to activate it typing. After following the installation instructions you can simply type "{Alt+x}generate-command-file{Enter}" (don't take the stuff between { } literally. {Alt+x} means pressing Alt and x together). The generator will now start to construct a global.dvc file in your ".../evc" directory and a "command_doc.rtf" file in your ".../evc/documentation" directory. It takes about 30 seconds (I haven't done any profiling and probably made some very inefficient choices, but what the heck, micro-pauses are good for your health).

If you have Dragon NaturallySpeaking running you'll have to close it down now.

After that you start Dragon NaturalImportDVC (to be downloaded from <http://www.synapseadaptive.com/joel/natmergeutilityaprogram.htm> (see readme.txt)). In this utility for "new commands input file" press the browse button and navigate to the file ".../evc/global.dvc" (you'll only have to do this once, the application saves this setting). select upgrade user named (and select the user you're using) and check "overwrite existing commands". Then press "import" and your global.dvc file will be updated.

If you restart Dragon NaturallySpeaking now you'll have all commands available.

The framework

The generated global.dvc file contains a number of macros available in each context. Most of them contain list elements (denoted by <...>). Before discussing their uses I'll first give a list of macros in each context (MENU refers to the active application, STATE denotes a string that should be present in the window title of the active window to activate the set of macros listed):

```
MENU "Global Commands"
  STATE "Global Commands"
  COMMAND "train word"
MENU "NATSPEAK"
  STATE "Command Wizard"
  COMMAND "cancel to emacs"
  COMMAND "confirm to emacs"
  COMMAND "confirm bookmark"
  COMMAND "confirm alias"
  STATE "Train Words"
  COMMAND "confirm to emacs"
  STATE "Training"
  COMMAND "okay"
MENU "EMACS"

global:
  STATE "emacs"
  COMMAND "end keyboard macro"
  COMMAND "<yes/no>"
  COMMAND "edit <modes> macro"
```

```

COMMAND "edit last <modes> macro"
COMMAND "new <modes> macro"
COMMAND "new <modes> command"
COMMAND "new <modes> statement"
COMMAND "new <modes> alias"
COMMAND "new bookmark"
COMMAND "goto <bookmarks>"
COMMAND "goto <bookmarks> next"
COMMAND "show <bookmarks>"
COMMAND "goto <other places>"
COMMAND "goto <other places> next"
COMMAND "show <other places>"

```

```

COMMAND "<commands>"
COMMAND "prefix <number1/30> <commands>"
COMMAND "<commands> <number1/30>"
COMMAND "describe <commands>"

```

semi-modes:

```

COMMAND "<..commands>"
COMMAND "prefix <number1/30> <..commands>"
COMMAND "<..commands> <number1/30>"
COMMAND "describe <..commands>"

```

etc.

modes (major and minor):

```

STATE "dired-mode"
COMMAND "<statement> statement"
COMMAND "<commands>"
COMMAND "prefix <number1/30> <commands>"
COMMAND "<commands> <number1/30>"
COMMAND "describe <commands>"
STATE "...-mode"
COMMAND "<commands>"
COMMAND "prefix <number1/30> <commands>"
COMMAND "<commands> <number1/30>"
COMMAND "describe <commands>"

```

...
etc.

Calling Emacs commands

The most important commands in each Emacs-context are:

```

COMMAND "<commands>"
COMMAND "prefix <number1/30> <commands>"
COMMAND "<commands> <number1/30>"
COMMAND "describe <commands>"

```

Here <commands> refers to the Emacs commands available in the active context. The context STATE "emacs" is always active, and thus contains all global Emacs commands. The contexts STATE "...-mode" are only active when in ...-mode. Semi-mode commands refer to groups of global commands with the same prefix of which it is preferable to strip off this common prefix. An example of a semi-mode are the commands defined for voicegrip (not standard in Emacs but presumably present if you use programming by voice). The command like "vg translate line" is better off without the prefix "vg" leading to "translate line" (see ".../evc/command_doc .rtf"). After you've run the generator the directory ".../evc/documentation" contains the file

"command_doc.rtf", which lists all commands for all contexts (mode specific commands appear at the end of this file).

Examples:

The command "next-line" is in the list of global commands. This means you can invoke it by saying "next line". If you want to invoke it with an argument you can say either "prefix 10 next line" or "next line 10" (they have exactly the same effect). If you want to see the full documentation of the Emacs command "next-line" you can say "describe next line". Mode specific commands can only be called when its mode is active. For most of these commands the mode-specific prefix is stripped off (see ".../evc/documentation/command_doc.rtf"). For instance the Emacs command "dired-mark" can be called in dired-mode by saying "mark". Marking more than one file in dired-mode can be done by saying either "prefix 5 mark" or "mark 5". The full documentation of "dired-mark" can only be retrieved directly by voice when dired-mode is active. To do this you say "describe mark".

New commands

Although new interactive Emacs commands are automatically included when the generator is run, you'll often want to use your newly defined interactive Emacs Lisp command straight away without having to run the generator first (and merging global.dvc files etc.). To make your interactive command available straight away you can say "new <modes> command"¹, which takes you straight to the right spot in the command Wizard². Here you can insert your command. You should, however, take care to use spaces instead of hyphens, otherwise your command will not be recognized. When you've inserted your command (keep the cursor on the same line) you say "confirm to emacs" which takes you straight back to emacs.

Examples:

Let's say you define a new command:

```
(defun do-something ()  
  (interactive)  
  ....)
```

After you've evaluated this command (for instance using "eval defun") you say "new Emacs command" (taking you to the list <commands> for state "emacs"). Here you insert "do something" and then say "confirm to emacs" and your new command is available.

Now let's say you define a new command:

```
(defun dired-do-something-else ()  
  (interactive)  
  ....)
```

After evaluating you say "new dired command" (taking you to list <commands> for state "dired"). Now you insert "do something else" and then say "confirm to emacs" and your new mode-specific command is available in dired-mode.

Aliases

SINGLE ALIASES

¹ For <modes> any of the following are available by default:

"emacs", "hs", "highlight changes", "view", "buffer menu", "apropos", "c", "c++", "emacs lisp", "info", "tex", "latex", "lisp interaction", "cperl", "perl", "edit abbrevs", "help", "lisp", "bookmark bmenu", "dired", "speedbar", "comint", "occur", "picture", "rmail", "shell", "gud", "else".

If you added some extra mode yourself in ".../evc/mode-list.el", it will also be available as a substitution for < modes>.

² When you invoke this macro for the first time in your session it sometimes happens (due to time delays) that you end up in the macro script of 'new <modes> command'. You can then just say "cancel to Emacs", and try again.

Often you won't like the pronunciation of a standard Emacs command. In these cases you can create an Emacs alias for that command. An alias is a standard Emacs concept. Aliases created by this package are kept in the file ".../evc/aliases.el". To make the process of creating aliases as smooth as possible (within the boundaries of this environment) I've created two macros.

```
COMMAND "new <modes> alias"  
COMMAND "confirm alias"
```

If you want to create an alias for a command you say "new <modes> alias" taking you to the right spot in the command Wizard. Here you can insert your alias (use spaces instead of hyphens!) (Keep the cursor on the same line) and then say "confirm alias" taking you back to emacs and prompting you for which command you want the alias to be defined. Insert this command and press Enter (or say "new line"). A new line will be added to the file ".../evc/aliases.el".

Examples:

You don't like the command "shrink-window-if-larger-than-buffer" and want to give it an alias "fit": say "new Emacs alias", insert "fit", say "confirm alias" and insert "shrink-window-if-larger-than-buffer" (you can use auto completion) and then you say "new line". Now your new alias is active and the following line has been added to the file ".../evc/aliases.el":

```
(defalias 'fit 'shrink-window-if-larger-than-buffer)
```

Now let's say you don't like the command "dired-find-file" (dired- is not stripped off because than the command would collide with the global command "find-file");

say "new dired alias", insert "go", say "confirm alias" and insert "dired-find-file" and say "new line".

The following line has been added to ".../evc/aliases.el":

```
(defalias 'dired-go 'dired-find-file)
```

A number of Emacs interactive commands contain short hands for typing, which are awkward to pronounce (e.g. "start-kbd-macro"). Although it still possible to pronounce them it is better to create an alias (e.g. "start-keyboard-macro").

MULTIPLE ALIASES

Sometimes you'll dislike the pronunciation of a component of a command that occurs in many other commands as well. For instance I don't like pronouncing the component "other window" in many commands. I'd rather say "next" instead (so I'd rather say "scroll next" than "scroll other window"). To create aliases for all commands containing "other window" simply say (or invoke using {Alt+x} etc.) "create aliases for"³. Then type "other-window", press enter, type "next", and press enter. This will create aliases in the file aliases.el for all commands containing "other-window". The new commands (aliases), however, are not available to NaturallySpeaking until you run the generator again and merge global.dvc files.

Bookmarks

Because bookmarks (in Emacs a bookmark is a way of remembering a certain location in a certain file) can be extremely useful for navigation by voice they've been hard coded in the framework. To create a bookmark in a specific spot in a file say "new bookmark", taking you to the right spot in the command Wizard. Here you insert your bookmark (no hyphens!) (Keep the cursor on the line where you inserted the bookmark), after which you say "confirm bookmark".

You can now say:

"goto <bookmarks>"	meaning select bookmark <bookmarks> in the current window
"goto <bookmarks> next"	meaning select bookmark <bookmarks> in another window
"show <bookmarks>"	meaning display bookmark <bookmarks> in another window, but don't select it

Examples:

³ it is a good idea to run the generator once before creating multiple aliases, because many mode specific commands may not have been loaded by emacs. This process only creates aliases for commands that are loaded.

In my file ".../.emacs" I want to create the bookmark called "login". First I issue the command "find file", and then type/dictate ".../.emacs" (or use some other method for visiting my .emacs-file). I go to the spot in this file that I want to remember. I say "new bookmark", insert "login" and say "confirm bookmark". If after moving around for a while I'd like to get back to this specific spot I can say either "goto login", "goto login next" or "show login". The bookmarks are usually saved across Emacs sessions, so even after I shut down Emacs a couple of times I can still issue these commands. Furthermore the bookmarks defined in Emacs will also be included in the generated global.dvc file by the generator.

Moving to standard buffers

Because, unfortunately, it is not possible to create a bookmark for a standard buffer in Emacs I decided to hard code some standard buffers to behave like bookmarks. This hard coding is represented by the macros:

```
COMMAND "goto <other places>"
COMMAND "goto <other places> next"
COMMAND "show <other places>"
```

in which <other places> can be any of: "Messages", "scratch", "Apropos", "Help" or "Buffer List".

Creating, editing and preserving NatSpeak macros

Although it is a good idea to implement functionality in Emacs-Lisp instead of Dragon macro language, many people still have a lot all of the latter or still prefer to create their own macros alongside Emacs-Lisp functionality. This package provides that option. To keep these Dragon macros reasonably separated from the generated functionality I've created states like "@...-mode" and a state "emacs@". I've also make some macros to quickly access these areas:

```
COMMAND "edit <modes> macro"
COMMAND "edit last <modes> macro"
COMMAND "new <modes> macro"
```

If you already have a large set of NatSpeak macros which you want to keep using, just cut them out of their or original place in the global.dvc file (in "C:\...\NatSpeak\users\<>current") and paste them in the part (in the global.dvc file): "STATE "emacs@". If you have any commands that will cause a name clash with the command of the framework (which is quite likely) it is a bit ambiguous which of the two NaturallySpeaking will take. Usually this will not be a problem because they will not only have the same name, but also do the same thing. But if at a certain stage things don't behave the way you want, remember this could be where the problem is.

Examples:

Saying "new Emacs macro" takes you to the command Wizard and then to the application "EMACS" and state "emacs@", where you can define your macro. After you're finished defining it just say "confirm to emacs", which takes you straight back to emacs. You can even define mode-specific (e.g. dired) macros by saying "new dired macro". A macro you define here will only be active in dired mode. I assume that the commands "edit <modes> macro" and "edit last <modes> macro" speak for themselves in this context.

Miscellaneous hard coded macros

```
COMMAND "end keyboard macro" : called directly via its key binding ("{Ctrl+x}")
COMMAND "<yes/no>"           : efficiently answer yes/no questions in the mini buffer
COMMAND "cancel to emacs"   : cancel the command Wizard and move focus to emacs
COMMAND "train word"        : activate the train word pop-up from anywhere.
COMMAND "confirm to emacs": click done after training words and move focus to emacs
```

Else functionality.

because I like "else" (<http://members.xoom.com/pmilliken/>) very much I've included some functionality for using it in Emacs voice Commander. This won't disturb the normal functioning of the package, so you won't be bothered if I didn't manage to convince you that else is great ;) A global macro "new <modes> statement" (e.g. "new Emacs lisp statement") is available to enter a new programming mode specific statement that can be automatically expanded. When I say for instance "new Emacs lisp statement" this takes me straight to the list <statement> used by the macro "<statement> statement" in Emacs lisp context. The statements in the list <statement> correspond to tokens defined in the file Emacs-lisp.lse in the else package. Unfortunately I don't think it's worth at this stage to generically make the generator fill this list from the else language files. Therefore you'll have to fill it your self. Subsequent global.dvc generations will not overwrite these definitions though.

Examples

I want to be able to say "defun statement" which inserts an else-defun-statement-construct:

```
(defun {identifier} ([defun_arguments]
 [Documentation]
 [interactive]
 {statement}...
 )
```

First-time make sure there's a token:

```
DELETE TOKEN DEFUN -
 /LANGUAGE="EMACS-LISP" -
 DEFINE TOKEN DEFUN -
 /LANGUAGE="EMACS-LISP" -
 /PLACEHOLDER=DEFUN
```

END DEFINE

in the file Emacs-lisp.lse in the else package.

Then I say "new Emacs lisp statement" and insert "defun" (without the quotes) after which I say "confirm to Emacs".

When I'm in Emacs-lisp-mode I can now say "defun statement" which has the desired effect.

Adding Emacs functionality

One of the main advantages of this package is that it's generic. This means that any new Emacs functionality you add (e.g. using (load...)) is simply speech enabled by re-running the generator and merging the generated global.dvc file to the one you are using. Any interactive command defined in your new Emacs Lisp file is automatically added to list of global speech commands. Sometimes, however, you'd like to add a new minor or major mode (or even a semi-mode). Emacs has no intrinsic mechanism to determine which mode a command belongs to. In Emacs any interactive command is a global command and can be activated anytime (using [M-x] ... {Enter}). Only key bindings are mode aware because a mode-map is activated upon mode activation. Now that we want to activate commands by voice (taking over the function of the key bindings) we do want them to be mode aware.

Fortunately (almost) all mode specific commands obey a convention: they start with a mode specific prefix. Unfortunately there's no internal list of mode specific prefixes, so we'll have to construct it ourselves. There is also some other information we need to provide to the generator to make mode-aware commands work correctly. All this information is contained in the file ".../evc/mode-list.el". To add a mode yourself you'll have to work your way through this file, following its instructions (it's quite heavily commented).

Training commands that are not recognized

NaturallySpeaking doesn't have a very pretty interface for training commands. Nevertheless you'll have to use it sometimes. Therefore I've included a global macro "train word" to go straight to the NaturallySpeaking train words pop-up.

When a command is not recognized the first thing to do is just try again pressing the Ctrl key. If a certain command keeps giving problems you can say "train word", type the command and train it. It often occurs however, that a certain component of a multiword command keeps giving problems. In my case for instance the words "other" and "up" were a real pain. If I said "other window" it would persistently type "although window". This problem can be solved by just training the component "other". You can say "train word", then type "other", and train it. This will help a great deal in all commands that contain the component "other".

Training your self for using the framework

'Just say what you want to do and the framework does the right thing!', 'no need to memorize a large set of commands, just say it your way!'. These types of claims are made by vendors of commercially available speech recognition products. Fortunately this is not a commercial product so there is no need to twist the truth. Emacs has very powerful editing capabilities. To access all that functionality a large command set is needed, and it takes (a lot of) time to learn how to use it. It definitely is necessary to memorize a large set of commands to be able to work smoothly and to benefit from Emacs' power.

If you're not used to Emacs and want to learn how to use it using speech recognition it is advisable to read a good book about it (e.g. learning GNU Emacs from O'Reilly) that always mentions both key bindings and command names (in case it doesn't you can just use 'describe key' to display the command name). It's not absolutely necessary but it will make you understand the big picture quicker than by reading the documentation of individual commands.

If you learn to use Emacs without speech recognition you'll have to learn a large set of key bindings. The good news is that Emacs commands will have much more meaning to you than key bindings, and therefore are much easier to memorize. Even better news is that they're all well-documented because Emacs is 'self documenting'.

After running the generator the file ".../evc/command_doc.rtf" contains all speech enabled commands. It's very useful to print this file. To grasp what is available just go through this file testing each command (or just display its full documentation) that seems useful to you. If it's useful indeed, mark it. After you've marked all useful commands you can start learning them by heart. Don't mark too many commands as useful, because you'll never be able to memorize all of them.

To retrieve the documentation of a single command just say 'describe <individual command>'.

Note, however, that you cannot say things like 'describe dired-unmark'. Instead you'll have to say 'describe unmark', because the dired- part is stripped off in the dired command list. Furthermore you can only get the documentation using this voice macro when you're in dired mode, because the list containing 'unmark' (<commands>) is only available in dired mode.

During the process of learning commands by heart you'll notice that you dislike the pronunciation of some commands. The answer is: create aliases. The framework offers macros for creating aliases. Your user aliases will be activated immediately, and will be stored in the file ".../evc/aliases.el". If you create an alias 'this' for the command 'delete-other-windows' you can get the documentation for 'this' by simply saying 'describe this'. Automatically the documentation string of delete-other-windows will appear! So don't be afraid to create many aliases. Use short aliases for commands you use often.

My preferences

In the directory .../EVC/personal you can find two files: removables.txt and aliases.el. If you want to use my settings you can copy those files to the .../EVC directory overwriting the standard files. If you can generate a global.dvc file you'll just get a subset of commands I use speech enabled plus you get my aliases. This results in a much shorter commands_doc.rtf file, but you get my preferences, which might not be what you want, but maybe it is :).

Implementation

Discussion of the files in the package

In most cases you won't need to read this chapter. Only if something goes wrong this is where you'll find clues to fixing it. Furthermore if you're interested in changing the design (or even better if you're making an SAPI implementation) this is where you should be.

Directory ".../evc/documentation":

user_manual.pdf:

this document in Acrobat format.

Command_doc.rtf:

generated documentation for all Emacs interactive commands. Unfortunately it contains no table of contents, but all mode-specific commands are at the end of this file.

Directory ".../evc":

.emacs:

contains the code to be added to your own .emacs file.

voice-commander.el:

loads the necessary files into Emacs. This file should be loaded from your .emacs file.

aliases.el:

contains aliases for Emacs interactive commands.

mode-list.el:

sets up a list of symbols representing all semi, major and minor modes. Some properties are added to the symbols (using (put ...)). These properties are used for properly inserting modes in the frame title (in the file frametitle.el) and for properly generating the global.dvc file (in a mode aware fashion). This file is necessary because not all modes obey the same convention and because Emacs stores no intrinsic information about the mode awareness of commands. If you add a mode yourself you'll have to update this file. Instructions for doing so are in the file itself.

generator.el:

the central file for global.dvc generation. It contains the command generate-command-file. Hopefully (for you) you won't have to go deeply into this file. It's been subject to a lot of evolution so it will take quite some time to figure out how it works. If you need to anyway (e.g. because you want to steel some code because you're making an SAPI implementation!), feel free to contact me. I might insert some more comments.

generator_utils.el:

all kinds of utilities used by generator.el which I've split out of that file to make it a little smaller (and less scary).

frametitle.el:

this file puts the active major and minor modes into the frame title. This frame title is used by NaturallySpeaking to activate certain "states" (i.e. contexts). Some modes are derived from other modes. For example emacs-Lisp-mode is derived from Lisp-mode. This means that whenever emacs-Lisp-mode is active the commands all of Lisp-mode should also be activated. This is achieved by setting the "subset" property for the symbol emacs-Lisp to a list containing the symbol Lisp in the file ".../evc/mode-list.el". This property is subsequently used in the file frametitle.el to also include the parent mode in the frame title.

Directory ".../evc/templates":

this directory contains all templates used by the generator to construct a global.dvc file, the command_doc.rtf file and the files global_but_mode_prefix.txt and removables.txt. Let's start with the latter.

removables.txt:

it contains the first characters of commands that don't need to be reflected as speech enabled commands. For instance the entry "ethio" achieves that all commands with prefix "ethio" (because I don't need any ethiopean functionality in my editor) will not be reflected in the

global.dvc file or command_doc.rtf file. Please check out this file to see if it doesn't remove commands you want to use. For instance I don't use Emacs for email. If you do you'll have to remove the entry "mail". Instead of removing groups of commands which have to saying prefix this file can of course also be used to remove single commands from consideration.

global_but_mode_prefix.txt:

some commands have a mode-specific prefix but should still be a global command (e.g. dired-jump or view-buffer). These commands are listed in this file.

Global.dvc generation.

```

Mainframe.txt
{
MENU "Global Commands"
...
MENU "NATSPEAK"
...
MENU "EMACS"
  <<global>> => global_macro_template.txt
  {
    STATE "emacs"
    <<globals>> => generic_group_template.txt
    {
      <<groupspecific>> =>
      specific_global_macros.txt
      ...
    }
    <<semi-modes>> => n *
    {
      generic_group_template.txt
      {
        << groupspecific>> =>
        specific_semi_mode_macros.t
        xt
        ...
      }
    }
  }
  <<modes>> => m *
  {
    generic_group_template.txt
    {
      <<groupspecific>> => specific_mode_macros.txt
      {
        STATE "<<mode>>-mode"
        ...
      }
      ...<<modestub>> => mode_stub.txt
    }
  }
  <<user-macros>> => (m+1) *
  {
    user_mode_macro_template.txt
  }
}

```

While tags (<<...>>) are replaced by template files, tags in those template files are replaced by stuff the generator extracts from Emacs. The scheme sketched above is therefore not extensive but only sketches the global perspective.

Command_doc.rtf generation

```
documentation_mainframe .txt
{
  ...
  <<global>> => global_docu_template.txt
  {
    <<globals>> => global_mode_docu_template.txt
    {
      x * normalrtftemplate.txt
    }
    <<semi-modes>> => n *
    semi_mode_docu_template.txt
    {
      y * normalrtftemplate.txt
    }
  }
  <<modes>> => m *
  mode_docu_template.txt
  {
    z * normalrtftemplate.txt
  }
  ...
}
```

Things to do

Emacs window configuration support

Soon I'll release some extra window configuration support for Emacs. When I have that ready I'll also incorporate support for it in Emacs voice Commander.

SAPI implementation

The semi-generic implementation I used here is limited: it is not dynamic and quite inflexible. This is due to the fact that after running the generator we have to close down NaturallySpeaking, merge global.dvc files and then restart NaturallySpeaking to make new functionality available for speech. Furthermore the limitations of Dragon scripting language prevent the use of global variables and code reuse. I've simulated code reuse by code replication, but it would be a bad idea to continue on that road.

A further disadvantage of using Dragon scripting language is that it ties this solution to Dragon NaturallySpeaking professional. Creating an SAPI implementation could resolve all of these problems. When Barry Jaspan released VR mode in November 1999 he said he was working on something like that.

The framework as sketched above could be seen as an initial default. Ideally, however, we'd like to be able to say things like 'next <number> lines', '<color> background' or 'jump to <tag>'. I've already hard coded 'goto <bookmark>', because it is extremely useful for now. Furthermore you can only say: '<any command> <number>' to give a number as an argument to an arbitrary command. As useful as they are, they are inflexible. Therefore I propose to introduce a number of global alists in Emacs that can serve as arguments (like 'color', 'tags' or 'number'). Furthermore I'd like to give any command that should behave differently than default ('<any command> <number>') a property (put 'AnyCommand' 'SpeechStructure ...') defining how it can be used as a speech command. The speech structure could be a list defining arguments and possible positions for these arguments within the command. An SAPI implementation could read out SpeechStructure for each command (if it's available) and use it to appropriately insert the command into the speech engine. For an example of a command that could be implemented this way see the next section.

Mode awareness of programming modes

Although I've included some programming modes in this package, I'm not very experienced in this field. Most programming modes somehow derive from cc-mode (as I read in the info manual). I haven't studied this carefully though, so I've treated this subject very roughly. If you run into problems because of this, please contact me.

Cheers,

Hans van dam