Second Climacs

Version 2 of the Climacs text editor.

Robert Strandh

2013

Contents

Chapter 1

Introduction

Second Climacs is an Emacs-like editor written entirely in Common Lisp. It is called Second Climacs because it is a complete rewrite of the Climacs text editor.

Climacs gave us some significant experience with writing a text editor, and we think we can improve on a number of aspects of it. As a result, there are some major differences between Climacs and Second Climacs:

- We implemented a better buffer representation, and extracted it from the editor code into a separate library named Cluffer. The new buffer representation will have better performance, especially on large buffers, and it will make it easier to write sophisticated parsers for buffer contents.
- The incremental parser for Common Lisp syntax of Climacs is very hard to maintain, and while it is better than that of Emacs, it is still not good enough. Second Climacs uses a modified version of the Common Lisp reader in order to parse buffer contents, making it much closer to the way the contents is read by the Common Lisp compiler.
- Climacs depends on McCLIM for its graphic user interface. Second Climacs is independent of any particular library for making graphic user interfaces, allowing it to be configured with different such libraries.

Part I

User manual

Chapter 2

Common Lisp mode

The contents of this chapter was largely written long before the Common Lisp mode was implemented or even fully designed. We still use the present tense to describe functionality, though, as if it existed.

Common Lisp mode consists of two parts:

- The *code analyzer*, which is responsible for analyzing the contents of the buffer and presenting the result of this analysis to the user.
- A set of *commands* that take advantage of the result if the analysis.

2.1 Code analyzer

2.1.1 Display of information from code analysis

Information derived by the code analyzer is presented as slight alterations of how the code elements in the buffer is displayed to the user. The following methods are used (roughly in decreasing order of frequency):

- Changing the background color.

- Changing the foreground color.
- Changing the font face.
- Using a different glyph.

In addition, a code element that is marked in this way also has associated textual information that can be read in the minibuffer when the cursor is positioned on the text, or as a tooltip when the pointer is positioned above the text. In the remainder of this chapter, we give the English version of the textual information. Internationalization may change the language according to user preference.

Names of specific colors that are mentioned in this section are merely *examples*. They may or may not be the default colors actually used, and every color can be customized by the end user.

Furthermore, in some cases a code element that is marked this way has a *context menu* associated with it.

In general, errors and warnings are indicated by a specific *background color*. Typically these colors are in shades of *red*.

Symbols that have no error or warning information associated with them are displayed using some dark foreground color so as to give good contrast with the background.

Except for symbol with lexical bindings and symbols used for `loop` keywords, the *hue* of the color is associated with the *home package* of the symbol. We use *blue* for the Common Lisp package, *green* for the current package, and *magenta* for other packages.

Symbols with lexical bindings use *cyan* colors, independently of the home package of the symbol. Symbols used as `loop` keywords use a dark brown color.

Within a family of similar hues, the darkest color is used for symbols used as *functions* and for symbols used as *global variables* The two categories can be distinguished because of the *position* within an expression. Slightly lighter colors are used for *macros* and *symbol macros*.

A *context menu* may be associated with certain symbols.

2.1.2 Analyzer phases

The code analyzer works in three phases.

1. Surface syntax analysis.
2. Structure syntax analysis.
3. Semantic analysis.

2.1.3 Surface syntax analysis

This phase consists of repeated applying a custom version of the Common Lisp `read` function to the contents of the buffer in order to obtain a *succession of top-level expressions*.

The following types of errors are detected during this phase:

- Invalid tokens. An invalid token is either a token that has an invalid constellation of package markers, or a token with an invalid constellation of escape characters. In the latter case, it could be a *single escape* character at the very end of the buffer (not even followed by a newline) or an odd number of *multiple escape* characters.
- A right parenthesis at the top level of the buffer.
- An *incomplete* expression, i.e., an expression that is not terminated, typically as a result of too few closing parentheses.
- Extraneous whitespace.

As mentioned previously, the result of this analysis is a succession of top-level expressions, where the last one may incomplete. An incomplete expression is nevertheless parsed in order to determine the presence of invalid tokens.

Invalid token syntax

A token which does not have any interpretation as a number is considered a potential symbol. If so, there are a few cases where token is nevertheless invalid as the name of a symbol.

The token might have an invalid constellation of package markers:

- Too many package markers.
- Two package markers that are separated by some other character.
- A package marker at the end of the token.

In this case, the background color of the package markers is vivid red, and the background of the symbol itself is pink.

The associated textual information says "Illegal constellation of package markers".

Figure 2.1 illustrates how this type of information is displayed.

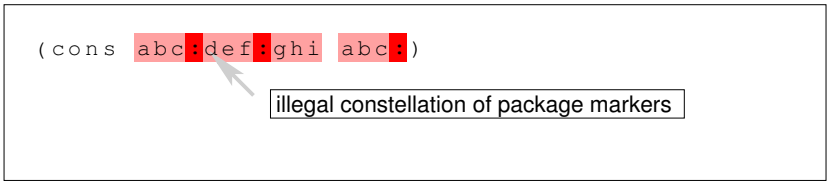

Figure 2.1: Display of potential symbol with illegal package markers.

A token with a *single escape* character immediately at the end of the buffer is displayed with a *pink* background. The single escape character itself has a vivid *red* background. The tooltip says "Single escape followed by end of buffer".

A token with an odd number of *multiple escape* characters is displayed with a *pink* background from the start of the token up to the character immediately preceding the last multiple escape character. The last multiple escape character

and the characters following it are displayed with a vivid *red* background. The tooltip says "Odd number of multiple escape characters".

No context menu is suggested for tokens with invalid syntax.

Incomplete expression

An incomplete expression is an expression that starts at some point in the buffer, but the end of the buffer is reached before the expression is complete.

Information about an incomplete expression is shown as an orange background of the first character of the expression. When several nested expressions are incomplete, the first character of each nested expression is marked this way.

Figure 2.2 illustrates how this type of information is displayed.

Figure 2.2: Display of incomplete expression.

Extraneous whitespace

Whitespace is considered *extraneous* in the following cases:

- When it follows a left parenthesis.
- When it precedes a right parenthesis.
- When it follows other whitespace that separate two expressions.
- When it follows the last non-whitespace character of a line.

It is *not* considered extraneous in the following cases:

- When it precedes the first non-whitespace character on a line.
- When it separates the first semicolon on a line from the last preceding non-whitespace character.

Extraneous whitespace is marked with a *pink* background. The tooltip associated with the marked background says "Extraneous whitespace".

Figure 2.3 illustrates how this type of information is displayed.

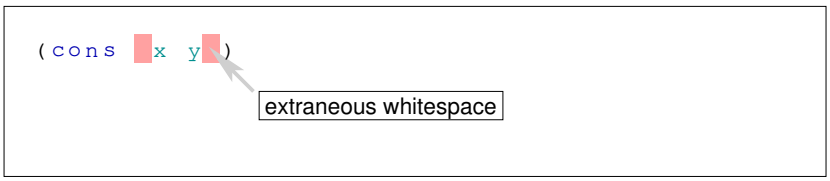

Figure 2.3: Display of extraneous whitespace.

2.1.4 Structure syntax analysis

A top-level expression that passes the first phase of the code analyzer is then analyzed as a *form* with respect to its *structure syntax*.

This phase checks the form as follows:

- If the form has the syntax of a symbol with one or two package markers (other than when the form has the syntax of a *keyword* symbol), then a check is made that the package exists.
- If the form has the syntax of a symbol with a single package marker (other than when the package marker is the first character), then a check is made that the symbol exists in the package indicated, and that it is exported from that package.

- If the form has the syntax of a symbol with two package markers, then a check is also made that the symbol exists in the package indicated, but with a less severe error display if it does not exist.

- If the form is a compound form where the `car` is not a symbol, a check is made that the `car` of the form is a plausible `lambda` expression.

- If form is a compound form, and the `car` of the compound form is a *special operator* or a *standard macro*, then the *structure syntax* is verified. For example, it is verified that the bindings of a `let` have the right form, and that `setf` has an even number of arguments.

Non-existing package

A symbol with one or two package markers but where the package indicated in the prefix does not exists is marked with a pink background, and the package name is marked with a vivid red background.

The associated textual information says "Non-existing package".

Figure 2.4 illustrates how this type of information is displayed.

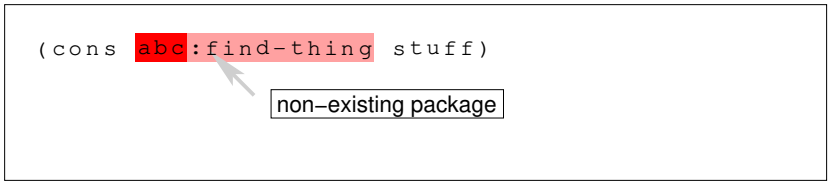

Figure 2.4: Display of potential symbol with a non-existing package.

The context menu has a single option: "Create the package".

Non-existing symbol

A symbol token with a single package marker that refers to a non-existing symbol is marked with a vivid red background under the symbol name and a

pink background under the package name. The associated textual information says "Non-existing symbol".

Figure 2.5 illustrates how this type of information is displayed.

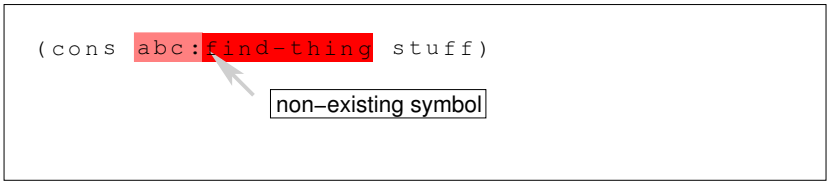

Figure 2.5: Display of a non-existing symbol with one package marker.

The context menu gives the following options:

- Create and export the symbol in the specified package.
- Import the symbol from a different package into the specified package, and also export it from the specified package. The user will be prompted for the package to import from.

A symbol token with a two package markers that refers to a non-existing symbol is marked with a pink background. The associated textual information says "Non-existing symbol".

Figure 2.6 illustrates how this type of information is displayed.

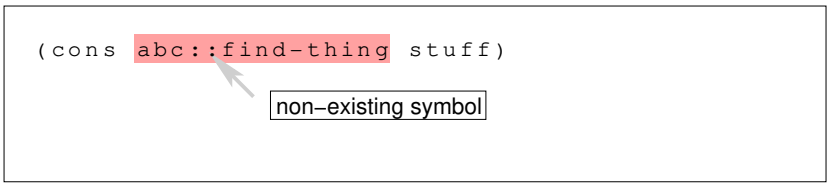

Figure 2.6: Display of a non-existing symbol with two package markers.

The context menu gives the following options:

- Create the symbol in the specified package.
- Import the symbol from a different package into the specified package. The user will be prompted for the package to import from.

Unexported symbol

A symbol token with a single package marker that refers to an existing, but unexported symbol is marked with a pink background. The associated textual information says "Unexported symbol.

Figure 2.7 illustrates how this type of information is displayed.

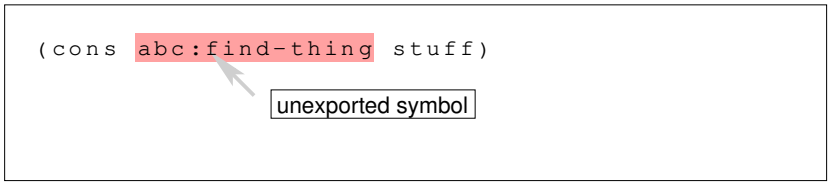

Figure 2.7: Display of an unexported symbol.

The context menu gives the following options:

- Export the symbol from the specified package.

2.1.5 Semantic analysis

Only when a top-level form passes the second phase of the analysis is it subject to *semantic analysis*.

Illegal use of Common Lisp symbols

When a symbol from the `common-lisp` package is used in a context that can be determined illegal, it is signaled by the use of an orange background.

Figure 2.8 illustrates how this type of information is displayed.

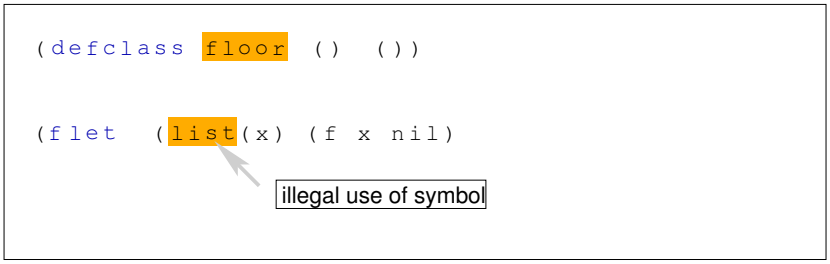

Figure 2.8: Display of illegal use of Common Lisp symbol.

Names of lexical functions

Names of functions introduced by `flet` or `labels` are shown with a dark *cyan* foreground color, independently of the home package of the symbol including when that package happens to be the `common-lisp` package.

When the pointer is located over such a name, the corresponding symbol in the introducing binding is highlighted with a light blue background.

One entry in the context menu is to jump to location where the binding was established.

Names of lexical variables

Names of lexical variables introduced by `let`, `let*`, `multiple-value-bind`, and by macros that expand into one of these special forms are shown with a dark *cyan* foreground color, independently of the home package of the symbol including when that package happens to be the `common-lisp` package.

When the pointer is located over such a name, the corresponding symbol in the introducing binding is highlighted with a light blue background.

One entry in the context menu is to jump to location where the binding was established.

Names of local macros

Names of local macros are shown with a slightly lighter *cyan* foreground color, independently of the home package of the symbol including when that package happens to be the `common-lisp` package.

When the pointer is located over such a name, the corresponding symbol in the introducing binding is highlighted with a light blue background.

One entry in the context menu is to jump to location where the binding was established.

Names of local symbol macros

Names of lexical variables are shown with a slightly lighter *cyan* foreground color, independently of the home package of the symbol including when that package happens to be the `common-lisp` package.

When the pointer is located over such a name, the corresponding symbol in the introducing binding is highlighted with a light blue background.

One entry in the context menu is to jump to location where the binding was established.

Names of global functions in the current package

A dark *green* foreground color is used to display a symbol in the current package that is the name of a global function, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named function.

If the source location where the function was defined can be determined, then one entry in the context menu is to jump to that location.

Names of global macros in the current package

A slightly lighter *green* foreground color is used to display a symbol in the current package that is the name of a global macro, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named macro.

If the source location where the macro was defined can be determined, then one entry in the context menu is to jump to that location.

Names of global symbol macros in the current package

A slightly lighter *green* foreground color is used to display a symbol in the current package that is the name of a global symbol macro, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named symbol macro.

If the source location where the symbol macro was defined can be determined, then one entry in the context menu is to jump to that location.

Names of special variables in the current package

A dark *green* foreground color is used to display a symbol in the current package if it is in a context where it is the name of a special variable.

The *tooltip* shows the `documentation` entry associated with the named variable.

If the symbol names a variable that is globally special, and if the source location where the variable was defined can be determined, then one entry in the context menu is to jump to that location.

Names of constant variables in the current package

Names of global functions in the `common-lisp` package

A dark *blue* foreground color is used to display a symbol in the `common-lisp` package that is the name of a global function, and that is not used as lexical name (of a variable or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named function.

One entry of the context menu is to show the entry in the HyperSpec associated with the named function.

Names of global macros in the `common-lisp` package

A slightly lighter *blue* foreground color is used to display a symbol in the `common-lisp` package that is the name of a global macro, and that is not used as lexical name (of a variable or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named macro.

One entry of the context menu is to show the entry in the HyperSpec associated with the named macro.

Names of special variables in the `common-lisp` package

A dark *blue* foreground color is used to display a symbol in the `common-lisp` package that is the name of a special variable.

The *tooltip* shows the `documentation` entry associated with the named variable.

One entry of the context menu is to show the entry in the HyperSpec associated with the named variable.

Names of constant variables in the `common-lisp` package

The *tooltip* shows the `documentation` entry associated with the named constant variable.

One entry of the context menu is to show the entry in the HyperSpec associated with the named macro.

Names of special operators

The *tooltip* shows the `documentation` entry associated with the named special operator.

One entry of the context menu is to show the entry in the HyperSpec associated with the named special operator.

Names of global functions in other packages

A dark *magenta* foreground color is used to display a symbol in a package other than the current one or the `common-lisp` package that is the name of a global function, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named function.

If the source location where the function was defined can be determined, then one entry in the context menu is to jump to that location.

Names of global macros in other packages

A slightly lighter *magenta* foreground color is used to display a symbol in a package other than the current one or the `common-lisp` package that is the name of a global macro, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named macro.

If the source location where the macro was defined can be determined, then one entry in the context menu is to jump to that location.

Names of global symbol macros in other packages

A slightly lighter *magenta* foreground color is used to display a symbol in a package other than the current one or the `common-lisp` package that is the name of a global symbol macro, and that is not used as lexical name (of a variable, a function, a macro, or a symbol macro).

The *tooltip* shows the `documentation` entry associated with the named symbol macro.

If the source location where the macro was defined can be determined, then one entry in the context menu is to jump to that location.

Names of special variables in other packages

A dark *magenta* foreground color is used to display a symbol in a package other than the current one or the `common-lisp` package if it is in a context where it is the name of a special variable.

The *tooltip* shows the `documentation` entry associated with the named variable.

If the symbol names a variable that is globally special, and if the source location where the variable was defined can be determined, then one entry in the context menu is to jump to that location.

Names of constant variables in other packages

Argument mismatch

Argument mismatch is a situation that may occur for the following types of forms:

- Function calls.

- Macro calls.
- Special forms.

Argument mismatch corresponds to one of the following situations:

- The number of arguments is less than the minimum number required by the function, the macro, or the special operator.
- The number of arguments is greater than the maximum number allowed by the function, the macro, or the special operator.
- The function, the macro or the special operator admit &key arguments, and there is an odd number of arguments in that part of the argument list.
- The function, the macro or the special operator requires the number of arguments to be even, but an odd number of arguments are given (e.g., for `setq` and `setf`).
- The function, the macro or the special operator requires the number of arguments to be odd, but an even number of arguments are given.

In the case where the number of arguments is greater than the maximum number allowed, each extraneous arguments is highlighted with a vivid *red* background. The tooltip associated with each arguments says "Extraneous argument."

In all other cases, the closing parenthesis is preceded by a *red* rectangle. Notice that this red rectangle does not constitute an item in the buffer, and as a consequence it is impossible to position the cursor between the red rectangle and the closing parenthesis. Rather, the red rectangle is part of the way the closing parenthesis is displayed. In the case where too few arguments were given, the tooltip says "Too few arguments", and in the other cases it says "Wrong argument parity".

Indentation

A line that is indented incorrectly is displayed with a *pink* arrow in the left margin. If the indentation of the line is insufficient, the arrow points to the right, and if the indentation of the line is excessive, then the arrow points to the left.

Figure 2.9 illustrates how this type of information is displayed.

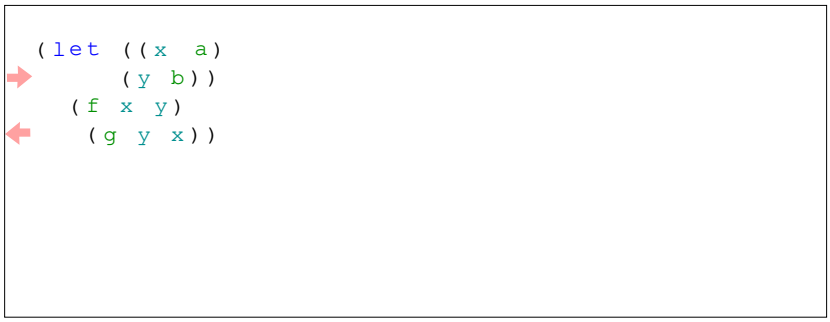

Figure 2.9: Display of extraneous whitespace.

2.1.6 Other analyses

Comments

Comments can be checked for spelling errors.

2.2 Commands

- `Forward expression`
- `Backward expression`
- `Beginning of expression`
- `End of expression`

- Beginning of top-level expression
- End of top-level expression
- Exchange expressions
- Down expression
- Indent line
- Indent region
- Indent top-level expression
- Indent buffer
- Complete symbol

Part II

Extension writer's guide

Chapter 3

General structure

At any point in time Second Climacs contains a certain number of *buffers*. A buffer can only be modified as a result of the execution of a *command*. Such a command can be executed as a result of a *key sequence*, of typing the *name* of the command into a command-line command processor, or of clicking on a button or a menu entry that was designed to execute a command. Generally speaking, most events, such as scrolling or resizing a window, do not result in a command being executed.

Before a buffer can be viewed, its *syntax* must be analyzed by a *syntax analyzer*. Normally, there is a single syntax analyzer for a buffer, and it is chosen based on the contents of the buffer, which is typically determined by the extension of the file from which the buffer was created. Thus, for instance, a buffer containing Common Lisp code is typically analyzed using the analyzer for Common Lisp code. However, it is possible for a single buffer to have several different simultaneous syntax analyzers. The syntax analyzers of a buffer generally analyze the syntax *incrementally* whenever possible. This incremental analysis is triggered by the command loop after the execution of a command. Every syntax analyzer is triggered after each iteration of the command loop, though typically most buffers have not been modified so the incremental analysis then terminates immediately. Though in most cases a syntax analyzer refers to a single buffer, it is possible for a syntax analyzer to refer to several buffer, or even to other syntax analyzers.

All user interaction, both input and output, is mediated through a *view*. The view typically contains a single syntax analyzer which in turn contains a single buffer. More complicated views can contain several syntax analyzers. Each view typically contains a *cursor* into the buffer of its syntax analyzer. Each view also contains a *command processor* to which key strokes are directed when the view is the *current view*. These key strokes may modify the buffer(s) associated with the view, or they may influence the view itself in some way (moving the cursor for instance).

A view may or may not be *visible*. Views that are not visible will still have they associated syntax analyzers updated after each iteration around the command loop, but the processing stops there. Views that are visible also have a *show* associated with them. They show is charged with transforming the result of the syntax analysis to elements of the graphic user interface. In a way analogous to that of a syntax analyzer, a show is updated incrementally from the result of the syntax analyses of the associated view. However, whereas the syntax analyzer is updated at each iteration of the command loop, the show is updated at each iteration of the *event loop*. The reason for this more frequent update is that the show might have to change as a result of events such as resizing a window or scrolling. Because the data structures of a show can take up considerable space, they are discarded when a view is no longer visible, and they are recomputed when the view again becomes visible.

Chapter 4

Writing backends

Chapter 5

Writing syntax analyzers

Part III

Internals

Chapter 6

Representation of the editor buffer

Second Climacs uses the Cluffer library[1] to represent its buffers.

We briefly describe the essential aspects of that library below. For detailed information on how it works, see the dedicated documentation.

Cluffer proposes two distinct protocols, namely the *edit protocol* and the *update protocol*.

The edit protocol provides operations for editing the buffer contents. It has been designed to be both simple and very efficient. As such, it does not provide operations on larger chunks of contents such as *regions*. It only provides operations on single items, and operations to split and join lines. These editing operations do not trigger any view updates which is why they can be invoked a large number of times for each user interaction without loss of performance. This features is taken advantage of in operations on regions and in keyboard macros.

The update protocol is designed to be run at the frequency of the event loop. It is based on the concept of *time stamps*. Any number of edit operations can be performed between two invocations of the update protocol, and the update

[1] See https://github.com/robert-strandh/Cluffer

protocol can be invoked at different times for different views, including very rarely for views that are not currently on display. Given that the amount of data displayed in a view is relatively modest, no attempt is made to minimize the modifications to the view. The smallest unit of an update is a *line* of items.

Chapter 7

General control structure

The general control structure was designed with the following goals:

- Most editing operations should be very fast, even when they involve fairly large chunks of buffer contents. Here, *fast* means that the response time for interactive editing should be short.
- From a software-engineering point of view, the buffer editing operations should not be aware of the presence of any *views*.

Notice that it was *not* a goal that editing operations use as little computational power as possible.

Input events can be divided into two categories:

- Input events that result in some modification to some buffer contents. Inserting and deleting items are in this category. Modifications can be the result of indirect events such as executing a keyboard macro that inserts or deletes items in one or more buffers.
- Input events that have no effect on any buffer contents. Moving a cursor, changing the size of a window, or scrolling a view are typical events in this category. These events influence only the *view* into a buffer.

When an event in the first category occurs, the following chain of events is triggered:

1. The event itself triggers the execution of some *command* that causes one or more items to be inserted and/or deleted from one or more buffers. Whether this happens as a direct result or as an indirect result of the event makes no difference. The buffers involved are modified, but no other action is taken at this time. Lines that are modified or inserted are marked with the *current time stamp* and the current time stamp is incremented, possibly more than once.

2. At the end of the execution of the command, the *syntax update* is executed for all buffers, allowing the contents to be incrementally parsed according to the syntax associated with the buffer.[1] Finally, visible views are repainted using whatever combination they want of the buffer contents and the result of the syntax update. The syntax update uses the time stamps of lines in the buffer and of the previous syntax update to compute an up-to-date representation of the buffer. This computation is done incrementally as much as possible.

3. Each view on display recomputes the data presented to the user and redraws the associated window. Again, time stamps are used to make this computation as incremental as possible.

[1] FIXME: There seem to be cases where the syntax of one buffer depends not only on its own associated buffer, but also on the contents of other buffers. It is not a big problem if the dependency is only on the *contents* of other buffers, but if the dependency is also on the *result of the syntax analysis* of other buffers, then one syntax update might invalidate another. In that case, it might be necessary to loop until all analyses are complete. This can become very complicated because there can now be circular dependencies so that the entire editor gets caught in an infinite loop.

Chapter 8

Common Lisp mode

8.1 Syntax

8.1.1 Parsing using the Common Lisp reader

We use a special version of the Common Lisp reader to parse the contents of a buffer. We use a special version of the reader for the following reasons:

- We need a different action from that of the standard reader when it comes to interpreting tokens. In particular, we do not necessarily want the incremental parser to intern symbols automatically, and we do not want the reader to fail when a symbol with an explicit package prefix does not exist or when the package corresponding to the package prefix does not exists.

- We need for the reader to return not only a resulting expression, but an object that describes the start and end positions in the buffer where the expression was read.

- The reader needs to return source elements that are not returned by an ordinary reader, such as comments and expressions that are skipped by certain other reader macros.

- The reader can not fail when an incorrect character is encountered, nor when end of file is encountered in the middle of a call.

We call the data structure referred to in the last item a *wad*. It contains the following slots:

- The start and the end location of the wad in the buffer. For details on how this wad is represented, Section 8.1.2.
- The expression that was read, with some possible modifications. Tokens are not represented as themselves for reasons mentioned above.
- A list of *children*. These are wads that were returned by recursive calls to the reader. The children are represented in the order they were encountered in the buffer. This order may be different from the order in which the corresponding expressions appear in the expression resulting from the call to the reader.

8.1.2 Data structure

A location in the buffer is considered a *top-level location* if and only if, when the buffer is parsed by a number of consecutive calls to `read`, when this location is reached, the reader is in its initial state with no recursive pending invocations.

The Common Lisp syntax maintains a sequence[1] of *top-level wads*. A wad is considered top-level if it is the result of an immediate call to `read`, as opposed to of a recursive call.

This sequence is organized as two ordinary Common Lisp lists, called the *prefix* and the *suffix*. Given a top-level location L in the buffer, the prefix contains a list of the top-level wad that precede L and the suffix contains a list of the top-level wads that follow L. The top-level wads in the prefix occur in reverse order compared to order in which they appear in the buffer. The top-level wads in the suffix occur in the same order as they appear in the buffer. the location L is typically immediately before or immediately after the top-level expression in which the *cursor* of the current view is located, but that is not

[1] It is not a Common Lisp sequence, but just a suite represented in a different way.

a requirement. Figure 8.1 illustrates the prefix and the suffix of a buffer with five top-level expressions.

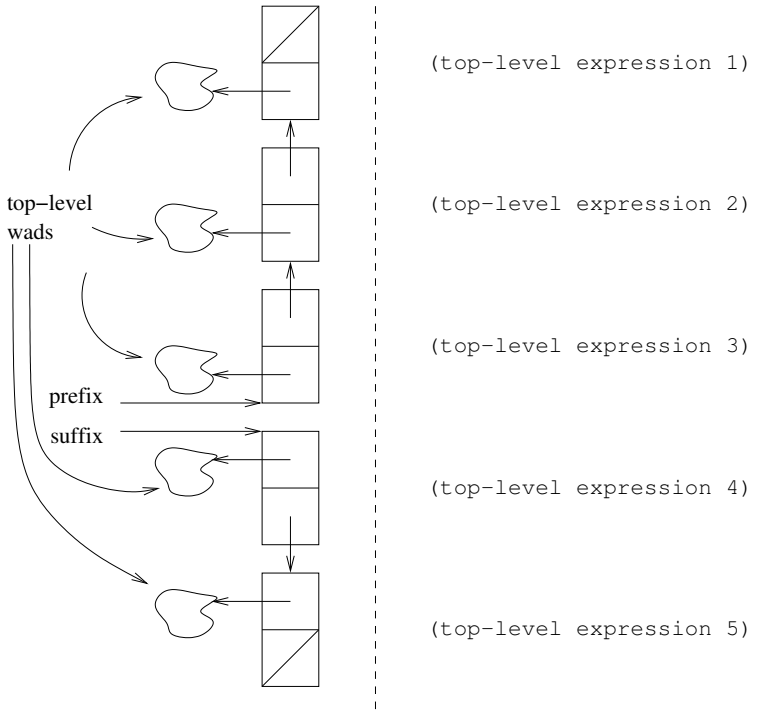

Figure 8.1: Prefix and suffix containing top-level wads.

Either the prefix or the suffix or both may be the empty list. The location L may be moved. It suffices[2] to pop an element off of one of the lists and push it onto the other.

The representation of a wad is shown in Figure 8.2.

Let the *initial character* of some wad be the first non-whitespace character encountered during the call to the reader that produced this wad. Similarly, let the *final character* of some wad be the last character encountered during the call to the reader that produced this wad, excluding any look-ahead character that could be un-read before the wad was returned.

[2]Some slots also need to be updated as will be discussed later.

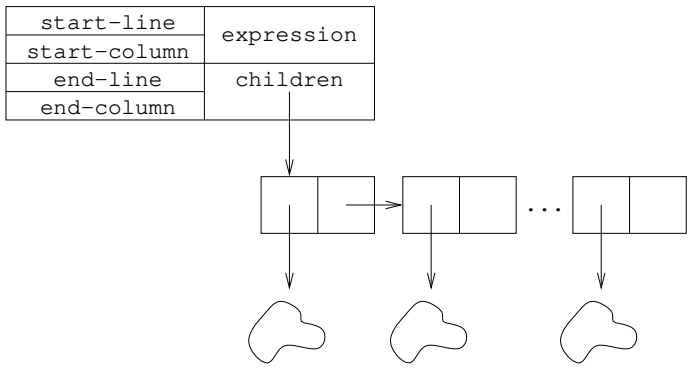

Figure 8.2: Representation of wad.

The slot named `start-line` is computed as follows:

- If the wad is one of the top-level wads in the prefix or the first top-level wad in the suffix, then the value of this slot is the absolute line number of the initial character of this wad. The first line of the buffer is numbered 0.
- If the wad is a top-level wad in the suffix other than the first one, then the value of this slot is the number of lines between the value of the slot `start-line` of the preceding wad and the initial character of this wad. A value of 0 indicates the same line as the `start-line` of the preceding wad.
- If this wad is the first in a list of children of some parent wad, then the value of this slot is the number of lines between the value of the slot `start-line` of the parent wad and the initial character of this wad.
- If this wad is the child other than the first in a list of children of some parent wad, then the value of this slot is the number of lines between the value of the slot `start-line` of the preceding sibling wad and the initial character of this wad.

The value of the slot `start-column` is the absolute column number of the initial character of this wad. A value of 0 means the leftmost column.

The value of the slot `end-line` of some wad is the number of lines between the value of the slot `start-line` of the same wad and the final character of the wad. If the wad starts and ends on the same line, then the value of this slot is 0.

The value of the slot `end-column` is the absolute column number of the final character of the wad.

To illustrate the data structure, we use the following example:

```
...
34 (f 10)
35
36 (let ((x 1)
37       (y 2))
38   (g (h x)
39      (i y)
40      (j x y)))
41
42 (f 20)
...
```

Each line is preceded by the absolute line number. If the wad starting at line 36 is a member of the prefix or if it is the first element of the suffix, it would be represented like this:

```
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
   00 01 ((x 1) (y 2))
      00 00 (x 1)
      01 00 (y 2)
   02 02 (g (h x) (i y) (j x y))
      00 00 (h x)
      01 00 (i y)
      02 00 (j x y)
```

Since column numbers are uninteresting for our illustration, we show only line numbers. Furthermore, we present a list as a table for a more compact presentation.

8.1.3 Moving top-level wads

Occasionally, some top-level wads need to be moved from the prefix to the suffix or from the suffix to the prefix. There could be several reasons for such moves:

- The place between the prefix and the suffix must always be near the part of the buffer currently on display when the contents are presented to the user. If the part on display changes as a result of scrolling or as a result of the user moving the current cursor, then the prefix and suffix must be adjusted to reflect the new position prior to the presentation.
- After items have been inserted into or deleted from the buffer, the incremental parser may have to adjust the prefix and the suffix so that the altered top-level wads are near the beginning of the suffix.

These adjustments are always accomplished by repeatedly moving a single top-level wad.

To move a single top-level wad P from the prefix to the suffix, the following actions are executed:

1. Modify the slot `start-line` of the first wad of the suffix so that, instead of containing the absolute line number, it contains the line number relative to the value of the slot `start-line` of P.
2. Pop P from the prefix and push it onto the suffix. Rather than using the straightforward technique, the `cons` cell referring to P can be reused so as to avoid unnecessary consing.

To move a single top-level wad P from the suffix to the prefix, the following actions are executed:

1. If P has a successor S in the suffix, then the slot `start-line` of S is adjusted so that it contains the absolute line number as opposed to the line number relative to the slot `start-line` of P.

2. Pop P from the suffix and push it onto the prefix. Rather than using the straightforward technique, the `cons` cell referring to P can be reused so as to avoid unnecessary consing.

We illustrate this process by showing four possible top-level locations in the example buffer. If all three top-level wads are located in the suffix, we have the following situation:

```
prefix
...
suffix
34 00 (f 10)
02 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

In the example, we do not show the children of the top-level wad.

If the prefix contains the first top-level expression and the suffix the other two, we have the following situation:

```
prefix
...
34 00 (f 10)
suffix
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

If the prefix contains the first two top-level expressions and the suffix the remaining one, we have the following situation:

```
prefix
...
34 00 (f 10)
```

```
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
suffix
42 00 (f 20)
...
```

Finally, if the prefix contains all three top-level expressions, we have the following situation:

```
prefix
...
34 00 (f 10)
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
42 00 (f 20)
suffix
...
```

8.1.4 Incremental update

Modifications to the buffer are reported at the granularity of entire lines. The following operations are possible:

- A line may be modified.
- A line may be inserted.
- A line may be deleted.

Several different lines may be modified between two incremental updates, and in different ways. The first step in an incremental update step is to invalidate wads that are no longer known to be correct after these modifications. This step modifies the data structure described in Section 8.1.2 in the following way:

- After the invalidation step, the prefix contains the wad preceding the first modified line, so that these wads are still valid.

- The suffix contains those wads following the last modified line. These wads are still valid, but they may no longer be top-level wads, because the nesting may have changed as a result of the modifications preceding the suffix.
- An additional list of *residual wads* is created. This list contains wads that have not been invalidated by the modifications, i.e. that appear only in lines that have not been modified.

The order of the wads in the list of residual wads is the same as the order of the wads in the buffer. The slot `start-line` of each wad in the list is the absolute line number of the initial character of that wad.

Suppose, for example, that the buffer contents in our running example was modified so that line 37 was altered in some way, and a line was inserted between the lines 39 and 40. As a result of this update, we need to represent the following wads:

```
...
34 (f 10)
35
36       (x 1)
37
38      (h x)
39      (i y)
40
41      (j x y)
42
43 (f 20)
...
```

In other words, we need to obtain the following representation:

```
prefix
...
34 00 (f 10)
residual
```

```
36 00 (x 1)
38 00 (h x)
39 00 (i y)
41 00 (j x y)
suffix
43 00 (f 20)
...
```

Processing modifications

While the list of residual wads is being constructed, its elements are in the reverse order. Only when all buffer updates have been processed is the list of residual wads reversed to obtain the final representation.

All line modifications are reported in increasing order of line number. Before the first modification is processed, the prefix and the suffix are positioned as indicated above, and the list of residual wads is initialized to the empty list.

The following actions are taken, depending on the position of the modified line with respect to the suffix, and on the nature of the modification:

- If a line has been modified, and either the suffix is empty or the modified line precedes the first wad of the suffix, then no action is taken.
- If a line has been deleted, and the suffix is empty, then no action is taken.
- If a line has been deleted, and it precedes the first wad of the suffix, then the slot `start-line` of the first wad of the suffix is decremented.
- If a line has been inserted, and the suffix is empty, then no action is taken.
- If a line has been inserted, and it precedes the first wad of the suffix, then the slot `start-line` of the first wad of the suffix is incremented.
- If a line has been modified and the entire first wad of the suffix is entirely contained in this line, then remove the first wad from the suffix and start the entire process again with the same modified line. To remove the first wad from the suffix, first adjust the slot `start-line` of the second

element of the suffix (if any) to reflect the absolute start line. Then pop the first element off the suffix.

- If a line has been modified, deleted, or inserted, in a way that may affect the first wad of the suffix, then this wad is first removed from the suffix and then processed as indicated below. Finally, start the entire process again with the same modified line. To remove the first wad from the suffix, first adjust the slot `start-line` of the second element of the suffix (if any) to reflect the absolute start line. Then pop the first element off the suffix.

Modifications potentially apply to elements of the suffix. When such an element needs to be taken apart, we try to salvage as many as possible of its descendants. We do this by moving the element to a *worklist* organized as a stack represented as an ordinary Common Lisp list. The top of the stack is taken apart by popping it from the stack and pushing its children. This process goes on until either the top element has no children, or it is no longer affected by a modification to the buffer, in which case it is moved to the list of residual wads.

Let us see how we process the modifications in our running example.

Line 37 has been altered, so our first task is to adjust the prefix and the suffix so that the prefix contains the last wad that is unaffected by the modifications. This adjustment results in the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
suffix
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

The first wad of the suffix is affected by the fact that line 37 has been modified. We must move the children of that wad to the worklist. In doing so, we make

the `start-line` of the children reflect the absolute line number, and we also make the `start-line` of the next wad of the suffix also reflect the absolute line number. We obtain the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
36 01 ((x 1) (y 2))
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first element of the worklist is affected by the modification of line 37. We therefore remove it from the worklist, and add its children to the top of the worklist. In doing so, we make the `start-line` of those children reflect absolute line numbers. We obtain the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
36 00 (x 1)
37 00 (y 2)
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first element of the worklist is unaffected by the modification, because it precedes the modified line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
worklist
37 00 (y 2)
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first wad of the top of the worklist is affected by the modification. It has no children, so we pop it off the worklist.

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
worklist
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The modification of line 37 is now entirely processed. We know this because the first wad on the worklist occurs beyond the modified line in the buffer. We therefore start processing the line inserted between the existing lines 39 and 40. The first item on the worklist is affected by this insertion. We therefore remove it from the worklist and push its children instead. In doing so, we make the `start-line` slot those children reflect the absolute line number. We obtain the following result:

```
prefix
...
```

```
34 00 (f 10)
residue
36 00 (x 1)
worklist
38 00 (h x)
39 00 (i y)
40 00 (j x y)
suffix
42 00 (f 20)
...
```

The first element of the worklist is unaffected by the insertion because it precedes the inserted line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
worklist
39 00 (i y)
40 00 (j x y)
suffix
42 00 (f 20)
...
```

Once again, the first element of the worklist is unaffected by the insertion because it precedes the inserted line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
```

```
36 00 (x 1)
38 00 (h x)
39 00 (i y)
worklist
40 00 (j x y)
suffix
42 00 (f 20)
...
```

The first element of the worklist is affected by the insertion, in that it must have its line number incremented. In fact, every element of the worklist and also the first element of the suffix must have their line numbers incremented. Furthermore, this update finishes the processing of the inserted line. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
39 00 (i y)
worklist
41 00 (j x y)
suffix
43 00 (f 20)
...
```

With no more buffer modifications to process, we terminate the procedure by moving remaining wads from the worklist to the residue list. The final situation is shown here:

```
prefix
...
34 00 (f 10)
residue
```

```
36 00 (x 1)
38 00 (h x)
39 00 (i y)
41 00 (j x y)
worklist
suffix
43 00 (f 20)
...
```

Recreating the cache

Once the cache has been processed so that only wads that are known to be valid remain, the new buffer contents must be fully parsed so that its complete structure is reflected in the cache.

Conceptually, we obtain a complete cache by applying `read` repeatedly from the beginning of the buffer, until all top-level wad have been found. But doing it this way essentially for every keystroke would be too slow. In this section we explain how the partially invalidated cache is used to make this process sufficiently fast.

8.2 Computing indentation

8.2.1 Introduction

There are two basic cases for computing indentation of some wad:

1. If the wad is a top-level wad, then the indentation is always 0, i.e. it should be positioned in the leftmost column.
2. If the wad is a nested wad, then the indentation is relative to some ancestor wad.

For each top-level wad, its absolute indentation is first set to 0 and then the function `compute-child-indentations` is called on that top-level wad.

The function `compute-child-indentations` takes two parameters, a wad and a *client* instance.

We must distinguish between the following cases:

1. P is an atomic wad.
2. P represents a special form or a function or macro call with its own indentation rule.
3. P represents a function call without its own indentation rule.
4. P represents a macro call without its own indentation rule.

In case 1, we are done because its indentation has already been computed and it has no children. The remaining cases are treated below.

The different cases are recognized because the function `function-information` of the Cleavir system `cst-to-ast` is called with the operator as an argument and the return value of that function determines whether the operator is a function, a macro, or a special operator.

8.2.2 Special indentation rules

Certain operators have their own indentation rules. In particular, most special operators do, but it is also possible to define special indentation rules for functions and macros.

Such an indentation rule is defined as a method on the generic-function named `compute-child-indentations-special`. This function takes the wad P, a symbol (the name of the operator), and a *client* instance. A method should specialize on the name of the operator (in the form of an EQL specializer) and on the client parameter (unless this is a default method).

The method computes and assigns indentation for the descendants that have fixed relative indentation according to the syntax of the operator, and then recursively calls `compute-child-indentations` in order to recursively compute indentation for the children of those descendants.

8.2.3 Indenting a function call

If P represents a function call, then by default, its children are indented according to the following rules:

- If P has at least two children and the second child is positioned on the same line as the first child, then the remaining children (starting with the third one) are indented so that they align with the second child.
- If either P has only one child (which must then be the function to be called), or the second child is not positioned on the same line as the first child, then every child is indented by two positions relative to P.

However, some special cases exist. For example, when the lambda list of the function has keyword arguments, then keywords are aligned vertically.[3]

8.2.4 Indenting a macro call

The indentation of a macro call depends on the lambda list for the macro. Two major cases are identified:

- The lambda list does not have `&body` in it.
- The lambda list has `&body` in it.

In the first case, the macro call is indented in the same way as a function call.

In the second case, there are two sub-cases:

- If any of the arguments of the body is positioned on the same line as the operator, then every child that is not positioned on the same line as the operator is aligned under the first body argument.
- Otherwise, every child that belongs to the body is indented by two positions relative to P.

[3] Provide a more detailed description.

If there is a child of P preceding the body arguments and the first such child is positioned on the same line as the operator, then every child line preceding the body arguments is indented below the first child. Otherwise, if there is a child of P preceding the body arguments, then every line of such a child is indented by four spaces.

8.2.5 Indenting lambda lists

Part IV

Interfaces

Chapter 9

McCLIM ESA

Part V

Contributing

Chapter 10

General Common Lisp style guide

10.1 Purpose of style restrictions

The purpose of imposing a particular style is based on a few simple facts that hold true for both natural languages and programming languages:

- The set of all idiomatic phrases is a tiny subset of the set of all grammatical phrases.
- The main purpose of these phrases is to serve as communication between people.

To illustrate the first fact, consider a natural language such as English. In English, we say "tooth brush", but "dental floss". The words "dental brush" and "tooth floss" would be perfectly grammatical, but they are not used. A person trying to communicate with other people must use the words that have been widely agreed upon, even though some other words are perfectly legitimate. It might seem that such idiosyncrasies would be limited to languages with multiple heritage such as English, but that is not the case. In French, we say "brosse à dents", "pâte dentifrice", and "fil dentaire". There are nine reasonable combinations, but only one is used.

The same thing is true for programming languages. The community has collectively decided on a particular subset of all the grammatical phrases, and a programmer who wishes to communicate with other programmers should stick to that subset.

It should also be emphasized that the choice of idioms is different in different languages. An example from natural languages would be that in English we say "I was my hands", in French "I wash myself the hands", and in Swedish we say "I wash the hands [on myself]". Just as it would be pointless trying to use an idiom form one language in a translated version in a different language, it is as pointless to translate idioms from one programming language to a different programming language.

Finally, the choice of what phrases are idioms and what phrases are not is almost totally arbitrary, and based on coincidences of history. Therefore it is rarely productive to ask oneself why a particular phrase is an idiom and a different one is not. There is no possible enlightening answer to such a question.

10.2 Width of a line of code

Horizontal space is a precious resource that should not be wasted. The width of a line should preferably not exceed 80 characters. This limit used to be hard, because some printers or printer drivers would truncate longer lines. Since it is less common to print code these days, the limit is now soft. The purpose of keeping lines somewhat short is so that it is possible on a reasonable monitor to display two documents side by side. One document is typically a Common Lisp source file, and the other document is typically the buffer containing interactions with the Common Lisp system.

The systematic use of long lines makes the practice of displaying two documents side by side impossible, or at least impractical. If a single monitor is used, the programmer then has to flip back and forth between the source code and the interaction loop. When two monitors are used, the effect is to waste half a monitor that could otherwise be used for displaying documentation or something else.

10.3 Blank lines

A single blank line is common in the following situations:

- Between two top-level forms.
- Between a file-specific comment and the following top-level form.
- Between a comment for several top-level forms and the first of those top-level forms.

A single blank line *may* occur inside a top-level form to indicate the separation of two blocks of code concerned with different subjects, but it would be more common to put those two blocks of code in separate functions.

There should never be any instance of two consecutive blank lines, and the last line of the file should not be blank.

10.4 `car`, `cdr`, `first`, etc are for `cons` cells

The Common Lisp standard specifies that the function `car`, `cdr`, `first`, `second`, `rest`, etc return `nil` when `nil` is passed as an argument. This fact should mostly be considered as a historical artifact and should not be systematically exploited. Take for instance the following code:

```
(if (first x) ...)
```

To the compiler, it means "execute the false branch of the `if` when either `x` is `nil`, or when `x` is a list whose first element is `nil`".

To the person reading the code, it means something different altogether, namely "`x` holds a non-empty list of Boolean values, and the false branch of the `if` should be executed when the first element of that list is *false*. See also Section 10.5.

10.5 Different meanings of `nil`

Consider the following local variable bindings:

```
(let ((x '())
      (y nil)
      z)
  ...)
```

To the compiler, the three are equivalent. To a person reading the code, they mean different things, however:

- The initialization of `x` means that `x` holds a *list* that is initially empty.
- The initialization of `y` means that `y` holds a Boolean value or a default value that may or may not change in the body of the `let` form.
- The absence of initialization of `y` means that no initial value is given to `z`. In the body of the `let` form, the variable `z` will be assigned to before it is used.

The following body of the `let` form corresponds to the expectations of the person reading the code:

```
(let ((x '())
      (y nil)
      z)
  ...
  (push (f y) x)
  ...
  (unless y (setf y (g x)))
  ...
  (setf z (h x))
  ...)
```

The following body of the `let` form violates the expectations of the person reading the code:

```
(let ((x '())
      (y nil)
      z)
  ...
  (push (f y) z)     ; z is used before it is assigned.
  ...
  (unless x          ; x is treated as a Boolean.
    (setf y (g x)))
  ...
  (push (f x) y)     ; y is treated as a list.
  ...)
```

10.6 Tests in conditional expressions

The *test* of a conditional expression should be a (possibly generalized) Boolean expression. The following expressions correspond to the expectations of the person reading the code:

```
(if visited-p ...)
(when (member ...) ...)
(cond ((plusp x) ...) ...)
```

The following code violates the expectation:

```
(let ((item (find ...)))
  (when item ...))
```

because `item` is not a (generalized) Boolean value. It is an item returned by `find`, though there is an *out of band* value (`nil`) indicating that no item was found by `find`. In this case, the corresponding code that corresponds to the expectations would look like this:

```
(let ((item (find ...)))
  (unless (null item) ...))
```

10.7 General structure of recursive functions

When possible, a recursive function should be structured like a mathematical proof by induction. By that we mean that the special case should be handled *first* so as to reassure the person reading the code that this case can be handled correctly by the function.

So for instance, assume we have want to write a function that counts the number of atoms in a tree, we should not write it like this:

```
(defun count-atoms (tree)
  (if (consp tree)
      (+ (count-atoms (car tree))
         (count-atoms (cdr tree)))
      1))
```

but rather

```
(defun count-atoms (tree)
  (if (atom tree)
      1
      (+ (count-atoms (car tree))
         (count-atoms (cdr tree)))))
```

Even when the base case does not return anything useful, it should be handled first. The following code violates the expectations:

```
(defun map-conses (function tree)
 (unless (atom node)
   (funcall function node)
   (traverse (car node))
   (traverse (cdr node))))
```

and should be written like this instead:

```
(defun map-conses (function tree)
  (if (atom node)
      nil ; nothing to do
      (progn (funcall function node)
             (traverse (car node))
             (traverse (cdr node)))))
```

though, admittedly, this example is a little too simple to illustrate the importance of this rule.

10.8 Using `car` and `cdr` vs. using `first` and `rest`

While the two functions `car` and `first` have the exact same definitions, as do `cdr` and `rest`, they send very different messages to the person reading the code.

The functions `car`, `cdr`, etc., should be avoided when the argument is to be considered as a *list*, and should be reserved for other uses of `cons` cells such as for *trees* or *pairs* of values.

It follows that the two families of functions should never be mixed for the same argument.

10.9 Commenting

Comments are meant to be read by the maintainers of the code. One can therefore safely assume that the reader is quite familiar with the Common Lisp language and the main structure of CLIM and Second Climacs.

Comments should be used to explain aspects of the code that are not obvious from reading the code itself. A comment is the ideal place to introduce definitions of concepts that must be understood in order for the code to make sense. If the code is structured in a particular way for performance reasons, then a comment is a good place to indicate such a fact, so as to avoid that the maintainer be tempted to introduce modifications that alter this structure.

10.10 Designators for symbol names

Always use uninterned symbols (such as `#:hello`) whenever a string designator for a symbol name is called for. In particular, this is useful in `defpackage` and `in-package` forms.

Using the upper-case equivalent string makes the code break whenever the reader is case-sensitive (and it looks strange that the designator has a different case from the way symbol that it designates is then used), and using keywords unnecessarily clutters the keyword package.

10.11 Docstrings

We believe that it is a bad idea for an implementation of a Lisp system to have docstrings in the same place as the definition of the language item that is documented, for several reasons. First, to the person reading the code, the docstring is most often noise, because it is known from the standard what the language item is about. Second, it often looks ugly with multiple lines starting in column 1 of the source file, and this fact often discourages the programmer from providing good docstring. Third, it makes internationalization harder.

For this reason, we will provide language-specific files containing all docstrings of Common Lisp in the form of calls to `(setf documentation)`.

10.12 Naming and use of slots

In order to make the code as safe as possible, we typically do not want to export the name of a slot, whereas frequently, the reader or the accessor of that slot should be exported. This restriction implies that a slot and its corresponding reader or accessor cannot have the same name. Several solutions exist to this problem. The one we are using for Second Climacs is to have slot names start with the percent character ('`%`'). Traditionally, a percent character has been used to indicate some kind of danger, i.e. that the programmer should be very careful before directly using such a name. Client code that attempts to use

such a slot would have to write package::%name which contains two indicators of danger, namely the double colon package marker and the percent character.

Code should refer to slot names directly as little as possible. Even code that is private to a package should use an internal protocol in the form of readers and accessors, and such protocols should be documented and exported whenever reasonable. It sometimes good practice to have multiple accessors for a slot, one for internal purposes and one for use by client code. This practice allows for :before, :after, and :around methods on one accessor but not the other.

10.13 Using other packages

The :use option of defpackage and the use-package function should be restricted to the common-lisp package as much as possible. The reason for this restriction is that using a package this way represents a commitment to accepting all exported symbols of that package, current and future, whereas in most cases there is no guarantee that future modifications of the package will not introduce symbol conflicts. If it is desired to avoid explicit package prefixes in some cases, then it is better to use the :import-from option of defpackage to import an explicitly-supplied list of symbols.

10.14 Conditions, restarts, and reporting

Conditions should not be simple conditions, because we want condition reporting to be subject to internationalization. For the same reason, the condition reporter should not be part of the define-condition form, and instead be written separately in a file that contains language-specific condition reporters.

Restarts should be provided whenever practical.

10.15 Internationalization

We would like for Second Climacs to have the ability to report messages in the local language if desired. For this, we use the library named `acclimation`.[1]

10.16 Threading and thread safety

Consider the use of locks to be free. A technique called "speculative lock elision" is already available in some processors, and we predict it will soon be available in all main processors.

[1] See https://github.com/robert-strandh/Acclimation

Part VI

Appendices

Appendix A

Common Lisp mode

In this chapter, we describe a preliminary design for the data structure to represent the result of the syntax analysis of Common Lisp programs. This data structure is to be considered obsolete.

A.1 Syntax

The Common Lisp syntax maintains a tree representing the contents of the buffer. A node in the tree is either a *code node* or a *binary tree node*.

⇒ `code` [*Class*]

This class has a slot representing the position in the buffer where this code fragment starts and ends.

⇒ `binary-tree` [*Class*]

A binary tree node represents a *sequence*[1] of code fragments. If a buffer contains more than one consecutive top-level expression, then the root node of the buffer representation is a binary-tree node.

The binary-tree nodes are treated as a *splay tree* [ST85], in that they can be dynamically reorganized according to the access pattern. A binary-tree node

[1] Not a Common Lisp sequence; just a suite.

has a *contents* slot that contains a code node. It also has a *left child* and a *right child* slot. The child of a binary-tree node can be another binary-tree node or a code node. Either the left child or the right child of a binary-tree node may be `nil`, but not both.[2]

There are three subclass of `code`:

⇒ `expression` [*Class*]

This class is a subclass of the class `code`. It represents a Common Lisp expression. The node has a slot containing the expression being represented. An expression node also contains a slot holding its *children*. This slot may contain `nil` if the node has no children, another expression node if the node has exactly one child, or a binary-tree node if the node has more than one child.

⇒ `whitespace` [*Class*]

This class is a subclass of the class `code`. It represents a sequence of whitespace characters. A node of this type is only present when the whitespace is located at the beginning of a line, and even there, it is optional. It is present only for longer sequences of whitespace.

⇒ `comment` [*Class*]

This class is a subclass of the class `code`. It represents a Common Lisp comment.

In order to avoid updating the entire tree whenever text is inserted or deleted, positions of code are *relative* to some other position p. A position takes the form $< l, c >$, where l indicates *lines* and c indicates *columns*. If $l = 0$, then c is the number of columns to add to the position indicated by p to obtain the new position. If instead $l > 0$, then l indicates the number of lines between p and the new position, and c is the *absolute* position from the beginning of the line.

The following rules apply in order to determine the origin of a relative start position of some node n.

- If the parent of n is a code node, then the start position of n is relative

[2] If both children were `nil` the contents node of the binary-tree node would replace the binary-tree node.

to the *start position* of the parent.

- If the parent of n is a binary-tree node, and n is the *left child* of its parent, then the start position of n is relative to the *start position* of its parent, and its value is always $< 0, 0 >$.
- If the parent of n is a binary-tree node, n is the *contents node* of its parent, and the left child of the parent is `nil`, then the start position of n is relative to the *start position* of its parent, and its value is always $< 0, 0 >$.
- If the parent of n is a binary-tree node, n is the *contents node* of its parent, and the left child of the parent is not `nil`, then the start position of n is relative to the *end position* of the left child.
- If the parent of n is a binary-tree node, n is the *right child* of its parent, then the start position of n is relative to the *end position* of the *contents node* of its parent.

The end position given in some node n is relative to the start position of n.

Consider the following buffer contents, where the initial left parenthesis is positioned in column 0:

```
(let ((x 1)) ; comment
  x)
```

The code tree for that code fragment is shown in Figure A.1.

Common Lisp syntax contains a special version of the Common Lisp *reader*. It differs from the standard reader in the following ways:

- It never signals an error.
- It records the start and end position of every call, as well as the object read.
- Instead of calling `intern` on symbols, it merely records that character sequence as being a symbol in the current package.

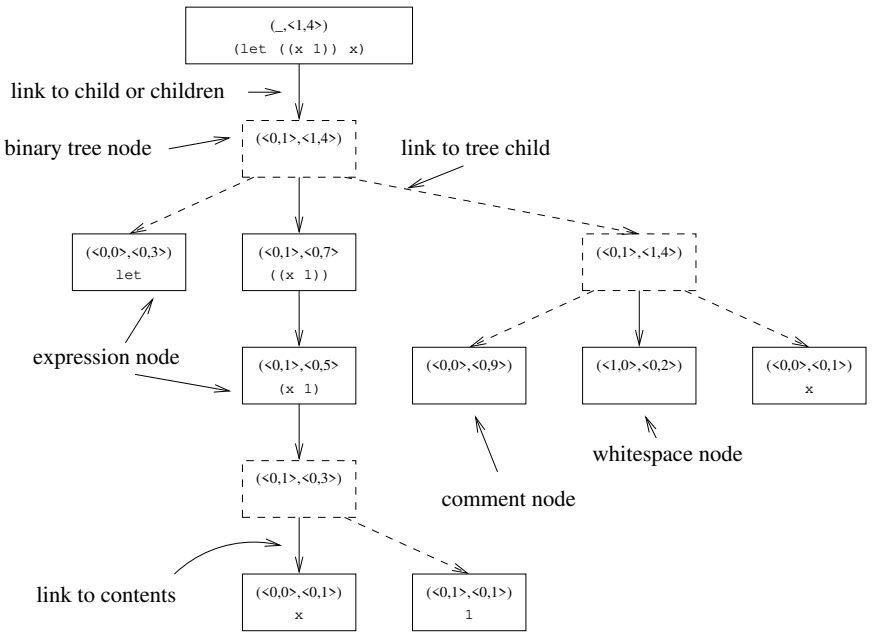

Figure A.1: Example of code tree.

On the other hand, it behaves like the ordinary Common Lisp reader in that it can handle custom reader macros, even though it provides reader macros for standard macro characters that behave slightly differently from the standard reader macros.

When some illegal syntax is encountered, it tries to do something reasonable. For instance if *end of file* is encountered in the middle of reading a list, the end of file is treated as terminating the list. When an illegal token is encountered, an object is returned that indicates this fact.

Bibliography

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

Index