Emacs Language Sensitive Editor (ELSE)

Peter Milliken[1]

---

[1] email contact:

`peterm@unwired.com.au`

`peterm@resmed.com.au`(my work address – which may change at a moments notice in these days of "outsourcing" jobs to other countries)

Table of Contents

Preface

This manual documents the use and creation/customisation of language templates for ELSE. There are two levels of reader assumed by this document; a user and a language template maintainer. The two are not mutually exclusive.

This manual is a mixture of user manual and reference manual. All readers are recommended to read the Tutorial at the end of the manual (See Chapter 13 [Tutorial], page 44). In addition, complete beginners to ELSE are recommended to read the first six chapters (Overview — Using ELSE) and then the chapter on Customising Variables (but only after they have become familiar with ELSE). The rest of the manual is for the more adventurous who wish to either customise existing ELSE templates or create their own templates for any language that ELSE does not currently support (please make them available to the wider community if you do).

If you dislike reading documentation then just focus on the sections for installing ELSE (See Chapter 3 [Installation Instructions], page 8,) and the Tutorial Chapter 13 [Tutorial], page 44. Once you have satisfied your initial curiosity about ELSE then you can come back and read the other portions of the manual. ELSE is an extremely powerful, but simple concept, so if you decide to persevere with using ELSE then you will sooner or later find yourself reading the entire manual. Much of ELSE's power comes from the ability for the user to customise the language templates for his/her own programming style.

1 Introduction

These days a number of packages exist for Emacs that provide what users think of as "templates" or "skeletons" or code "abbreviations". When they first go in search of something to help generate code they are thinking of something that will allow the generation of common code constructs (usually control structures or file/function headers) with a small number of keystrokes (similar to the Emacs abbreviation facility). For instance, a user might be interested in a package that generates an "if" statement, they envisage such as package generally working like an abbreviation i.e. "if<some Emacs command/key sequence>" which then leaves a tailored "if" construct in their buffer, where they can then "fill in the blank" or missing portions of the code.

ELSE (**E**macs **L**anguage **S**ensitive **E**ditor) provides this functionality via what is called "tokens" (see Section 6.4 [Expanding Tokens], page 13) plus much more. The following example shows the use of a token definition taken from the C language templates, this definition generates an "if" construct. The key–stroke sequence to generate this command (assumes installation of ELSE has been achieved) is `ifC-c / e` with the following result in the buffer:

```
if ({expression}) {
  {statement}...
}
[elsif_part]...
[else_part]
```

After the expansion, the cursor is automatically positioned within the first placeholder (`{expression}` and the user can just start typing the condition expression, ELSE will automatically delete the placeholder text and replace it with the typed information.

ELSE takes what I will term a "whole language" approach to helping generate code. Not only does it provide code "abbreviations" but it also provides facilities that allow generation of a code file from initial empty buffer to completed file - all without having to leave the template system. Indeed, it is possible to code a typical language file without having to manually type a single language keyword or construct. This "whole language" approach utilises what is termed "placeholders" (see Section 6.3 [Expanding Placeholders], page 12). Given a sufficiently detailed language template file (one that closely mirrors the EBNF[1] for the language, for instance), then it is possible to achieve this goal.

A brief tutorial is offered (see Chapter 13 [Tutorial], page 44) showing the use of both placeholders and tokens.

Most people, when they start to explore language sensitive editing, start out by looking for a system that allows them to specify language "templates" — a facility that will allow them to type in an abbreviation which is then somehow expanded to the full language construct i.e. 'if' is expanded to the full syntax of the particular language if statement. This is as far as they typically envisage a templating package to work. ELSE provides this basic functionality via what is termed "tokens". But ELSE goes much further than this, it provides a system that allows the mimicking of the entire language syntax i.e. it offers not only expansion of abbreviations but also allows menu choices and the provision of information messages to the user — all within the syntax of an ASCII readable language definition file. The structure of the language definition file is described fully in the reference portions of this manual (Chapter 7 [Creating and Modifying Templates], page 15).

You might be asking the question - how is ELSE different from any of the other template packages available for Emacs? ELSE differs from other packages that aim to achieve the same goals in several ways:

---

[1] Extended Backus–Naur Form is a mechanism used by computer scientists to describe the syntax of a computer language i.e. check section A.13 of Kernighan and Ritchie's "White book" on the C language

1. ELSE language template definitions are specified using a purely ASCII, textural syntax with no attempt at making them look ELisp-like. At least one user has seen this as a disadvantage though :-). To be fair, his argument was that why should he have to learn yet another "language" when if the template definitions could be defined using Elisp style syntax? There is no easy answer for this. As the author, I (obviously) thought that staying away from Elisp like syntax would be a benefit to a user i.e. they don't have to learn Elisp to use ELSE! At first glance, ELisp is not an easy language to pick up with its nested brackets etc - miss one and you are history! :-).

2. ELSE language definition syntax conforms with a commercially available Editor[2].

3. The user interface for using ELSE is completely visual i.e. other template/skeleton packages use an interface of questions/answers and have invisible "markers" to points of interest in the syntax generated. Thus their interface is not easy to use (once you embark into a Q&A session you are stuck with completing it - many such sessions require you to know the full extent of your code before you embark on it! e.g. how many case statements will be used in a switch statement and exactly what are the cases in which the program should take action? If you need further information to answer any of the questions then there is no way to just "interrupt" the season, switch to another buffer and look up the required information — so you are either forced to continue on as best you can or abandon the Q&A session entirely). ELSE's placeholders or "markers" persist across edit sessions because they are actual text in the buffer, other template/skeleton packages use Emacs markers which are lost when the edit session for that file is ended.

4. ELSE offers features that are missing in other packages e.g. the ability to have text automatically repeated as you type at a single place in multiple places within the language syntax i.e. function names can be repeated at the end of the function body as the name is typed by the user at the start of the body[3]. It is possible to create flexible templates for commonly used language constructs using this feature i.e. coding for loops in C often have a similar, repetitive "pattern", using this feature of ELSE, typing can be kept to a minimum as the variable name is repeated in multiple places e.g.

   ```
   for (no_items = 0; no_items < 100; no_items++)
   ```

   is a very common construct - the auto-substitute feature of ELSE could allow the user to create the above code pattern such that the variable "no_items" was automatically repeated in two places as the first instance was typed by the user.

5. ELSE language definitions can model the original language syntax either as loosely or closely as the template definition maintainer desires using a menu system that provides choices that follow language syntax branching. Thus a user can see what code syntax is possible at different points in the file without necessarily having to be an expert in the language in which the code is being generated. This means that ELSE can be a aid to beginning programmers as they learn the language in which they are generating code.

6. ELSE offers a more "dynamic" interface than other similar packages. The user can "extract" and modify language template definitions "on the fly" whilst in an Emacs edit session. These changes can be either temporary i.e. last for the current edit session only or can be saved for future use in later sessions. In fact, ELSE is so dynamic that you can actually extract the contents of the entire language template set for the current edit session and save that for later (re)use.

7. And lastly (but not least :-)), ELSE is extensively documented. Documentation for competitive packages are only just starting to emerge, use of those packages have thus been

---

[2] well, it is as close as my memory allowed when I moved from using DEC LSE and creating a mimic of it — users have reported very minor syntactical differences that are easily catered for by editing the template file

[3] this is mandatory for some languages, also some project coding standards demand a repetition of information that is found at the head of construct to be repeated at the end of the construct i.e. copying the expression portion of an if statement into a comment at the end of the if structure

limited to those few who were prepared to invest the time to read Elisp and a few scattered examples to learn how to use the packages. ELSE template definition syntax and documentation hopefully allows even users who are not interested in programming in Elisp to enjoy the fruits of language sensitive editing quickly and (relatively) easily.

2 Overview of ELSE

ELSE is an implementation of a minor mode for Emacs that provides language sensitive editing capability to the currently enabled major mode (for the current buffer) of Emacs. It is aimed fairly and squarely at providing support to programmers for input of program text but as you will see it can be easily customised for any task that involves repetitious input of common textual sequences.

Use of ELSE will improve programmer productivity by reducing the amount of time to enter the program/text in the first instance and in the second instance, it will cut-down on time consuming errors due to typing mistakes i.e. syntax errors that are usually found by the compiler during the compile - edit cycle e.g. missing ;'s become a thing of the past.

The aim of ELSE is to reduce programming to an exercise of "filling in the blanks", hence the use of the term *template* language. Language constructs are the templates and the variable, procedure and function names are the "blanks".

There are a number of ways of implementing so called "language sensitive editing" (lse) in an editor. The most common approaches seen in many editors that offer this feature take a very "primitive" form where the programmer is left feeling that he/she could just as easily do without e.g., many implementations offer the following behaviour:

1. offer macros that would fill in some of the language syntax by straight generation of a portion of the chosen language statement leaving the user to position the cursor manually to where variable entry is required and manual deletion of (optional) portions of the syntax that are not required; or

2. through a series of question/answer sessions, where at the end of the sequence, the language statement is deemed "complete" and is entered into the text buffer.

These approaches tend to feel fairly "intrusive" to the programmer and are generally awkward to use. In addition, they offer little or no help to the novice in possible selections of syntax.

The approach to Language Sensitive Editing offered here for Emacs is modelled after a feature found in a editor offered by Digital Equipment Corporation[1] called LSE. This approach to the problem does not suffer from any of the usual awkwardness associated with typical lse implementations. It provides a natural framework for the programmer where the work is limited to menu selections and "filling in the blanks" e.g., a typical *template* (see Section 2.2 [Definitions], page 7) in ELSE looks like:

```
[context_clause]...
package {program_unit_name} is
  {basic_declarative_item}...
[private_part]
end [program_unit_name];
```

This is the template for a `package` specification in Ada. Each of the textual strings inclosed by '`{}`''s or '`[]`''s are language *placeholders* (see Section 2.2 [Definitions], page 7) that offer further expansion possibilities through either menu selection, text substitution or language prompts e.g., if the placeholder '`[context_clause]`' is *expanded* then the contents of the buffer will become:

```
with {library_unit_name}...; [use_clause]
[context_clause]...
package {program_unit_name} is
  {basic_declarative_item}...
[private_part]
end [program_unit_name];
```

---

[1] DEC has been purchased by Compaq, which has in turn been purchased by Hewlett-Packard, but the product lines lives on AFAIK

Observe that the placeholder '`[context_clause]...`' has been expanded and replaced with the two lines:

```
with {library_unit_name}...; [use_clause]
[context_clause]...
```

Upon expansion, ELSE replaced the placeholder `context_clause` with its definition, namely the text `with {library_unit_name}...; [use_clause]`. It detected that the placeholder was to be repeated (the trailing ...) and thus duplicated the placeholder being expanded onto the next line.

After the expansion, the cursor will be automatically re-positioned between the first set of '`{}`''s. The user then has options of further expansion or performing straight text entry, if the text entry option is exercised then the text within and including the '`{}`''s is automatically deleted by the minor mode and replaced by the entered text i.e. no awkward "killing" or "deleting" of text is required, the minor mode recognises a valid[2] placeholder and responds appropriately e.g.,

```
with TEXT_IO, [library_unit_name]...; [use_clause]...
```

Note that the user has only typed the text `TEXT_IO`, the ',' and the repetition of the (optional) syntax/placeholder `[library_unit_name]...` was automatically supplied by the minor mode functionality. Just as 'conveniently', if the programmer decides that a second `library_unit_name` is not required then the placeholder can be deleted (`else-expand-placeholder` (*C-c / e*)) using a single command with the following to result:

```
with TEXT_IO; [use_clause]...
```

Note that ELSE has performed appropriate "housekeeping" and that the ',' has been automatically deleted as nolonger required and the ';' character is flush against the package name (TEXT_IO). Similarly, the `[use_clause]` placeholder can be expanded upon or deleted.

The preceeding examples have shown how ELSE offers language sensitive editing via the *placeholder* mechanism. Another mechanism offered by ELSE for quick generation of language syntax is via the expansion of *token's*. A token is usually used where no placeholders are available i.e. the programmer has deleted all placeholders but wants to add further code to a particular section. He/She has the option of either typing in a placeholder, such as, `{statement}...` or if a simple construct is all that is needed, then he/she can type in a token and perform expansion upon it. A *token* is an abbreviated string that has been defined in the language templates to be expanded to a full construct e.g.

Code before entry `with the ''token'' inserted at the desired point:`

```
Value1 := 10;
Value2 := 20;

if                      <---- require an ''if'' statement here

Value3 := Value1 * Value2;
```

----

[2] valid placeholder or token strings are defined as a string that exists as a definition in the currently enabled language definitions

Code after token is expanded :

```
Value1 := 10;
Value2 := 20;

if {condition} then
  {statement}...
[elsif_part]
[else_part]
end if;

Value3 := Value1 * Value2;
```

In the previous example, the abbreviation "if" is a token defined in the Ada language templates which is defined to expand to an "if statement" template.

2.1 Typographical Conventions

ELSE uses the following typographical conventions:

1. Curly braces '{}''s — denote a mandatory entry e.g. `with {library_unit_name}`, the language requires that a package name be supplied. Mandatory entries cannot be deleted (using the `else-kill-placeholder` (`C-c / k`) command, that is), the user is warned by a error message in the command line of the editor.

2. Square braces '[]''s — denote an optional entry e.g. `[context_clause]` can be either deleted or 'expanded'.

3. 3 Dots ... — denote the fact that the preceeding (or 'attached') placeholder will be 'automatically' repeated by the minor mode functionality when the user performs any actions other than the `else-kill-placeholder` (`C-c / k`) command.

4. ↦ — denotes the results of an *expansion* of either a placeholder or token e.g.

   `[context_clause]... ↦ with {library_unit_name}...; [use_clause]`

2.2 Definitions

The following terminology is used in this manual:

1. '`placeholder`' — Term used to denote a textual string that is recognisable or *defined* in the currently selected 'language' mode. The string is enclosed by either '[]''s or '{}''s (see Section 2.1 [Typographical Conventions], page 7).

2. '`token`' — A (usually short) textual string that has been *defined* in the currently active language template. It can be expanded to provide a language template. This is (usually) used as a shorthand way of inserting a particular language construct. Tokens are textual strings which are not enclosed by '[]''s or '{}''s.

3. '`expand`' or '`expansion`' — Denotes the execution of the command '`else-expand-placeholder`'.

3 Installation Instructions

To install ELSE, copy the Emacs Lisp file 'else-mode.el' anywhere into the load path of your installation of Emacs (I use the 'site-lisp' directory). Place the following command into your .emacs file:

```
(require 'else-mode)
```

ELSE also optionally supports the use of a support package called setnu.el (mirrored/available from the same place that you copied else-mode.el). This package is written by Kyle E. Jones and provides line numbering support (if enabled - See Chapter 8 [Custom Variables], page 37.) to the display of menu choices. This feature is available so that individuals using voice coding systems can easily pick the desired menu choice. The setnu.el package should be copied to the same spot as else-mode.el.

It is recommended, but not necessary, that you install the ELSE info documentation. ELSE documentation consists of a TexInfo file (else.texi), an info file (else.info) and a PDF file (else.pdf). Copy the info file (else.info) into the Emacs Info directory and add the following line to the 'dir' file that can be found in the Emacs info directory:

```
* ELSE: (else.info).     Emacs Language Sensitive Editor.
```

ELSE comes with a number of template definition files (see Chapter 12 [Template Availability], page 43). Place the desired template definition files anywhere in the Emacs 'load-path' (the site-lisp directory is fine).

4 Default Keybindings

Following the recommendations of the Elisp manual regarding minor modes, ELSE provides a minor-mode map i.e. a map that is active only when the minor mode is active, that binds the four main commands of ELSE as follows:

1. else-expand-placeholder - `C-c / e`
2. else-next-placeholder - `C-c / n`
3. else-previous-placeholder - `C-c / p`
4. else-kill-placeholder - `C-c / k`

Some lessor used commands and their bindings are:

1. else-comment-placeholders - `C-c / c`
2. else-uncomment-placeholders - `C-c / u`
3. else-insert-placeholder - `C-c / i`

Note that these bindings are purely provided to conform with the conventions for such things as specified in the Emacs Lisp manual. My personal preference is to bind the main four commands to `F3 - F6`. A 'quirk' of the operation of ELSE that is worth mentioning is that when selecting items from the menu display, the command `else-expand-placeholder` can also be used to select an item i.e. first instance of the command over a placeholder will bring up a menu of choices and then a second press of the keybinding will provide a selection request - this provides faster and convenient selection for the user at times e.g. a commonly selected menu option can be a quick as `F3`/`F3` (assuming the desired option is the first entry in the menu — if it isn't then "customise" that placeholder definition and swap the order of menu item!).

For the convenience of novice/beginning Emacs programmers, here are the key definitions that I use (place them in your .emacs or emacs.el file — whichever one you use):

```
(global-set-key [f3] 'else-expand-placeholder)
(global-set-key [f4] 'else-next-placeholder)
(global-set-key [f5] 'else-previous-placeholder)
(global-set-key [f6] 'else-kill-placeholder)
```

A further "nice to have" in your .emacs file is to have else–mode turned on automatically for each file that you edit. An example of how to turn on ELSE for c-mode (C source files) is:

```
(add-hook 'c-mode-hook
          (lambda ()
             ;; this is shown as a lambda so you can add further interesting
             ;; minor mode definitions here.
             (else-mode)))
```

Refer to the Emacs manual for further information on major mode hooks, when and how they are run to achieve customisation of an edit environment.

5 Command Summary

The following user commands are provided by ELSE.

else-mode   *Command*
>   Toggles the minor mode for the current buffer. If the buffer is empty then it inserts the *initial_string*.

else-expand-placeholder   *Command*
>   If the cursor is positioned within a *placeholder* (see Section 2.2 [Definitions], page 7) then the placeholder is expanded[1] according to the rules for the definition of that placeholder name. If the command is executed with the cursor immediately positioned after a valid token then the rules defined for that token are used in the expansion.

else-next-placeholder   *Command*
>   Moves the cursor to the next valid placeholder in the current buffer.

else-previous-placeholder   *Command*
>   Moves the cursor to the previous valid placeholder in the current buffer.

else-kill-placeholder   *Command*
>   Kills or deletes the placeholder in which the cursor is currently positioned. Note that a numeric argument `C-u` will force a kill even when the placeholder is mandatory.

else-cleanup-placeholders   *Interactive Command*
>   Command to delete every placeholder remaining in the current buffer. Mandatory and optional placeholders are all deleted using the else-kill-placeholder command.

else-comment-placeholders   *Command*
>   Uses the comment syntax for the currently defined major mode to "comment out" any placeholders in the source file that have not been expanded or deleted. This command is useful for when a compilation is desired but coding is not yet deemed complete ie there are still placeholders in the buffer.

else-compile-buffer   *Command*
>   Command to "compile" the language definitions found at 'point' to the end of the buffer. When supplied with a numeric argument (`C-u`) will compile definitions from the beginning of the current buffer.

else-compile-fast-load   *Command*
>   Command to generate a "fast" load version of a language template file. A version of the language template file is created using the Emacs Lisp "read" syntax. When ELSE attempts to load a new language definition file it will first look for a "fast load" version of the file. Please note that this command is a "hold-over" from the days when PC's were very slow, this command probably should **not** be used as it will be removed from future versions of ELSE.

---

[1] Note that if the placeholder text within the '{}''s or '[]''s is not defined then the command will not recognise the placeholder string, this is a common error when the user has manually (mis-)typed a placeholder and can't work out why ELSE won't expand it.

else-extract-all   Interactive command

Extract all of the placeholders, tokens and the language definition for the current language into the current buffer at 'point'.

else-extract-placeholder *placeholder*   Interactive command

Prompts the user for a valid placeholder name and then extracts the placeholder definition into the current buffer (see Section 7.2 [Customising An Existing Template Language], page 15).

else-extract-token *token*   Interactive command

Prompts the user for a valid token name and then extracts the token definition into the current buffer (see Section 7.2 [Customising An Existing Template Language], page 15).

else-insert-placeholder   Interactive Command

Command to insert a placeholder string at point into the current buffer. It expects some leading characters of the placeholder prior to point at the time the command is invoked (these character must be preceeded by a `"{"` or `"["` character). It will use this abbreviation to perform auto-completion on the placeholders loaded for the current language template file. If there are more than one possibilities, then the command will provide auto-completion of the unique portion of the placeholder and then stop, a second invocation will split the window and display a list of possible completions. This second window must be deleted manually (in the current version - a future release will probably fix this problem).

else-move-n-placeholders   Interactive Command

Command to move to the "next" placeholder where "next" is controlled by the `else-direction` custom variable i.e. if `else-direction` is on then this command will invoked the `else-next-placeholder` (`C-c / n`) command, if the `else-direction` flag is off then it will invoke the `else-previous-placeholder` (`C-c / p`). This command was added for "usability" for VR Programming, it helps reduce the number of voice commands required to use ELSE.

else-show-token-names   Interactive Command

Display names of all of the Tokens in the current language template set, sort them alphabetically and display them in a temporary buffer.

else-show-placeholder-names *Display names of all of*   Interactive Command

the Placeholders in the current language template set, sort them alphabetically and display them in a temporary buffer.

else-toggle-direction   Interactive Command

Command to toggle the custom variable `else-direction`.

else-uncomment-placeholders   Command

This command will go through the current buffer looking for placeholders that have been "commented out" using the comment syntax of the current major mode. The language comment syntax will be removed.

else-wrap-region   Interactive Command

"Wrap" a template around a region of code. Mark the region to be enclosed by the template and then run this command, it will prompt for the placeholder name.

6 Using ELSE

ELSE has been implemented as a minor mode of Emacs. This means that each buffer within Emacs may have it own set of language templates enabled, the only limit is the system resources that is running the editor. So, Emacs may have multiple language definitions loaded at any one time i.e. the user may be interfacing an Ada program to a C program and so editing an Ada module in one buffer and a C file in another buffer, ELSE could be enabled for both buffers and they will each have the appropriate set of language templates enabled. The following sections detail how to start up ELSE and use it in the course of normal code entry.

6.1 Invoking ELSE

ELSE has been implemented as a minor mode of Emacs, it determines which language speci-fication to load either from the buffer's major mode or, if it can't locate an appropriate template file name (see Section 9.4 [Building the Language Template File Name], page 40), then it will prompt the user to enter the template name. For example, if the major mode for the current buffer is "C", then the major mode name will be "C" and ELSE will attempt to load the "C" template file using the name "C.lse" (see Section 9.4 [Building the Language Template File Name], page 40). Note that ELSE will first check if the template file for the major mode is already loaded, if not then it will search (see Section 9.4 [Building the Language Template File Name], page 40) for the file, if it can't find the file then it will prompt the user to enter a file name.

After loading the language template file, ELSE will then search for a "customisation" file for that particular language. The name of this file is of the form <language_name>-cust.lse i.e. in the case of loading a template file for the C language it would look for a customisation file called C-cust.lse located anywhere in the load path. Refer to see Section 7.2 [Customising An Existing Template Language], page 15 for more information on this feature.

To invoke ELSE use the command `else-mode`. If the current buffer is empty then ELSE will insert the *initial_string* for the language (see Chapter 7 [Creating and Modifying Templates], page 15) and position the cursor within the placeholder[1].

6.2 Navigating Using ELSE

Use the `else-next-placeholder` (`C-c / n`) and `else-previous-placeholder` (`C-c / p`) commands to navigate forwards and backwards through the buffer being edited. These com-mands will locate the next/previous valid placeholder and move point into the middle of the placeholder text. Each command can take a numeric argument, if the desired number of place-holders are not available, then point will be positioned to the last/first placeholder in the buffer.

6.3 Expanding Placeholders

Placeholders are expanded by positioning the cursor between the '{}''s or '[]''s and exe-cuting the `else-expand-placeholder` (`C-c / e`) command. If the placeholder string is a valid placeholder then ELSE will provide one of the following responses:

1. Replace the placeholder with a lower level language construct e.g.

    `[context_clause]` ↦ `with {library_unit_name}...; [use_clause]`

2. Provide a choice of possible lower level language constructs via a menu selection scheme e.g.

---

[1] The four primary else commands (else-expand-placeholder, else-next/previous-placeholder and else-kill-placeholder) each check whether else-mode is enabled and will invoke it automatically if not set

```
[type_declaration] ↦ full_type_declaration
                     incomplete_type_declaration
                     private_type_declaration
                     private_extension_declaration
```

3. Provide a prompt to the user that this is the lowest possible expansion and that the user must type in a replacement string i.e. expansion of the Ada placeholder {expression} will display the prompt:

```
Enter an Ada expression as defined in section 4.4 of the LRM,
for example:
4.0, Pi, (1 .. 10 => 0), Integer'Last, Sine(X), not Destroyed
Color'(Blue), 2*Line_count, -4.0 + A, B**2 - 4.0*A*C
```

If a placeholder is followed by ellipses (...) then ELSE will reproduce the placeholder automatically as the user either expands or types into a placeholder. The placeholder may be replicated in either the horizontal (same line) or vertical direction. The direction of replication (see Section 7.5.1.6 [Duplication Specifier], page 25) is determined in the definition of the placeholder.

6.4 Expanding Tokens

Tokens can best be thought of as a handy abbreviation for a placeholder that can be expanded e.g. the token *if*, when expanded, gives the language construct for the if statement (example taken from the C language templates but this is a very common token definition) i.e.

```
if ↦ if ({expression})
     {
         {statement}...
     }
     [else statement]
```

Tokens are most commonly used when the main portion of the code has been entered and the programmer/user is in the compile/fix-up cycle i.e. the placeholders have been removed from the source file to make it compilable but the programmer needs to add further statements. The options available at this point depends on the scope of the change required, if a major piece of code needs to be added then it is usually best to manually type in a placeholder that can be used as a good starting point e.g. [statement]... is a pretty common definition in almost any language definition template.

6.5 Deleting Placeholders

Use the command `else-kill-placeholder` (*C-c / k*) to delete a placeholder located under point. Note that if the placeholder is a mandatory entry (enclosed by '{}''s rather than '[]''s) then ELSE will issue an error message and ring the bell.

To delete all placeholders in a buffer, use the command `else-cleanup-placeholders`. This command will start at the beginning of the buffer and delete every placeholder (mandatory or optional) contained in the buffer. This is a convenient method of performing a "final" clean-up after completing the coding of an program file.

6.6 Preparing for Compilation

Often a user will wish to compile the current buffer even though code entry has not been completed. Two commands are provided for convenience during this phase of coding:

1. `else-comment-placeholders` — Comment all lines that contain placeholders, this uses the comment syntax defined by the current major mode. It uses the Emacs `comment-region` command to accomplish this process[2]

2. `else-uncomment-placeholders` — Uncomment all placeholders in the current buffer. This command is provided to reverse the effects of the `else-comment-placeholders` command and return the buffer to a state ready for more code entry by the user.

6.7 Disabling ELSE

To disable ELSE just type the command 'else-mode' to toggle ELSE off. Note that any template language definitions that have been loaded will remain loaded into Emacs memory. To explicitly delete a set of language definitions the user must either stop and re-start the Emacs session or use a DELETE LANGUAGE specifier (see Section 7.4.1.1 [Language Identification], page 18) which must be "compiled" by ELSE using the command `else-compile-buffer` (see Chapter 5 [Command Summary], page 10).

---

[2] you definitely want to do this because the typical compiler won't compile ELSE placeholder templates! But it is a "nice to have" because you might not have finished writing code and thus don't want to lose all of your placeholders. By using `else-uncomment-placeholders` you can quickly reverse the effect and start coding again.

7 Creating and Modifying Templates

This section covers the details of how to create and modify language template definitions. The aim is to give the user a basic understanding, which when coupled with looking at real template definitions, will allow the creation of new language templates and the customisation of existing template definitions.

Commands that are used in this activity are:

1. `else-compile-buffer` — "compiles" ELSE template definitions from point until the end of the current buffer. When supplied with a numeric argument (`C-u`), will compile from the beginning of the current buffer.

2. `else-extract-all` — will "extract" the definition of the entire enabled language definition file into the current buffer at point.

3. `else-extract-placeholder`[1] — extract the template definition of a placeholder into the current buffer at point. The command will prompt the user for the name of the placeholder to extract.

4. `else-extract-token` — extract the template definition of a token into the current buffer at point. The command will prompt the user for the name of the token to extract.

7.1 Syntactic Conventions for Template Definitions

General syntactic conventions used in a language template file:

1. Template definitions **are not** case sensitive i.e. a placeholder definition for If_Statement is the same as a definition for IF_STATEMENT. Also the current version of ELSE expects all "standard" template definition strings (such as DEFINE, LANGUAGE etc) to be upper case. I don't really see any reason to fix this up, it would just slow down the regular expression searching when the template file is loaded. The only exception to the case sensitivity rule is that the language name is case sensitive i.e. if you are defining a new definition for Emacs-Lisp and call it EMACS-LISP then you will get an error message to the effect that language Emacs-Lisp doesn't exist!

2. Enclose text strings that contain embedded spaces with quotes.

3. Enclose *text strings* to the right of an equal (=) sign in quotes, this rule does not apply to "standard" values such as `NONTERMINAL, TERMINAL, MENU` etc.

7.2 Customising An Existing Template Language

Probably the first thing you will want to do after the initial exploration of using ELSE is to start to customise the templates for your own environment and use. Experience will show that only a small number of templates require customisation and then it will largely depend upon the "phase" in your code entry i.e. you might find yourself coding a section that has a large number of 'case' constructs and not so many 'if' constructs, so rather than have to pass over the 'if' construct in a menu to get to the 'case' construct continually, just customise the *statement* placeholder so that the 'case' construct occurs before the 'if' construct i.e.

Before:

```
DELETE PLACEHOLDER STATEMENT -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER STATEMENT -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
```

---

[1] These commands provide "auto-completion" for the convenience of the user

```
        /DESCRIPTION=""
        /DUPLICATION=CONTEXT_DEPENDENT -
        /SEPARATOR="" -
        /TYPE=MENU -

        "null_statement"/PLACEHOLDER
        "if_statement"/PLACEHOLDER
        "case_statement"/PLACEHOLDER
        "loop_statement"/PLACEHOLDER
        "block_statement"/PLACEHOLDER
        "accept_statement"/PLACEHOLDER
         .
         .
         .


     END DEFINE
```
After:
```
   DELETE PLACEHOLDER STATEMENT -
        /LANGUAGE="Ada" -
   DEFINE PLACEHOLDER STATEMENT -
        /LANGUAGE="Ada" -
        /NOAUTO_SUBSTITUTE -
        /DESCRIPTION=""
        /DUPLICATION=CONTEXT_DEPENDENT -
        /SEPARATOR="" -
        /TYPE=MENU -

        "case_statement"/PLACEHOLDER
        "if_statement"/PLACEHOLDER
        "null_statement"/PLACEHOLDER
        "loop_statement"/PLACEHOLDER
        "block_statement"/PLACEHOLDER
        "accept_statement"/PLACEHOLDER
         .
         .
         .


     END DEFINE
```
To achieve the above change you would perform the following sequence:

1. Extract the definition for STATEMENT using the command `else-extract-placeholder` and supplying STATEMENT as the argument;
2. Edit the definition by swapping the "if_statement" and "case_statement" lines and moving the "null_statement" below the "if_statement" line;
3. "compile" the new definition by positioning point at the beginning of the DELETE PLACEHOLDER line and running the command `else-compile-buffer`.

Note that you can accomplish the above in any buffer in your Emacs session. The change will only exist for that edit session unless you either save the changed template definition (either into the main language template file or the language custom file).

Note also that the language templates are held as a 'global' variable within the Emacs session, so any changes you make to the template definitions in one buffer will be in effect for every other buffer that has ELSE mode enabled and that particular set of language templates selected.

Other reasons for the desire to perform customisation of a set of language templates also exist. One such is the establishment of a "global" set of common templates across a number of users (usually to help enforce project coding standards). In this case, each user may want to provide their own customisations but make sure that they don't impinge on other users. For this reason, ELSE searches and loads a "customisation" language file immediately after loading the primary language file i.e. loading a language file comprises two steps, the loading of the <language>.lse file and then the searching and loading of a customisation file <language>-cust.lse. The customisation file may be located anywhere in the Emacs load path i.e. it doesn't have to be in the same directory as the primary template file and can be used to not only provide new template definitions but also provide "overrides" for definitions in the "global" language template file. In this manner, we can achieve several goals:

1. distribution of a standard language template file which remains constant; and

2. allow projects and groups to use the same template file.

and yet still provide the facility to have individual template characteristics. An example of this is the first release of ELSE contained language template files that contained file and function headers that had been customised for previous projects and coding standards that I had worked on. These definitions went out in the original distribution as an example of how people could produce such things but neglected the very real problem of providing a central repository for a standard set of templates for a particular language. The current language template file distribution have these definitions extracted from the main template file and <language>-cust.lse files are provided with these definitions in them. These definitions can then be readily edited for the particular project the user is working on without incurring incompatibilities with the central distribution.

The main feature of the ELSE definition language that allows these customisations to work is the sequence of a template definition where the placeholder/token is first **DELETE'd** and then **DEFINE'd** (see Section 7.5.1.1 [Delete Placeholder Statement], page 23 and Section 7.5.1.3 [Define Placeholder Statement], page 23). This means that any existing template definition is wiped and then completely redefined for the current edit session.

7.3 Extracting Placeholders or Tokens

To customise a set of language template definitions you can either edit the original definition file to make the change permanently or, if you are after just a temporary change to a definition, you can 'extract' the definition into the current buffer, make your change and then recompile the definition into the current editing session.

The commands to access the currently loaded language definitions are `else-extract-all`, `else-extract-placeholder` or `else-extract-token`. The latter two commands will extract an individual definition of the type indicated whereas the first command will extract the entire definition. The process of "extraction" will leave the desired definition in the current buffer at 'point'. Note that definition name completion is available when typing in the name of the definition to extract. Once the modifications have been made then the definition can be recompiled into ELSE using the command `else-compile-buffer` (See Chapter 5 [Command Summary], page 10.

These changes go into effect at a global level i.e. if there are multiple buffers loaded with the same language template then they all see the same change.

7.4 Creating a New Language

A language template file is best started from a set of EBNF for the target language. The structure of the language definitions take the form of the classical inverted tree structure commonly found in computer science literature i.e. portion of the Ada language template structure looks like:

```
                               initial string
                                    |
         -----------------------------------------------------------------
         |                              |                       |         |
     library_unit_declaration     library_unit_body       separate_body  ...
         .                              |                       .
         .                    --------------------              .
         .                    |                  |              .
                      subprogram_body      package_body
```
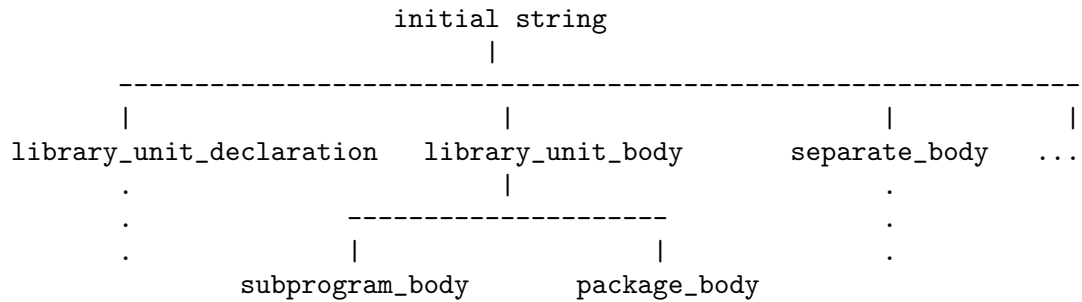
Here the placeholder definitions of `subprogram_body` and `package_body` expand directly to the language constructs and the user is then taken to the top of the tree again for the "lower level" definitions.

Each template file starts out with a *language definition* template (see Section 7.4.1 [Language Definition Template], page 18) (the root of the tree) and then proceeds into the individual branches using placeholder and token definitions. Each element of the template syntax is defined and explained in the following sections.

7.4.1 Language Definition Template

The language definition template is (note that the valid identifier characters are shown here as split across two lines for documentation purposes, this should not be done in a language definition file)[2]:

```
1  DELETE LANGUAGE "Ada" -
2  DEFINE LANGUAGE "Ada" -
3      /INITIAL_STRING="{compilation_unit}..." -
4      /PUNCTUATION_CHARACTERS=",|;"
5      /SELF_INSERT_CHARACTERS=" '~!@#$%^&*(),.<>/?;:'[]{}=\-+|"" 
6      /VALID_IDENTIFIER_CHARACTERS="abcdefghijklmnopqrstuvwxyzABCDEF \
                                      GHIJKLMNOPQRSTUVWXYZ_0123456789-"
7      /INDENT_SIZE=2 -
8      /VERSION=1.7

END DEFINE
```

<div align="center">Example Language Definition</div>

7.4.1.1 Language Identification

`Lineno 1 - 2` — Any given Emacs session can have multiple source files loaded, each with their own set of language templates i.e. the user could be editing a C source file, a C++ source file and an Ada source, ELSE could be turned on for each buffer. So, language definition files need a mechanism to uniquely identify the language to which the definition will be applied. A language definition file always starts out with the same two lines, the `DELETE LANGUAGE {language name}`[3] and `DEFINE LANGUAGE {language name}`. Because each ELSE session is customisable, each template definition construct (`LANGUAGE`, `PLACEHOLDER` and `TOKEN`) includes a `DELETE` and `DEFINE` command pair. By using a convention of `DELETE` and then `DEFINE` we ensure that the definition being "replaced" or changed will be deleted and then defined anew. Thus to define a new language the following two lines are:

---

[2] Line numbers are added to aid in the referencing in later sections
[3] Note that language name is case sensitive i.e. Ada and ADA are two different language names

```
DELETE LANGUAGE "XXX" -
DEFINE LANGUAGE "XXX" -
```

This deletes a complete language definition called "XXX" (and all of its associated placeholder and token definitions) and then commences defining a new language called "XXX".

7.4.1.2 Initial String

`Lineno 3` — The specifier `/INITIAL_STRING` defines the textual string to be inserted on the condition that the buffer in which ELSE is being turned on is empty. When ELSE mode is enabled for a buffer it will make sure the appropriate set of language definitions are loaded and then check if the buffer is empty, if the buffer is empty then the text string defined by this specifier will be inserted into the buffer. This text string is usually the template that appears at the top of the language definition tree i.e. `{compilation_unit}`.

7.4.1.3 Punctuation characters

`Lineno 4` — The specifier `/PUNCTUATION_CHARACTERS` defines the punctuation characters for the language being defined. This affects how the *housekeeping* efforts of ELSE work. It helps the code determine where whitespace should or should not occur. e.g. when deleting the optional placeholder for the parameters of the following procedure specification:

```
procedure TEST [formal_part];
```

then we want the following:

```
procedure TEST;
```

rather than:

```
procedure TEST ;
```

In this example, the function `else-kill-placeholder` (`C-c / k`) command noticed that the ';' character is defined as a punctuation character by the Ada Language Definition and thus it should ensure that no whitespace exists between the punctuation character and the preceeding function name.

7.4.1.4 Self Insert Characters

`Lineno 5` — When the user inserts keystrokes into an ELSE enabled buffer the code has to check if the cursor is within an placeholder, if it is then the desired behaviour is for ELSE to automatically delete the placeholder and replace it with the keypresses that are coming from the keyboard. The strings defined by `/SELF_INSERT_CHARACTERS` and `/VALID_IDENTIFIER_CHARACTERS` are both used for this purpose.

7.4.1.5 Valid Identifier Characters

`Lineno 6` — This attribute is used during token searches to allow ELSE to correctly identify the text string that the user may be attempting to expand. When the user runs `else-expand-placeholder` (`C-c / e`) ELSE "compiles" the string specified by the /VALID_IDENTIFIER_CHARACTERS attribute into the following Elisp regular expression:

```
[^%s+]
```

where the "%s" is replaced by the text string /VALID_IDENTIFIER_CHARACTERS. This regular expression means "search for any character which is not in the range of characters". The entire string between "point" (current cursor location) and the character found by this search is taken to be the text of the token the user is requesting an expansion for.

Since this search uses Emacs regular expression syntax, care needs to be exercised in the text actually placed in the /VALID_IDENTIFIER_CHARACTERS attribute. If the user wishes to

alter this attribute, they are advised to become familiar with Emacs regular expression syntax prior to attempting any changes.

As an example, the following situation is put — the user wants to modify a set of language templates so that they can use a token consisting of the character sequence "?:" (the "C" ternary operator could be generated this way). Just creating a token with a name of "?:" will not work as the two characters will not be part of the /VALID_IDENTIFIER_CHARACTERS set and therefore attempting to "expand" the string "?:" would fail to find a valid string to search the token database with.

So the logical first cut by the unwary would be to add these characters at the end of the existing string attribute for /VALID_IDENTIFIER_CHARACTERS thusly:

```
/VALID_IDENTIFIER_CHARACTERS="abcdefghijklmnopqrstuvwxyzABCDEF \
                              GHIJKLMNOPQRSTUVWXYZ_0123456789-?:"
```

However, since this string is used as a regular expression, the character sequence "9-?" will be interpreted as "all characters in the range from 9 to ?" rather than as the user intended i.e. the individual characters "9", "-", "?" and ":". The "correct" place to add the sequence "?:" would be anywhere in the string *other than* at the end of the /VALID_IDENTIFIER_CHARACTERS string i.e. this would be the correct method of implementing this change:

```
/VALID_IDENTIFIER_CHARACTERS="abcdefghijklmnopqrstuvwxyzABCDEF \
                              GHIJKLMNOPQRSTUVWXYZ_0123456789?:-"
```

In fact, it might be better for template maintainers to make the treatment of /VALID_IDENTIFIER_CHARACTERS more "obvious" by using something like this:

```
/VALID_IDENTIFIER_CHARACTERS="a-zA-Z_0-9-"
```

If a "user" was faced with modifying this, they might be tempted to read further?

7.4.1.6 Indentation Size

`Lineno 7` — The line

```
/INDENT_SIZE=2 -
```

allows customisation of the identation of the code generated using ELSE. This attribute allows the user to specify the indentation of each line of the template. It provides a single point for the user to change the indentation of the code generated by ELSE (previous versions of ELSE did not contain this feature and thus, if your coding standard required indention differently to that specified by the standard base template, then you would have to edit the entire template file and change all indented lines to the level required by your coding standard).

As ELSE loads and scans the template definition file it determines a "normalised" value for the indentation of each line within each definition by making the first line that shows an indentation (leading space(s)) from the first line of the definition. If subsequent lines show indentation which is greater that the "normalised" value then it is assigned a further multiple of the INDENT_SIZE. This continues for the scanning of the definition and the indentation of each line is stored by ELSE. When ELSE is asked to insert the text lines for a definition then it multiples each indentation value by the value specified by INDENT_SIZE i.e. as an example, the text definition of a switch statement might be:

```
"switch ({expression}) {"
"   [case_part]..."
"      [default_part]"
"}"
```

Here we see at line 2 that we have an indentation of 3 spaces — this value is calculated as a value of indentation of 1 times the value contained in INDENT_SIZE. When line 3 is scanned, ELSE notes that it is further indented and assigns a value of 2 to that line. When line

4 is scanned ELSE notes that it has no indentation compared with line 1 and is assigned an indentation level of 0 to that line.

When ELSE inserts this definition, it will take the indentation value of each line (0 for lines 1 and 4, 1 for line 2, 2 for line 3) and insert the number of spaces determined by the indentation level multipled by the value of INDENT_SIZE. Thus line 0 would be inserted with no extra spaces, line 1 would have 2 spaces inserted, line 3 would have 4 spaces inserted and line 4 would have 0 spaces inserted e.g.

```
switch ({expression}) {
  [case_part]...
    [default_part]
}
```

To change the indentation of lines in ELSE, all the user has to do is change the value of /INDENT_SIZE and "recompile" the language template definition file.

Note that if there is no indentation size specified then a default value of 4 will be assumed. ELSE will produce a message when compiling a language that contains no indentation size attribute specifier.

Where the user might want to override the spacing of templates as they are scanned by ELSE i.e. textual headers for functions/files might be an example. ELSE has the facility of allowing "hard" spaces to be inserted at the beginning of each line of a definition. These hard spaces are indicated by a '@' character i.e.

```
DELETE PLACEHOLDER MODULE_LEVEL_COMMENTS -
    /LANGUAGE="C" -
DEFINE PLACEHOLDER MODULE_LEVEL_COMMENTS -
    /LANGUAGE="C" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION="" -
    /DUPLICATION=CONTEXT_DEPENDENT -
    /SEPARATOR="" -
    /TYPE=NONTERMINAL -

    "/*  ===[ {module} ]=========================================//"
    ""
    "@Description: {text}"
    ""
    "@Revisions:"
    ""
    "@@REV     DATE      BY      DESCRIPTION"
    "@------  ---------   ---      --------------------------------//"
    ""
    "@------------------------------------------------------------//"
    ""
    "@@This item is the property of ResMed Ltd, and contains confid//"
    "@@secret information. It may not be transfered from the custod//"
    "@@ResMed except as authorised in writing by an officer of ResM//"
    "@@item nor the information it contains may be used, transfered//"
    "@@published, or disclosed, in whole or in part, and directly o//"
    "@@except as expressly authorised by an officer of ResMed, purs//"
    "@@agreement."
    ""
    "@@Copyright (c) 2002 ResMed Ltd.  All rights reserved."
    "@=========================================================//"
```

```
        ""

    END DEFINE
```

7.4.1.7 Template Version

`Lineno 8` — The line

```
        /VERSION=1.7 -
```

indicates what version this particular language template file is. This is a better solution to placing version labels in the comments portion of language template files as if a user decides to "extract" the entire language set then any comment headers are lost — this attribute allows people to see what version of the template file was used as the base when the language template set was compiled.

7.4.2 Overriding Language Attributes

You can "override" any of the attributes of a language (attributes have been described above) by providing a single `DEFINE LANGUAGE` statement without the (normally) preceding `DELETE LANGUAGE` statement. Just include the attribute that you wish to "override" and the new value will take effect when you "compile" the new language definition. Currently this feature of ELSE is only really useful for the `/INDENT_SIZE` attribute i.e. the baseline language file may define a value that is not appropriate for your project so you can redefine it to some other value by placing a `DEFINE LANGUAGE` statement in the `<lang>`-cust.lse file. For example, the following will re-define the indentation size of C Language template definitions to be 4 spaces:

```
    DEFINE LANGUAGE C -
        /INDENT_SIZE=4 -

    END DEFINE
```

When ELSE meets such a situation it will issue a "warning" message i.e. "Language XXX exists, assuming attribute modification".

7.5 Definition of the Template Structure

The remainder of the language definition file may contain a mixture of placeholder and token definitions. The order in which they appear is not important as no attempts are made at cross checking definitions and their use whilst the language definition file is being loaded. Perhaps one day there will be an explicit command to aid template developers to check for "holes" by performing a consistency check.

7.5.1 Placeholder Definition

A typical placeholder definition is shown below. Note that the line numbers are added as an aid in the following sections that explain each portion of this structure. Each section will reference the appropriate section by use of a line number. Texinfo doesn't seem to offer appropriate x-referencing in this area, so each section will just mention `Lineno: X`.

```
 1  DELETE PLACEHOLDER IF_STATEMENT -
 2      /LANGUAGE="Ada" -
 3  DEFINE PLACEHOLDER IF_STATEMENT -
 4      /LANGUAGE="Ada" -
 5      /NOAUTO_SUBSTITUTE -
 6      /SUBSTITUTE_COUNT=2 -
 7      /DESCRIPTION="" -
 8      /DUPLICATION=CONTEXT_DEPENDENT -
 9      /SEPARATOR="" -
10      /TYPE=NONTERMINAL
11
12      "if {condition} then"
13      "   {statement}..."
14      "[elsif_part]"
15      "[else_part]"
16      "end if;"
17
18  END DEFINE
```

The following sections explain each construct used in this definition.

7.5.1.1 Delete Placeholder Statement

`Lineno 1` — This statement tells ELSE to delete a placeholder called `IF_STATEMENT` from the language template definitions for the language "Ada" (as defined in the next line — see Section 7.5.1.2 [Language Specifier], page 23). The placeholder name may be any combination of characters between the range of SPC to ~. If the name contains one or more spaces, then it must be wholly enclosed by quotation marks.

7.5.1.2 Language Specifier

`Lineno 2 & 4` — The language specifier `/LANGUAGE="Ada"` defines the particular language set to which the preceeding template command applies. This line must follow the commands `DEFINE PLACEHOLDER` or `DELETE PLACEHOLDER`.

7.5.1.3 Define Placeholder Statement

`Lineno 3` — This statement tells ELSE to define a placeholder called `IF_STATEMENT` from the language definition identifier in the next line (see Section 7.5.1.2 [Language Specifier], page 23). The placeholder name may be any legal combination of the following characters:

`A-Z 0-9_`

If the name contains embedded spaces then it must be enclosed by quotation marks.

7.5.1.4 Auto Text Substitute

`Lineno 5 --- Lineno 6` — The attribute on Lineno 5 has two possible values: `/AUTO_SUBSTITUTE` or `/NOAUTO_SUBSTITUTE`. It works in conjunction with count specified by the attribute on Lineno 6. This attribute exists because it is quite common in many languages (or coding styles) to repeat a text string multiple times within a language construct e.g. in Ada the package name (`designator`) is an optional entry at the end of the package body:

```
1 package body {designator} is
2   {declarative_item}...
3 [begin_package_body]
4 end [designator];
```

In the above example, it would be desirable if the second occurrence of the *designator* string, shown at lineno 4, could be replaced with the text of the first *designator* string (lineno 1) as the user types it into the buffer. To achieve this behaviour in ELSE, the definition of the *designator* placeholder would have the auto-substitute attribute set and the substitution count attribute set as follows:

```
DELETE PLACEHOLDER DESIGNATOR -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER DESIGNATOR -
    /LANGUAGE="Ada" -
    /AUTO_SUBSTITUTE -
    /SUBSTITUTE_COUNT=2 -
    .
    .
    .

END DEFINE
```

The general behaviour of ELSE is that whenever the user starts to enter text whilst positioned within a placeholder, ELSE will examine the auto-substitute attribute for that placeholder and if it is set to `AUTO_SUBSTITUTE` it will then search forward in the buffer looking for `SUBSTITUTE_ COUNT` matches of the placeholder string. Each matching portion of the buffer is tagged using Emacs markers, then as the user types in the text string at the first placeholder, the same text (including backspaces etc) is repeated at each of the subsequent matching sites. The duplication of changes to the subsequent textual areas continues until the user makes a change to the buffer that is outside the area of the original placeholder. Once such a change occurs then all auto-substitution makers are erased.

The default value for the `/SUBSTITUTE_COUNT` attribute is 1 i.e. there will be one other place to perform a substitution.

The following example shows an interesting example of the use of the auto-substitute feature in the C language to create a "custom" placeholder definition for the for loop i.e. it is quite common to have a for loop where the count variable is repeated at three different points in the same line. By defining a special placeholder name with an auto-substitute count set appropriately then we can save some typing i.e.

```
DELETE TOKEN FOR -
    /LANGUAGE="C" -
DEFINE TOKEN FOR -
    /LANGUAGE="C" -
    /DESCRIPTION="conditional, repeated statement execution"

    "for ({loop_var} = [0]; {loop_var} [<] {value}; {loop_var}[++])"
    "{"
    "   {statement}..."
    "}"

END DEFINE

DELETE PLACEHOLDER LOOP_VAR -
    /LANGUAGE="C" -
DEFINE PLACEHOLDER LOOP_VAR -
    /LANGUAGE="C" -
    /AUTO_SUBSTITUTE -
    /SUBSTITUTE_COUNT=2 -
```

```
        /DESCRIPTION="" -
        /DUPLICATION=CONTEXT_DEPENDENT -
        /SEPARATOR="" -
        /TYPE=TERMINAL

        "Enter a variable name."

    END DEFINE
```

Here, the *loop_var* placeholder is defined to auto-substitute with 2 further occurrences. As the user types into the first occurrence of the placeholder then that text will be repeated twice more.

Note that if the placeholder definition specifies a repeat count but that number of matches cannot be found at run-time then only the placeholders that were matched will have the substitution performed.

Another thing to note about this example, is that it shows a for loop definition with some "common" options i.e. the initial value is defined as a placeholder with a value of '0' when expanded, the loop test is defined with a default of '<' and the loop counter is defined with a default value of increment i.e. '++'. Since all three of these items are defined as placeholders, the user has the option of navigating between them and either selecting the default value or typing in a new value.

7.5.1.5 Description Specifier

`Lineno 7` — This specifier defines a text string which will appear with the placeholder if it is referenced from a menu item. This acts as a one line help string for the user in menu displays. An empty description is defined by `""`.

7.5.1.6 Duplication Specifier

`Lineno 8` — This specifier is used to define how the template will be duplicated if the ellipses are found after the placeholder text in the buffer i.e. `[statement]...` tells ELSE to keep repeating the placeholder `[statement]` whenever the user expands (or types into) it. The possible values are:

1. VERTICAL — the placeholder is duplicated vertically onto the next line.

2. HORIZONTAL — the placeholder is duplicated on the same line to the right of the placeholder being expanded.

3. CONTEXT_DEPENDENT — means the placeholder will be duplicated in either the vertical or horizontal direction. Some simple rules are applied in this case, if the placeholder is alone on the line then it is duplicated vertically, if there is text on the same line before the placeholder then it will be duplicated in the horizontal e.g.

```
        [statement]... ↦ if {condition} then
                            {statement}...
                         [elsif_part]
                         [else_part]
                         end if;
                         [statement]...
```

Example of Vertical Duplication

```
    when {discrete_choice}... => ↦ when RED | [discrete_choice]... =>
```

<div align="center">Example of Horizontal Duplication</div>

7.5.1.7 Separator Specification

`Lineno 9` — The specifier `/SEPARATOR` is used to indicate the characters that should be inserted automatically when the placeholder is duplicated e.g. Ada uses the '|' character as a logical 'or' symbol when multiple conditions are applicable in a case statement, so when each occurrence of the placeholder is expanded then ELSE automatically inserts the character(s) defined by this specifier as part of the placeholder duplication e.g. the placeholder definition for *discrete_choice* is:

```
    DELETE PLACEHOLDER DISCRETE_CHOICE -
        /LANGUAGE="Ada" -
    DEFINE PLACEHOLDER DISCRETE_CHOICE -
        /LANGUAGE="Ada" -
        /NOAUTO_SUBSTITUTE -
        /DESCRIPTION="" -
        /DUPLICATION=CONTEXT_DEPENDENT -
        /SEPARATOR=" | "
        /TYPE=MENU

        "expression"/PLACEHOLDER
        "discrete_range"/PLACEHOLDER
        "others"

    END DEFINE
```

and the expansion would be:

```
    when {discrete_choice}... => ↦ when RED | [discrete_choice]... =>
```

Note that the separator characters, `" | "` were automatically inserted before the placeholder was repeated. ELSE also uses the character(s) defined by the `/SEPARATOR` specifier when killing an unwanted placeholder e.g.

```
    when RED | [discrete_choice]... => ↦ when RED =>
```

7.5.1.8 Type Specifier

`Lineno 10` — The specifier `/TYPE` informs ELSE how to treat the expansion of the placeholder. This specifier can have three possible values:

1. TERMINAL — This is the "end of the line", no further expansions are defined and the text string(s) held in the body of the definition are to be used as a prompter to the user e.g. with the following definition for *identifier* placeholder

```
    DELETE PLACEHOLDER IDENTIFIER -
        /LANGUAGE="Ada" -
    DEFINE PLACEHOLDER IDENTIFIER -
        /LANGUAGE="Ada" -
        /NOAUTO_SUBSTITUTE -
```

```
/DESCRIPTION="" -
/DUPLICATION=CONTEXT_DEPENDENT -
/SEPARATOR="" -
/TYPE=TERMINAL

"Any Ada identifier will do"

END DEFINE
```

An expansion of this placeholder will provide the prompt of `Any Ada identifier will do` to the user. The prompt is displayed for a customisable period (see Chapter 8 [Custom Variables], page 37) in seconds and then erased from the screen. Note that is the user performs any entry during the display of the prompt string then the prompt buffer will be terminated immediately and the user input actioned.

2. NONTERMINAL — informs ELSE that the body of the placeholder definition contains one or more text strings which should be used to replace the placeholder ie. the following definition will provide this expansion

```
[context_clause] ↦ with {library_unit_name}...; [use_clause]

DELETE PLACEHOLDER CONTEXT_CLAUSE -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER CONTEXT_CLAUSE -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION=""
    /DUPLICATION=CONTEXT_DEPENDENT -
    /SEPARATOR="" -
    /TYPE=NONTERMINAL -

"with {unit_simple_name}...; [use_clause]"

END DEFINE
```

3. MENU — informs ELSE that the body of the placeholder definition contains a menu selection that must be presented to the user for resolution i.e. the following definition will present a menu of choices for the `statement` placeholder:

```
DELETE PLACEHOLDER STATEMENT -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER STATEMENT -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION=""
    /DUPLICATION=VERTICAL -
    /SEPARATOR="" -
    /TYPE=MENU -

"assignment_statement"/PLACEHOLDER
"if_statement"/PLACEHOLDER
"case_statement"/PLACEHOLDER
"loop_statement"/PLACEHOLDER
"block_statement"/PLACEHOLDER
```

```
                    .
                    .
                    .
            "select_statement"/PLACEHOLDER

        END DEFINE
```

7.5.1.9 Template Body

`Lineno 12 - 16` — The "body" of the template definition may contain one or more text strings. These text strings will be used by ELSE according to the setting of the `/TYPE` specifier (see Section 7.5.1.8 [Type Specifier], page 26 for examples).

When the placeholder body is a menu then each text line may contain optional keywords. These keywords are:

1. `/PLACEHOLDER` — informs ELSE that the preceding text is the name of a placeholder.

2. `/TOKEN` — informs ELSE that the preceding text is the name of a token.

3. `/FOLLOW` — informs ELSE that if the line item is a placeholder, that is itself a menu, then incorporate the entries of that definition into the menu display[4]. The default for a line item in a menu template definition is `FOLLOW` and so this attribute may be omitted. To explicitly override this behavour for individual instances use the `/NOFOLLOW` option as documented below.

4. `/NOFOLLOW` — informs ELSE that even though the line item is a placeholder, that is itself a menu, do not "follow" the definition by incorporating the entries of that definition into the menu display. Note that this option can be globally overriden by the `'else-follow-menus` and `'else-nofollow-menus` flags (See Chapter 8 [Custom Variables], page 37.)

For example, the `FULL_TYPE_DECLARATION` placeholder in the Ada language templates has a body of:

```
        "type {identifier} [known_discriminant_part] is {type_definition};"
        "task_type_declaration"/PLACEHOLDER
        "protected_type_declaration"/PLACEHOLDER
```

This body tells ELSE that if the user selects the first line, then perform a direct text substitution, if one of the other two lines are selected then ELSE is being told that they refer to a further placeholder and will be treated according to the `/TYPE` specifier of that placeholder i.e. the indicated placeholder will be expanded. See also See Section 7.4.1.6 [Indentation Size], page 20, for further information on applying "hard" spaces to the beginning of a line of text in the body.

An example of the use of the `FOLLOW` and `NOFOLLOW` is taken from the Emacs-Lisp template file for the definition of the `STATEMENT` placeholder[5]:

```
        "let_statement"/PLACEHOLDER
        "if_statement"/PLACEHOLDER
        .
        .
        .
        "property-list-stmt"/PLACEHOLDER
        "symbols"/PLACEHOLDER/NOFOLLOW
        "sequences"/PLACEHOLDER/NOFOLLOW
        .
```

---

[4] This is default behaviour for ELSE

[5] The entry for `property-list-stmt` has been modified for the purposes of this example

.

Here the placeholder `property-list-stmt` is a menu which in turn has multiple entries/options. If `else-follow-menus` is `nil`, then ELSE will "follow" the menu definitions for `property-list-stmt` and include its items in the menu display, however, because `symbols` is followed by `/NOFOLLOW` then its definitions will not be included in the primary menu dispolay i.e. if the user wishes to select one of the options in the `symbols` placeholder then they must explicity select the `symbols` entry to be presented with the possible options at that level.

7.5.1.10 Placeholder Cross-Referencing

Finally, the "body" of a placeholder definition may contain some or none of the above attributes and purely reference another existing placeholder i.e.

```
DELETE PLACEHOLDER XYZ -
  /LANGUAGE="ABC"
DEFINE PLACEHOLDER XYZ -
  /LANGUAGE="ABC"
  /PLACEHOLDER=DEF

END DEFINE
```

Here we have a placeholder called "XYZ" which will automatically use the definition of the placeholder "DEF" when it is referenced or invoked by the ELSE routines. This can be extremely useful in situations such as the following where the EBNF for a language might specify two forms of "identifier" which you wish to ideally make the same w.r.t ELSE behaviour e.g. when the user enters a string for `subprogram_identifier` ELSE should perform an auto-substitution on the following `defining_identifier` placeholder.

```
procedure {subprogram_identifier} is
   ...
end [defining_identifier];
```

We may not want to necessarily change the template definition to generate:

```
procedure {defining_identifier} is
   ...
end [defining_identifier];
```

and then have to define a placeholder definition for `defining_identifier` that includes auto-substitution behaviour, because `defining_identifier` may appear somewhere else as a "singleton". The ideal would be to have a placeholder definition for `subprogram_identifier` that defined auto-substitute behaviour but also referred to the definition of `defining_identifier` for its basic behaviour i.e. if we use the following definition then when the user types in a `subprogram_identifier` placeholder ELSE will search for a matching `defining_identifier` placeholder and perform auto-substitution there:

```
DELETE PLACEHOLDER SUBPROGRAM_IDENTIFIER -
  /LANGUAGE="ABC"
DEFINE PLACEHOLDER SUBPROGRAM_IDENTIFIER -
  /LANGUAGE="ABC"
  /AUTO_SUBSTITUTE -
  /PLACEHOLDER=DEFINING_IDENTIFIER

END DEFINE
```

7.5.1.11 End Define Command

`Lineno 18` — The `END DEFINE` specifier informs ELSE that the end of a placeholder or token definition has occurred.

7.5.2 Token Definition

There are only two forms to the definition of a Token, they are both shown below. These two definitions show two different ways of specifying the same thing. In the first instance, the definition is using an existing placeholder definition and just referring to it as the action to perform on expansion. The second form shows a textual substitution form which acts in the same manner as a `NONTERMINAL` placeholder (see Section 7.5.1.8 [Type Specifier], page 26).

```
DELETE TOKEN IF -                    DELETE TOKEN IF -
    /LANGUAGE="Ada" -                    /LANGUAGE="Ada" -
DEFINE TOKEN IF -                    DEFINE TOKEN IF -
    /LANGUAGE="Ada" -                    /LANGUAGE="Ada" -
    /PLACEHOLDER=IF_STATEMENT
                                     "if {condition} then"
END DEFINE                           "  {statement}..."
                                     "[elsif_part]"
                                     "[else_part]"
                                     "end if;"
                                     END DEFINE
```

The `DELETE` and `DEFINE` lines follow the same rules as for the equivalent placeholder specifiers (see Section 7.5.1.1 [Delete Placeholder Statement], page 23) and (see Section 7.5.1.3 [Define Placeholder Statement], page 23. Similarly for the `LANGUAGE` specifier (see Section 7.5.1.2 [Language Specifier], page 23.

The greatest strength of the token definition is that it provides convenient "abbreviations" for the user i.e. the user doesn't have to type out the full placeholder name and enclose it in braces. Tokens provide the facility of a convenient stand alone string which can be expanded into some other entity, either another placeholder definition or as a textual string. Note that token definitions do not provide "non-terminal" and "menu" facilities like the placeholder definition however, this is off-set by the ability of the token definition to refer to a placeholder definition which does provide these facilities.

Of special note, ELSE searches the appropriate definition array based upon the context of the string being expanded i.e. if the string is enclosed in '{}'s or '[]'s then it will search the placeholder array, if the string is "free-standing" then ELSE will search the token array. Thus, template definition names do not have to be unique between placeholders definitions and token definitions e.g. you could have a placeholder definition called "if" and a token definition called "if" without having any conflict.

7.5.3 Hooking Elisp Code into ELSE Templates

On rare occasions it might be useful to associate some Elisp code with ELSE templating activities. This can be "linked" to both placeholder and token definitions using the same syntax (the following examples show usage with placeholder definitions but the same syntax applies equally to token definitions). The syntax is:

```
/RUN_CODE=<elisp-defun><phase>
```

where *elisp-defun* is the name of a Elisp defun and *phase* is the "phase" of operation during which ELSE will call the defun. The currently[6] defined "phases" are:

/BEFORE — Call `elisp-defun` *before* the placeholder/token is expanded (invoked during execution of `else-expand-placeholder` (`C-c / e`));

/AFTER – Call `elisp-defun` *after* the placeholder/token has been expanded (invoked during execution of `else-expand-placeholder` (`C-c / e`));

/ONINSERT — Call `elisp-defun` when a *self-insert* character is typed with point in the placeholder (this phase option is meaningless for tokens).

A definition (placeholder or token) may have multiple `/RUN_CODE` lines indicating a separate elisp-defun to call for each line. Each `/RUN_CODE` line may have only one elisp-defun but can have multiple phase indicators i.e. the following example would have *elisp-test-dfn* called during both the `/BEFORE` and `/AFTER` phases of the expansion.

```
DELETE PLACEHOLDER XXXX -
    /LANGUAGE="yyy" -
DEFINE PLACEHOLDER XXXX -
    /LANGUAGE="yyy" -
    .
    .
    /RUN_CODE=else-test-dfn/BEFORE/AFTER
    .

END DEFINE
```

A more concrete example is shown using a definition found in the ELSE template language itself (Template.lse). The definition for placeholder "nonterminal|terminal_placeholder" is (partially shown for brevity):

```
DELETE PLACEHOLDER nonterminal|terminal_placeholder -
    /LANGUAGE="Template" -
DEFINE PLACEHOLDER nonterminal|terminal_placeholder -
    /LANGUAGE="Template" -
    .
    .

    "    /{substitute}"
    "    [substitute_count]"
    "    /DESCRIPTION={descriptive_text} -"
    .
    .

END DEFINE
```

The "{substitute}" placeholder is a menu that leads to either `/NOAUTO_SUBSTITUTE` or `/AUTO_SUBSTITUTE` — however, if the user selects `/NOAUTO_SUBSTITUTE` then they have to manually delete the redundant placeholder "[substitute_count]". A possible solution (this can be solved with slightly different ELSE template definitions, BTW) is to use the `/RUN_CODE` attribute and accompany it with an appropriate elisp defun. In this scenario, the definition of "substitute" changes from:

---

[6] Other possibilties are /ON_ENTRY and /ON_EXIT i.e. calling defuns as ELSE moves point into or out of a placeholder — if there is demand for these "phases" I will implement them

```
    DELETE PLACEHOLDER SUBSTITUTE -
        /LANGUAGE="Template" -
    DEFINE PLACEHOLDER SUBSTITUTE -
        /LANGUAGE="Template" -
        .

        .
        /TYPE=MENU -

        "NOAUTO_SUBSTITUTE -"
        "AUTO_SUBSTITUTE -"

    END DEFINE
```

to this (incorporating the run-time attribute):

```
    DELETE PLACEHOLDER SUBSTITUTE -
        /LANGUAGE="Template" -
    DEFINE PLACEHOLDER SUBSTITUTE -
        /LANGUAGE="Template" -
        .

        .
        /RUN_CODE=process-substitute/AFTER
        /TYPE=MENU -

        "NOAUTO_SUBSTITUTE -"
        "AUTO_SUBSTITUTE -"

    END DEFINE
```

and the user "compiles" the following elisp defun into their edit session (place in a convenient .el file that gets loaded, your .emacs file or whatever):

```
    (defun process-substitute ()
      "Determine whether the next placeholder after a [substitute] should
    be deleted or left intact - could make this more 'robust' by testing
    whether the placeholder is a [substitute_count] before deleting......"
      (let ()
        (if (else-scan-for-match "/NOAUTO_SUBSTITUTE" nil t)
            (progn
              (else-next-placeholder)
              (else-delete-placeholder)))))
```

Now when "{substitute}" is expanded, the Elisp defun `process-substitute` is run *after* the expansion and decides whether the "[substitute_count]" placeholder remains or not depending upon which option the user selected.

Note that this example relies on "published" ELSE defuns (`else-next-placeholder` and `else-delete-placeholder`) and an "unpublished" defun `else-scan-for-match`. The later is used to scan the text before point (search limited to the beginning of the line in this case) and look for the text `/NOAUTO_SUBSTITUTE` — if it is found then the "[substitute_count]" is not wanted and is deleted.

7.6 Example of Creating A New Language Template

Use of ELSE shines for "verbose", structured languages such as Ada, Modula-2, Pascal etc and performs not so well for what I will call unstructured "minimalist" languages such as C

and C++. Obviously, the aim in the definition of languages such as Modula-2, Pascal etc was to provide code that was more readable and therefore easier to check and maintain. Languages such as C and C++ were designed (C++, due to its desire to be backward compatible with C) for minimal code entry effort which results in code with minimal readability and therefore maintainability.

When creating language templates it is important to keep in mind that there is a point where the trade-off of template availability versus user effort is reached i.e. even for "verbose" languages, I would not recommend providing templates for the "expression" construct. It is easier to just provide a prompt string at this point rather than to follow the path that the language syntax lays out, because at this stage the user is only required to type in something like `A = B` or something equally simple. Remember the aim of ELSE is to reduce keystrokes and improve productivity, so providing templates that allow the user to select from the many permutations that lead from a typical language "expression" syntax is not warranted either on your part of template design or on the users part of code entry.

If you are faced with developing a set of language templates then there are two possible courses of action. You can either establish a bare minimum of constructs that will suit your needs i.e. template definitions for the common language constructs such as "statement", "if statement", "case statement", "procedure or function statement" etc or you can go the "whole hog" and provide a "complete" set of templates right down to definitions for common library calls etc.

One of the benefits of the "bare minimum" approach is that you can cater for the large portion of code entry with only a couple of hours work to develop the template definitions. Whereas if you opt for the "whole hog" approach then it may involve quite a few hours work. Creation of language template definitions is a very manual, mechanical task and is not always easy since you are usually working from a specification for the language such as a set syntax diagrams. Such syntax diagrams are rarely suitable for straight incorporation into a set of language template definitions. There is usually a fair amount of manually going through the syntax looking for simplifications etc.

If you are going for the complete approach then you are advised to start from a set of EBNF syntax diagrams for the language. To relieve the amount of work required for this approach, I have written a program that helps enormously in the effort of going from the EBNF to the language definition file. This program is available upon request, just email me and I will send you the latest copy (its comes in binary form because I am too embarrassed to release it, it was my very first effort at using both literate programming and C++ :-), perhaps one day I will re-do it based upon experiences learnt). The program is available for Win95 platforms only.

The following attempts to provide some definition of the EBNF structure and examples with accompanying language template definitions. EBNF syntax diagrams are useful because they can be used to determine when a particular construct should be optional, mandatory, whether it should auto-repeat and whether there are any "punctuation" strings required.

It was somewhat difficult to run down a definition of EBNF syntax, hopefully the following is accurate, please provide corrections if I am wrong :-).

In EBNF the following rules apply:

1. Square brackets enclose optional items, thus the two following rules are equivalent.

```
return_statement ::= return [expression];
return_statement ::= return; | return expression;
```

2. Curly braces enclose a repeated item. The item may appear zero or more times, the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent:

```
term ::= factor {multiplying_operator factor}
term ::= factor | term multiplying_operator factor
```

3. A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

```
constraint ::= scalar_constraint | composite_constraint
discrete_choice_list ::= discrete_choice {| discrete_choice}
```

4. Items that repeat are enclosed within braces, any character that appears after the opening curly brace defines a "separator" string ie. a string that appears between each occurrence of the repeating construct (refer to example in previous item)

It is important to note in the following examples, that the keywords of the language being converted are copied verbatim into the language template definition, any other text is converted to either a mandatory or optional placeholder depending on how the EBNF defines their usage. Examples that illustrate these points are shown below. The EBNF construct is shown first followed by the equivalent ELSE template definition.

```
return_statement ::= return [expression];

DELETE PLACEHOLDER RETURN_STATEMENT -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER RETURN_STATEMENT -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION="" -
    /DUPLICATION=CONTEXT_DEPENDENT -
    /SEPARATOR="" -
    /TYPE=NONTERMINAL -

    "return [expression];"

END DEFINE
    sequence_of_statements ::= statement {statement}

DELETE PLACEHOLDER SEQUENCE_OF_STATEMENTS -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER SEQUENCE_OF_STATEMENTS -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION="" -
    /DUPLICATION=VERTICAL -
    /SEPARATOR="" -
    /TYPE=NONTERMINAL -

    "{statement}..."

END DEFINE
```

This example shows the definition of an item that appears zero or more time, because a typical language syntax demands at least one occurrence of a `statement` the EBNF shows a single instance (the mandatory single instance) followed by the instance enclosed in curly braces which is specified as repeating "zero or more times". Thus we "code" the template definition as single item enclosed by curly braces (which in ELSE means a mandatory entry). Remember that the ... denote auto-repeat, so when the placeholder is repeated it will be enclosed by square brackets and will thus be an optional entry. Thus we achieve the original intent of the ENBF.

In practice, you would not define the construct `sequence_of_statements` as it appears by itself and is used in other constructs such as:

```
if_statement ::=  if condition then
                     sequence_of_statements
                  {elsif condition then
                     sequence_of_statements}
                  [else
                     sequence_of_statements]
                  end if;
```

So rather than define the language template such that the user is forced to expand `sequence_of_statements`, we perform a "substitution" ourselves so that our `if` construct looks like:

```
if_statement ::=  if condition then
                     statement {statement}
                  {elsif condition then
                     statement {statement}}
                  [else
                     statement {statement}]
                  end if;
```

Which would lead us to a language template of:

```
DELETE PLACEHOLDER IF_STATEMENT -
    /LANGUAGE="Ada" -
DEFINE PLACEHOLDER IF_STATEMENT -
    /LANGUAGE="Ada" -
    /NOAUTO_SUBSTITUTE -
    /DESCRIPTION=""
    /DUPLICATION=CONTEXT_DEPENDENT -
    /SEPARATOR="" -
    /TYPE=NONTERMINAL -

    "if {condition} then"
    "  {statement}..."
    "[elsif_part]..."
    "[else_part]"
    "end if;"

END DEFINE
```

This step also shows the `elsif` and `else` portion being modified for incorporation into the template language. Placeholders may not span lines, so any EBNF construct that does so must be reduced to a placeholder that can be produced on one line. The syntax for the `if` statement is a good example of this having to be done. The following shows the equivalent EBNF and language templates for these structures:

```
elsif_part ::=                        DELETE PLACEHOLDER ELSIF_PART -
                                          /LANGUAGE="Ada" -
    elsif condition then              DEFINE PLACEHOLDER ELSIF_PART -
       statement {statement}            /LANGUAGE="Ada" -
                                          /NOAUTO_SUBSTITUTE -
                                          /DESCRIPTION=""
                                          /DUPLICATION=CONTEXT_DEPENDENT -
                                          /SEPARATOR="" -
                                          /TYPE=NONTERMINAL -
```

```
                                              "elsif {condition} then"
                                              "  {statement}..."

                                          END DEFINE

                                          DELETE PLACEHOLDER ELSE_PART -
    else_part ::=                             /LANGUAGE="Ada" -
        else                              DEFINE PLACEHOLDER ELSE_PART -
          statement {statement}             /LANGUAGE="Ada" -
                                            /NOAUTO_SUBSTITUTE -
                                            /DESCRIPTION=""
                                            /DUPLICATION=CONTEXT_DEPENDENT -
                                            /SEPARATOR="" -
                                            /TYPE=NONTERMINAL -

                                            "else"
                                            "  {statement}..."

                                          END Define
```

The following is an example of syntax that repeats with a separator string present. The construct is the (partial) Ada syntax for an object declaration. Just to provide the context, the (partial) EBNF is shown first.

```
    object_declaration ::=
      defining_identifier_list : [aliased] [constant] subtype_indication [:= \
    expression];

        defining_identifier_list ::=
          defining_identifier {, defining_identifier}
```

Here, `defining_identifier_list` is defined as a `defining_identifier` that must occur one or more times and is separated by the string ", " when it does repeat. Thus, this EBNF leads to:

```
    DELETE PLACEHOLDER DEFINING_IDENTIFIER -
        /LANGUAGE="Ada" -
    DEFINE PLACEHOLDER DEFINING_IDENTIFIER -
        /LANGUAGE="Ada" -
        /NOAUTO_SUBSTITUTE -
        /DESCRIPTION=""
        /DUPLICATION=HORIZONTAL -
        /SEPARATOR=", " -
        /TYPE=TERMINAL -

        "Type in a valid Ada identifier"

    END DEFINE
```

Here we can see the /SEPARATOR attribute be used, duplication is set for the horizontal direction and the placeholder is defined as a TERMINAL placeholder with an appropriate prompt message.

8 Custom Variables

ELSE has a small (but growing) number of custom variables, these are available through the Customisation Groups under the customisation group `Programming – Tools – Emacs LSE`. These variables are (listed alphabetically):

else-direction   *Custom Variable*

    This variable determines which direction the command `else-move-n-placeholders` will attempt i.e. if the variable is "on" then the command `else-next-placeholder` (`C-c / n`) will be used, if the variable is "off" then the command `else-previous-placeholder` (`C-c / p`) will be used. The default for this variable is 't' (on).

else-follow-menus   *Custom Variable*

    If true (`t`) then menu definitions are 'followed' or expanded until no sub-entry menu is found and all are combined into a single menu selection display at the 'top level'. If `nil`, then menu's are not expanded and the user has to traverse sub-menu entries (useful when combining menu's leads to huge menu selections).

    Setting this flag to `t` will explicitly override the settings of individual template definitions that use the `/NOFOLLOW` attribute (See Section 7.5.1.9 [Template Body], page 28.)

    The default for this variable is `nil` (off) as the default behaviour of ELSE is to "follow" menu definitions unless explicityly told not to by the `/NOFOLLOW` attribute.

    This variable may be overriden by the setting of `'else-nofollow-menus` to `t`.

    This flag is a buffer-local variable so individual buffers can have individual settings.

    This also means that setting the variable through the Emacs Customisation menu will set the global value for the variable but not actually affect the setting of any "current" buffers that are using this variable. Only "new" buffers will inherit the new value set via the customisation menu.

    See the Emacs manual for instructions on changing a variable from the command line (Evaluating Emacs-Lisp Expressions).

else-ignore-case-in-name-sorts   *Custom Variable*

    Used by ELSE when sorting output (else-extract-all, else-show-token-names, else-show-placeholder-names) into alphabetical order. A value of `t` will ignore case when sorting the names i.e.

```
PASS
print
RETURN
```

A value of `nil` will produce a sorted list that is case sensitive i.e.

```
PASS
RETURN
.
.
.
print
```

The default for this variable is `t` (ignore case).

else-kill-proceed-to-next-placeholder   *Custom Variable*

    This is a "usability" variable that controls the behaviour of the command `else-kill-placeholder` (`C-c / k`) after it has executed. If this variable is set (on) then ELSE will attempt to perform a `else-next-placeholder` (`C-c / n`) after successfully killing a placeholder. If the variable is "off" then the cursor will remain at the point at which the kill command was executed. The default is `nil` (off).

else-move-and-execute   Custom Variable
> Another "usability" variable. If this variable is "on" then after either an expansion or
> kill request (`else-expand-placeholder` (`C-c / e`) or `else-kill-placeholder` (`C-c / k`))
> which has failed, then a *movement* (either a "next" or "previous" placeholder - direction
> is dependent upon the setting of the `else-direction` flag) command is executed and the
> original operation (expand or kill) will be attempted again at the new placeholder[1]. The
> default for this variable is `nil` (off).

else-nofollow-menus   Custom Variable
> If true (`t`) then menu definitions are **not** 'followed' — when `t`, this variable takes precedence
> over the setting of `'else-follow-menus` and any `/FOLLOW` or `/NOFOLLOW` attibutes of the
> template definition. If `nil`, then the variable becomes "inactive" and determination of
> menu "follow" or not is determined based upon the settings of `'else-follow-menus` and
> thence the `/NOFOLLOW` and `/FOLLOW` attributes in that order.
>
> The default for this variable is `nil` (off).
>
> This flag is a buffer-local variable so individual buffers can have individual settings.
>
> This also means that setting the variable through the Emacs Customisation menu will
> set the global value for the variable but not actually affect the setting of any "current"
> buffers that are using this variable. Only "new" buffers will inherit the new value set via
> the customisation menu.
>
> See the Emacs manual for instructions on changing a variable from the command line
> (Evaluating Emacs-Lisp Expressions).

else-only-proceed-within-window   Custom Variable
> This flag is used only if the else-kill-proceed-to-next-placeholder flag is set. The movement
> to the next placeholder will only occur if the placeholder is visible in the current window.
> If this flag is off then the movement will proceed (possibly) causing visual disorientation
> to the user when the screen display jumps. The default for this variable is 't' (on).

else-prompt-time   Custom Variable
> This variable supplies the time which a prompt string to the user is displayed for a TERMI-
> NAL placeholder (see Section 7.5.1.8 [Type Specifier], page 26). The default is 3 seconds.

else-set-lineno   Custom Variable
> Controls the display of line numbers in the menu display window. The default for this
> variable is `nil` (off).

---

[1] this behaviour was added specifically for VR Coding and helps reduce the number of voice commands required
to achieve a specific objective

9 Technical Notes

This chapter provides some "technical" information that doesn't fall into any particular category for inclusion in previous chapters. The information offered covers:

1. "useful" variables/functions that are contained within else-mode.el. This is a brief description of the some of the variables/functions that an individual might want to access/call from outside of ELSE to perform various tasks. This list is probably not exhaustive but hopefully helpful.[1]

2. Editing Template Files

3. Hooks used in ELSE.

4. How ELSE determines the file name of a language template.

5. How to speed up load times of templates on a slow computer.

9.1 Useful ELSE Defuns

Description of some useful Defuns that ELSE contains that users can access (they are non-interactive) from their own elisp Defuns to perform useful actions.

else-after-token   apropos
   Return `t` if point is situated immediately after a valid token string for the language definition in force for that buffer.

else-in-placeholder   apropos
   Returns `t` if point is within a valid placeholder for the language definition in force for that buffer. See also else-placeholder-start and else-placeholder-end.

else-dump-language   apropos
   Dump the current buffer language template to the named file. Note that the file name parameter must have been already vetted to make sure it complies with the else naming conventions i.e. .esl

else-establish-language *language-name*   apropos
   Set language template set 'language-name as the current template set for this buffer.

else-look-up *name-string* &optional *ignore-forward-refs*   apropos
   Look-up the definition of a placeholder/token called 'name-string. 'ignore-forward-refs allows functions like `'else-kill-placeholder` to stop the forwarding referrals i.e. we wish to kill what is there not what might have been there

else-placeholder-start   apropos
   Position in the buffer of the start of the last placeholder detected/found (see else-in-placeholder).

else-placeholder-end   apropos
   Position in the buffer of the end of the last placeholder detected/found (see else-in-placeholder).

---

[1] Customisable variables and interactive defuns are documented elsewhere in this manual, the variables/functions mentioned here are for "internal" use and should be used with discretion

9.2 Editing Template Files

ELSE comes with a template definition language file for creating and maintaining language template files. It is called template.lse. When editing template definition files you can have ELSE enabled for the buffer and it will load the template.lse definitions. However, when performing common ELSE functions (common to editing a language template file that is) it will perform in terms of the language of the buffer contents rather than the ELSE template language itself i.e. if you are editing C.lse and wish to perform an extraction of a current C language definition then you can run `else-extract-placeholder` and it will look in the C language definitions for the requested definition rather than the template definitions. Likewise, when compiling the current buffer the definitions will go into the C language definitions rather than the ELSE language template definitions. The only exception to this rule is the commands that are obviously relevant to editing a language definition file in its own language i.e. `else-expand-placeholder` will still be expecting to expand placeholders and tokens that below to the TEMPLATE definitions rather than the C definitions.

9.3 ELSE and Hooks

Probably the most single significant feature that sets ELSE above any other equivalent templating system for Emacs is the ability to generate portions of text that can be easily located (placeholders) and to which the cursor can be positioned (see Section 6.2 [Navigating], page 12). Once there, then typing by the user will cause the automatic deletion of the placeholder and the insertion of the typed text. This functionality is achieved by ELSE "hooking" into the before and after change hooks of Emacs. Here it can monitor what is happening in the changes to the buffer and take appropriate action (such as the deletion of a placeholder when 'self-insert' text is typed or the duplication of typed text when the placeholder has identifier and auto-substitute situation (see Section 7.5.1.4 [Auto Text Substitute], page 23)).

9.4 Building the Language Template File Name

When ELSE mode is invoked for a buffer it will read the name of the major mode currently in effect and append the file extension ".lse" and attempt to load a file by that name using the 'load-path'. Note that this function is case sensitive i.e. if the major mode for the buffer is 'C' then a file name of 'C.lse' will be constructed and searched. Similarly, if the major mode is Emacs-Lisp then ELSE will construct a file name of 'Emacs-Lisp.lse'.

If the constructed file name cannot be located in the 'load-path' then the user will be prompted for a file name.

10 Compatibility

ELSE is definitely known not to be compatible with versions of Emacs prior to Emacs 20.X. Emacs Lisp introduced a change at 19.29 that I took advantage of and then Emacs 20.X introduced the use of custom variables which are also used in ELSE.

11 Notes for Voice Recognition Coding

It is recommended that people using Emacs and ELSE for programming using Voice Recognition tools should set the following custom variables (see Chapter 8 [Custom Variables], page 37).

1. `else-kill-proceed-to-next-placeholder`
2. `else-set-lineno`
3. `else-move-and-execute`
4. `else-only-proceed-within-window`

Having these variables set result in a minimum of voice commands to navigate and input code using ELSE. Refer to demonstration videos using ELSE, that Hans van Dam has kindly produced, at `http://home.hetnet.nl/~vandamhans/index.htm`

12 Language Template Availability

The following list indicates what language templates are currently available from `http://www.zipworld.com.au/~peterm`. For further (and a growing list) of other language templates, kindly made available by Douglas Harter refer to `http://mywebpages.comcast.net/dharter46`. Douglas currently has template files for Basic, Bliss, Bourne Shell, C Shell, COBOL, DEC DCL, Fortran90, Fortran, PASCAL, Perl and the list is growing.

1. Ada83 — must be renamed to Ada.lse prior to use. Has seen some extensive use in a programming environment so should be quite "usable"[1]

2. Ada95 — must be renamed to Ada.lse to be used. Usability index is moderate, I used these templates for about 6 months of coding, some obvious paths have not been fine tuned.

3. ELSE Template Language (template.lse). Quite usable - use these templates to create ELSE language definition templates.

4. LaTeX — Usable but extremely small sub-set - I don't do documentation this way anymore - the world is succumbing to MS-Word :-(.

5. Emacs-Lisp — Usable not terrifically extensive.

6. C — Usable.

7. Python — Usable.

8. C++ — Not very Usable. Some work is being done by Stephen Leake.

9. Java — Not very Usable. Nobody is working with this template set to my knowledge.

---

[1] Language template files have what I would term a *usability* index as far as ELSE is concerned. The more extensively they are used in programming the more "fine tuning" has been done to the template definitions and thus they become more "usable" by the programmer. Template files that have been freshly translated from the EBNF of a language have a low usability index and should be regarded by beginners with extreme caution.

13 Tutorial

This tutorial assumes that you have followed the installation instructions and have downloaded the C language templates (`C.lse` and `C-cust.lse`), these files represent the base C language template files (`C.lse`) and the customisation templates that I use for my C programming (`C-cust.lse`). For the tutorial to be as shown, it is important that these two files are used, otherwise the code samples shown may not be accurate.

The tutorial uses the C language templates, as C is probably the most recognised language (everybody has at some time or another been exposed to some form of C syntax, using any of the other languages for which ELSE templates are available might not have such a wide recognition factor).

13.1 Using ELSE for Abbreviation Coding

If you are in a "maintenance" type role i.e. working with modifying or fixing existing code files, or perhaps you are completely new to ELSE, then you might be more interested in the "abbreviation" powers of ELSE.

In these kind of coding situations, you are more likely to wish to just add a couple of lines of code rather than write completely new code from scratch. In the case of just adding several lines, then it is convenient to have some quick and easy way to generate a common code construct. You could type a placeholder directly into the buffer and then expand it but this is somewhat laborious and is not necessarily true to the underlying aim of ELSE, which is to reduce typing. To meet these needs, ELSE offers something called "tokens". Tokens are a template definition that takes a minimal number of characters and turns them into some desired construct using the expand command (`C-c / e`). Most of the common language constructs available to ELSE should have the common constructs available as tokens as well as placeholders i.e. one such common construct is the "if statement". To generate an if construct at point in the buffer, just perform the following key sequence:

`ifC-c / e`

and you will have the following if construct appear:

```
if ({expression}) {
  {statement}...
}
[elsif_part]...
[else_part]
```

Again, the cursor will be automatically positioned into the first placeholder and options similar to that explained in the previous section (Section 13.2 [Whole Language Coding], page 45).

Another example of a simple, but common construct is the humble comment statement. In `C-cust.lse` there is a token that allows quick and easy comment generation. Type the following anywhere in the buffer:

`cC-c / e`

and you should see the text:

```
/* {text} */
```

The cursor will be positioned within the "text" placeholder and the user can start typing the comment immediately.

13.2 Using ELSE for Whole Language Coding

Open a new file, "example.c". When the new file is created, enable ELSE for the buffer using *M-x else-mode*. You should see the following in the buffer:

```
{compilation_unit}
```

The cursor should be positioned within the placeholder shown. This is a top level language construct that should lead to all of the possible paths to create a complete C language file. Expand the placeholder by executing *C-c / e*, which should result in the buffer containing:

```
[module_level_comments]
[#include]...

{translation_unit}...
```

Again, the cursor will be positioned in the first placeholder (`module_level_comments`) of the expansion. Perform another expansion (*C-c / e*) and this placeholder will be replaced by the following text:

```
/* ===[ {module} ]=========================================================

   Description:

     {text}

   Revisions:

     REV     DATE       BY       DESCRIPTION
     ------  ---------  ---      -------------------------------------------

     ----------------------------------------------------------------------
     This item is the property of GTECH Corporation, Providence,
     Rhode Island, and contains confidential and trade secret information.
     It may not be transfered from the custody or control of GTECH except
     as authorized in writing by an officer of GTECH.  Neither this item
     nor the information it contains may be used, transfered, reproduced,
     published, or disclosed, in whole or in part, and directly or
     indirectly, except as expressly authorized by an officer of GTECH,
     pursuant to written agreement.

     Copyright (c) 1996 GTECH Corporation.  All rights reserved.
   =========================================================================*/
```

The cursor will now be the next available placeholder (`module`). If this placeholder is expanded (*C-c / e*), then the user will receive a prompt screen with the following text:

```
Enter the name of the module i.e. gs_lib.c
```

This text will be visible for approximately 3 seconds (ELSE prompt time - a customisable variable). This indicates to the user that there is no further expansions available from the templates. The prompt string itself offers informative text meant to help the user provide the required information. The fact that the placeholder is a "mandatory" placeholder (see Section 2.1 [Typographical Conventions], page 7) means that the user must provide a value i.e. the placeholder cannot be deleted or killed because the syntactic conventions require an entry to be made at this point.

Type in the text, *example.c* — the placeholder will be deleted automatically when the first letter is typed by the user.

Now navigate to the placeholder `include_files` using repeated next placeholder commands (`C-c / n`). Expand the placeholder (`C-c / e`) and you will be presented with a menu screen showing two possible choices (as shown below):

```
#include <{file_name}>
#include "{file_name}"
```

Select the first option by pressing the letter `s` on the keyboard (selection can also be achieved by re-executing the expansion command `C-c / e` — this is a convenience option for individuals who may map the ELSE expand command to a more convenient key sequence i.e. I have the else-expand-placeholder command mapped to <F3> - by allowing this behaviour, the user doesn't have to move fingers to achieve menu selection.

Enter the text `stdio.h`, the buffer should now contain (excluding the file header that we have already developed in the interest of saving space):

```
#include <stdio.h>
[#include]...
```

Note that the `#include` placeholder has been automatically repeated (repetition of the placeholder is achieved by the inclusion of the ellipses (. . .) at the end of the placeholder. This allows multiple constructs to be easily repeated i.e. ELSE does the repetition — it would be extremely tedious if the user was forced to type in each placeholder by hand.

Navigate to the next placeholder and delete it using `C-c / k`, this will delete the text of the second instance of the `#include` placeholder, including the braces and ellipses.

Now navigate to the "translation_unit" placeholder and expand it, you will be presented with a menu consisting of the following choices:

```
function_definition
declaration
```

Select the first possibility (`function_definition`) to receive the following text in the buffer:

```
[declaration_specifiers] {declarator}
[declaration]...
{
  [declaration]...
  [statement]...
}
```

You can navigate backwards (`C=c /p`) and forwards through these placeholders, expansion will lead to either prompt strings, menu choices or simple text substitutions, depending upon the language template definition for that placeholder. From these simple placeholders an entire language file can be constructed. Since ELSE generates all of the "house-keeping" characters such as opening and closing braces, semi-colons that end the line etc, when it comes time to compile this file, then the compiler should report a very much reduced number of syntactic errors that if the file had been entered by more traditional methods.

Hopefully this tutorial has given you some small taste for the power of ELSE and has whetted your appetite to learn more. Any questions about the contents of this manual, using ELSE or creating/modifying new templates then just contact me at the email address on the front page of this manual, or, if that doesn't work, then you can reach me through the gnu.emacs.help newsgroup, I monitor that group on all work days of the week (Australian time that is).

Concept Index