Emacs Language Sensitive Editor (ELSE)

A minor mode for Emacs.
ELSE Version 2.
Manual edition 2.1, 21st November, 2017.

Peter Milliken

This manual is for *Emacs Language Sensitive Editor (ELSE)* (version 2, 21st November, 2017), a minor mode written for the GNU Emacs editor.

Short Contents

Table of Contents

1 Overview

ELSE (Emacs Language Sensitive Editor) is a system for template generation (typically focused on programming but can be used for any editing task that involves repetitive blocks of text) aimed at reducing the amount of user typing and to operate seamlessly with a minimum of interference to the user.

ELSE is implemented as a minor mode and can work with any major mode. As ELSE is invoked the first time for each major mode, it will load a template file specific to that mode, subsequent invocations for buffers using that major mode will access the same set of template definitions i.e. there is one template file for each major mode for which ELSE is invoked. There is no limit to the number of language templates that can active during an edit session and ELSE seemlessly switches templates as the user switches buffers.

The fundamental components/building block of ELSE is the "placeholder". A placeholder is a piece of text surrounded by either curly braces ('`{}`') or brackets ('`[]`'). Placeholders can be navigated to via the next/previous commands (`else-next`/`else-previous`).

Expanding a placeholder (user places point within the '`{}`' or '`[]`''s and executes the command `else-expand`) will result in one of three possible responses:

- A menu (list) of possible completions;
- A block of text is inserted and replaces the placeholder text; or
- A prompt to the user is presented indicating possible text entries to replace the placeholder.

An abbreviation for a placeholder i.e. text not surrounded by braces or brackets, can also be expanded with the same results as if point was situated within the "full" placeholder e.g. a common placeholder in most language template files is the `statement` placeholder, so expanding the abbreviation "sta" would be the same as expanding `[statement]`. If the abbreviation is ambiguous i.e. there is more than one placeholder that starts with "sta", then ELSE will generate a menu popup showing the list of possible completions.

1.1 What makes ELSE different from other templating systems?

Probably the biggest, single advantage that ELSE offers over other available templating systems is the way it uses and implements placeholders. With other systems, such as skeleton and yasnippet, the placeholders (the points where the user expects to fill in the syntactic construct generated by the template system) are non-visible markers in the buffer to which the user can navigate. However, these markers are not "persistent" i.e. if the user navigates away from the entry point (and who doesn't switch all over the place, referencing other pieces of code as they write new code?) then on returning the markers may nolonger be available i.e. they are implemented as invisible entities that may be silently deleted as a result of edit actions by the user elsewhere in the edit session.

With ELSE, placeholders are visible pieces of text in the buffer and are treated no differently (unless the user attempts to type within them, in which case they are seamlessly replaced with the typed text) than any other text. They 'persist' until the user actively deletes them.

Another difference is that other template systems, in an attempt to aid the user, sometimes resort to a question/answer session, requesting the user provide information through text prompts before the system fills in the construct being generated - hardly a "natural", seamless editing experience! The only time ELSE requires "user input" is when the placeholder being expanded can lead to multiple choices, in which case ELSE offers a menu of possible completions and the user selects (or cancels) the selection.

1.2 Reporting Bugs

I welcome comments, suggestions and bug reports for any aspect of ELSE. Please email them to `peter.milliken@gmail.com`.

For enhancements, please include:

- An example of the suggested behaviour.
- A scenario of when and how it might be used.

For bug reports, please include enough information to reproduce the problem. Generally, this could include:

- A copy of the templates used at the time of the bug.
- Precisely the environment/context in which the template was being used.
- A description of the problem and, if possible, any samples of the erroneous behaviour e.g. results before/after an expansion for instance.
- Version of Emacs under which the error occurred.

1.3 History

ELSE has it's roots in an editor I first met and used in 1985 – DEC's LSEDIT (Language Sensitive Editor) – which ran under the VMS operating system. At the time, I was working on a project writing code in Ada – a very "verbose" language and one that I was totally unfamiliar with. I was impressed that LSE provided templates for a number of different languages and was not at all "awkward" or intrusive to use – its interface was (IMO) intuitive and made writing code much easier, mainly by generating the syntactic "sugar" that was part of the language but also providing a system for guiding somebody unfamiliar with such a feature rich language through its tree-like system of templates.

When I moved away from that project, I became increasingly frustrated at having to switch editors as I moved from project to project – mainly using Unix (vi/Emacs) or VMS (Eve/LSE). I have always liked to "enhance" my editing experiences by writing/customising my editing environment (LSE was written using a text processing utility and language called TPU – Text Processing Utility that DEC created to write editors). So I searched for an editor that was (a) customisable/extendable; (b) available on multiple OS platforms and (c) provided a consistent type of user interface with other text entry programs. At about that time, the computing industry was moving away from multiple terminals attached to a single (mainframe/mini) computer and towards a personal computer on each desk – running DOS initially.

I decided upon Emacs – it was available under Unix and there was a DOS "port" called MicroEmacs (mg). I could have just as easily gone with vi, but I found its user interface annoying since no other text entry system (Microsoft Word for instance) used it – and continually switching between the two (sometimes multiple times per day!) meant I soon developed a certain amount of angst against vi and any of its ports.

As I moved from project to project I had looked back often with longing at the functionality that LSE had provided and thus ELSE was born – I finally had the right tools and environment to imitate LSE's functionality. I had kept a copy of some of the language templates (initially on magnetic tape, then transferred to floppy disk as technology changed) and a snippet of the LSE user manual explaining the template language itself. So I had a basis to work from. ELSE's first incantation was implemented entirely in C as an extension to mg. Then GNU Emacs became available for PCs and I completely rewrote the functionality in Elisp and thus ELSE (version 1) was born, it was 1997. In the spirit of contributing to the Emacs community, I "threw it out there", but wasn't totally surprised by the apparent lack of "users". Usage is almost impossible to gauge, but I only ever received a handful of emails from users, so I assumed that was about the limit of its take-up, plus my earlier experience with extending mg, which was mandated/used

by a project team of 15 programmers, indicated that the majority of programmers just did not want or need something that generated language syntax at the touch of a single command[1].

ELSE version 1 existed for many years in a very stable form with only very small and minor enhancement modifications.

Enter the year 2017 and Emacs 25.1 and a fairly bored author. I had lost access to the original web-site that I used to host ELSE (I moved ISPs) and Emacs and supporting packages had overtaken some of the "features" used in ELSE - mainly the menu system that I had originally implemented was very crude and basic compared with packages such as *popup*. So I decided to do a complete re-write of ELSE – plus ELSE version 1 was my first foray into the land of Elisp programming and I knew that the code was pretty ugly, so I hoped to do a better job (even though I hadn't gained any further Elisp programming experience) since I was not under the pressures of attempting to produce a tool that helped with productivity while still remaining productive on an active project!

At the user interface level, ELSE version 2 functionally duplicates ELSE version 1, with some little used commands/features removed[2] because (a) I ended up using them infrequently or (b) I never received any feedback by the originating requester of the feature to tell me they had ever used it![3]

ELSE version 2 extends the (ELSE version 1) user interface by treating any string[4] as a potential abbreviation for a placeholder e.g. *if* ↦ results in the expansion of the *if_statement* placeholder. If there are multiple *completions* of an abbreviation (from the list of placeholder names) then the user is presented with a menu showing the available completions.

1.4 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 1.5 [Typographical Conventions], page 3).
2. '`expand`', '`expanded`', '`expanding`' or '`expansion`' — Denotes the execution of the command `else-expand` when point is either within a placeholder or directly after an abbreviation.

1.5 Typographical Conventions

This manual uses the following typographical conventions:

1. ↦ — denotes the *expansion* operation applied to a placeholder e.g. *expansion* of the placeholder '`{import_stmt}`' looks like:

   ```
   {import_stmt} ↦ import {module} [as {name}]
   ```

   whereas *expansion* of the textual abbreviation *imp* looks like:

   ```
   imp ↦ import {module} [as {name}]
   ```

[1] an attitude I have never understood – I mean if a construct is generated automatically, surely there is less chance of syntactic errors at compile time due to typos!

[2] they can be easily re-instated if there is any demand

[3] Version 1 allowed linking of Elisp functions into various stages of the template expansion process – something that I never saw the need for personally, however I did implement the request

[4] ELSE version 1 had an entity called a *token* which acted like an abbreviation for placeholders

2 Installation Instructions

To install ELSE, make sure the Emacs Lisp files are in your load path. If you downloaded ELSE from the package archives then this will have been done for you. Place the following command into your .emacs file:

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

Note: This version of ELSE is specifically written to use features of Emacs 25 and may not work with any previous version. If you need a version that works with Emacs prior to 25 (right back to 19), just drop me a line.

Joe Schafer (`joe@jschaf.com`) very kindly donated a basic else template mode for use when editing ELSE template files. To use this mode, include the following in your .emacs file:

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

and invoke it using `else-template-mode`.

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. Place the desired template definition files anywhere in your load path (if ELSE was downloaded from the package archives then the template files will be in the same place as the .el files — or you can move them into a more convenient spot, just make sure they are in the load path[1]).

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

```
(add-hook 'c-mode-hook
          (lambda ()
            ;; using lambda here so you can add further interesting minor
            ;; mode definitions easily.
            (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.

[1] Be careful, because ELSE searches the load path for the template files, to avoid confusion, make sure you have one copy only of the template files

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

Note that these bindings are purely provided to conform with the conventions as specified in the Emacs Lisp manual. My personal preference is to bind the main four commands to `F3 - F6`.

3.1 Popup Menu Mappings

ELSE uses the popup package to present menu choices to the user. The key bindings for the menu map is:

- `q/Q` – quit from the selection process.
- `s/S` – select the menu item.

  Note: No matter what key sequence you bind to `else-expand`, as part of the activation sequence, ELSE will also bind this key sequence to the "select" operation in the menu map– this is a "convenience" inserted so that the user can use the same key to select a menu item as started the expansion process i.e. in the expectation of no further finger movement.

4 Using ELSE

4.1 Invoking ELSE

To turn ELSE on/off in a buffer use the command *M-x else-mode*.

If ELSE is activated in a buffer that is empty then it will automatically insert the *initial_string* placeholder defined in the template language definition(see Section 6.5.2 [Language Definition], page 10). This is usually the placeholder `{compilation_unit}`. Point will automatically be positioned in the placeholder.

4.2 Navigating

The user can quickly move between placeholders using the commands `else-next` (*C-c / n*) and `else-previous` (*C-c / p*). After the command, ELSE will center point in the placeholder i.e. ready for an `expand`.

Both commands accept an optional numeric argument i.e. *C-u n C-c / n* will move point forward "n" placeholders. If there aren't "n" placeholders after point, then ELSE will stop at the last placeholder encountered and signal the user there are no more placeholders.

4.3 Placeholder Expansion

The expansion of a placeholder will produce one of the following responses:

- The user is presented with a menu directing the user deeper into the language templates e.g. all current templates have a `{statement}` placeholder which contains a menu of possible selections i.e. `expanding` the placeholder `{statement}` (when editing a Python file) would result in the user being presented with the following menu options:

```
"if_stmt"
"while_stmt"
"for_stmt"
.
.
"exec_stmt"
```

  Example 4.1: Menu on expansion of "statement" placeholder

- The placeholder is replaced with the text construct defined for that placeholder e.g. Example 4.2 shows the expansion of the "if" abbreviation[1].

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

  Example 4.2: C/C++ If statement

- The user will see a hint with suggestions regarding the form of information they should type to replace the placeholder[2] e.g. a trivial example would be `expanding` the `{text}` placeholder, the hint would be "Enter some text."

[1] The same result would be achieved by selecting the `if_statement` in the menu of Example 4.1

[2] Assuming the template language author put in tips on expected entries – the original LSE templates had useful tips, but ELSE, on the main, doesn't

4.4 Deleting Placeholders

To delete a placeholder, use the command `else-kill` (*C-c / k*) when "point" is within a placeholder. If the placeholder is a mandatory placeholder then ELSE will signal that the placeholder cannot be deleted. To deliberately override this behavour, use the sequence *C-u C-c / k*.

5 Command Summary

The following user commands are provided by ELSE.

`else-mode` [Interactive command]
Toggles the minor mode for the current buffer. If the buffer is empty when *else-mode* is enabled then it inserts the *initial_string*[1].

`else-expand` [Interactive command]
Expand the enclosing *placeholder*/preceding *abbreviation* (see Section 1.4 [Definitions], page 3) according to the rules for the definition of that placeholder[2]

`else-next` [*n*] [Interactive command]
Moves the cursor to the next valid placeholder in the current buffer. The optional argument *n* moves "n" placeholders (or to the last placeholder in the buffer if "n" cannot be satisfied).

`else-previous` [*n*] [Interactive command]
Moves the cursor to the previous valid placeholder in the current buffer. The optional argument *n* moves "n" placeholders (or to the earliest placeholder in the buffer if "n" cannot be satisfied).

`else-kill` [*n*] [Interactive command]
Kills or deletes the placeholder in which the cursor is currently positioned.

> **Note:** a numeric argument (`C-u`) will force a kill even when the placeholder is mandatory.

`else-compile-buffer` [*n*] [Interactive command]
Command to "compile" the language definitions from '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-extract-all` *language* [Interactive command]
Prompts the user for the language name and extracts the language definition and all of the placeholders into the current buffer at 'point'.

> **Note:** This command works from any buffer.

`else-extract-placeholder` *placeholder* [Interactive command]
Prompts the user for a placeholder name defined in the current language template set and then extracts the placeholder definition into the buffer at point.

> **Note:** Placeholder names are stored internally in upper-case. This function uses the Elisp *completing-read* function to provide name completion and that function is case sensitive, so the user must use upper-case when typing the placeholder name to extract.

`else-show-placeholder-names` [Interactive Command]
Display names of all of the Placeholders in the current language template set, sorted in alphabetical order, including the line number and file name where the definition occurs.

---

[1] The *initial_string* is defined as part of the language definition statement

[2] Note that if the placeholder/abbreviation is not defined in the language template set then the command will not recognise the placeholder/abbreviation and the command will do nothing.

6 Template System

6.1 Template File Naming

Language template files should have a ".lse" extension and should have the same name as the (language) major mode i.e. Emacs-Lisp ⇒ Emacs-Lisp.lse. Where such a direct correspondence cannot be used i.e. the C/C++ major mode is named "C/l" and "C++/l" respectively, ELSE maintains a "translation" table of major mode name to file (prefix) name (refer to customisation variable else-Alternate-Mode-Names) i.e. the default ELSE installation already defines an entry in this table for the C/C++ major modes e.g. C/l ↦ C and C++/l ↦ C++, which leads to C.lse and C++.lse respectively.

ELSE generates and uses a "fast load" variant of the templates files that has a ".esl" extension see Section 6.4 [Fast Load Files], page 9.

6.2 Template Loading

When ELSE is first invoked, it must "locate" the appropriate language template set to use. ELSE uses the following steps to determine which language template file to use/load for a given major mode:

1. Check the translation table to see if there is an entry for the current major mode name, if there is use that translation i.e. "C/l" ↦ "C", otherwise use the major mode name for the following steps;
2. Check if there is a template set of that name already loaded in memory, if so, then set the template set for the buffer and exit the loading process;
3. Search the Emacs load path for files using <name>.esl, <name>.lse and <name>-cust.lse. If no file is found, prompt the user for a file name and repeat the search process;
4. Determine which (.esl or .lse) is the "newer" and "load" that file(s) i.e. "compile" the <name>.lse followed by the <name>-cust.lse (and write a new <name>.esl) or "load" the contents of the <name>.esl;

6.3 Project Wide Templates and Local Customisations

The very first version of ELSE (written for the MicroEmacs editor "mg" - refer to see Section 1.3 [History], page 2) was written to be used by a project team of 15+ programmers. It was hoped, through "mandating"[1] the use of mg/ELSE, that the project coding standards would see better adherence. To allow some "flexibility" to the individual, when ELSE loads a template set, it first loads a file named <major mode name>.lse and then searches the load path for a customisation file named <major mode name>-cust.lse – thus a project wide coding standard can be implemented in some central path for the entire project and the individual can provide personal customisations using the customisation file (located in a separate path).

6.4 Fast Load Files

For convenience, ELSE manages a "fast loading" system for templates i.e. whenever a template set is loaded/compiled for the very first time, then ELSE will write the resulting Emacs Elisp objects to a separate file. In subsequent edit sessions, when ELSE is invoked, it checks the template file(s) against the fast load variant and if the fast load variant has a newer time stamp, then it loads the template set from the fast load file rather than the template files. This is a faster process on older/slower computers. The fast load files are stored/located in the path

[1] When did that ever work with herding cats... er, I mean programmers? :-)

indicated by the custom variable else-fast-load-directory. If either of the .lse files have a newer time stamp, then ELSE first loads the main template file, followed by the customisation file and (re)writes a new fast load file.

6.5 Template File Layout

6.5.1 Syntactic Conventions for Template Definitions

General syntactic conventions used in a language template file:

1. Case – use upper case for all keywords of a template definition. ELSE will generate an error if it can't scan a keyword e.g. "END DEFINE" is good, "End DEFINE" will fail during the compile phase. Placeholder names do not have to be upper case, however, the compile process will convert them to upper case and any template extraction will read them back in upper case.
2. Enclose any text strings that contain embedded spaces with quotes.
3. The `@` character is used to define "hard spaces" in the body of a definition.

6.5.2 Language Definition

The language definition defines the global aspects of a language and serves as the vehicle to contain and group the placeholder definitions. Each template set may have only one language definition. A typical language definition is shown in Example 6.1.

```
1  DELETE LANGUAGE C
2  DEFINE LANGUAGE C
3      /INITIAL_STRING="{compilation_unit}"
4      /PUNCTUATION_CHARACTERS="*.;(),"
5      /VALID_IDENTIFIER_CHARACTERS="a-zA-Z_0-9-''
6      /INDENT_SIZE=2
7      /VERSION=1.7
8
9  END DEFINE
```

Example 6.1: Sample Language Definition

With reference to Example 6.1:

- `Lines 1 - 2` – defines the language and its name;
- `Line 3` – defines the string that ELSE will autmatically insert when it is invoked in an empty buffer e.g. `{compilation_unit}`
- `Line 4` – used by the kill placeholder process to "neaten" the line after a placeholder is killed e.g. killing the `declarator` placeholder in the following line

  Before:

  ```
  typedef const float abc, [declarator]...;
  ```

  After:

  ```
  typedef const float abc;
  ```

  If the "`;`" character was not included in Line 4 then the line (after the kill) would have looked like this:

  ```
  typedef const float abc ;
  ```

- `Line 5` – (Elisp regexp) used to define the range of characters that might constitute a valid placeholder name in the language i.e. this expression is used to define the legal characters when searching backwards in the buffer for a valid abbreviation to expand e.g. if the string `@if` was expanded then ELSE would use "if" as the possible (placeholder) abbreviation – because the '@' character is not part of the character range.

- `Line 6` – Indentation to use with the placeholder definitions – to avoid having "fixed" indentation that must be changed (manually) every time the coding standards change, all code indentation in the placeholder definitions are "normalised" during the compilation phase and computed relative to the first indentation in a definition i.e. Example 6.2 shows the (partial) definition of a switch statement. In this example, when ELSE "compiles" this definition, it allocates (normalises) the indentation at Line 8 to be an indent value of 1 and any further indentation (such as the indentation at lines 10 - 11, 14 - 15) are calculated as multiples of the first occurrence. When the placeholder is expanded, ELSE multiplies the normalised values by the value at Line 6. Thus, if Line 6 of Example 6.1 was changed to 4 then when the case statement is expanded lines 8 - 9, 12 - 13 of Example 6.2 would be indented 4 spaces, but lines 10 - 11, 14 - 15 of Example 6.2 would be indented 8 spaces (since in the definition they are "twice" the indentation of line 8).

```
1  DELETE PLACEHOLDER SWITCH_STATEMENT
2      /LANGUAGE=C
3  DEFINE PLACEHOLDER SWITCH_STATEMENT
4      /LANGUAGE=C
5      .
6      .
7      "switch ({expression}) {"
8      "  case {constant_expression}:"
9      "  [case {constant_expression}:]..."
10     "    [statement]..."
11     "    break;"
12     "  [case_part]..."
13     "  default :"
14     "    [statement]..."
15     "    break;"
16     "}"
17
18 END DEFINE
```

Example 6.2: Indentation space interpretation

Note: For convenience, the language customisation file (<major mode name>-cust.lse) may contain a language definition statement that redefines the indentation value – this is intended to allow the user to override any "project" wide coding standards for their own personal use i.e. easily swapping backwards and forwards. This statement would look like this:

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

This does not define a new language but rather signals to ELSE that an attribute is about to be overridden.

Addendum: Since space characters at the start of a line in a definition are "normalised" an alternate method is required when the user wants to make sure the spacing is not transformed – this usually occurs in "headers" within files for instance. To "force" a space at the beginning of a line use the `@` character for each "hard" space desired e.g. the following line, when inserted during an expansion, will have 3 spaces:

```
"@@@The rain in Spain falls mainly on the plain"
```

- `Line 7` – Version information for the language set.

6.5.3 Placeholders

Placeholders come in four possible flavours:

1. Menu – offers a range of choices leading to further expansion;
2. NonTerminal – defines a language structure to be inserted at point i.e. the switch statement in Example 6.2;
3. Terminal – specifies an informational (hint) message to the user regarding valid values to replace the placeholder; and
4. Referential – a placeholder that references another placeholder definition.

The first three types of placeholder have the common attributes/structure shown in Example 6.3.

```
1  DEFINE PLACEHOLDER STATEMENT
2      /LANGUAGE=C
3      /AUTO_SUBSTITUTE
4      /SUBSTITUTE_COUNT=1
5      /DESCRIPTION=""
6      /DUPLICATION=CONTEXT_DEPENDENT
7      /SEPARATOR=""
8      /TYPE=MENU
9      .
10     .
11     .
12 END DEFINE
```

Example 6.3: Placeholder Common Attributes

1. `Line 1` – starts the the placeholder definition and assigns a placeholder name;
2. `Line 2` – defines the language set the placeholder belongs too – the language referenced must have already been "created" using a language definition template (see Section 6.5.2 [Language Definition], page 10);
3. `Line 3` – defines an attribute that informs ELSE whether edit actions in a placeholder of this name should be repeated in one or more other instances of the placeholder in the buffer:
   a. /AUTO_SUBSTITUTE – specifies (when the placeholder is expanded or typed into) whether the text that replaces the placeholder should be repeated (auto substituted) into any other placeholders of the same name
   b. /NOAUTO_SUBSTITUTE – no auto substitution should be attempted.

   Typical useage would be in the following situation:

   ```
   @node {node-text},,
   @chapter {node-text}
   [cindex]
   {text}
   ```

   The placeholder `node-text` is defined with `/AUTO_SUBSTITUTE` and a `/SUBSTITUTE_COUNT=1`. When the user types into the first occurrence of the placeholder the text/edit actions are repeated in the second instance of the placeholder. All edit actions are repeated until the user navigates away and performs an edit operation somewhere else in the buffer.
4. `Line 4` – (optional) specifies the number of occurrences ELSE should look for when it starts performing an auto-substitution i.e. this line is only valid when `Line 3` is `/AUTO_SUBSTITUTE`. This value defaults to 1 if it is not present. ELSE can handle substitution counts up to 7 and will issue an error message if the value specified is greater[2].

[2] The most I have ever used is 2!

5. `Line 5` – specifies some descriptive text that is shown if the placeholder is referenced from a menu (shown beside the menu entry if it is not empty).
6. `Line 6` – if the placeholder is followed by ellipses (. . .) then this attribute specifies the "direction" of where the placeholder is duplicated. It has three possible values:
   a. VERTICAL – the placeholder is repeated on the next line;
   b. HORIZONTAL – the placeholder is repeated horizontally (to the right); and
   c. CONTEXT_DEPENDENT – ELSE will determine, based on the context of the surroundings, whether the placeholder is repeated vertically or horizontally.
7. `Line 7` – contains the "separator" text (string) to be used when the placeholder is being duplicated i.e. as part of the duplication, the separator string is appended to the placeholder undergoing expansion
8. `Line 8` – specifies the placeholder type i.e. MENU, NONTERMINAL or TERMINAL.

A referential placeholder has the following form:

```
DEFINE PLACEHOLDER CLASSNAME
    /LANGUAGE=Python
    /PLACEHOLDER=IDENTIFIER

END DEFINE
```

Example 6.4: Referential Placeholder

The template designer would use this form of placeholder to cater for situations where a placeholder is "just like this" other placeholder. Sometimes template sets can be created from an EBNF of the language in question (I have a program I wrote to help do this) and the EBNF does this type of thing with the language definition. These situations can be easily factored out by the template designer but the "feature" is left in because some of the original templates use it and I never bothered factoring them out – the user never notices.

6.5.3.1 Menu Placeholders

A typical menu placeholder and the menu popup presented to the user is shown in Example 6.5. The definition lists the possible choices offered to the user when the `statement` placeholder is expanded.

```
DEFINE PLACEHOLDER STATEMENT
    .
    .
    /TYPE=MENU
                                        ⇒
    "labeled_statement"/PLACEHOLDER          labeled_statement
    "compound_statement"/PLACEHOLDER         compound_statement
    "{assignment_expression};"               {assignment_expression};
    "selection_statement"/PLACEHOLDER        selection_statement
    "iteration_statement"/PLACEHOLDER        iteration_statement
    "jump_statement"/PLACEHOLDER             jump_statement
    "comment_statement"/PLACEHOLDER          comment_statement

END DEFINE
```

Example 6.5: Menu Placeholder and Resulting Expansion

Each line in the "body" of a menu definition can have the following properties or attributes:

1. self–insert – that is, the text line will be inserted (if selected) at point;
2. placeholder reference – references another placeholder definition and is indicated with the `/PLACEHOLDER` attribute. The `/PLACEHOLDER` attribute may be followed by the one or more of these attributes:
   a. `/NOFOLLOW`(default) – don't "follow' the definition of the referenced placeholder (see `/FOLLOW` below);
   b. `/FOLLOW` – indicates to ELSE that it should "follow" the reference and substitute the referenced placeholder definition into the menu at this line. The referenced placeholder may be a menu or nonterminal placeholder (it doesn't make sense for it to be a terminal placeholder). If it is referencing a nonterminal placeholder then the body of that placeholder must be a single line of text – for obvious reasons, otherwise ELSE ignores the follow request. Example 6.6 shows what happens when the referenced placeholder is a menu and Example 6.7 shows what happens if the referenced placeholder is a nonterminal (with a single insert line);
   c. `/DESCRIPTION` – Provide a description to display when the menu item is displayed to the user.

```
DEFINE PLACEHOLDER STATEMENT
    .
    .
    /TYPE=MENU

    "labeled_statement"/PLACEHOLDER
    "compound_statement"/PLACEHOLDER
    "{assignment_expression};"
    "selection_statement"/PLACEHOLDER
    "iteration_statement"/PLACEHOLDER
    "jump_statement"/PLACEHOLDER/FOLLOW
    "comment_statement"/PLACEHOLDER

END DEFINE
```

```
DEFINE PLACEHOLDER JUMP_STATEMENT
    .
    .
    /TYPE=MENU

    "goto {identifier};"
    "continue;"
    "break;"
    "return [expression];"

END DEFINE
```

```
                    labeled_statement
                    compound_statement
                    {assignment_expression};
                    selection_statement
                    iteration_statement
        ⇒           goto {identifier};
                    continue;
                    break;
                    return [expression];
                    jump_statement
                    comment_statement
```

Example 6.6: Menu Referencing another menu definition

```
DEFINE PLACEHOLDER STATEMENT
    .
    .
    /TYPE=MENU

    "labeled_statement"/PLACEHOLDER
    "compound_statement"/PLACEHOLDER
    "{assignment_expression};"
    "selection_statement"/PLACEHOLDER
    "iteration_statement"/PLACEHOLDER
    "jump_statement"/PLACEHOLDER/FOLLOW
    "comment_statement"/PLACEHOLDER

END DEFINE
```

```
DEFINE PLACEHOLDER JUMP_STATEMENT
    .
    .
    /TYPE=NONTERMINAL

    "goto {identifier};"

END DEFINE
```

```
                    labeled_statement
                    compound_statement
                    {assignment_expression};
                    selection_statement
       ⇒            iteration_statement
                    goto {identifier};
                    jump_statement
                    comment_statement
```

Example 6.7: Menu Referencing a NonTerminal Placeholder

6.5.3.2 NonTerminal Placeholders

A "nonterminal" placeholder is where all the action is with ELSE. This is the placeholder that actually inserts code constructs into the buffer. Example 6.8 shows the template for a function definition in Emacs Elisp. After the expansion, ELSE will place point in the first placeholder – `{identifier}`.

```
DEFINE PLACEHOLDER DEFUN
    /LANGUAGE="Emacs-Lisp"
    .
    .
    /TYPE=NONTERMINAL

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

END DEFINE
```

Example 6.8: Nonterminal Placeholder

6.5.3.3 Terminal Placeholders

A "terminal" placeholder sits at the end of the expansion tree i.e. there are no further expansions available. The template designer uses terminal placeholders to provide "hints"/are informational as to what the user is expected to enter for a valid results e.g. Example 6.9 shows an elaborate example of a terminal placeholder that can be found in the Ada language template file. ELSE will put this block of text up to the user as a popup hint when the user attempts to expand an `expression` placeholder.

```
DEFINE PLACEHOLDER EXPRESSION
  /LANGUAGE="Ada"
  .
  .
  /TYPE=TERMINAL

  "Enter a valid expression eg. VOLUME, not DESTROYED, "
  " 2*LINE_COUNT,4.0,4.0 + A, B**2 4.0*A*C, PASSWORD(1 .. 3) = "BWV","
  "COUNT in SMALL_INT, COUNT not in SMALL_INT, INDEX = 0 or ITEM_HIT"
  "(COLD and SUNNY) or WARM, A**(B**C), (1 .. 10 => 0), SUM,        "
  "INTEGER'LAST, SINE(X), COLOR'(BLUE), REAL(M*N), (LINE_COUNT + 10)"

END DEFINE
```

Example 6.9: Terminal Placeholder

Realistically, there is (obviously) as lot of work for the template designer when producing this kind of information and it is really of limited use to a knowledgeable programmer. When the basic templates sets were produced – sometimes from EBNF – I took the easy way out and produced hints like this for the Emacs Elisp `identifier` placeholder:

"Just type in a legal Elisp identifier".

6.6 Customisation

There are several ways to customise a placeholder with ELSE:

1. Edit/add[1] a definition in a template file and compile it; or
2. Extract a definition into the current buffer, modify it and compile it.

The first mechanism is the obvious preferred approach when you want the change to be permanent or available across edit sessions. The second approach is preferred/easier for temporary customisations or when experimenting with something you might want to make permanent i.e. use the 2nd method, get it working and then add it to one of the template files for the template set.

To extract a placeholder definition (from a currently loaded template set):

1. Go to the end of the buffer in which ELSE is enabled – because the template compile process only expects to see template definitions from point to the end of the buffer;
2. extract the definition you want customised using the command `else-extract-placeholder`;
3. modify the template definition; and
4. "compile" it into the current edit session using `else-compile-buffer`.

For example, say you have a (C language) `if_statement` placeholder that generates an if construct like this:

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

and you want the opening brace on the same line as the "if". Then you would perform the following sequence (assumes you are in a C language buffer with ELSE mode enabled):

[1] ELSE comes with a template set designed to create templates – Template.lse, just edit a file with a .lse extension, turn on ELSE mode and specify `template` when prompted for a template file name

1. `else-extract-placeholder` – respond with IF_STATEMENT at the prompt;
2. make the change as shown here:

```
DELETE PLACEHOLDER IF_STATEMENT
    /LANGUAGE=C
DEFINE PLACEHOLDER IF_STATEMENT
    /LANGUAGE=C
    .
    .
    .
    /TYPE=NONTERMINAL

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

END DEFINE
```

⇒

```
DELETE PLACEHOLDER IF_STATEMENT
    /LANGUAGE=C
DEFINE PLACEHOLDER IF_STATEMENT
    /LANGUAGE=C
    .
    .
    .
    /TYPE=NONTERMINAL

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

 END DEFINE
```

3. Position point at the beginning of the `DELETE` and run *`M-x else-compile-buffer`* – the modified construct/definition will be available for the rest of the edit session.
4. (optionally) copy the new definition at the end of the C-cust.lse – it is not necessary to delete the previous version because the "DELETE" statement will delete any prior definitions of `if_statement` and then the new definition will replace it.

7 Custom Variables

The custom variables used by ELSE, are available through the Customisation Groups under the customisation group Programming → Tools → Emacs LSE. These variables are (listed alphabetically):

`else-Alternate-Mode-Names` [Custom Variable]
With Emacs 22.1, the default mode name for files with the .c extension changed from "C" to "C/l" - this caused problems with the basic rule that ELSE uses to formulate the name of the template file from the mode name (see Section 6.1 [Template File Naming], page 9). This variable was created to allow the user to provide an alternate "mode" name that ELSE can use in locating the required template files.

`else-kill-proceed-to-next-placeholder` [Custom Variable]
This is a "usability" variable that controls the behaviour of the command `else-kill` (*C-c / k*) after it has executed. If this variable is set (on) then ELSE will attempt to perform a `else-next` (*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-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-fast-load-directory` [Custom Variable]
This variable contains the path to where ELSE expects to write/read template fast load files (see Section 6.4 [Fast Load Files], page 9).

`else-overwrite-placeholder-on-conflict` [Custom Variable]
If non-nil, on encountering a DEFINE statement, any previous definition will be (silently) deleted i.e. a DELETE statement is not required if this variable is non-nil. The default for this variable is `nil` (off).

Concept Index