Emacs Skill-mode 3.2
User's manual

Edition February 1996

Jonas Jarnestrom

LZF 032 28/1M RD

1 Preface

This manual documents the use and customization of Skill-mode, a language sensitive package for Skill programming in Emacs. The reader is expected to have a programming background, to be familiar with Emacs, and preferrably to have read the basic chapters of the GnuEmacs manual. The common terms and concepts from the GnuEmacs manual will be used throughout this manual.

1.1 Text Conventions

Italics is used for emphasis, and for the first occurrence of special terms.

Bold is used for strong emphasis, and for the accurate command and variable names.

`Courier` is used for examples and for commands that should be typed in exactly as printed.

<headline> - "Nnnnn" denotes the corresponding entry in the menu system.

<command headline> : C-c C-k
shows the command key binding if applicable.

<option headline> : t shows the standard default value.

1.2 Credits

Richard Stallman for being the originator of the most powerful and most fascinating general-purpose text editor ever written. Further, for being the only president of any software house known to me, who frequently personally responds to and takes action on incoming bug reports. Very persistenly, year after year.

Jamie Zawinski for excellent work on Lucid Emacs.

Wouter Batelaan and Alan Cochran for the GateToOpus link.

Steve Atkins for help with the DFII 4.3 symbol libraries.

Jerry Duggan for the original file header code.

Shiv Sikand, Alan Cochran, Robert Shaw, Anette Haggstrom, Jacek Kaim, Rolf Engstrand, Henry Chalup, Tony Watering for good feed-back and support.

Barry Warsaw, Jim Blandy, Dan LaLiberte, Joe Wells, Dale Worley, Barry Margolin, Jeff Peck, Kyle Jones, for good help in the News group gnu.emacs.bug

1.3 Dedication

I dedicate this release to Mrs Miep Gies, 87, the sole still living guardian angel of the jewish Anne Frank family during their hiding in Amsterdam from june 1942 to august 1944 in World War II.

Miep has honoured Anne's memory every day ever since, among other things by corresponding with thousands of school kids from all over the world, and by keeping her home open to a seemingly never-ending supply of fascinated Diary readers.

1.4 If You Like Skill-mode...

A lot of work and dedication has gone into Skill-mode since its first stumbling steps back in january 1989. Since 1991 all the development has been done entirely on unpaid spare-time, and all in all it's a 2 man-year effort.

Thus, If you like skill-mode and find that it saves you time and money, or promotes your business in any other way, I hereby ask you for donations of money, if possible on an annual basis. In particular, if you use support resources or is granted specific enhancement requests, I ask you for donations of money.

The bank details is as follows :
Bank : Nordbanken Sweden , swift-address : NBBKSESS
Account No : 590415-3912
My name : Jonas Jaernestroem

My snail-mail :
Jonas Jarnestrom
Kransbindarvag 9
126 36 Hagersten
Sweden

My email :
erajonj@kieras90a.ericsson.se

2 Introduction

The main objectives for Skill-mode were :

- Boost the productivity of the Skill developer.
- Minimize human keyboard input and eliminate spelling errors.
- Improve readability, documentation, and maintainability of the source.
- Support top-down as well as bottom-up programming styles.
- Provide on-line language knowledge for non-expert users.
- Avoid imposing any particular programming style.

2.1 Features

- Gate-to-Opus link
- Incremental completion mechanism.
- Comprehensive symbol libraries for DF-II 4.3
- Automatic argument help for 2200 standard functions.
- Brief documentation on 1200 standard functions.
- Hierarchy browser for browsing and moving through hierarchies.
- Hierarchical motion for effective navigation.
- Full-featured documentation headers for functions and files.
- Automatic declaration of local variables.
- Automatic indentation
- Automatic generation of end-comments.
- Auto-spacing for symmetry, uniformity and readability.
- Automatic check of parenthesis balance.
- Automatic case-unfolding.
- Syntax highlighting with colour or font variations.
- Comprehensive menu system
- Extensive, entirely menu-controlled customization.
- On-line user's manual.

3 General Characteristics

Skill-mode supports both C style and Lisp style syntax, though not simultaneously in the same file.

The help info on formal arguments and incremental completion can be shown in focus, right at the cursor, to be clearly visible without the need to change eye focus.

Skill-mode is very strong at handling program structures of function trees. Thus, it's recommended that you strive to partition your programs in many small functions (say, 10-30 lines) and make massive use of argument lists for data transport between functions.

Skill-mode provides support for bottom-up as well as top-down programming methodologies. Functions can be defined before use or vice versa. In either case, skill-mode stores the names and arguments when they first appear for use later on.

3.1 Context Sensitive Command Input

A general UI principle is that whenever a command requires an argument it tries to collect it from the context of the cursor. If pointing at a valid object, it is taken as argument without prompting for user input. If not pointing at a valid object, user input is prompted for.

It is normally sufficient to enter unique substrings. Case is ignored and completion is normally available on SPC. Default value is normally taken from the text preceding point, when applicable.

In Emacs-19, completion is also available on the mouse. For example, in Xemacs the mouse can be used to find candidates for the descend command whereas in GnuEmacs the mouse can pick items in the completion help window.

3.2 Optional Arguments for Added Flexibility

Most commands in Skill-mode takes optional arguments in order to make them more flexible. In many cases they define a repeat count for the command. If you want to get the most out of Skill-mode, it's recommended that you check the on-line documentation for your most frequently used commands.

There are two ways to supply prefix arguments, `M-<digit>` or `C-u <digit>`. **C-u is generally recommended in skill-mode since M-1 through M-6 have been rebound to completion commands**. C-u without digit has the special meaning of "multiply by four", so that `C-u` yields 4, `C-u C-u` yields 16 and `C-u C-u C-u` yields 64.

Prefix argument is entered before the basic command, and can be used with key bindings as well as extended command names. Thus, if you for example want to indent the current function with the rewrite-comment option, you can type either `C-u C-c Tab` or `C-u M-x il-indent-fun.`

4 A Note on the Skill Syntax

The Skill syntax is a peculiar blend between Franz-Lisp and C. As a consequence, you can do most things in at least two different ways. In other words, the resulting grammar is quite cluttered and unfortunately a rather poor base for building a comprehensive language mode with reasonable effort. As a consequence, Skill-mode contains more syntactic restraints than would have been needed for a cleaner language. My work with Skill-mode has revealed some of the drawbacks of blending two unrelated, self-contained languages like Lisp and C.

Genuine Lisp syntax has an obvious simplicity that is lost to a large extent when the C style syntax is used, but for obscure "consistency" reasons, Cadence has always recommended the C style syntax. Lisp seem to lend itself fairly well for elaborate syntax extensions such as the Skill format, but it is hard to hide the fact that it is still little else than a Lisp interpreter in disguise.

4.1 Cluttered Grammar Creates Problems

The most obvious example of the cluttered grammar is function calls, where either the algebraic or the prefix notation is allowed, when placing the opening parenthesis. Skill-mode supports both notations, though not simultaneously in the same file.

Another significant example is that two types of comment are allowed ; block-oriented (C style) and line-oriented (Lisp style). This creates problems for the Skill-mode parser since block-oriented comments are hard to detect from an arbitrary cursor position point of view. It virtually requires scanning from beginning of file to reliably answer the question "Is the cursor inside a comment ?". As a consequence, the use of block-oriented multi-line comments is not fully supported and not recommended to use.

Another example is the algebraic functions, which exist both as infix operators and as functions. For example : plus(x y) vs x + y .

4.2 Do's and Don'ts

- The beginning-of-function must start at left margin and match the setting of admin-option **il-fun-begin-regexp** , to be recognized as a local function.
- The end-of-function syntax must match the setting of admin-option **il-fun-end-regexp**, to be recognized as the end of a local function, i.e with a closing parenthesis followed by an optional single-line comment and a blank line, if the default setting is used.
- Don't mix C style and Lisp style syntax in the same file.
  Skill-mode assumes the syntax used in the first clause starting at left margin. When you initially start with an empty buffer, C style syntax is assumed. To switch to Lisp style, restart by typing `M-x skill-mode` when you have written the first function header.
- For assignments, always use the '=' operator in C style, and *setq* in Lisp style. If you don't, those variables will be ignored by the symbol manager. Thus, variables created in other ways, as in `sprintf(var "foo")` or `foreach(var '(1 2 3)...` are not detected.
- Use line-oriented comments if possible.
- Avoid question mark in local symbol names.
  If you insist, let Skill-mode auto-declare it, or make sure to quote it with a '\', as in `finished\?` . There must never be unquoted question marks in a local declaration list.
- Don't delimit function arguments with commas.
- Don't use the "super right bracket" ']' to end functions or expressions.

5 A Note on Keybindings

5.1 Keybinding Conventions

The GnuEmacs manual outlines some general rules for major modes, mainly for the indentation and comments. Skill-mode adheres closely to these rules and to the de facto standard found in Lisp-mode and C-mode, who are the closest relatives. **Notable exceptions are M-1 through M-6 and M-@ through M-%, which are rebound to completion commands.**

5.2 Using the Meta Key

The great majority of the more powerful commands in Emacs are accessed through the Meta key. Since many (older) terminal models lack meta keys, the ESC key is used as stand-in, although it's a poor replacement for a genuine meta key. You simply cannot enjoy the full benefit of Emacs without it.

The meta key works very similar to the Cntrl key, and is pressed simultaneously with Cntrl or Shift to form the more complex key sequences. To repeat the command, simply keep these prefix keys down while repeating the last key in the sequence. (As opposed to ESC, where you need to repeat the entire key sequence.)

Thus, if your terminal is equipped with meta keys, I recommend you to start using them right away. If they are not operational in Emacs, contact your system administrator and kindly ask him to bring home an Emacs binary compiled for X windows. Don't discard ESC for good though; it's still a good complement when within close reach.

If you're interrested in minimizing your hand movements, try this finger map : left pinky on Cntrl, right pinky on Meta. This pattern allows both hands to share the fingerwork more evenly and work more in parallell, i.e. faster.

6 Grey-shading in the Menu System

In order to achieve proper grey-shading of the region-oriented commands in the menu system, the standard option **transient-mark-mode** (GnuEmacs) / **zmacs-regions** (XEmacs) must be set to t in your .emacs file. (This setting will also cause the selected region to vanish at a subsequent region-command.)

7 Motion Commands - the "Move" main-Menu

The move commands fall in three categories; hierarchical, unbalanced and balanced motion. Hierarchical motion follows the function calling trees, balanced motion follows the parenthesis clauses, whereas unbalanced motion does not.

7.1 Hierarchical Motion

Hierarchical motion consists of commands for vertical motion : **il-move-descend** & **il-move-ascend**, and commands for horizontal motion : **il-move-next-child** & **il-move-prev-child**. A descend-path stack keeps track of previous locations as you descend progressively further down so that subsequent ascends will follow the path in reverse. (These markers are separate from user markers that are set by C-SPC (**set-mark-command**) so they can never interfere.) The horizontal move-commands strive to stick to the current function; if no more children found, point will not move on until the command is repeated.

7.1.1 Commands

il-move-descend - "Descend..." : M-n

Descend to a local, external or unknown function at point.
If no function found, prompts for a name. Default is the next local call in current fun (including callback strings). Use SPC for completion (or in XEmacs: move mouse over code to get options hilited and select with button2).

If you enter an unknown fun name or substring, prompts for a case-insensitive grep search in adjacent directories, according to **il-move-search-path**. If you enter a non-child function or if you have abandonned the descend path created by previous descends, the descend-path is reset.

With argument, unconditionally move to beginning of function.

Point is moved to: (in falling priority)
a) A previous descend/ascend point from current session: "(marker)".
b) An edit point from current session: "(new edit)".
c) An edit point from a previous session: "(month day)" and userid.
d) Beginning of function: "(beginning)".

With **il-move-descend-ignore-old-edit** set, item (c) above is bypassed. The echo area displays point status enclosed by parentheses.

Options: il-move-search-path, il-move-descend-ignore-old-edit
Minor mode: il-tree-mode

il-move-ascend - "Ascend" : M-p

Ascend to a local or external parent by following a descend path.
If you have left the descend path, it is reset. If no descend path, search backward for a parent (including callback strings). If multiple parents, pick the nearest backward one. If the browser is used, parent is instead fetched from the tree. External parent is considered only when there is no local parent and a previous descend to current fun has been made from another buffer. Point must be inside a fun when calling.

Minor mode: il-tree-mode

il-move-next-child - "Next Child" : C-c C-n

Find the next child call in the current fun (or file).
With argument, find next external child. Comments and strings are not searched. If no more children found in current fun, point will not move on until command is repeated.

Minor mode: il-tree-mode

il-move-prev-child - "Prev Child" : C-c C-p

Find previous child call in the current fun (or file).
With argument, find external child. Comments and strings are not searched. If no more children found in current fun, point will not move on until command is repeated.

Minor mode: il-tree-mode

7.1.2 The Hierarchy Browser - Skill-tree Mode

skill-tree-mode

Major mode for browsing and moving through hierarchy trees.
Used as input & output device for the hierarchical move commands.
'Point-follow-mouse' mechanism for simple commuting between Skill and Skill-Tree windows. Scrolling is done with the scroll-bar.

With **il-tree-mode** on, the browser is linked to the hierarchical move commands. New tree-versions are stored in **il-tree-archive** and old versions are automatically pruned in the process.

Admin-options: il-tree-archive

7.1.2.1 Browser Commands

il-tree-move-child-call : mouse-1

Go to the first occurrence of the mouse-selected function call.
Requires that selected child function has a local parent, to be meaningful. Applies to Skill-Tree buffers only.

il-tree-move-descend : mouse-2, M-n, right-arrow

Descend to the mouse-selected function.
A Skill-tree wrapper to il-move-descend

il-tree-move-ascend : M-p, left-arrow

Ascend to the parent function.
A Skill-tree wrapper to **il-move-ascend**

il-tree-describe-fun : mouse-3

Describe the mouse-selected function.
A Skill-tree wrapper to il-describe-fun

il-tree-move-next-child : C-c C-n, down-arrow

Move to the next child call.
A Skill-tree wrapper to il-move-next-child

il-tree-move-prev-child : C-c C-p, up-arrow

Move to the previous child call.
A Skill-tree wrapper to il-move-prev-child

7.1.2.2 Skill-tree-mode's Admin-options

il-tree-archive: "~/.skill-mode"

The archive dir for the hierarchical trees created by the browser.

7.1.3 Hierarchical Move Options - "Customize->Options->Move"

il-move-descend-ignore-old-edit : t

If t, ignore old edit point and instead goto beginning of fun.
If nil, old edit points come third in priority (see il-move-descend).

il-move-function-menu : t

Add a `jump-table of functions' menu to the menubar.
Provides direct access to each function in the file. Slows down find-file slightly. A mark is set at the old position, so you may return with C-u C-SPC. In XEmacs the menu can also be popped-up with Shift-button3 or F8.

Uses the standard packages *func-menu* (XEmacs) and *imenu* (GnuEmacs), which are both highly customizable. See the sources for further details. In XEmacs, skill-mode sets fume-menubar-menu-name,fume-scanning-message and fume-rescanning-message. For performance reasons fume-scanning-message is nil and auto-rescan turned off, why rescan must be done manually with "Rescan buffer".

il-move-search-path: "~/*.il ../*/*.il ../*.il"

Directories and file names to search for unknown functions.
May be absolute or relative paths (ie relative to the current buffer-file)
Used by **il-move-descend** when searching the file system.

7.2 Unbalanced Motion

Unbalanced motion consists of commands that move to interesting spots independently of the parenthesis clauses.

7.2.1 Commands

il-move-begin-fun - "Begin of Fun" : C-M-a

Move backward to the nearest beginning-of-fun.
If already at beg-of-fun, move to the previous one.
With numeric argument, do this n times.

Admin-options: il-fun-begin-regexp

il-move-end-fun - "End of Fun" : C-M-e

Move forward to the nearest end-of-function.
If already at end-of-fun, move to the next end-of-fun.
With numeric argument, do this n times. If unbalanced parentheses, issue warning message and beep terminal, then move forward to next beginning-of-fun and search backward for end-of-fun.

`End-of-fun regexp mismatch' means that the pattern following the end-parenthesis doesn't look like an end-of-fun (though it might still be syntactically correct).

Admin-options: il-fun-end-regexp

il-move-last-edit - "Last Edit" : C-c C-l

Find last edit in current function or current buffer.
With argument (C-u), move to last edit from previous session. If not in a function when called, move to the last top-level edit in current session, or to last edit from previous session (same point as when visiting the file). Point status is displayed in the echo area.

il-move-declaration - "Local Decl." : C-c C-d

Find the local declaration of the current function. Set mark.
Return with C-x C-x.

il-move-forward-cexp - "Forward Cexp" : C-c C-f

Move forward to next control expression.
With numeric argument, do this n times.

il-move-backward-cexp - "Backward Cexp" : C-c C-b

Move backward to previous control expression.
With numeric argument, do this n times.

7.2.2 Unbalanced Move Admin-options

il-fun-begin-regexp : "^procedure(\\|^(procedure \\|^(defun "

Search pattern used for determining the syntax of fun start.
The search is case-sensitive so the pattern must match exactly. Note that this regexp is only used to find the first fun definition in the file and store that pattern in the var il-function-start. All subsequent funs must conform to this pattern to be recognizable. Fun definitions must begin at left margin to be recognized.

il-fun-end-regexp : "[ ]*\\([;/].+\\)? *\\(\\|\\'\\)"

Syntax of end-of-function, as seen after the closing parenthesis.
The default setting means: white space(s) or a comment followed by a blank line or end-of-buffer. Avoid fiddling with unless you are well familiar with Emacs regexps.

7.3 Balanced Motion - the "Balanced" Menu

If you want to move (over), copy, kill or transpose Skill expressions, this is best done with the commands that works on balanced expressions. By using these, you make shure that the parenthesis balance is always retained, as opposed to when using word- or line-oriented commands.

By convention, Emacs keys for dealing with balanced expressions are usually C-M- characters. They tend to be analogous in function to their M- and C- equivalents.

These commands fall into two classes. Some deal only with lists (parenthetical groupings). They see nothing except parentheses, and escape characters that might be used to quote those.

The other commands deal with expressions or sexps. The word `sexp' is derived from s-expression, the ancient term for an expression in Lisp. But in Emacs, the notion of `sexp' is not limited to Lisp. It refers to an expression in whatever language your program is written in. Sexps typically include symbols, numbers, and string constants, as well as anything contained in parentheses, brackets or braces. *For Skill editing, the sexp is normally a much more versatile and useful object than the list.*

In languages that use prefix and infix operators, such as Skill, it is not possible for all expressions to be sexps. For example, Skill-mode does not recognize foo + bar as a sexp, even though it is a Skill expression; it recognizes foo as one sexp and bar as another, with the + as punctuation between them. This is a fundamental ambiguity: both foo + bar and foo are legitimate choices for the sexp to move over if point is at the f. Note that (foo + bar) is a sexp in Skill-mode.

The sexp definition has been changed in Skill-mode to include function calls written in the C style syntax. This is to achieve a consistent behaviour regardless of whether Lisp style or C style syntax is used. Thus a function call is always a sexp.

7.3.1 Commands

il-forward-sexp - "Forward Sexp" : C-M-f

Move forward across one balanced sexp, including end-comment. With numeric argument, do this n times. In skill-mode, a function call is always a sexp.

il-backward-sexp - "Backward Sexp" : C-M-b

Move backward across one balanced sexp, ignoring end-comments. With numeric argument, do this n times. In skill-mode, a function call is always a sexp.

forward-list - "Forward List" : C-M-n (Emacs standard function)

Move forward across one balanced group of parentheses.
With argument, do it that many times. Negative arg -N means move backward across N groups of parentheses.

il-backward-list - "Backward List" : C-M-p

Move backward across one balanced group of parentheses.
With numeric argument, do this n times.

down-list - "Down List" : C-M-d (Emacs standard function)

Move forward down one level of parentheses.
With argument, do this that many times.A negative argument means move backward but still go down a level. In Lisp programs, an argument is required.

il-backward-up-list - "Up List" : C-M-u

Move backward out of one level of parentheses.
With numeric argument, do this n times.

8 Completion Commands - the "Complete" main-Menu

The completion commands uses symbol libraries that originates from the Skill on-line documentation and from the sourcecode of the visited working files. Thus, by using them the spelling will always be correct. Another advantage is speed ; completion is normally faster than manual keybord input, especially for a novice programmer.

Good and descriptive symbol names tend to be rather long, but when writing code programmers generally avoid long names, as they are tedious to write and easier to misspell. Using completion, you can keep the long names without paying the normal penalty.

8.1 The Completion Mechanism - How it works

There are two ways to call the completion mechanism:

- Explicitly, with one of ten Meta keys.
- Implicitly , with the electric SPC key, as realized by *Complete* minor mode, see Implicit Completion - "Complete" Minor Mode on page 48.

The completion mechanism performs three basic tasks :

- Completes symbols
- Lists symbol summaries
- Lists symbols that match a regular expression (apropos)

8.1.1 Characteristics

8.1.1.1 Context sensitive

If there is a pattern preceeding point, try to complete that pattern. If no pattern, show a summary of the seleceted symbol type.

8.1.1.2 Incremental Completion loop

The hierarchical prefix naming conventions used in DFII creates large and very regular prefix trees poorly suited for completion. This difficulty is addressed by an incremental completion loop that works on every new character that is entered, serving as a "guide" through the prefix tree. You move "down the tree" by entering characters and "up the tree" with DEL. Once you have entered the loop, you are bound to end up with an existing and correctly spelled symbol.

8.1.1.3 Options for the next segment

At each node in the name tree, your options for the next segment are shown. If you enter a non-existent character, it is rejected and you are prompted for a new try. A special option is "--" which represents the *empty string* . It always appears as the first option and is selected with the SPC key.

8.1.1.4 Control keys for the input loop

ESC	Quit.
SPC	Select the first option.
DEL	Undo last segment (Quit if no more segments to undo).
C-v	Scroll help window forward.

8.1.1.5 Case Unfolding

You can do most of the typing in lowercase letters, and disregard the case used in the original symbol definition. The keyboard input are unfolded to uppercase whenever needed, allowing for a smoother typing style eliminating the shift keys. This feature is controlled by the option **completion-ignore-case** .

8.1.1.6 Apropos search

When calling completion with a prefix argument, an apropos search is made for all occurrences that match a regular expression. Default regexp is the pattern before point, so if you try to complete a pattern and fail, simply repeat the command with an argument to make an exhaustive apropos search. Case is ignored if **completion-ignore-case** is set.

8.1.1.7 Undoing function completions

To undo a completed function, use `M-DEL` (**il-backward-kill-word**) or `DEL`. These keys always removes the formal argument help along with the function name itself.

8.1.1.8 Append parentheses (C style)

An opening parenthesis is appended when completing functions. For functions with no arguments the closing parenthesis is appended as well, provided that the library contains the neccessary info.

8.2 Symbol Terminology

A *block* is a function/*prog*/*let* clause written at the outermost level and starting at the left margin.

8.2.1 Variable Types

A *local variable* is a variable declared in an enclosing *prog/let* clause or a formal argument of an enclosing function definition. The *prog/let* clauses may be nested and a local variable may be declared at any of these levels.

A *parent-local variable* is a variable (or formal argument) that is locally declared in a parent function and accessible from within a current child function, through the dynamic scoping scheme used in Skill.

A *block-global variable* is a variable that is set in the current *block* but not locally declared.

A *global variable* is a variable that is set in the current buffer but not locally declared. (It can thus be set at the "top level" or inside a *block*.)

A *global carrier* is a global variable that carries data between two or more local functions in the current buffer.

A *global dummy* is a global variable with seemingly no global purpose. The only reason it exists is probably because somebody forgot or didn't care to declare it locally. It remains as waste after execution and is typically a result of poor programming style.

A *basic system variable* is reserved for system functions in a naked Skill system. A normal application program is not supposed to modify it. Example: editor, _globalindent, poport.

An *API system variable* is a variable used by one of the API packages of DFII. A normal application program is not supposed to modify it. Example: dmbLibCreateForm, schPinMasters, schZoneFormatString.

8.2.2 Function Types

An *unknown function* is a function currently unknown to Skill-mode.

A *local function* is a function that is defined in the current buffer.

An *external function* is a function that is defined in a coexisting buffer of the current Emacs session, i.e that is a *local function* in another buffer. Thus, visiting an arbitrary skill-file makes its functions known and accessible in all existing skill-mode buffers.

A *basic function* is a basic Lisp or C primitive of general-purpose character. They form the skeleton and structure of the Skill language. Example: cons ,mapcar, foreach, sprintf, strcat.

An *API function* is part of one of the API packages of DFII, normally with a 2-3 character package prefix.
Example: dbChangeGroupType , hiCreateMenuItem.

A *prototype function* is a non-standard user-created function call, previously unknown to skill-mode, that is yet to be defined, that has been recorded as a prototype on the user's confirmation. The prototype is of permanent nature and is saved between sessions. It is used for top-down programming or to maintain a private library of miscellaneous funs.

A *temporary function* is a non-standard user-created function call, previously unknown to skill-mode, that has been recorded as a temporary, either silently or interactively as the result of the user's refusal to a prototype query. As the name suggests, the temporary is ephemeral and can only exist within the current editing session. It is sort of a "sloppy cousin" to the prototype and may also be used for top-down programming.

8.3 The Symbol Manager

Although Skill-mode is certainly not a debugger, it tries to some extent to emulate the dynamic variable environment created by the Skill interpreter at runtime. This applies in particular to the variable types *local*, *parent-local*, *block-global* and *global*, where execution can be regarded as temporarily "halted" in the function currently containing the cursor. Skill-mode includes a *symbol manager* that identifies and classifies any symbol that is declared or is set in the file. When *Declare* minor mode is active, this is done implicitly during the editing session for every new line that is written.

8.3.1 Handling Global Variables

One major problem in the design of Skill-mode was how to provide flexibility and reasonable performance for a wide range of program structures and sizes. For example, the search for global variables (and carriers in particular) is rather time consuming for large files, and unsolicited search would tend to annoy the user. The chosen approach is to search globals only on explicit request, by doing M-q (**il-complete-block-global-var**) or M-$ (**il-complete-global-var**).

8.3.1.1 Protection of System variables

If you assign a new value to a system variable (which are normally not to be modified by application programmers), terminal will beep and a warning message will be issued in the echo area.

8.3.1.2 Local-global Name conflict sentinel

If you try to complete a global variable that is shaded by a local variable with the same name, an error message is displayed in focus. If you complete a local var that also exists as global, a warning message will be issued in the echo area.

8.4 Complete-function Commands

il-complete-local-fun - "Local Fun" : M-1

Complete local function or show summary.
With argument, make an apropos search for all occurrences that match <regexp>. With argument 16 (C-u C-u), re-search buffer for local functions.

il-complete-basic-fun - "Basic Fun" : M-2

Complete basic function or show summary.
With argument, make an apropos search for all occurrences that match <regexp>.

il-complete-api-fun - "API Fun" : M-3

Complete API Function or show summary.
With argument, make an apropos search for all occurrences that match <regexp>. Note that the summary shows only the `miscellaneous API funs', ie the funs lacking package prefix. Do **il-list-summary-api-libs** for a summary of current API libraries, or **il-view-api-fun-library** to view the contents of a particular library.

il-complete-external-fun - "External Fun" : M-4

Complete external function or show summary.
With argument, make an apropos search for all occurrences that match <regexp>.

il-complete-prototype-fun - "Prototype Fun" : M-5

Complete prototype fun or show summary.
With argument, make an apropos search for all occurrences that match <regexp>. See **il-arghelp-mode** for more info.

il-complete-temp-fun - "Temporary Fun" : M-6

Complete temporary fun or show summary.
With argument, make an apropos search for all occurrences that match <regexp>. See **il-arghelp-mode** for more info.

8.5 Complete-variable Commands

il-complete-local-var - "Local Var" : M-TAB

Complete local variable or show summary.
Try to complete if there is a pattern before point, else show summary.
With argument, make an apropos search for all occurrences that match <regexp>.
Help window suffixes : '-->' means formal argument.

il-complete-block-global-var - "Block-global Var" : M-q

Complete block-global variable or show summary.
With argument, make an apropos search for all occurrences that match <regexp>.

il-complete-global-var - "Global Var" : M-$

Complete global variable or show summary.
When first called, the file is searched for global variables.
With argument, make an apropos search for all occurrences that match <regexp>. With argument 16 (C-u C-u), re-search file for global variables.
Help window suffixes :

'==>' means global carrier.
'!!' means System variable is changed.

Options: il-complete-carrier-search

il-complete-parent-local-var - "Parent-local Var"

Complete parent-local variable or show summary.
If multiple parents, the first one found by backward search from the child fun is selected. As default, limit upward search to **il-complete-parent-local-depth**. With argument, search to that stop level.

Options: il-complete-parent-local-depth

il-complete-basic-var - "Basic System Var" : M-@

Complete basic system var or show summary.
With argument, make an apropos search for all occurrences that match <regexp>.

il-complete-api-var - "API System Var" : M-#

Complete API system var or show summary.
With argument, make an apropos search for all occurrences that match <regexp>.

8.6 Complete Options - "Customize->Options->Complete"

il-complete-carrier-search : nil

Search for carriers when searching for global variables.
'carrier' means a global var that carries info between local funs. Slows down the search for global vars. Not recommended for large files.

il-complete-parent-local-depth : 1

Default upward max depth used by **il-complete-parent-local-var**.

8.7 Complete Admin-options

il-complete-verbose-help : t

Give verbose completion help in a separate help window.
When only initials are shown in the help info slot, the corresponding full words are displayed in the help window. Slows down performance somewhat.

8.8 Library Commands - "Complete->API Library Info"

The library commands give full listings and survey of all the current API function libraries that are linked to the **il-complete-api-fun** command, but not possible to browse in the normal way.

8.8.1 Commands

il-view-api-fun-library - "View API Library..."

View the contents of a specific API function library.
Use SPC for completion or to see all available libraries.

Admin-option: il-symbol-lib-version.

il-list-summary-api-libs - "List Summary API Libs"

List summary of the API libs in the current symbol-library.

Admin-option: il-symbol-lib-version.

8.8.2 Library Admin-options

il-symbol-lib-version : "4.3"

Defines which symbol library (ie Framework version) to use.
Corresponds to a directory name in the *<installdir>/skill-symbol-lib* dir. To add new official (or private) libraries, see documentation on the variable **il-api-libraries**.

9 The Symbol Library

The bulk of the symbol library consists of a prefabricated static read-only part.

9.1 Selecting Library version

Currently there are two official library versions : **4.2.1** and **4.3**. Updated Skill symbol libraries will hopefully emerge in the future. Preferrably, each major new DF-II release should include a comprehensive and well-matching Skill-mode library. Skill-mode is built to accomodate multiple library versions, and the version to use is defined by the admin-option **il-symbol-lib-version** .

9.2 Contents of the DFII-4.3 API function Library

Table 1 DFII-4.3 symbol library

API package name	prefix	size
Cadence waves	cw	58
Compactor	sy	89
Component description format	cd	34
Database access	db	137
Design editor	de	41
Design flow	df	55
Design management	dm	106
Display list	dl	51
Diva interface	iv	5
Enter shape	en	16
Floor planning	ad	24
FrameMaker interface	fm	21
FrameMaker interchange format	mi	40
Global routing	gr	11
Global & detail routing	pr	53
Graph browser	da	19
Graphics editor	ge	147
Human interface	hi	272
Layout editor	le	116

Table 1 DFII-4.3 symbol library

API package name	prefix	size
Layout synthesizer	la	8
Library development	am	9
Miscellaneous non-prefix	misc	48
Module maker	mm	99
Parameterized cell	pc	45
Plot fun	pl	21
Schematic human interface / Strucure compiler	sc	109
Simulation interface	si	8
Technology file	tc	62
Technology file	tf	39
Waveform display	wa	87
Template for building new packages	zz	2
Grand total API functions:		1849

9.3 Adding New API Packages

New API packages can be added as official or private libraries quite easily, since no reconfiguration of the skill-mode installation is needed. At start-up Skill-mode checks the actual contents of */skill-symbol-lib/<version>* and *~/.skill-mode* and recognizes all packages with the correct name conventions. For private non-API function additions, the prototype library should be used. See the section on "The Protoype Mechanism".

9.3.1 Check-list for adding a new API package

(This info can also be find in the document-string for the **il-api-libraries** variable.)

1 Goto directory <installdir>/*skill-symbol-lib*/<version> .

2 Copy the template *zz_funs.el* library to a new two-letter prefix name. (If you want a private library, move it to the *~/.skill-mode* dir.)

3 Change the variable name on line 1 & 2 to match the file name.

4 Change the symbol-type value on line 1, it must end with *fun* .

5 Add the functions. Each fun is a list of 2 or 3 strings:
string1 = fun name.
string2 = arglist. No arglist is written with an empty string.
string3 = document string. Optional.
Note that every fun name must begin with the two-letter prefix name.

6 Ensure that there are 2 closing parentheses at the end of file.

7 Save file and start a new Emacs.

8 Check installation of the new library with the **API Library Info->View API Library** command.

9.4 The Dynamic Libraries

The dynamic part of the symbol library exists only in memory and is maintained continuously by the *symbol manager* during the editing session. The dynamic library includes : Local variables, local functions, external functions, block-global variables, global variables, parent-local variables. Since the symbols are collected from the actual source, an exact correspondance to the running Skill interpreter environment are always ensured.

The dynamic libraries are local to the current Emacs buffer, except for the external functions. In other words, if you are visiting two Skill files simultaneously, they are always treated as separate independendent modules with respect to the variable scope.

10 Analysis Commands - the "Analys" main-Menu

Skill-mode contains a set of commands that helps you to analyze unfamiliar source code and to test arbitrary code chunks when writing new code. The analysis toolbox contains code evaluation, printing, hierarchy analysis, function documentation, timestamp survey, and syntax checking.

10.1 Evaluation Commands - "Analys->Eval"

The evaluation commands send arbitrary code chunks to Opus for instant evaluation. These commands shortens the turn-around times and improves the programming environment notably. The mechanism uses the *GateToOpus* link included in the Skill-mode distribution. The commands are greyed out if the link has not been installed properly. (The actual test made is the existence of an executable *~/.gateToOpus.wakeMeUp* file in your home directory.)

il-eval-last-sexp - "Eval Last Sexp" : C-x C-e

Sends the previous sexp to Opus.
Requires that gateToOpus.il is loaded by Opus at startup.

il-eval-fun - "Eval Fun" : C-M-x

Sends the current function to Opus.
Requires that gateToOpus.il is loaded by Opus at startup. The parenthesis balance is checked, and if mismatch no sending is done.

il-eval-region - "Eval Region" : C-c C-e

Sends the current region to Opus.
Requires that gateToOpus.il is loaded by Opus at startup.

il-eval-buffer - "Eval Buffer" : C-c C-x

Sends the current buffer to Opus.
Requires that gateToOpus.il is loaded by Opus at startup.

10.2 Print Commands - "Analys->Print"

The Print commands are used for printing out selected portions of the code instead of the entire file, in order to save paper and printer time.

il-lpr-fun - "Print Fun" : C-c l

Prints out the current function without comment-header.
Related option : lpr-switches.

lpr-region - "Print Region" : C-c L

Print region contents as with Unix command `lpr'.

`**lpr-switches**' is a list of extra switches (strings) to pass to lpr.

10.3 Hierarchy Commands - "Analys->Hierarchy"

The hierarchy commands are closely related to the hierarchy browser ; they perform fragmented pieces of the browser functionality in a less user-friendly and non-continuous way. In general, the browser is therefore a better alternative. The "trace variable" command is very useful for tracing variables of unknown origin.

il-hierarchy-list-down - "List Downward..." : C-c d

List downward hierarchy of current fun or adjacent local fun call.
If nothing found, prompts for a name. Default is local call found on current line, else current fun. Default depth is defined by il-hierarchy-down-depth but can be overridden by a prefix argument. Hierarchy suppressed by the max depth is denoted by trailing dots.

Repeated calls to the same function are ignored. Recursive loops in the calling chain are always detected and marked with the message `Recursive Call'.

Options: il-hierarchy-down-depth, il-tree_indent

il-hierarchy-list-up - "List Upward" : C-c u

List the hierarchy from current function up to the top level.
The top level is either a main function or the genuine top level, which is displayed as "--top-level--".

If there are multiple parents to any child in the chain, select the first one found by searching backwards from the child. Point must be in a function when calling this command. If you have made a descend path with M-n (il-move-descend), this path is used in the ascent.

Options: il-tree_indentil-tree_indent

il-hierarchy-list-unreferenced - "List Unreferenced Funs..."

List the local funs unreferenced from a given hierarchy.
Prompts for a root function, default is the first fun in the file. Can be used to pinpoint unused functions in the file.

il-hierarchy-trace-var - "Trace Variable" : C-c v

Trace variable at or before point by searching upward hierarchy.
Shows where variable is declared, read and set. The trace will only display such settings/readings that precede the current point in execution order. If pointing at a variable, use that name as argument. Otherwise prompts for a name, using the variable preceding point as default. Use SPC for completion.

10.3.1 Hierarchy Options - "Customize->Options->Hierarchy"

il-hierarchy-down-depth : 2

Default downward depth used by il-hierarchy-list-down.
Set to 1 or 2 if you want il-hierarchy-list-down to be quick.

10.4 Document Commands - "Analys->Document"

The Document commands supplies function descriptions and global data flow. For the best benefit of the **il-describe-fun** command, it's important that you make a habit of filling in the function headers properly whenever creating new functions.

il-describe-fun - "describe Fun..." : C-c f

Display the documentation of the function at or before point.
If pointing at a fun, use that name as argument. Otherwise prompts for a name. Use SPC for completion.

Admin-options:
il-fundoc-search-limit,il-fundoc-begin-regexp, il-fundoc-end-regexp

il-list-carriers - "List Carriers"

Display the data flow for each global carrier in this file.
`carrier' means a global var that carries info between local funs. The source function is printed to the left and the destinations to the right. Only ONE source fun can be detected - the one that comes first in the file.

Makes a simple linear search of the file (and disregards the actual execution view with dynamic scoping). As soon as a non-local var assignment has been found in a function, all the other functions are searched for references to that var. The first fun is then regarded as `source', and the subsequent ones as `destinations'.

10.4.1 Document Admin-options

il-fundoc-begin-regexp : "^;\\|/*"

Regexp for the start of doc-string of local functions.
The default pattern matches both line-oriented and block-oriented comments. Avoid fiddling with unless you are familiar with Emacs regexps.

il-fundoc-end-regexp : "^;.+ [^;]\\|*/"

Regexp for the end of doc-string of local functions.
The default pattern matches both line-oriented and block-oriented comments. Avoid fiddling with unless you are familiar with Emacs regexps.

il-fundoc-search-limit : 3

Number of lines from beginning-of-fun to the fun doc-string.
Defines the scope of the forward-search. Must be a positive number.

10.5 Timestamp Commands - "Analys->Timestamps"

Skill-mode automatically maintains two types of timestamps; one that is part of the standard file header, the other that is stamped on every edited function. The *file header timestamp* looks like this :

; Modified: Nov 7 16:28 95 TranslateSheet euajojm

It tells a) when the last change was made, b) where it was made, and c) who did it. This timestamp is somewhat similar to the $ Header $ substitution keyword used by RCS/CVS, but has the advantage of always reflecting the current state of the file, since it is updated whenever the buffer is saved. Moreover, it can be used for restarting from last edit when visiting the file next time.

The *function timestamp* looks like this :

procedure(TranslateSheet(keyClick) ;_Nov 7 95 euajojm 541

It tells a) when last change was made, b) who did it, and c) where it was made (as an offset count from the beginning of function). With this timestamp enabled, you can easily trace arbitrary last-edit-points in the local functions by using the hierarchical motion commands, or get a survey of what functions were edited at the previous occasion by doing the "List Recent" command.

il-timestamp-list-recent - "List Recent" : C-c t

List the most recently edited (and timestamped) functions.
These are the ones that were edited the same day as the last edit.
Requires the existence of a standard file header and enabled **il-timestamp-file-header** & **il-timestamp-funs** to be meaningful. The listed funs can be visited by clicking mouse-2 on them.

Options: il-timestamp-file-header,il-timestamp-funs

il-timestamp-list-all - "List All" : C-c T

List all timestamps of the local functions.
Requires that **il-timestamp-funs** is enabled to be meaningful. The listed funs can be visited by clicking mouse-2 on them.

Options: il-timestamp-funs

il-list-unsaved-funs - "List Unsaved Funs" : C-c C-t

List the functions that have been edited since last save.

10.5.1 Timestamp Options - "Customize->Options->Timestamp"

il-timestamp-file-header : t

Timestamp the standard file header when saving the file.
Requires the existence of a standard file header to be meaningful. See also il-create-file-header and il-timestamp-restart-from-last-edit.

il-timestamp-funs : t

Timestamp all edited functions when saving the file.
The timestamp contains date, userid and a last-edit pointer. The pointer is used by il-move-descend.

il-timestamp-restart-from-last-edit : t

Restart from last edit when visiting a file.
Requires that the file has previously been saved with il-timestamp-file-header enabled, to be meaningful.

10.5.2 Timestamp Admin-options

il-timestamp-fun-column : 50

Indent column for function timestamps.

10.6 Syntax check Commands - "Analys->Syntax Check"

The syntax check commands deal with parenthesis and double-quote checking. The parenthesis checking can be done either explicitly with the "Check Parentheses" command, or implicitly when the buffer is saved with the option **il-syntax-check-parentheses** .

10.6.1 Automatic Parenthesis Check

When the buffer is saved, every function that has been edited may optionally be checked regarding the parenthesis balance. If any mismatch is found, the terminal beeps, faulty function is pointed out, and you are asked to confirm the save operation. This feature is controlled by the option **il-syntax-check-parentheses**. The admin-option **il-fun-end-regexp** defines the syntax of the end-of-fun's searched for.

As a complement, Skill-mode also utilizes the standard parenthesis-matching feature that automatically shows how parentheses match in the text when you type closing parentheses. This feature is controlled by the standard option **blink-matching-paren** .

10.6.2 Finding Faulty Parentheses

A good first measure is to indent the function. The commands for moving over balanced sexps and lists are useful to check the scope of the expressions. A similar and more convenient command is C-c m (**il-match-parenthesis**), which is identical with the automatic parenthesis-matching feature mentioned earlier in this chapter.

10.6.3 Automatic Double-qoute Matching

Skill-mode contains a "blink-matching-double-qoute" feature, that is similar in operation to the standard "blink-matching-parenthesis" feature. It is controlled by the option **il-syntax-blink-matching-quote**.

10.6.4 Syntax Commands

il-check-parentheses - "Check Parentheses" : C-c p

Check the parenthesis balance in the current function.
`End-of-fun regexp mismatch' means that the pattern following the end-parenthesis doesn't look like an end-of-fun (though it might still be syntactically correct).

Admin-option: il-fun-end-regexp

il-match-parenthesis - "Matching Parenthesis" : C-c m

Blink matching opening parenthesis before point.

10.6.5 Syntax Options - "Customize->Options->Syntax Check"

il-syntax-check-parentheses : t

Check the parenthesis balance when saving the file.

il-syntax-blink-matching-quote : t

Blink matching quote when writing string constants.
Set to nil to turn off.

il-syntax-blink-time : 0.2

Blink time (in seconds) for matching parenthesis and double-quote.
Suitable time depends on your typing speed.

10.6.6 Syntax Admin-options

il-syntax-matching-quote-distance : 2

Search limit (num of lines) for matching opening double-quote.

11 Modification Commands - the "Modify" main-Menu

The "Modify" main-menu is the place holder for all commands that modify the source code in one way or another. The Modify toolbox contains indentation, mark & kill, comment, header, clean-up and debug commands.

11.1 Indentation Commands - "Modify->Indent"

The best way to keep a Skill program properly indented is to use Skill mode to re-indent it as you change it. Skill-mode has commands to indent a single line, a parenthetical grouping, the current function, the region or the entire buffer. (The indent mechanism also contains an automatic end-comment generator, see Automatic Generation of End-comments on page 37.)

There are two ways to call the basic indent primitive **il-indent-line** :

- Explicitly, with the TAB key.
- Implicitly, with the electric RETURN key, realized by *Indent* minor mode, see Automatic Indentation - "Indent" Minor Mode on page 51.

All the batch-indent commands use the same rather slow **il-indent-line** primitive, why these are often felt to be sluggish. Therefore, the smaller chunk you select for batch-indent, the better. C-M-q (**il-indent-sexp**) is perhaps the most versatile indent command for multiple lines. Do as follows: move back to beginning of the target expression, re-adjust indentation with TAB if needed, and hit C-M-q.

11.1.1 Indent Commands

il-indent-line - "Indent Line" : TAB

Indent current line as Skill code.
In indent mode, call indent-line after each RET to indent the empty new line. Use TAB to indent line explicitly. The amount of indenting is controlled by 3 options.

End-comments are generated if control expression spans more than il-comment-end-of-cexp-limit lines and closing parenthesis is the only entry on the line. End-comments can be rewritten or deleted at any time. See also **il-indent-fun**.

Comments are treated according to number of semicolons:

; --> indent to **comment-column**
;; --> indent as program code.
;;; --> don't touch.

Exception : Comments beginning at left margin are never touched.

Options: il-indent-body,il-indent-fun-body,il-indent-follow-first-arg-limit, il-comment-end-of-cexp-limit

il-indent-fun - "Indent Fun" : C-c C-i , C-c TAB

Indent the current function and check the parenthesis balance.
With argument, rewrite all end-comments. With argument 16, delete all end-comments.

il-indent-sexp - "Indent Sexp" : C-M-q

Indent each line of the sexp starting at or after point.
With argument, rewrite all end-comments. With argument 16, delete all end-comments.

il-indent-region - "Indent Region" : C-c i

Indent each line of the active region.
With argument, rewrite all end-comments. With argument 16, delete all end-comments.

il-indent-buffer - "Indent Buffer" : C-c I

Indent the entire current buffer.
With argument, rewrite all end-comments. With argument 16, delete all end-comments.

11.1.2 Indent Options - "Customize->Options->Indent"

il-indent-body : 3

Indent control expressions (if,cond,foreach,...) this value.

il-indent-fun-body : 2

Indent function bodies (defun/procedure,prog/let) this value.

il-indent-follow-first-arg-limit : 20

Follow first arg if length of fun name is less than value.
Otherwise indent according to il-indent-body. Following first arg is neat but less suitable for long function names in combination with large nesting depths. Set to a high number to activate unconditionally.

11.2 Edit Commands - "Modify->Mark & Kill"

With the advent of mouse-operated cut & paste in Emacs, the use of the mark-<entity> commands has declined significantly. Still, they are the fastest and safest means of selecting an arbitrary code-chunk for copy or kill, and might be worthwhile to learn if you want to become proficient with Emacs.

For example, **il-mark-sexp** selects the following sexp, accurately like a surgeon, without ever upsetting the parenthesis balance. There is simply no way that you could obtain the same accuracy with mouse-operated cut & paste.

il-mark-sexp - "Mark Sexp" : C-M-SPC

Set mark after the sexp following point.
With numeric argument, set mark n sexps after point.

il-mark-fun - "Mark Fun" : C-M-h , M-BackSpace

Put mark at end of function, point at beginning.

il-kill-sexp -"Kill Sexp" : C-M-k

Kill the sexp after point.
With numeric argument, kill n expressions after (or before) point.

il-transpose-sexps - "Transpose Sexps" : C-M-t

Swap the two sexps following point.

11.3 Comment Commands - "Modify->Comment"

Two types of comments are allowed in Skill ; block-oriented (C style) and line-oriented (Lisp style). Although the recognition of block-oriented comments has improved since V2.1, line-oriented comments still works best and is still the recommended choice. The comment commands described below are biased accordingly.

For efficient creation and maintenance of end-of-line comments, it's a good idea to learn the key binding `M-;` (**indent-for-comment**) by heart.

11.3.1 Indenting Comments

Comments are indented according to the number of semicolons:

; --> indented to **comment-column**
;; --> indented as program code.
;;; --> not touched.

Exception : Comments beginning at left margin are never touched.

11.3.2 Automatic Generation of End-comments

Comments at the end of control expressions, here referred to as *end-comments,* is a common technique to make the end of lengthy control expressions reasonably visible. Skill-mode includes an *automatic end-comment* mechanism that creates and maintains the end-comments for you.

An end-comment is generated if the control expression spans more than **il-comment-end-of-cexp-limit** lines and right parenthesis is the sole printable character on the current line, by pressing TAB (**il-indent-line**) which is the basic and more explicit method.

The implicit method is obtained by activating **il-indent-mode**, which binds **il-indent-line** to the electric right-parenthesis key. Further, by setting **il-indent_newline-at-end-of-cexp**, the right-parenthesis will always end up on a separate line, thus ensuring the end-comment creation.

The end-comments can be rewritten at any time by calling indent-fun,indent-sexp,indent-region or indent-buffer with a prefix argument, or be deleted by calling them with prefix argument 16 (i.e. C-u C-u). When updating the end-comments in this wholesale manner, only the items matching the admin-option **il-end-comment-start** will be rewritten, while all non-matching items are regarded as manually written and *write-protected end-comments*. In other words, you may occasionally write protected end-comments all from scratch, or by modifying generated items, and write-protect them by altering the start-string from the default " ; ** " to e.g. " ; *** ".

11.3.3 Commands

indent-for-comment - "Indent For Comment" : M-; (standard)

Indent this line's comment to **comment-column**, or insert an empty comment.

kill-comment - "Kill Comment" : C-c C-k (standard command)

Kill the comment on this line, if any.
With argument, kill comments on that many lines starting with this one.

il-comment-newline - "Comment Newline" : M-LFD

Continue comment on the next line if presently within one.
The body of the continued comment is indented under the previous comment line. If continuing a single semicolon comment starting at comment-column, no newline will be inserted.

il-comment-region - "Comment Region" : C-c ;

Comment (or uncomment) every line in the region.
Insert the value of comment-start at the beginning of every line in the region and check the parenthesis balance. With argument, uncomment the region.

11.3.4 Comment Options - "Customize->Options->Comment"

il-comment-end-of-cexp-limit : 5

Controls printing of end-comments for control expressions.
If control expression spans more than n lines and the closing parenthesis is written on a separate line, an end-comment is printed when indenting the line. Set to a high number to disable.

il-comment-fill-column : 60

Column beyond which automatic line-wrapping should happen.
Triggers auto-fill for every all-comment line in the file, ie. NOT for end-of-line comments. Set to a high number to disable.

11.3.5 Comment Admin-options

il-end-comment-start : " ; ** "

The start-signature of an automatically generated end-comment.
Check with your department if there exists a local preference for the end-comment syntax, the first time you install Skill-mode, and adjust once and for all if necessary. Avoid changing once people have started to use Skill-mode, since all end-comments created so far will then become write-protected.

il-end-comment-end : " **"

The end-signature of an automatically generated end-comment.
Not as critical as il-end-comment-start; may be changed after initial release to the users.

11.4 Header Commands - "Modify->Headers"

Headers fall in two categories : function level and file level. On the file level there are three independent sub-sections that together form the header; they are the *file header*, the *revision header* and the *symbol header*. The file header, which is strongly recommended for every file, must be created first, before the two optional sections. Due to the division in three sections and three separate commands, it is possible to update each part independently from the others.

The file-level commands are greyed out when point is not at the beginning of the file, whereas the function-level dittos are greyed out when there is no function at point.

The *revision header* can be maintained either manually, by explicitly doing the command when you have completed a new revision, or can be submitted to the version control system you are using. This setup is done by your administrator.

The *symbol header* gives a survey of the function hierarchy, very similar to the hierarchy browser, and a survey of the global variables used. It's an effective way of visualizing the program structure, but may grow to unwieldy proportions for large structures unless you limit the searched hierarchy depth. It also tends to require repeated re-creation as the code evolves.

Function headers are always prompted for unsolicitedly, as an integral part of the work-flow when defining new functions, where each sub-section are voluntary. In this way, Skill-mode strongly encourages "a header for every function" policy, which is a corner-stone for proper documentation of the source code. The function header is deliberately fairly compact, trying to avoid being perceived as too unwieldy for very small functions.

il-create-fun-header - "Create Fun Header..." : C-c h

Creates a standard function header.
With argument, recreate symbol header. The header consists of a general description, a description of the formal arguments and a note on the return value. Press RET to end user-input in each section.

il-create-file-header - "Create File Header..." : C-c H

Create a standard file header.
With argument, re-create file header. It's strongly recommended that a file header is included in every skill source file. Don't touch the first and last line of semicolons; they are used for identification.

Options: il-header-author-name,il-header-user-name

Admin-options: il-header-VCS,il-header-VCS-header-keyw, il-header-copyright-notice,il-header-user-company-name, il-header-search-limit,il-header-max-size

il-create-revision - "Create Revision Header..."

Make a revision entry, creating the revision header if needed.
As a prerequisite, buffer must have a standard file header. If **il-header-VCS** and **il-header-VCS-history-keyw** are set, the VCS will control the header once it has been created.

Admin-options: il-header-VCS,il-header-VCS-history-keyw

il-create-symbol-header - "Create Symbol Header"

Create a standard symbol header.
With argument, re-create symbol header. As a prerequisite, buffer must have a standard file header. The following sections are generated on request: Global Carriers, System Variables Used.
Press RET to end user input in each section.

For the function list you have two options :
a) A hierachical calling tree, starting from the main fun.
b) A linear list, sorted alphabetically or according to sequence in file.

Don't touch the first and last line of semicolons; they are used for identification.

Options: il-tree_indent

11.4.1 Header Options - "Customize->Options->File Header"

il-header-author-name : "user-full-name <userid@host>"

Author name for the file header. Must be a string.

il-header-user-name : "userid"

User name for the file header. Must be a string.

11.4.2 Header Admin-options

il-header-VCS : "CVS"

Name of the version control system you are using, if any.
Use uppercase letters when entering the name. non-nil yields a VCS-line containing the VCS header keyword. Set to nil if no version control system is used.

il-header-VCS-header-keyw : "$ Id $"

Header substitution keyword to put in the file header.
At check-in the VCS substitutes the keyword with header info. The default value is appropriate for CVS and RCS.

il-header-VCS-history-keyw : "$ Log $"

History substitution keyword to put in the revision header.
The default value is appropriate for CVS and RCS. The benefit of this keyword is somewhat debatable. For example, the change history may progressively grow to unwieldy proportions. Set to nil to prohibit VCS from controling the revision header.

il-header-copyright-notice : nil

Overrides the default copyright notice in the file header.
Default is: `(C) Copyright <year>, <company-name> , all rights reserved.' Use `\n ;' to insert newlines if you need a multi-line notice.

il-header-max-size : 30

Maximum size of a file header.

il-header-search-limit : 3

Number of lines to search for the file header.

il-header-user-company-name : "Your company"

User's company name to be included in the copyright string.
Default value is taken from environment variable ORGANIZATION, if it exists.

11.5 Cleaning-up Source-code - "Modify->Clean-up"

So far we only have got one clean-up measure : removing global dummy variables by declaring them locally wherever possible.

il-query-localize-block-vars - "Query Localize Vars in Block..."

Offer to declare global dummy variables as locals in current block. With argument, scan entire file for localization candidates. Variable name starting with the package prefix is assumed to be global and thus ignored.

For safety reasons you are asked to confirm every modification. Beware of local variables used in a global context. If for example a child fun modifies a parent-local variable, it will be a candidate for localization in the child fun.

11.6 Debugging Source-code - "Modify->Debug"

So far we only have got one debugging measure : insertion of *println* trace calls with an optional *lineread* for halting.

il-insert-trace-message - "Insert Trace Message"

Insert a *println* trace that prints out current sourcecode line.
With argument, append a *lineread()* for halting and waiting for a RET.

12 Managing Skill-mode itself - the "Mode" mainMenu

The "Mode" main-menu accomodates all the commands used for the operation and maintenance of the Skill-mode itself, and is by far the largest and deepest main-menu. It's basically a place holder for menus originally aspiring for the menu bar but eventually couldn't be lodged there. They are the "Help" menu, the extensive "Customize" menu, and the "Debug" menu.

13 Help on Skill-mode - the "Mode->Help" Menu

The menu is divided in four sections : General help, Command help, Option help and Admin-option help. The most generally useful "General help" command ought to be "View User Manual", which brings up this manual in a postscript viewer. The Help menu is fairly comprehensive which to some extent is a historic remainder of a time when this User's manual was non-existent.

13.1 General Help

il-view-mode-news - "View News"

Display info on recent changes to Skill-Mode.

il-view-user-manual - "View User Manual"

View the Skill-mode user manual.
Launches the postscript viewer asynchronously in an inferior shell.
Admin-Option: il-manual-viewer-postscript

il-show-mode-version - "Show Version"

Echo the current version of skill-mode in the minibuffer.

13.2 Command Help - "Commands"

il-describe-command - "Describe Command..."

Show the documentation for a Skill-mode option.
Use SPC for completion.

il-list-commands-brief - "List Commands Brief"

Display a brief summary of the Skill mode commands.

il-list-commands-verbose - "List Commands Verbose"

Display a verbose summary of all Skill mode commands.

13.3 Option Help - "Options"

il-view-options - "View Options"

View the options. For debugging.
With argument, select the admin-options.

il-describe-option - "Describe Option"

Show the documentation for an option.
With argument, select admin-option. Use SPC for completion.

il-list-options-brief - "List Options Brief"

Display a brief summary of the options.
With argument, select admin-options.

il-list-options-verbose - "List Options Verbose"

Display a verbose summary of the options.
With argument, select the admin-options.

13.4 Admin-option Help - "Admin-Options"

The admin-option help uses the same commands as in previous paragraph, but called with arguments.

13.5 Help Admin-options

il-manual-viewer-postscript : "pageview -w 8.27 -h 11.69 -dpi 85"

Postscript viewer command to view the user manual.
The default setting works for SUNOS and Solaris. `ghostview' from FSF is another possible alternative, but doesn't seem to give as good readability as pageview. See the man pages on `ghostview'.

14 Customization - the "Mode->Customize" Menu

You may customize Skill-mode according to your own preferences. For this purpose there is a set of built-in *minor modes*, a set of user-variables called *options*, and a set of administrator-variables called *admin-options* which, as the name suggests, are controlled by your local Skill-mode administrator.

All options have default values that should be appropriate for the novice user. In contrast, the minor modes are all initially turned off and must be activated either from the "Minor Modes" menu or through the customization walk-through that is promted for when you start a new Skill-mode version for the first time.

With its 7 minor modes, 45 options and 21 admin-options, Skill-mode is notably more customizable than most other major modes. To make this extensive flexibility manageable, the minor modes and the options are managed more or less automatically by Skill-mode itself and controlled primarily through the "Customize" menu. Customization is generally best done interactively with this menu, on a one by one basis, where you observe the changes of a single behaviour one at a time, and then decide whether to keep it or not.

All personal Skill-mode customization of permanent nature are stored in the *.skill-mode* directory in your home directory. Your options and minor mode defaults are stored in the file *defaults.el* (and should never need any manual editing under normal conditions). Your prototype library is stored in the file *prototype_funs.el* .

14.1 Customize Minor Modes - "Customize->Minor Modes"

The "Minor Modes" menu controls the 7 minor modes administered by Skill-mode. Most of them have associated options for further tailoring of their behavior, accessible through the "Options" sub-menu of each minor mode. These *minor mode options* are distinguished from their *regular option* counterparts by the underscore delimiter after the minor mode prefix-name in the symbol name, as in **il-font-lock_more-hilite** .

Apart from the standard *Font-lock* minor mode, all the minor modes found in the menu are effectively part of Skill-mode and can only exist in a Skill-mode buffer. In that respect they are thus not genuine minor modes in the strictest sense, but in most other ways their basic behaviour is identical to these.

14.1.1 Syntax Highlighting - "FontLock" Minor Mode

Syntax highlighting is done with the standard *Font-lock* minor mode in both Emacses. Its basic customization is controlled by Skill-mode in the same manner as for the built-in minor modes.

14.1.1.1 Font-lock-mode's Options - "Minor Mode->Options"

il-font-lock_more-hilite : nil

Increase the degree of high-lighting. Font-lock mode.
Slows down fontifying and buffer redisplay notably. We strongly recommend using a speedup-method together with this one. The behaviour is defined by the variables **il-font-lock-keywords-C-more** and **il-font-lock-keywords-lisp-more**, and as default the control expressions are hilited.

il-font-lock_use-colour : t

Highlight with colors instead of font variations. Font-lock mode.
Cannot be turned off interactively in GnuEmacs.

il-font-lock_speedup-method : turn-on-lazy-lock

Method of speeding up font-lock mode. Font-lock mode.
Lazy-lock fontifies only the visible window-region. Fast-lock saves away font-lock caches in separate cache files which are autoloaded the next time the source-file is visited.
Set to one of: nil, turn-on-lazy-lock, turn-on-fast-lock

14.1.2 Automatic Argument Help - "ArgHelp" Minor Mode

After completion of a function name, the corresponding formal arguments (often including datatype mnemonics) are displayed at point, and remain there until you close the argument list. When writing nested function calls, the outermost arguments are pushed on a stack, and retrieved as you close the innermost argument lists.

il-arghelp-mode

Minor mode for automatic argument help in function calls.
Without arg, toggle mode. With positive arg, turn on mode.

If the current fun call is known to skill-mode, the formal arguments are displayed at point or in the echo area (**in-focus**). If it is unknown, it is either recorded silently as a temporary fun, or interactively as a prototype/temporary (**prototype-make**). For very frequent funs, the arghelp can be muted (**muted-funs**). Funs with `key'-args may be processed interactively (**interactive-key-args**).

Arghelp-mode works as an implicit spelling checker; with correct spelling the argument list is displayed or, in case of no args, the closing parenthesis is inserted.

Contains a sentinel against unintentional re-use of existing local fun names. When enabled, '**Ar**' appears in the mode line.

Options:
il-arghelp_in-focus,il-arghelp_muted-funs
il-arghelp_interactive-key-args, il-arghelp_prototype-make

14.1.3 The Prototype Mechanism

The use of *prototype functions* are twofold :

1 Top-down programming, i.e. calling a fun name that is not yet defined. When you define it later on, it is removed from the prototype library.

2 Maintaining a private library of useful funs, as a complement to the supplied standard fun library. The proceeding is similar to case 1 except that the prototype is never defined as local fun, but remains as a prototype indefinitely.

If **il-arghelp_prototype-make** is on and you complete an unknown fun call of minimum 4 chars and an arglist of maximum 60 chars, you are promted to save name, arglist and docstring. (If you decline the query, it will be recorded as a *temporary* fun.) If the actual arglist contains plain args only, it's used as default for the formal arglist. It's good practice to enter a correct formal arglist complete with datatype mnemonics, as in : "t_string x_max"

The temporary fun is sort of a "sloppy cousin" to the prototype, as it doesn't ask for any user-input. It also lends itself for top-down programming, although not across sessions.

The *prototype library* is stored in a separate file, *~/.skill-mode/prototype_funs.el*, and is thus non-volatile data. In order to correct or enlarge the prototype library, it may be edited manually, although this implies a higher risk of corrupting the contents than when Skill-mode maintains it. If you edit the file manually, make sure there are no active Skill-mode buffers in the current emacs session when you save the buffer - or you run the risk of loosing your edits.

14.1.3.1 ArgHelp-mode's Options - "Minor Modes->Options"

il-arghelp_in-focus : t

Defines where to display the argument help. Arghelp-mode.
t displays arghelp in a slot below point, nil displays it in the echo area. t gives the most efficient arghelp while nil gives the least disturbance in the focus area.

il-arghelp_interactive-key-args : t

Defines how to process `key' arguments. Arghelp-mode.
If t, interactively prompts for key args in the minibuffer. If nil, no special treatment of key args. The start/end of the prompting session is signalled by a twin-beep. Press RET to skip an argument. Compulsory args are then inserted as empty entries, while optional args are discarded. Press SPC to get local var completion for symbol entries. NOTE ! Unless input is a local var, strings and symbols are automatically quoted by the input reader and expected list arguments are quoted with parentheses.

il-arghelp_prototype-make : t

Prompts for prototyping of unknown function calls. Arghelp-mode.
If t, interactively prompts for prototyping of unknown fun calls, including formal args and document string. If you decline they are recorded as temporary funs. If nil, they are silently recorded as temporary funs.

il-arghelp_muted-funs : (if car cdr set setq procedure defun prog)

Functions with muted argument help. Arghelp-mode.

14.1.4 Implicit Completion - "Complete" Minor Mode

Complete-mode is a more convenient way to call completion, where triggering is caused by the SPC bar when "applied to" certain pattern lengths. This implicit calling method is of course inherently more hazardous as accidental triggering may occur, but this can normally be held at a satisfactory low level by using a conservative setup of the completion behaviour.

The completion behaviour is defined by four options (see below) where the length of the pattern before point decides what kind of completion to be applied. The default setup, for example, is that "abc" is regarded as a local variable, "abcd" as a basic function, "abcde" as a local function, and "abcdef" as an API function.

il-complete-mode

Minor mode for length-sensitive completion by the SPC-bar. Without arg, toggle mode. With positive arg, turn on mode. In this mode, 4 completion commands are available on SPC-bar, triggered by pattern lengths as defined by 4 options.

Complete is inhibited if pattern is a local/global variable, or if point is in a comment or string constant. Use M-SPC to inhibit momentarily. Undo accidental triggering with DEL. If case unfolding is enabled, you might be more exposed to accidental triggering. If it troubles you, disable it with the standard option **completion-ignore-case** .

See also **il-complete-by-length**. When enabled, '**Co**' appears in the mode line.

Options: il-complete_trigger-local-var,il-complete_trigger-basic-fun il-complete_trigger-local-fun,il-complete_trigger-api-fun

14.1.4.1 Complete-mode's Options - "Minor Modes->Options"

il-complete_trigger-local-var : 3

Complete this pattern length to a local variable. Complete-mode. Set to zero to disable.

il-complete_trigger-basic-fun : 4

Complete this pattern length to a basic function. Complete-mode. Set to zero to disable.

il-complete_trigger-local-fun : 5

Complete this pattern length to a local function. Complete-mode. Set to zero to disable.

il-complete_trigger-api-fun : 6

Complete this pattern length to an API function. Complete-mode. Set to zero to disable.

14.1.5 Activate Tree Browser - "Tree" Minor Mode

il-tree-mode

Minor mode to link the tree browser to the hierarchical move commands. The browser is used as input/output device for hierarchical movement. The first move command in a skill buffer launches the browser and prompts for the main function, unless the file includes a standard file header. The tree is normally loaded from file but the initial scan may be quite slow for files of large depth. See skill-tree-mode for more info. When enabled, '**Tr**' appears in the mode line.

Options: il-tree_search-depth,il-tree_window-height, il-tree_indent, il-tree_rescan-trigger

14.1.5.1 Tree-mode's Options - "Minor Modes->Options"

il-tree_indent : 5

Indent function hierarchy trees to this value. Tree-mode.
Used by all commands that either displays or writes hierarchy trees.

il-tree_rescan-trigger : 5

Defers re-scan until N new functions has been added. Tree-mode.
The trigger condition is checked when the working file is loaded.
Newly created functions are accumulated under `APPENDED FUNCTIONS:'.

il-tree_search-depth : 4

Depth searched by the hierarchy browser. Tree-mode. The setting is a trade-off between survey, resolution and scan-times. Re-scan is triggered by:
a) adding il-tree_rescan-trigger number of new funs (at next visit)
b) changing this option (at next hierarchical move)

Be conservative about modifying permanently; it will make your accumulated tree archive outdated and force re-scan of each file at next visit. Functions not part of the main tree are listed at the end under `UNREFERENCED FUNCTIONS'. Truncated branches are marked by trailing dots and listed at the end under `UNREFERENCED & TRUNCATED FUNCTIONS'.

il-tree_window-height : 0.15

Relative height of browser window as a fraction of total screen height. For example, a value of 0.1 implies one tenth of the total screen height. Appropriate range is somewhere between 0.1 and 0.20. Note that window will be deleted if the computed height is less than window-min-height. Tree-mode.

14.1.6 Automatic Indentation - "Indent" Minor Mode

il-indent-mode

Minor mode for indent-line triggered by RETURN,')' and 'else'.
Without arg, toggle mode. With positive arg, turn on mode.
If **il-indent_newline-at-then&else** is non-nil, newline is inserted after `then' and `else'. If **il-indent_newline-at-end-of-cexp** is non-nil, newline(s) are inserted around the closing parenthesis of control expressions. When enabled, '**In**' appears in the mode line.

Options: il-indent_newline-at-then&else,
il-indent_newline-at-end-of-cexp

14.1.6.1 Indent-mode's Options - "Minor Modes->Options"

il-indent_newline-at-end-of-cexp : t

Auto-insert newline(s) around End-of-Control-expr. Indent-mode. Enforces uniform layout and high visibility of the End-of-Cexp. Ensures printing of end-comments.

il-indent_newline-at-then&else : t

Auto-insert newline after `then' and `else'. Indent-mode.

14.1.7 Automatic variable declaration - "Declare" Minor Mode

In Declare-mode the symbol manager spots every new symbol as you leave the line, and queries as soon as there is any ambiguity whether you want it to be local or global. If you answer "local", the local declaration is written in the source, which is referred to as *auto-declaration*. If you answer "global", the symbol is from now on regarded as global. This way, you never need to make any local declarations manually.

A prerequisite is the existence of an enclosing prog/let clause, so to be user-friendly Declare-mode queries if you want a prog/let clause whenever you create new functions. Let or prog clause is controlled by the option **il-declare_use-let-clause**.

il-declare-mode

Minor mode for declaring new local variables on the fly.
Without arg, toggle mode. With positive arg, turn on mode. A declare query is made whenever you leave a line inside a prog/let clause containing an assignment of a new variable made with '=' in C-style or 'setq' in Lisp-style. Declaration is made at the innermost prog/let clause.

Interactively inserts a prog/let clause when you define a new function. Variable name starting with the package prefix is assumed to be global and thus ignored. (To qualify as package prefix, at least 70% of the local funs must share the same prefix.)

When enabled, '**De**' appears in the mode line.

Options: il-declare_use-let-clause, il-declare_fill-column

14.1.8 Declare-mode's Options - "Minor Modes->Options"

il-declare_use-let-clause : t

Use let-clause to declare local variables. Declare-mode.
Set to nil to use prog-clause.

il-declare_fill-column : 75

Defines the max permitted column for the local variable list.
Set to 1 for one variable per line, if you want to document each variable separately.

14.1.9 Automatic Case Unfolding - "Unfold" Minor Mode

il-unfold-mode

Minor mode for automatic case-unfolding of functions and local vars. Without arg, toggle mode. With positive arg, turn on mode. Unfold is triggered by three keys: the opening parenthesis, the closing parenthesis and the SPC-bar. If the current pattern, disregarding the case, is either a known function or a local var, it is unfolded to the proper case.

Unfolding is indicated by a flashing of the symbol-name. It's inhibited if pattern contains any uppercase characters. Unfold mode works as an implicit spelling checker; with correct spelling, the symbol is unfolded to its proper form.

When enabled, '**Uf**' appears in the mode line.

14.2 Customizing Options - "Customize->Options"

All the regular options are controlled from the "Customize->Options" menu. Most of these options tailor the behaviour of explicit menu commands and share the topic prefix-name with their related commands. These have already been covered in previous sections of this manual. There are, however, two groups of options that fall outside this category, namely the "User Interface" and the "Automatic Spacing" group. They are described below.

14.2.1 User Interface - "Options->User Interface"

il-ui-experienced-user : t

Makes user-interface less verbose.
Suppresses the confirmation query when pushing menu toggle-buttons.

il-ui-popup-menu : il-menu-move (Only in XEmacs)

Defines a skill-mode specific popup menu on mouse button3.
Can be a user-defined menu or one of the skill-mode pulldowns:
il-menu-move,il-menu-complete,il-menu-analys,il-menu-modify,
il-menu-mode.
Set to nil to get the default menu 'Emacs Commands'.

14.2.1.1 User Interface Admin-options

il-ui-remove-menus : (("View") ("Find") ("Utilities") ("SPARCworks") ("SCCS"))

Defines menu items to be removed from the skill-mode menu-bar. Particularly intended for XEmacs >= 19.11 which in some cases is rather cluttered with standard menus out of the box, leaving no room for the skill-mode menus. For more info on syntax, see documentation on **delete-menu-item**.

14.2.2 Automatic Spacing

Spacing is needed very frequently in every source, either for syntax delimiting or for decent readability. In order to minimize the human keyboard input, Skill-mode provides an *automatic spacing* mechanism that maintains symmetry and a uniform layout according to your own personal taste. Auto-spacing is disabled when you are in comments or strings.

An optional space can be appended after successful completion. This is controlled by the user variables **il-space-append-var** for variables and **il-space-append-fun** for functions.

14.2.2.1 Make Right Hand Side follow Left Hand Side

If **il-space-rhs-follow-lhs** is set when typing closing parentheses, spacing is adjusted according to the spacing at the matching opening parenthesis. In other words, if the list starts with a space, it will end with a space and vice versa. This mechanism also works on assignment, relational and logical operators. Combined with **il-space-append-cexp**, it will give you a hint of where the control expressions ends, in case you like the Lisp style habit of piling many parentheses on a single line.

14.2.2.2 Spacing Options - "Options->Auto-space"

il-space-append-cexp : t

In C-style, append a space to every control expression.

il-space-append-fun : nil

In C-style, append a space to every function call.

il-space-append-var : t

Append a space to every successful variable completion.

il-space-before-operator : t

Insert a space before operator if non-existing. Applies to assignment,boolean and relational operators that are formed by the following keys: " = < > & | ! ^ ". Legal combinations of these keys are automatically merged when typing ahead. For example, typing two successive '=' yields "==".

il-space-rhs-follow-lhs : t

Make right hand side spacing of operator follow the left hand. Applies for the following keys: ") = < > & | ! ^ ". The closing parenthesis spacing follows the matching opening parenthesis's, thus deleting any surplus trailing spaces.

14.3 Manage Session Changes - "Custom->Session Changes"

The "Session Changes" menu manages the temporary customizations that you have done so far in the current Emacs session. It allows you to browse, undo, or save them to make them permanent.

il-session-show-changes - "Show"

Show what session changes have been done so far.

il-session-undo-changes - "Undo"

Undo the session changes that have been done so far.

il-session-save-changes - "Save"

Save the current setup of user options and minor modes.
Created eLisp code is saved in *~/.skill-mode/defaults.el* .This file will be loaded each time skill-mode is initially loaded.

14.4 Permanent Customization - "Customize->Permanently"

The "Permanently" menu is, not surprisingly, for permanent customization. The average user should under normal circumstances hardly ever need it. The **il-perm-customize-all** command is called implicitly when Skill-mode prompts you to customize it the first time you start a new Skill-mode version.

il-perm-customize-all - "Customize All..."

Customize skill-mode permanently through an exhaustive interactive query session. Created eLisp code is saved in *~/.skill-mode/defaults.el* which is loaded each time skill-mode is initially loaded.

il-perm-visit-init-file - "Visit Init-file"

Visit the skill-mode init-file for manual editing.

il-perm-reload-init-file - "Reload Init-file"

Reload the skill-mode init-file.
This is used for loading changes that you have manually edited in the init-file, to avoid the need for exit and restart. It resets the options unconditionally. This command is very different from il-session-undo-changes and is not intended for mainstream use. Always use il-session-undo-changes for undoing plain session changes.

15 Debugging Skill-mode - the "Mode->Debug" Menu

The "Debug" menu is for Skill-mode debugging. The most important command for an average user is perhaps the "Submit Bug Report" command. If you come across something you believe to be a bug, first check with "View Known Bugs", then compose a bug report according to the guidelines that follows below.

Let me remind you that a sloppily composed bug report is a waste of my time as well as yours, and although it's true that sloppy reports are deplorably predominant, it doesn't absolve you from the responsibility of writing proper ones.

15.1 How to Write a Proper Bug Report

A proper bug report includes a complete command sequence from Emacs startup upto the point where the bug occurs, and a backtrace printout of the bug event. Please also include the function (or any other relevant code chunk) containing the cursor where the problem occurs. I promise to treat such code confidentially.

For more detailed guidelines on how to assemble a proper bug report, please refer to the GnuEmacs Manual chapter 29.3.2 "How to Report a Bug", also available online through the Help->Info menu command, major topic: Emacs, Topic: Recovery from Problems, Node: Bugs.

15.2 How to Create a Backtrace printout

In order to create a backtrace you must first enable the debugger. Then you repeat the command sequence that evoked the error. There are three types of events that may invoke the debugger (as can be seen in the Mode->Debug menu):

- Bugs that terminate in an error, enabled by toggle button `Debug on Error'.
- Bugs that never terminates (e.g. closed loops), enabled by `Debug on Quit'. When you have reached the faulty spot, you terminate it with `C-g'.
- A certain function is called, enabled by `Debug on Entry'.

Invoking the debugger implies entering a recursive edit level, which is manifested by the encompassing square brackets on both mode lines. The backtrace info can now be copied to the mail buffer.

To exit the debugger, goto the backtrace buffer (if not already there) and type `q'. Now the backtrace buffer and the square brackets should vanish.

To finish the sequence, disable the debugger by reversing the menu command that was used to enable it. If you forget to do this, the debugger will inevitably be invoked by accident later on in the session, which certainly isn't harmful but often a bit annoying."

il-view-known-bugs - "View Known Bugs"

Display info on known bugs in Skill-Mode.

il-submit-bug-report - "Submit Bug Report"

Submit a bug report to the skill-mode maintainer.
Displays an instruction message and queries if you want to go on.
Queries if you want to append your current customization in the mail and leaves you in the mail buffer.

il-view-buffer-local-vars - "View Buffer-local Vars"

View the buffer-local skill-mode variables. For debugging.

il-view-global-mode-vars - "View Global Vars"

View the global skill-mode variables. For debugging.

16 Electric Commands

Emacs commands are normally bound to Control- or Meta-key sequences. However, some language modes also use plain printable-character keys to call various language-specific commands by stealth. Such a key is often regarded as "electric" and consequently, the command bound to it, as an *electric command*. Electric commands are used extensively in Skill-mode, primarily by the auto-space and auto-indent mechanisms.

il-electric-ampersand

Makes space adjusting.

il-electric-caret

Makes space adjusting.

il-electric-delete

Skill-mode wrapper to the standard backward-delete-char-untabify. Maintains the arg help stack.

il-electric-double-quote

Blink matching double quote.

il-electric-e

Auto-newline after `else' if indent-mode is on and il-indent_newline-at-then&else is on.

il-electric-equal

Makes space adjusting.

il-electric-exclamation

Makes space adjusting.

il-electric-greater

Makes space adjusting.

il-electric-left-parenthesis

Pushes the arglist stack and displays the top element.

il-electric-less

Makes space adjusting.

il-electric-n

Auto-newline after `then' if indent-mode is on and il-indent_newline-

at-then&else is on.

il-electric-return

Looks for new symbols and declares them.

il-electric-right-parenthesis

Pops the arglist stack and displays the top element.

il-electric-space

Calls the minor modes that are currently active.

il-electric-vertical

Makes space adjusting.

17 Wrappers to Standard Commands

The reason for this fairly extensive set of wrappers is to keep track of the location of the last edit, to trigger syntax checks and auto-indent when point leaves a modified line. (This wrapper methodology is a very old design choice from the good old Emacs-18 era, and may no longer be the best way to solve these tasks.)

il-backward-char

Skill-mode wrapper to the standard backward-char.

il-backward-kill-word

Skill-mode wrapper to the standard backward-kill-word.
With argument, do this n times. Maintains the arghelp stack.

il-beginning-of-buffer

Skill-mode wrapper to the standard beginning-of-buffer.

il-beginning-of-line

Skill-mode wrapper to the standard beginning-of-line.

il-end-of-buffer

Skill-mode wrapper to the standard end-of-buffer.

il-end-of-line

Skill-mode wrapper to the standard end-of-line.

il-forward-char

Skill-mode wrapper to the standard forward-char.

il-next-line

Skill-mode wrapper to the standard next-line.

il-previous-line

Skill-mode wrapper to the standard previous-line.

il-scroll-down

Skill-mode wrapper to the standard scroll-down.

il-scroll-up

Skill-mode wrapper to the standard scroll-up.

18 Miscellaneous Other Commands

These are commands that for some reason didn't rank (or were irrelevant) for the menu system.

il-describe-command-brief

Display a short description on a Skill mode command.
Use SPC for completion.

il-popup-menu (Only in XEmacs)

Popup a skill-mode-local menu on button3.

il-print-timestamp

Prints a timestamp with userid at comment-column. Can be used anywhere in the source to document changes. It looks like this "; Oct 6 90 etxjojm "

il-set-option

Set option to VALUE, which must be a Lisp object. Prints current value in the input field so that it can be edited. Use SPC for completion.

skill-mode

Major mode for editing Skill code. Prefix keys are C-c Meta and C-x. Tab indents for Skill code. Line-oriented comments (delimited by ';') is strongly recommended when using skill-mode, although block-oriented comments is partly supported.

Turning on Skill mode calls the value of the variable skill-mode-hook, if it's non-nil. Requires at least [LX]Emacs 19.10 or GnuEmacs 19.28.

19 Appendices

Keybinding Index

tab	il-indent-line
return	il-electric-return
space	il-electric-space
!	il-electric-exclamation
“	il-electric-double-quote
&	il-electric-ampersand
(	il-electric-left-parenthesis
)	il-electric-right-parenthesis
<	il-electric-less
=	il-electric-equal
>	il-electric-greater
^	il-electric-caret
e	il-electric-e
n	il-electric-n
|	il-electric-vertical
delete	il-electric-delete
button3	il-popup-menu
down	il-next-line
f8	function-menu
left	il-backward-char
right	il-forward-char
up	il-previous-line
C-a	il-beginning-of-line
C-b	il-backward-char
C-e	il-end-of-line
C-f	il-forward-char
C-i	il-indent-line
C-m	il-electric-return
C-n	il-next-line
C-p	il-previous-line
C-v	il-scroll-up

M-backspace	il-mark-fun
M-tab	il-complete-local-var
M-linefeed	il-comment-newline
M-return	il-comment-newline
M-#	il-complete-api-var
M-$	il-complete-global-var
M-1	il-complete-local-fun
M-2	il-complete-basic-fun
M-3	il-complete-api-fun
M-4	il-complete-external-fun
M-5	il-complete-prototype-fun
M-6	il-complete-temp-fun
M-<	il-beginning-of-buffer
M->	il-end-of-buffer
M-@	il-complete-basic-var
M-n	il-move-descend
M-p	il-move-ascend
M-q	il-complete-block-global-var
M-v	il-scroll-down
M-delete	il-backward-kill-word
M-C-space	il-mark-sexp
M-C-@	il-mark-sexp
M-C-a	il-move-begin-fun
M-C-b	il-backward-sexp
M-C-e	il-move-end-fun
M-C-f	il-forward-sexp
M-C-h	il-mark-fun
M-C-i	il-complete-local-var
M-C-j	il-comment-newline
M-C-k	il-kill-sexp
M-C-m	il-comment-newline
M-C-p	il-backward-list
M-C-q	il-indent-sexp
M-C-t	il-transpose-sexps
M-C-u	il-backward-up-list
M-C-x	il-eval-fun

C-c tab	il-indent-fun
C-c ;	il-comment-region
C-c H	il-create-file-header
C-c I	il-indent-buffer
C-c L	lpr-region
C-c T	il-timestamp-list-all
C-c d	il-hierarchy-list-down
C-c f	il-describe-fun
C-c h	il-create-fun-header
C-c i	il-indent-region
C-c l	il-lpr-fun
C-c m	il-match-parenthesis
C-c p	il-check-parentheses
C-c t	il-timestamp-list-recent
C-c u	il-hierarchy-list-up
C-c v	il-hierarchy-trace-var
C-c C-b	il-move-backward-cexp
C-c C-d	il-move-declaration
C-c C-e	il-eval-region
C-c C-f	il-move-forward-cexp
C-c C-i	il-indent-fun
C-c C-k	kill-comment
C-c C-l	il-move-last-edit
C-c C-n	il-move-next-child
C-c C-p	il-move-prev-child
C-c C-t	il-list-unsaved-funs
C-c C-x	il-eval-buffer
C-x C-e	il-eval-last-sexp

Command, Variable, Terminology-Index