Top Level, Inc.

When you're ready for the future™

Top Level Common Lisp Reference Manual

Release 1.0.1

Top Level, Inc.
196 North Pleasant St.
Amherst, Massachusetts 01002
(413) 256-6405

March 1990

Top Level, Inc.

When you're ready for the future™

Top Level Common Lisp Reference Manual

Release 1.0.1

Top Level, Inc.
196 North Pleasant St.
Amherst, Massachusetts 01002
(413) 256-6405

March 1990

Contents

Chapter 1

Introduction

Top Level Common Lisp is a high-performance implementation of Common Lisp. It is currently available for the Sequent Symmetry and Encore Multimax shared-memory multiprocessors.

Much of the basic Lisp-level system is based on the public domain Carnegie-Mellon University Common Lisp system. To achieve high performance, the compiler and lower-level support system were redesigned to fully utilize the architecture of the Intel 80386 processors used in the Symmetry and the National Semiconductor 32000 series processors used in the Multimax.

Top Level Common Lisp is a full implementation of the Common Lisp standard, with the exception of support for complex numbers. This manual describes only those aspects of the system that are specific to this implementation, and generally documents only those extensions and facilities that are not already defined by Common Lisp, as specified in Common LISP: The Language by Guy L. Steele, Jr., published by Digital Press.

Top Level Common Lisp extends the Common Lisp language to include programmer-specified parallel programming constructs for shared-memory multiprocessors. It uses the future construct as the underlying synchronization mechanism and provides three grain-sizes of parallel operators. Chapter 13, "Parallel Programming," presents the parallel programming extensions.

Any function or variable documented in this manual is generally in the system package, and is shown in **this typeface**. Common Lisp functions are shown in `this typeface`.

Chapter 2

Lisp Listener

The Lisp Listener provides a convenient and productive interface to the Lisp top-level loop, or can be used whenever the system requires input. It allows you to edit your input, recall previous input, and execute other useful functions through keystrokes. Editing commands are similiar to those in Emacs. Functions and variables are provided that allow you to customize the interaction and command bindings.

The listener will begin evaluating a form as soon as it has been completed, without requiring a `Return`. When it begins to evaluate a form, the listener prints out a message. The result of the evaluation is then printed, and top-level variables are bound as defined in CLtL. If multiple values are returned, each one is printed. The listener then reprompts for another form.

use-listener-p [*variable*]

If `non-nil`, the listener is used when typed user input is required. It defaults to `t`. Using the `-nolistener` boot option will bring up the system without the listener, and set this variable to `nil`.

Keystroke Notation

Keystrokes that are presented here as `META` keys are two-keystroke sequences in which the first keystroke actually is an `Escape`, i.e., the character `META-a` actually represents the sequence `Escape` followed by `a`. In addition, the case of the character following the `META` is significant. Thus `META-a` is a different sequence than `META-A`. The TI Explorer keyboard will automatically transmit `META` keystrokes as two-character sequences, but keyboards vary greatly, and others may require using the two-keystroke sequence.

2.1 Getting Listener Information

`META-?` Displays a short help notice.

ll-show-bindings `&Optional` *full-info self-inserts? table* [*function*]

Displays all the current key bindings. If *full-info* is `non-nil`, the default, it also displays the documentation string for the function the keystroke is bound to. If *self-inserts?* is `non-nil`, it shows those keys bound to **ll-self-insert**. If *table* is supplied, it is used instead of the current one.

`CTRL-x,/`	Prompts for a keystroke and displays what function it will invoke.

2.2 Editing Input

You can edit the form you are currently typing. Editing commands and command bindings are very similiar to Emacs.

2.2.1 Character Commands

`CTRL-d`	Deletes the character the cursor is on.
`CTRL-h`	Deletes the character preceding the cursor.
`Rubout`	Deletes the character preceding the cursor.
`CTRL-b`	Moves the cursor backward one character.
`CTRL-f`	Moves the cursor forward one character.
`CTRL-t`	Transposes the character the cursor is on with the preceding one.

2.2.2 Word Commands

`META-b`	Moves the cursor back one word.
`META-f`	Moves the cursor forward one word.
`META-h`	Kills (deletes) the word preceding the cursor.
`META-d`	Kills (deletes) the word following the cursor.
`META-t`	Transposes the two words preceding the cursor.

2.2.3 Line Commands

`CTRL-a`	Moves the cursor to the beginning of the line it is on.
`CTRL-e`	Moves the cursor to the end of the line it is on.
`CTRL-n`	Moves the cursor down one line.
`CTRL-p`	Moves the cursor up one line.
`CTRL-k`	Kills (deletes) text from the cursor to the end of the line it is on.

2.2.4 Lisp Code Commands

`META-F`	Moves forward one S-expression.
`META-B`	Moves backward one S-expression.
`META-D`	Kills forward one S-expression.
`META-H`	Kills backward one S-expression.
`META-A`	Moves to the beginning of the current List.
`META-E`	Moves to the end of the current List.
`CTRL-x,CTRL-a`	Displays the current function's arguments.
`CTRL-x,CTRL-d`	Displays the current function's documentation.
`META-x`	Enters a loop, prompting for forms to macroexpand. Entering a symbol will return to the normal listener loop.
`META-a`	Prompts for arguments to `Apropos`.

2.2.5 Kill Buffer Commands

Whenever something is deleted, it is generally pushed into the kill history. Single-character deletes are not, but word deletions are. All listeners share the same kill history.

`CTRL-space`	Sets the mark at the current cursor position.
`CTRL-w`	Kills (deletes) the text from the mark to the current position.
`META-y`	Rotates the kill history forward, deletes the previously yanked text, and replaces it with the new top element of the kill history.
`META-Y`	Rotates the kill history backward, deletes the previously yanked text, and replaces it with the new top element of the kill history.

2.3 Input Buffer and History

These are the commands that deal with the input buffer. Each listener maintains its own input history.

2.3.1 Input Buffer Commands

`CTRL-u`	Clears out the input buffer, reprompting for new input.
`CTRL-l`	Redisplays the input buffer.
`META-l`	Clears the screen and redisplays the input buffer.
`META-<`	Moves to the beginning of the input.

`META->`	Moves to the end of the input.
`CTRL-x,q`	This will insert the next character typed into the input buffer without executing its command binding, if any.

The listener keeps a history of input strings, with a maximum of 100 entries. Before evaluating an input form, it is saved in the history. It doesn't save the form in the following two cases: 1) The input string is identical to the previous input string, or 2) its length is shorter than the value of ***ll-ignore-string-length***.

You can recall the previous input using `CTRL-i`. After you have recalled a previous input, you can continue to move through the history using `META-i` to go backward, and `META-I` to go forward. When you move in the history, the previously recalled input is replaced by the next one.

Another way to view the history is to use `CTRL-x,CTRL-h`. This will display the last 10 input strings. To recall one of these input strings, use `CTRL-x,h` and then enter the digit for the input string. This will recall the input string and place you there in the history. Repeating `CTRL-x,CTRL-h` will move back through the input history, displaying 10 items each time.

2.3.2 Input History Commands

`CTRL-i`	Recalls the previous input string.
`META-i`	Moves back in the history, replacing the recalled input.
`META-I`	Moves forward in the history, replacing the recalled input.
`CTRL-x,CTRL-h`	Displays previous 10 inputs.
`CTRL-x,h,`*digit*	Recalls selected *digit*'s history element.

reset-ll-history *[function]*

This function will clear out the input history.

2.4 Other Useful Commands

`META-e`	Calls the Editor (the function `ed`).
`META-m`	Calls the Unix shell. This will work only from a Lisp process connected to a login terminal. It will hang if the process is attached to a network or Xterm window.
`META-G`	Exits the Lisp listener, returning to the basic Lisp reader.

2.5 User Customization

The listener has a number of global variables that define its behavior. You can alter their values to customize the listener to your own liking. If you want to create your own read-eval-print loop, see the function **ll-reader-loop** below.

ll-set-prompt *prompt-string* [*function*]

Sets the prompt to be *prompt-string.* If *prompt-string* is a function, it will call this function to return a string each time it prompts. This string shouldn't have any newlines in it, or the listener will not work properly.

package-prompt [*function*]

Changes the prompt to be the current package name followed by a colon.

ll-print-function [*variable*]

Bound to a function used to print the evaluation results. Its default value is `#'pprint`.

ll-eval-notice [*variable*]

Bound to a string that gets printed when the form is being evaluated. Its default value is " ...".

ll-eval-function [*variable*]

Called on a completed form. Its default value is `#'eval`.

lisp-listener [*function*]

Enters into the top-level listener read-eval-print loop. This should *not* be called recursively, since the `*terminal-io*` stream will get two levels of editing, which doesn't work.

ll-read `&Optional` *read-line prompt* [*function*]

The functions `read` and `readline` will automatically use the listener for terminal input when ***use-listener-p*** is `non-nil`, but you can use this function if the listener is not being used in general. It reads a form from the terminal using the listener. All the key bindings are in effect, and the history is available and is updated. If *read-line* is `non-nil`, it will operate the same as the Common Lisp function **read-line**, returning a string. The prompt should be a string. If not given, no prompt is used. Note that a newline is NOT issued before the read.

ll-reader-loop `&Key` *read-only read-line prompt eval-notice print-function eval-function empty table* [*function*]

The general reader loop. The function **lisp-listener** calls this function. Note that it is different from the **lisp-listener** function in that it is exited when an error occurs, or when `META-G` is entered. The parameters are as follows:

`:read-only`	If `non-nil`, then the function returns exactly one form, and the eval-function is not called. A newline is not issued before reading.
`:read-line`	If `non-nil`, then a form consists of a string of one line.
`:prompt`	The prompt used. It defaults to the current one.
`:eval-notice`	Binds ***ll-eval-notice*** to the value during the loop.
`:print-function`	Binds ***ll-print-function*** to the value during the loop.
`:eval-function`	Binds ***ll-eval-function*** to the value during the loop. If you want to returna a value from the reader-loop you can do a throw to `'ll-read`.
`:empty`	When this parameter is supplied (and is `non-nil`), if the buffer is ever cleared **ll-reader-loop** returns it as the value of the call.
`:table`	This parameter allows you to supply a different binding table. Don't use this unless you know what you are doing. It should be a 256 element array. Use the function **ll-bind-key** to fill it up.

The **ll-bind-key** function is provided to bind keystroke sequences to functions. You can use this function to re-bind commands to different keystrokes. First find out what function the command is currently bound to using the `CTRL-x,/` command. Then use **ll-bind-key** to bind the keystroke to the function symbol. In general, you can use **ll-bind-key** to bind a keystroke to any arbitrary function.

ll-bind-key *function keypath* `&Optional` *table* [*function*]

Binds *keypath* to the function. When *keypath* is typed, *function* is called with the edit-stream as an argument. *Keypath* can be either a character or a list of characters, indicating a multiple-keystroke command. Note that **ll-bind-key** does not verify that *function* is a legal function. The *table* argument defaults to the current table. It should be a 256-element array.

ll-terminal-mode *switch* [*function*]

Toggles the terminal state for the listener. Turning it on with a *switch* argument of `:on`, `:off` will set the terminal to a normal state. This should be used if a key's function requires the terminal state be restored to its "normal" state.

ll-redisplay [*function*]

Redisplays the input buffer. It is bound to `CTRL-l`, but you may want to call it after you execute a command that may mess up the screen.

Chapter 3

The Debugger

When an error occurs, the system will invoke the Debugger. The debugger closely resembles the top-level loop of Lisp, except certain keystrokes are interpreted as debugger commands. Unlike the top-level loop, the debugger can be entered recursively. When in the debugger, the prompt is changed to `->`. The number of dashes indicates how many levels deep in the debugger you are.

The debugger interface takes advantage of the capabilities of the listener. It also allows you to recall and edit previous input. Debugger commands are invoked using single keystrokes.

Primitive Debugger Interface

> If for some reason the Lisp Listener is not being used, the debugger interface is different. It does not use keystroke commands, but will recognize certain symbols as debugger commands. The keystroke commands described below also indicate their corresponding symbol commands. The symbols are not required to be in the keyword package. If you are using the primitive debugger interface, type `:help` for a list of commands.

Whenever you have a debugger prompt, the debugger will look at the next character typed. What happens next depends on the character.

- If the character is bound to a debugger command the command will be executed.
- Otherwise, the debugger will assume you are starting to type a form to be evaluated and prompts for same using the listener.

If you clear out the input buffer while being prompted for an expression to evaluate or for a command line, you will be reprompted with the debugger prompt and the debugger will again look at the next character typed as described above. When the debugger prompts for arguments to a command, it will use the input editor.

The default debugger command bindings are as follows:

`META-A`	Exits all levels of the debugger and restarts the top-level loop. It actually does a `throw` to the tag `'%top-level`. Symbol command `:abort`.

`CTRL-g`	Exits the current level of the debugger and returns to the previous error. If only one level deep, it behaves exactly like `:abort`. Symbol command `:quit`.
`META-a`	This is the same as `CTRL-g`.
`CTRL-b`	Prints a backtrace of stack frames from the current frame down to the bottom frame. Symbol command `:backtrace`.
`CTRL-d`	Disassembles the function for the current call frame. It shows only the code around the return PC. Symbol command `:disassemble`.
`META-d`	Disassembles the function for the current call frame, prompting for the number of bytes before and after the PC. Instructions are typically 2 or 3 bytes long. This works only if the PC can be determined, which is sometimes not possible. Symbol command `:disassemble-more`.
`CTRL-e`	Attempts to edit the source file in which the current function is defined and move you to that location in the file. Symbol command `:edit-definition`.
`CTRL-h`	Prints out a short list of debugger commands. Symbol command `:help`.
`CTRL-l`	Clears the screen and prints the current call frame, displaying the function and its arguments. Symbol command `:clear-print`.
`META-l`	Clears the screen and prints the current call frame, displaying the function and its arguments, the frame's locals, and also disassembles the function around the PC. Symbol command `:where`.
`META-CTRL-l`	Clears the screen and prints the current call frame, displaying the function and its arguments and the frame's locals. Symbol command `:locals`.
`CTRL-n`	Moves down a call frame and displays it. Symbol command `:next`.
`CTRL-p`	Moves up a call frame and displays it. Symbol command `:previous`.
`CTRL-r`	Returns from the current call frame. If the caller does not accept multiple values, then it will prompt for a single form to evaluate. The value of the form is returned as the value of the function call. If the caller will accept multiple values, it will prompt for the number of values to return. It will then prompt for that many forms to evaluate as the return values. It will execute any `unwind-protects` from above before returning. Symbol command `:return`.
`META-r`	Continues from a continuable error or returns from a call to **debug** or `break`. Symbol command `:continue`.
`META-CTRL-r`	Restarts the computation from the current call frame, reinvoking the call with the current arguments. It will reinvoke the call supplying all optional or keyboard arguments. See the function **arg** to see how to modify the arguments before reinvoking a call. Symbol command `:reinvoke`.

`META-<`	Moves to the top call frame of the call stack, which is usually the call to **debug**. Symbol command `:top`.
`META->`	Moves to the bottom of the call stack, which is usually the function **%initial-loop**. Symbol command `:bottom`.

You can change the debugger command bindings using the function **bind-debug-command** as described below.

bind-debug-command *character value* `&Optional` *documentation* [*function*]

Binds *character* to a debugger command. *Value* can be either a debugger command, such as `:backtrace`, or a function of no arguments that will return a debugger command. If documentation is supplied it is used when displaying the debugger command bindings.

get-debug-command *character* [*function*]

Returns the debugger binding for *character*.

default-debug-bindings [*function*]

Binds the debugger commands to their defaults as presented above.

The debugger will evaluate forms using **%eval**. This will evaluate the form in the interpreter's lexical environment that was in effect when **debug** was called. Evaluation of dynamic or special variable bindings is also in the environment in which **debug** was called. Thus, the debugger is not sensitive to the context of the current call frame, either for special bindings or for lexical bindings. Lexical bindings in compiled code are represented as locals in a call frame. They can be accessed using the function **arg**. To evaluate a variable without using the interpreters current lexical environment, use the function `eval`, e.g., `(eval 'a)`, will return the non-lexical binding of a.

There is currently no easy way to access a lexical binding that has been closed over in the debugger. A local with the name `closure-vector-home` will contain the current closure vector used to hold closure variables. The contents of the vector can be looked at, but the names of the variables kept in each vector slot are not maintained by the system.

When returning or reinvoking a call frame, the stack is unwound down to the current frame, evaluating any protected forms from any `Unwind-Protects` and unbinding any special variables bound above the call frame to reinvoke or return from. NOTE: If the global state is modified by a function and that modification was meant to be undone and is *not protected* by an `Unwind-Protect` the lisp system can be corrupted. Be careful when returning from or reinvoking call frames.

For some errors, it is possible to return values from the call to **%error**, allowing the function calling **%error** to proceed. This can be used safely for unbound variables and undefined functions, allowing you to return a value as the value of the variable and the function's definition, respectively. Other cases are possible. A thorough understanding of disassembled code is required to know when it is safe to do so for other kinds of errors.

arg *n* [*function*]

Returns the *n*th argument or local (shown as `Arg` *n* or `Local` *n*) from the current call frame. If *n* is `:rest`, then the `&Rest` argument is returned. If *n* is `:fun`, then the frame's function is returned.

This form has a `setf` method, so call frames can be reinvoked with different arguments. However, it is currently not possible to change the function in a call frame.

debug-print-array [*variable*]
debug-print-structure [*variable*]
debug-print-level [*variable*]
debug-print-depth [*variable*]

The above four variables are bound when printing the arguments and locals for a call frame. These variables do not affect the printing of values returned by evaluating a form in the debugger.

debug [*function*]

Invokes the debugger. This can be called anytime to invoke the debugger to inspect the current call stack.

debug-hidden [*variable*]

Bound to a function that is called on the function name for a call frame. The call frame is essentially hidden from the user's view if this function returns `nil`. It defaults to `#'identity`. This feature can be used to hide calls to interpreter functions that can clutter up the call stack.

debug-flush-errors [*variable*]

Limits how many levels of recursion the debugger can be invoked. If a number, it is the number of calls allowed before a call to **debug** will just return `nil`. If `non-nil` and not a number, then the debugger cannot be invoked and all errors will restart at the top level.

backtrace-on-error [*variable*]

If `nil`, no backtrace is performed when an error occurs. If a number, it is the number of calls that are displayed in the backtrace. If bound to anything else, a full backtrace will be performed.

debug-disassemble-distance [*variable*]

Specifies how many bytes of code are to surround the disassembled code from the return address. Instructions are typically 2 to 4 bytes long. The default value is 20.

backtrace `&Optional` *n* [*function*]

Prints a backtrace of the current call stack, down to *n* calls or the bottom. This can be called outside of the debugger context.

backtrace-call *function* `&Optional` *n* [*function*]

Calls *function* on each call frame's function. It will move down *n* calls or to the stack bottom. It will not call the ***debug-hidden*** function.

inside-debugger-p [*variable*]

Bound to `t` when inside the debugger.

debug-command-level [*variable*]

Bound to the recursion level of debug calls.

debug-prompt [*variable*]

Bound to a function that is called to print the prompt in the debugger. The function is called with no arguments.

3.1 Error Handling

All errors detected will invoke the debugger by default. The three forms defined below allow a user program to deal with errors and provide a simple, effective method for doing so.

handle-errors *error-value* `&Body` *body* [*macro*]

First evaluates *error-value.* If any of the forms in *body* signal an error, the form immediately returns two values, *error-value* and the error string.

call-catch-errors *error-value function* `&Rest` *args* [*function*]

Essentially the same as **handle-errors.** If an error occurs during the application of *function* to *args*, the form immediately returns two values, *error-value* and the error string.

with-error-handler *(*`lambda` *handler-args* `&Body` *handler-body)* `&Body` *body* [*macro*]

If no error is signaled during the evaluation of *body*, *body*'s values are returned. If an error is signaled, the lambda function is invoked. If the lambda function returns, then the error is handled normally, invoking the debugger. Forms in the body of the lambda can call **error-handler-return** to immediately return values from the **with-error-handler** form. A call to **error-handler-proceed** will attempt to return values from the call to **%error.** ONLY use **error-handler-proceed** for errors with a known way to proceed.

The system will abort to top level if any errors occur while evaluating the handler function. The symbols **error-handler-return** and **error-handler-proceed** are in the SYSTEM package, so be sure to import them or supply the system package prefix to avoid errors in the lambda.

3.1.1 With-Error-Handler Example

The following example using **with-error-handler** will return `''local''` for the value of `*system-host*` if an unbound reference is made during the evaluation of `(load-files)`. For any other unbound variable error, the **with-error-handler** form will exit with a value of `nil`. If the error is not an unbound variable error, the handler returns `nil`, which will invoke the debugger normally.

```
(with-error-handler
   (lambda (format &rest args)
     (when (eq format *unbound-error*)
       (if (eq (first args) '*system-host*)
           (ERROR-HANDLER-PROCEED "local")
           (ERROR-HANDLER-RETURN nil))))
   (load-files))
```

3.2 Debugging Tasks Without Windows

When using the background-window facilities, tasks that invoke the debugger will get their own input-output window. The debugger works just the same using these background windows.

If you are not using background windows, a difficult problem occurs because one terminal must be shared among all tasks. Only one task at a time can be reading from the terminal. If a background task tries to read from the terminal while another task "owns" the terminal, it will hang forever. Therefore, if you aren't using background windows, you should never write code that reads from `*standard-input*`. However, more than one task can *write* to the terminal. The output will likely be interleaved, and possibly not be readable, but at least it won't hang.

Even if you write code that does not read from `*standard-input*`, any errors that occur in the task will invoke the debugger, which requires input from the terminal. The debugger is set up so that you can "timeshare" among tasks that have invoked the debugger. When a background task calls **debug**, it will store itself on a list of tasks waiting to be debugged, and then suspend itself. The function **task-debug** is used to switch terminal control over to the debugger for these suspended tasks.

debugging-tasks [*variable*]

Supplies a list of all tasks that are waiting to run the debugger. If you modify this variable, make sure it is done atomically.

debug-task `&Optional` *task* [*function*]

Switches control over to debug *task*. The *task* argument is normally not supplied, in which case the function atomically pops off a task from the variable ***debugging-tasks***. Only tasks that are in a suspended state waiting for the terminal can be called with this function. This will wake up the task in the debugger with the terminal attached to `*debug-io*`.
When you exit from the debugger in any way, such as returning values from a call frame, reinvoking a call frame, or aborting the computation, control is returned back to the previous task, and the suspended task continues with its computation. The function **pause-debug** will re-suspend the task being debugged, returning control back to the previous task. Calling **debug-task** again will switch control back to where you left off in the debugger.

pause-debug [*function*]

Suspends a debugging session with a background task and switches back to the top-level task. This puts the current task back on the ***debugging-tasks*** list.

3.2.1 Background Debug Example

On the next page is an example session when no background windows are being used. The error message and backtrace are printed before the task is suspended. The text followed by ;; consists of comments, and is not part of the terminal interaction session.

```
USER: (setf val (process 'car 1)) ...     ;; Fork a process to do (car 1).
#<Future #<Process-10420 CAR {10420} 108003>>      ;; return value is a future.
;;
;; Indicates the error occured, and prints the error message
;; and backtrace before suspending the process.
;;
No Background IO. Background Task needs IO stream.
>Error: 1 was not a CONS.
DEBUG < %ERROR < CAR < PROCESS-TOP-LEVEL <
#<Process-10420 CAR {10420} 108003>> waiting to be debugged.
Use (sys::debug-task) to debug it.
USER: val ...                  ;; val is still a future object.
#<Future #<Process-10420 CAR {NEXT-TASK} 108003>>
USER: *debugging-tasks*  ...   ;; The task is waiting to be debugged.
(#<Process-10420 CAR {NEXT-TASK} 108003>)
USER: (sys::debug-task) ...         ;; Switch terminal control to task.
;;
;; Now running background task.
;;
CAR (PC 56D490) :
 Arg 0 (LIST): 1
-> <control-r>                 ;; We use control-r to return a value.
 Return value from CAR: 100
Return 100
from CAR (Y or N)? y => yes.
NIL
;; NIL is returned from the call to (task-debug), and
;; control is returned back to the previous task.
USER: val  ...                 ;; val is now determined to be 100.
100
```

Chapter 4

Pathnames

One of the most widely varying properties among different computer systems is the method used for specifying files. Common Lisp provides *pathnames* as a mechanism for dealing with filenames in a more portable manner.

A pathname has six logical components: *host*, *device*, *directory*, *name*, *type*, and *version*. By specifying these six logical components, it is possible to construct new pathnames that can share the same components and therefore represent related files. For example, the *type* component is almost always used to identify a particular format for a file. In Top Level Common Lisp, a file with a type of `''lisp''` represents a lisp source file; the same pathname with a type of `''zoom''` indicates a compiled version of the source file (it actually uses the value of `*compiled-file-type*` and `*source-file-type*`). Without pathnames a filename could not have components, but would exist only as a single entity, and therefore a program could not specify component relationships in a portable way.

By having components, pathnames also provide a mechanism for being *merged.* A default pathname can be used to supply any missing or unspecified components. With a default pathname established, redundant specifications can be eliminated.

A pathname is generally specified through a *namestring.* A namestring is intended to be an implementation-specific specification of a file name for a particular file system. To use such a namestring, there must be a mapping between a namestring and the components of a pathname.

4.1 Logical Namestrings

Common Lisp provides the functions `namestring` and `parse-namestring` to perform the mapping between a namestring and a pathname. While Common Lisp indicates that a namestring is an implementation-dependent representation, a namestring can still exist as a logical entity that is independent from a particular host's file system. Only when attempting to access a file is it necessary to have a host-dependent namestring. Top Level Common Lisp attempts to simplify pathnames by providing *logical namestrings* with a standard syntax for specifying pathname components. Thus, only a host-specific pathname-to-namestring translation is needed to access a file.

A logical namestring also can define *logical translations* that translate components from one logical pathname into another logical pathname. This provides a more portable and robust mechanism for specifying files in programs that will be used at different sites. The logical translations can be defined and modified as appropriate for any given site configurations and user needs.

A logical namestring has the following form:

host.device:directory;filename.type.version

A *directory* can have subdirectories separated by periods, such as `usr.local.lib`.

4.2 Logical Translations

In addition to providing a straightforward translation from a namestring to its pathname components, logical pathnames can have translations that are associated with specific values of a pathname's components. They can be used to provide a site- or user-specific access path to a file. A typical usage is to put translations in your `lisp-init` file that would define directory translations for the local host. For example, you can define a logical directory "`home`" that would translate to your home directory, such as "`users.staff.fred`".

```
(define-logical-host "local" "local"
  '(("home" "users.staff.fred")
    ("gbb"  "usr.local.lib.gbb")))
```

After this is evaluated, you can specify local files using a logical directory:
"`home;login-init.lisp`", which would translate into
"`users.staff.fred;login-init.lisp`" or
"`gbb.shell;load-shell.lisp` which would translate into
"`usr.local.lib.gbb.shell;load-shell.lisp`".

Logical pathnames are also used when specifying the pathnames for *module* files. This allows you to easily change where module files are located.

define-logical-host *logical-host translated-host translations* `&Optional` *redefine* [*function*]

Defines *logical-host* to translate to *translated-host.* You can define translations that translate to the same host. *Translations* should be an alist of the form (*logical-directory translated-directory*), where *logical-directory* and *translated-directory* are both strings. If *redefine* is `non-nil`, any previous translations are replaced. Otherwise the translations are added in, possibly shadowing any previous ones with the same *logical-directory.* Only translation of directory components is currently supported.

The Translation Process

Logical host pathnames are translated by finding the first matching translation. If the translated host is different, then the translation is again applied to the new pathname.

Translations are matched with the longest directory specification first. If that fails, then progressively shorter specifications will be tried, by dropping subdirectories from the end. For example, when translating a logical host pathname of the form "parser: syntax.words; nouns.lisp", "syntax.words" is tried first, followed by just "syntax". Directory translations replace the part where they match, and the translation process is then repeated.

Consider the logical host "parser" below.

```
(define-logical-host "parser" "zip"
  '(("syntax" "local.syntax")
    ("syntax.words" "syntax.zipwords")
    ("semantics" "local.semantics")))
```

The pathname "parser:syntax;nouns.lisp" would match with "syntax", and be translated to "zip:local.syntax;nouns.lisp".
The pathname "parser:syntax.words;nouns.lisp" would match with "syntax.words" and be translated to "syntax.zipwords", which is then matched with "syntax", and finally translates into "zip:local.syntax.zipwords; nouns.lisp".

If "zip" is also a logical host, then this result would also be subject to translation.

translate-pathname *pathname* [*function*]

Performs any logical pathname translations applicable to *pathname* and returns the translated pathname.

find-logical-host *hostname* [*function*]

Returns non-nil if *hostname* is a logical host.

4.3 Host-Dependent Namestring Translations

Top Level Common Lisp currently supports only local files, so all pathnames ultimately must translate with a host of "local" or with the name of the local host, and therefore translate into UNIX namestrings for actual file access.

While Top Level Common Lisp supports parsing of only logical pathnames, it must be able to translate from a pathname into a host-dependent namestring when accessing a file. This is accomplished by associating a translation function for each *host*.

The function **define-pathname-translation** will establish this translation function.

define-pathname-translation *hostname function* [*macro*]

When a namestring is needed for a pathname with a host of *hostname*, *function* is called with the pathname and should return a namestring for accessing the specified file.

Top Level Common Lisp currently supports only TI Explorer and Unix pathname translations. The following functions are used to perform the translations:

explorer-namestring *pathname* [*function*]

Translates *pathname* into a namestring for the Explorer. This is actually the same as `namestring`, since the Explorer will accept the pathname syntax used in Top Level Common Lisp.

unix-namestring *pathname* [*function*]

Translates *pathname* into a namestring for Unix.

4.3.1 Unix filenames

Unix pathnames can be specified using logical namestring syntax, such as
`"local:/etc/passwd"`,
but the pathname parses with a *name* component of `"/etc/passwd"` with no directory specified. Thus, if the pathname is merged with one that has a directory specified, the "expected" behavior will not occur. Specifying `"local:etc; passwd"` would produce the correct merging.

This is no way to specify relative directories.

4.4 Namestring Components with Syntactic Markers

A potential difficulty with using a logical namestring syntax is that a filename can have the syntactic markers of logical pathnames as a part of its name. For example, a directory can be `"a.b"`. Vertical bars can be used in these situations. All characters between vertical bars are not parsed. For example, to specify `"host:|a.b|;bar.lisp"` would specify a single directory named `"a.b"`.

Another case common in Unix occurs when a filename has dots in its name that are not used in the "normal" way, i.e., with only a single name and type. A filename `"read.me.now"` can be specified using vertical bars to prevent the `".now"` from being considered a version specification. i.e., `"read.|me.now|"` would specify a type of `"me.now"`.

To allow vertical bars to be specified, two contiguous vertical bars are considered a single vertical bar if they are not themselves inside a pair of vertical bars. For example, `"a||b.lisp"` would parse as `"a|b.lisp"`, but `"|a||b|.lisp"` parses as `"|a|"` followed by `"|b|"`, which is `"ab.lisp"`.

Chapter 5

The Inspector

The inspector provides a convenient way to browse through lisp objects. It is particularly useful for examining networks of structures, arrays or lists. The top level function is **inspect**.

inspect *object* `&Key` *level length array structure* [*function*]

Displays *object* and allows inspection and modification of its components. The keyword arguments *level*, *length*, *array*, and *structure* control the printed representation of objects during inspection. They correspond to the global variables `*print-level*`, `*print-length*`, `*print-array*`, and `*print-structure*`.

Each component of the current object is displayed and assigned a number. Entering its number will recursively inspect that object. The inspector maintains a stack of inspected objects, so you can pop the stack to return to the previous object being inspected. The variable `*` is bound to the current object being inspected. The variables `**` and `***` will be bound to the two previous objects inspected as well.

Since some lists or arrays can be quite large, "typing ahead" (e.g., hitting the space bar) will stop the display of their components. You can also set the range of objects to display, so only a portion of the object will be displayed. Even when a range is being used, all components of the object may still be selected.

Components of an object can be modified without regard to the semantic content of the component, so be careful when modifying objects using this facility.

5.1 Inspector Commands

Below are the inspector commands. They can be invoked either through single keystrokes, or entered as symbols. The symbols can be abbreviated.

`?` or `HELP`	Prints out available commands and their documentation.
`REDISPLAY` or `CTRL-l`	Clears the screen and redisplays the current object.
`RANGE` or `CTRL-r`	Prompts for object indexes to select the range of an object to display.
`HISTORY` or `CTRL-h`	Shows the top of the inspection stack.
`NEW` or `CTRL-n`	Prompts for a form to evaluate. The result is pushed as a new object to inspect.
`BACK`, `POP`, or `CTRL-p`	Pops the inspect stack.
`PPRINT`	Pretty prints the current object.
`EVAL` or `,`	Prompts for a form to evaluate and prints the result. The variable `*` is bound to the current object being inspected, so you can directly manipulate the current object.
`SET` or `=`	Prompts for an index and a form to evaluate. The result is stored into the specified component of the current object.
`EXIT`	Exits the inspector, returning the original inspect argument.
`RETURN`	Exits the inspector, returning the current object.
`CTRL-G`	Exits the inspector, returning `nil`.

An example interaction with the Inspector is presented on the next page.

5.2 Inspector Example

```
USER: (inspect 'bar) ...     ;; Call inspect on the symbol 'bar.
It is the symbol BAR.
[0] Package:   #<Package USER>
[1] Value:     Unbound
[2] Function:  Undefined
  Plist:  ()

Inspect: 0    ;; Select the package slot of symbol.

It is a PACKAGE.
[0] :TABLES
(NIL
 #<Package-Hashtable: Size = 2003, Free = 1077, Deleted = 0>
 #<Package-Hashtable: Size = 21, Free = 8, Deleted = 0>)
[1] :NAME
"USER"
[2] :NICKNAMES
NIL
[3] :USE-LIST
(#<Package MODULES>
 #<Package LISP>)
[4] :USED-BY-LIST
NIL
[5] :INTERNAL-SYMBOLS
#<Package-Hashtable: Size = 21, Free = 9, Deleted = 0>
[6] :EXTERNAL-SYMBOLS
#<Package-Hashtable: Size = 9, Free = 9, Deleted = 0>
[7] :SHADOWING-SYMBOLS
NIL
[8] :LOCK
#:<LOCK>

Inspect: 8   ;; Select the :LOCK slot of the package.

It is the symbol #:<LOCK>.
[0] Package:   NIL
[1] Value:     NIL
[2] Function:  Undefined
  Plist:  ()
;;
;; A , prompts for form to evaluate. We access the
;; current object using * and set it to variable plock.
;;
Inspect: Evaluate: (setf plock *)
#:<LOCK>
Inspect: <control-g>    ;; Quit from inspector, returning nil.

NIL
USER: plock ...         ;; now plock bound to the lock.
#:<LOCK>
USER:
```

Chapter 6

Tracing and Stepping

6.1 The Trace Facility

The trace facility is a debugging tool that allows you to monitor the calling of a function. Many options are available to have various actions performed when a traced function is called.

trace `&Optional` *function* `&Key` *step wherein break break-after break-all print print-after print-all* [*macro*]

With no arguments, **trace** returns a list of traced functions. **untrace** turns off tracing. Keyword arguments are as follows:

`:step`	If form evaluates to `non-nil`, single-stepping is turned on when *function* is called.
`:condition`	If form evaluates to `non-nil`, trace output is surpressed.
`:wherein`	Argument is a list of functions. Trace output is surpressed unless it is called from within one of these functions.
`:break`	If form evaluates to `non-nil`, **break** is called before *function* entry.
`:break-after`	If form evaluates to `non-nil`, **break** is called after *function* returns.
`:break-all`	The form is used as the value for both `:break` and `:break-after`.
`:print`	Argument is a list of forms that are evaluated and printed before *function* entry.
`:print-after`	Argument is a list of forms that are evaluated and printed after *function* returns.
`:print-all`	The argument is used for both `:print` and `:print-after`.

untrace `&Rest` *functions* [*macro*]

Turns off tracing for the specified function names. If none are supplied, all traced functions are untraced.

trace-args [*variable*]

Bound to the arguments of a traced function while evaluating any trace forms.

trace-values [*variable*]

Bound to the return values of a traced function while evaluating any trace forms.

trace-function-list [*variable*]

A list of function names currently being traced.

trace-print-level [*variable*]

Bound to `*print-level*` while printing any trace output.

trace-print-length [*variable*]

Bound to `*print-length*` while printing any trace output.

max-trace-indentation [*variable*]

The maximum number of spaces that should be used to indent trace output.

6.2 Encapsulations

The trace facility is built using the encapsulation facility. Encapsulations provide a convenient way to put wrappers around a symbol's function. A function can have many encapsulations, each independent from the others. Each encapsulation has an associated *type*, which is generally used to maintain only one encapsulation for each type, but it is possible to have many encapsulations of the same type.

Macros cannot be encapsulated, but their *expansion* functions can. If the symbol argument in the following operators names a defined macro, the *macro-function* of the symbol will be encapsulated. Note that a macro-function takes two arguments, the macro form, and an environment. The `argument-list` in an encapsulation for a macro-function will be a two-element list, containing these two arguments.

define-encapsulation *(symbol type)* `&Body` *body* [*macro*]

Defines an encapsulation of *symbol*'s function of the specified *type*. Both *symbol* and *type* should be `non-nil` symbols and are not evaluated. *Body* is replaced as the function definition of *symbol*. While in the body, `basic-definition` is bound to the encapsulated definition of the function, and `argument-list` is bound to the argument-list supplied for the function call. By evaluating `(apply basic-definition argument-list)`, the encapsulated definition can be called with the original arguments. **Define-encapsulation** will remove all previous encapsulations of *type* from *symbol* before installing this one, so it is useful for specifying a relatively permanent encapsulation that might be loaded in more than once. It also allows *Body* to be compiled, unlike the **encapsulate** function.

encapsulate *symbol type body* [*function*]

Encapsulate is similiar to **define-encapsulation**, but it is a function, not a macro. Therefore, all arguments are evaluated, including *body*. It is also different in that it will *not* remove any previous encapsulation of *type* for *symbol*.

encapsulated-p *symbol type* [*function*]

Returns `t` if *symbol* has an encapsulation of type *type*.

unencapsulate *symbol type* [*function*]

If the definition of *symbol* contains an encapsulation of *type*, it is removed and **unencapsulate** returns `t`. Otherwise, **unencapsulate** returns `nil`.

basic-definition *symbol* [*function*]

Returns a symbol that is `fbound` to the innermost definition of *symbol*. For a function that is not encapsulated, this simply returns *symbol*

6.3 The Stepper

The `step` macro can be used to single-step through interpreted code.

step *form* [*macro*]

Single-steps through the evaluation of `form`, prompting for commands before each subform evaluated. The recognized commands are as follows::

`N`	Evaluates the expression, stepping subforms.
`S`	Evaluates the expression without stepping subforms.
`Q`	Evaluates the expression without further stepping.
`p`	Prints the expression about to be evaluated.
`P`	Pretty prints the expression about to be evaluated.
`B`	Enters a Break loop.
`E`	Prompts for a form to evaluate.
`H`	Prints some help on available commands.
`?`	Prints some help on available commands.
`R`	Prompts for a form to evaluate and return as the value of the form to be stepped through.
`A`	Aborts the evaluation.

6.3.1 Stepper Printing Variables

The following variables are used to control the way objects are printed during stepping.

step-print-level [*variable*]

Bound to `*print-level*` while printing any step output.

step-print-length [*variable*]

Bound to `*print-length*` while printing any step output.

max-step-indentation [*variable*]

The maximum number of spaces that should be used to indent step output.

Chapter 7

The Compiler

Top Level Common Lisp has extended the Common Lisp function `compile-file` with additional arguments.

compile-file `&Optional` *input-pathname* `&Key` *output-file error-file lap-file errors-to-terminal lap-file load verbose package readtable declarations parallel* [*function*]

`input-file`	The source file to compile. If this is not supplied, it will be prompted for. The file type "lisp" will be defaulted into the filename if needed.
`output-file`	The output file. If `nil`, no output is produced. If `t`, it defaults to the same file as the input, but with the type "zoom". Otherwise, it should be the name of a file.
`error-file`	The error message output file. If `nil`, no error file is created. If `t`, it defaults to the same file as the input, but with the type "err". If it is a stream, then error messages are written to the stream. Otherwise, it should be the name of a file.
`lap-file`	The lap output file. Lap output will display the psuedo assembly language form of the compiled functions. If `nil`, no lap output is produced. If `t`, it defaults to the same file as the input, but with the type "lap". Otherwise, it should be the name of a file.
`errors-to-terminal`	If `non-nil`, error messages are printed to `*standard-output*`.
`load`	If `non-nil`, all forms compiled are effectively loaded into the compiler environment as they are compiled.
`verbose`	If `non-nil`, the compiler prints out a message as each function is compiled.
`package`	Compiles the file in the specified package. It defaults to `*package*`. It does *not* override any `in-package` form in the file.
`readtable`	Uses the specified readtable when reading the file.

`parallel`	If `non-nil`, the compiler uses multiple processes to compile the file. This defaults to the value of ***parallel-compile***.
`declarations`	If this argument is supplied, it should be a list of declarations, e.g., `'((special *foo*))`. The file is then compiled with the specified declarations in effect. The previous declaration state is restored after the file is compiled.

7.1 Compiler Optimizations

The compiler translates Lisp source code into a much more efficient form for run-time execution and run-time loading. The cost of this efficiency is generally a reduction in the ability to debug the resulting code. This results mostly from the inability to determine where in the source code an error occurs. This is made difficult by the transformation of function calls. Function calls can be transformed in three ways. They can be:

- compiled into a subroutine call to an assembly language routine,
- transformed into a more efficient Lisp form, or
- replaced by an inline expansion of the function's definition.

An understanding of disassembled code can help in the first case, but the others are more difficult.

In addition, the use of declarations can allow the compiler to eliminate run-time type-checking. Violation of these declarations may not be detected at run-time, and can result in obscure and hard-to-find bugs, or even complete system failure.

The amount of compiler optimization is controlled by the values of a number of variables. To provide finer control over the operation of the compiler, the Common Lisp `optimize` declaration accepts many additional specifications that can be used to set various optimization parameters. Some of these variables are modified by the use of the `speed`, `space`, and `safety` declarations. If a `declare` form with the extended specifications also includes `speed`, `space`, or `safety` specifications, their effect is processed *before* any extended specifications.

The best strategy is to debug your program using either interpreted code or code compiled with almost all optimizations turned off. After the code is debugged, you can then compile it with full optimization so the code can run as fast as possible.

The following variable can be used to turn off *all* processing of optimize declarations:

ignore-optimizations [*variable*]

If `non-nil`, `optimize` declarations are ignored.

The extended `optimize` parameters are described on the following page. Unless otherwise specified, the effective action of the parameter is turned on when its value is `non-nil`. If a parameter is not specified, its current value is not modified. Note that these parameters specify only that it is *possible* to perform the optimizations.

`all-lisp-calls`	If this is non-`nil`, then the compiler will perform no optimizations at all, but simply generate calls to the source-level Lisp functions. Even functions such as `car` and `+` will be compiled as a function call. True if `(and (eq safety 3) (>= speed space))`. Default is `nil`.
`inline-lisp`	If `nil`, Lisp functions declared inline will not be expanded inline. True if `(> speed space)`. Default is `t`.
`inline-sequence`	Sequence functions are transformed into `DO` loops. True if `(> speed space)`. Default is `t`.
`inline-list`	List accessors are coded inline, e.g., `car`, `cdr`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-cons`	List creation is coded inline, e.g., `cons`, `list`, `list*`. True if `(and (eq speed 3) (eq space 0))`. Default is `nil`.
`inline-array`	Array references are coded inline, e.g., `svref`, `schar`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-structure`	Structure references are coded inline. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-logic`	Logic operations are coded inline, e.g., `logior`, `logbitp`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-symbol`	Symbol operations are coded inline, e.g., `symbol-value`, `symbol-function`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-char`	Character operations are coded inline, e.g., `int-char`, `code-char`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`inline-predicate`	Type predicates are coded inline, e.g., `symbolp`, `consp`. True if `(and (= speed 3) (= space 0))`. Default is `nil`.
`fold-constants`	Constants are folded together if possible. This includes constants declared using `defconstant`. Default is `t`.

`tail`	Self-tail recursion is converted to a branch if possible. Other tail position calls are merged so the current call frame is reused instead of building a new one. This can make debugging difficult, since the call history is somewhat hidden. True if `(> speed space)`. Default is `t`.
`peep`	If `nil`, all peep optimization is turned off. If `t`, peep optimization is done to eliminate push-pops of registers, which is fairly fast, and results in much smaller and faster code. If `:best`, then many other optimizations occur, such as multiple branch eliminations, elimination of redundant move operations, etc. This produces the best possible code, but is fairly time-consuming and makes debugging even disassembled code difficult. False if `(or (< space cspeed) (< speed compiler-speed))`. Bound to `:best` if `(or (= speed 3) (= space 3))`. Default is `t`.
`array-bounds`	Inline array access checks array bounds. Default is `t`.
`no-type-checks`	Type-checking is not performed when the type is known or must be of a particular type. For example, if disabled, `car` will not check that its argument is a list. True if `(and (eq speed 3) (eq safety 0))`. Default is `nil`.
`check-keywords`	If `nil`, extra unknown keyword arguments are *not* flagged as errors. Essentially, this makes all keyword parameters have an implicit `&Allow-other-keys` . Default is `t`.
`let-bindings`	If `t`, typical trivial let bindings are eliminated when possible, replacing the binding reference with the value in which it would be bound. If `:all`, then all `let` forms are checked for trivial let bindings. This is fairly time-consuming. If `nil`, no let bindings optimizations are performed. Default is `t`.
`list-rest-args`	This declares `&Rest` arguments to be lists. They all start out that way, but the rest argument can be modified to be something else. Default is `t`.
`all-fixnums`	This declares that all integer values will be fixnums. Default is `nil`.
`all-common`	No future objects will ever be seen by the code. Default is `nil`.

never-tail-call [*variable*]

This is a list of function names whose calls are never optimized into a tail call. The system includes `error`, `cerror`, and `break`, since it is important to know which function they were called from.

7.2 Compiling Defstructs

In addition to the above declaration specifications, the operation of defstruct can be controlled using these variables.

inline-structure-functions [*variable*]

> If this is `non-nil`, functions created by `defstruct` will be declared to be `inline`. This is at the Lisp level, whereas the `inline-structure` declaration affects whether *these* inlined functions are compiled with inline instructions. The default is `t`.

safe-structure-accessors [*variable*]

> If this is `non-nil`, structure accesses will always be checked for *full* type correctness. This checking has a substantial time and space penalty. The default is `nil`.

7.3 Compiler Transforms

deftransform *fn-name arglist* `&Body` *body* [*macro*]

> Defines a compiler macro. When a call to *fun-name* is encountered by the compiler, it will expand it as a macro using the defined transform. This allows different behavior for interpreted and compiled code, so be very careful to maintain the same semantics. If the transform decides not to transform the form, it should return the value `'sys::%pass%`. *fn-name* can be `(fun transform-name)`, or just `fun`, in which case "`Fun`"`-transform` is used as the transform name. If *body* is `nil`, then *arglist* must be a transform name that is added to the transforms for the *fn-name*.

7.4 Parallel Compilation and Read-Time Evaluation

When a file is being compiled in parallel, the compiler will use a three-process pipeline, where one process reads from the file, another process compiles the forms, and another assembles and writes the output file. This maintains the semantics defined by Common Lisp, but an unexpected problem can arise if a file uses reader macros. Since the reader can essentially read the entire file before any forms are processed, using reader macros will not work if they depend on functions, macros, variables or constants defined in the same file.

You should move the code that the reader depends on to another file, and load that file in before compiling the one that uses the code. Alternatively, you can use the `#.` reader macro to have the needed code evaluated in the reader process. For example, if you want to define a new data type that is a subrange of integers, one of the limits could be obtained from a constant with your code looking something like this:

```
(eval-when (compile load eval)
(defconstant *x-max* 255) ;; Define the *x-max*
)

(deftype x-range () '(integer 0 #.*x-max*))
```

The problem is that `*x-max*` will not be bound in the reader process when it encounters the `#.*x-max*`. To solve this problem, you can also put a `#.` before the `defconstant` form so the reader will evaluate the defconstant:

```
#.(defconstant *x-max* 255) ;; Define the *x-max*
(deftype x-range () '(integer 0 #.*x-max*))
```

The difficulty with this fix is that the defconstant form won't be seen by the *compiler*, since `#.` will return the *result* of the evaluation, not the form itself, and the constant definition won't be placed in the output file. The solution is to define a macro that will evaluate a form, and then return it, so the compiler process will see the form. The macro **eval-when-read** is defined for this purpose.

eval-when-read *form* [*macro*]

Evaluates form for side-effect, and returns it.

Using this macro, you can get the reader to evaluate the defconstant form, and also have it be seen by the compiler:

```
#.(eval-when-read
(defconstant *x-max* 255) ;; Define the *x-max*
)
(deftype x-range () '(integer 0 #.*x-max*))
```

Chapter 8

Modules

When a program consists of more than one file, tools are needed to help manipulate those files. The Common Lisp functions `require` and `provide` make up only a rudimentary facility, which is inadequate for most purposes. Top Level Common Lisp has extended the notion of *modules* into a more powerful facility that provides a set of functions for grouping files together, describing their relationships, and manipulating them.

The module facility also has specifications that allow loading and compiling files in parallel. Top Level Common Lisp will use multiple processes to load and compile parallel modules, which can substantially speed up the process of compiling and loading a program.

The macro **defmodule** is used to associate a name with a set of files. It is also provided a pathname that supplies any defaults for the filenames that make up the module. The function **load-module** is used to load in the files. For example, the following can be used to define the compiler program:

```
(Defmodule (Compiler "sys: compiler;")
  "pass1"
  "pass2")
```

To load in the compiler, use `(Load-Module 'Compiler)`. This will load in the compiled files "`sys: compiler; pass1.zoom`" and "`sys: compiler; pass2.zoom`." Another call to **load-module** will not load any files, since the module system remembers it has already loaded in the `Compiler` module. Therefore, all programs that use the Compiler program can call **load-module** without having the files reloaded for each program.

During development of a module, any files that are changed must be recompiled to keep the program updated. The function **compile-module** is used to recompile and load in any files that have changed since the last time the program was loaded. When developing the program, you typically start your development by calling **compile-module** to compile and load in the newest versions of all the module files. With the latest files loaded, you can edit the functions within the files, incrementally modifying the latest version of the program. After saving these changes, calling **compile-module** will once again compile and load any modified files.

This process usually is more complicated because of *interdependencies* between the files. For example, you can have a file that contains macro definitions. The macros must be defined before any files that use them can be compiled. Moreover, if changes are made to this file, the files that use the macros must be recompiled to use the new macro definitions.

There are two different kinds of dependencies between files. The above example is a *compile* dependency, but *load* dependencies also exist between files. For example, a file may call a function to initialize the value of a variable. The file containing this function must be loaded before the file initializing the variable can be loaded. Changes in the file containing the function do not require the file using the function to be recompiled, only reloaded. The full range of dependencies is quite complex, and their specification is equally complex. The module facility in Top Level Common Lisp provides a fairly simple and straightforward way of expressing dependencies that captures most cases and is not syntactically cumbersome.

In addition to specifying dependencies, the files of a module can also be specified by using an existing module. Thus, you can have a set of modules that can be combined to form larger modules for specific purposes. For example, the `Compiler` module given earlier may contain an `Assembler` module as a sub-module.

While the module facility is generally used for compiling and loading Lisp files, it is actually a general facility that can be used for manipulating files of any kind. For example, a system for understanding natural language system may have a set of files that make up a dictionary. These files can be defined and manipulated as modules as easily as Lisp files.

The macro defmodule is used to define modules. It is described on the following page.

8.1 Defmodule

defmodule *(name merge-pathname* `&Key` *keyword-options)* `&Rest` *filespecs* [*macro*]

Defines the module *name* consisting of the associated files specified in *filespecs*. The *merge-pathname* argument is used to supply any defaulting of pathname components for the files in *filespecs*. There are many *keyword-options* that supply additional information or specifications to the module. The following keywords are recognized:

`:full-name`	A string that provides a more verbose name for the module.
`:documentation`	A documentation string for the module.
`:source-only`	Specifies that the files in the module have a source-type only and do not get compiled.
`:source-file-type`	Overrides the default file type that identifies the source files.
`:compiled-merge-pathname`	Overrides the pathname that is merged into the filenames to produce a compiled filename. It defaults to the load merge-pathname that is a required argument.
`:compiled-file-type`	Overrides the default file type that identifies compiled files.
`:loader`	Names a function that is called to load a file. It is passed the pathname to load and a `:verbose` keyword argument. It defaults to the `load` function.
`:compiler`	Names a function that is called to compile a file. It is passed the source pathname to compile, an `:output-file` keyword argument that is the compiled pathname to produce, and a `:verbose` keyword argument. It defaults to `compile-file`.

In addition to these keywords, **defmodule** also takes the file-interface-form keywords `:in-package`, `:shadow`, `:export`, `:use-package`, `:import`, and `:proclaim`. These are used for the expansion of the **in-module** form, which is described in Section 8.2.

8.1.1 Defmodule File Specifications

The *filespecs* of a **defmodule** can take many different forms. A string (surrounded by double quotes) specifies the name of a file. It should *not* include a file type, but otherwise can be any pathname. The merge-pathname of the module is merged with this filename to produce the required pathname.

A symbol is a reference to another module. Operations on the defined module will be passed on to the sub-module.

If the specification is a list, it can be in one of five forms. If the first element of the list is `:using` or `:using-self`, it specifies a dependency. If it is `:serial`, it specifies that the filespecs are to be processed serially from left to right. If it is `:parallel`, each filespec in the list can be both loaded and compiled in parallel. If the first element is not a keyword, then it is equivalent to a `:parallel` specification, so each filespec can be both loaded and compiled in parallel.

Top Level Common Lisp will use multiple processes to load and compile parallel modules, which can substantially speed up compiling and loading of code.

These specifications are best understood through examples. We start with the simple Compiler example given earlier. The following specifies the serial ordering of two files, "pass1" and "pass2". The file "pass1" is loaded and compiled before "pass2":

```
(Defmodule (Compiler "sys: compiler;")
  "pass1"
  "pass2")
```

The following example specifies that the two files, "pass1" and "pass2", use definitions in "macros", so "macros" must be compiled and loaded before "pass1" and "pass2", and also that any changes in "macros" require recompilation of the two files:

```
(Defmodule (Compiler "sys: compiler;")
  (:Using "macros"
    "pass1"
    "pass2")
  )
```

The first specification after the `:using` can be any specification. The following example specifies two files that are used by "pass1", and furthermore, that the two files can be processed in parallel.

```
(Defmodule (Compiler "sys: compiler;")
  (:Using ( "macros" "structs" )
    "pass1")
  )
```

By including the `:serial` keyword, the file "macros" is processed before the file "structs":

```
(Defmodule (Compiler "sys: compiler;")
  (:using (:serial "macros" "structs" )
    "pass1")
  )
```

The following example specifies that "pass1" and "pass2" can be compiled and loaded in parallel after the file "macros" is loaded:

```
(Defmodule (Compiler "sys: compiler;")
  (:using ( "macros" )
    ( "pass1" "pass2" )
  )
```

The next example shows the use of sub-modules. The assembler module is included in the compiler module and can be compiled and loaded in parallel with the other compiler files:

```
(Defmodule (Assembler "sys: assem;")
  "assembler"  ;; Contains only one file.
  )

(Defmodule (Compiler "sys: compiler;")
  ( Assembler  (:using ( "macros" )
                 ( "pass1" "pass2" )) )
  )
```

The final example demonstrates the use of the more general capabilites of modules. The primops module consists of assembly language routines compiled using the `assemble-file` function. They can all be assembled in parallel:

```
(Defmodule (Primops "sys: primop;"
             :SOURCE-FILE-TYPE "prim"
             :COMPILER assemble-file)
  ( "arrays" "lists" "symbols" "calls" )
  )
```

8.2 In-Module

Programs often create their own packages, exporting from the package the functions and macros that are part of the documented interface to the program. Conversely, when using a program, you will also use the program's package. The module facility allow you to specify which file-interface forms should be part of the files that make up the module. The **defmodule** form accepts the keyword options `:in-package`, `:shadow`, `:export`, `:use-package`, `:import`, and `:proclaim`. The values of these arguments are used to construct a file-interface form that is returned by the macro **in-module**. By placing an `(In-Module 'Compiler)` form in each of the module files, you insure they are compiled and loaded with the correct package manipulation forms for the module. In addition, this allows all of this information to be kept with the definition of the module, instead of spreading it throughout the files of the module. Therefore, any changes to the package structure of the module need be done only in the **defmodule** form.

Here is an example:

```
(Defmodule (modules "sys: code;"
             :IN-PACKAGE modules
             :USE-PACKAGE ("FILE-SYSTEM" "LISP")
             :EXPORT (defmodule in-module load-module compile-module)
           )
 "modules")

(In-Module 'Modules)   ==>

 (progn
    (in-package 'modules)
    (use-package '("FILE-SYSTEM" "LISP"))
    (export '(defmodule in-module load-module compile-module))
  )
```

8.3 Module Operations

load-module *name* `&Key` *print verbose reload source loader compiler version shadows* [*function*]

Loads in the module *name*. The keywords control how the loading is done.

`:print`	If `nil`, disables printing of what is happening.
`:load-print`	Passed on to the `load` function.
`:verbose`	Passed on to the `load` function.
`:reload`	If `non-nil`, the module is reloaded.
`:source`	If `non-nil`, only the source files are loaded.
`:loader`	If `non-nil`, it should be a function used to perform the loading.
`:compiler`	If `non-nil`, it should be a function used to perform any compiling.
`:version`	If `non-nil`, specifies the version of the module to load in. If the argument is `:released`, the "released" version is loaded. Otherwise, it should be a number. This numbered version of the system is loaded. **release-module** is used to create a module version. If this is `nil`, the newest version of the module files are loaded.
`:shadows`	If `non-nil`, it is a list of pathnames that are merged into all module's filenames. Any files found using these pathnames are loaded instead of the regular files. This allows you to have locally modified copies of module files. See also **shadow-module.**
`:parallel-load`	If `nil`, parallel modules will be loaded serially. It defaults to the value of ***parallel-load-modules***.
`:parallel-compile`	If `nil`, parallel modules will be compiled serially. It defaults to the value of ***parallel-compile-modules***.

compile-module *name* `&Key` *print verbose reload source loader compiler shadows no-force sub-modules* [*function*]

Compiles any modified module files and loads in the module. The keywords are similiar to those allowed for **load-module**, but no `:version` keyword is allowed, since **compile-module** always refers to the newest version of the files.

`:print`	If `nil`, disables printing of what is happening.
`:load-print`	Passed on to the `load` function.
`:verbose`	Passed on to the `load` function.
`:recompile`	If true, all module files are recompiled and loaded.
`:reload`	If `non-nil`, the module is reloaded.
`:source`	If `non-nil`, only the source files are loaded.
`:loader`	If `non-nil`, it should be a function used to perform the loading.
`:compiler`	If `non-nil`, it should be a function used to perform any compiling.

`:shadows`	If `non-nil`, it is a list of pathnames that are merged into all module's filenames. Any files found using these pathnames are compiled and loaded instead of the regular files. This allows you to have local modified copies of module files. Also see **shadow-module**.
`:no-force`	If `non-nil`, it does not use the module dependencies to force recompiling and loading of modules. It will only recompile and load any files that have changed since the last call to **compile-module**. Use this if you know changes in modified files are not significant (e.g., only documentation strings were changed).
`:sub-modules`	If `nil`, then any sub-modules within the module are not compiled, but only loaded.
`:parallel-load`	If `nil`, parallel modules will be loaded serially. It defaults to the value of ***parallel-load-modules***.
`:parallel-compile`	If `nil`, parallel modules will be compiled serially. It defaults to the value of ***parallel-compile-modules***.
`:error-output`	If `nil`, then all error-output from the compiler is discarded. If `t`, then error-output is sent to `*standard-output*`. If `:file`, then error-output files are created for each file compiled. These are shown after the module has been compiled and loaded. If `:both`, then the output is sent to `*standard-output*` and error-output files are also created, but they will not be shown.

release-module *name* `&Optional` *version* [*function*]

Creates a version of the *name* module. It takes all the module files that are currently loaded and copies them, but changes their file type. If *version* is not supplied, then it uses "`rel`" as the file type, which becomes the "`:released`" version of the module. If *version* is a number, it uses "`REL`*n*" as the file type, where *n* is the version number. **Load-module** can then be called with this version number (or `:released`), and only these files will be loaded.

copy-module *name host* `&Key` *source* [*function*]

Copies all files from the *name* module to the same place on *host*. By using logical hosts with directory translations, the module files can be copied easily to a new location. If *source* is `nil`, only compiled versions of the files will be copied.

shadow-module *name pathlist* `&Optional` *sub-modules* [*function*]

Makes *pathlist* shadow the filenames for the *name* module. When a module has a shadow *pathlist*, it will first try to find module files by merging the shadow pathnames into the filenames. Any files found using these pathnames are used instead of the regular files for both compiling and loading. This allows you to have your own local modified copies of module files. If *sub-modules* is `non-nil`, then *pathlist* is used to shadow all the sub-modules within *name* also. It defaults to `nil`. You can also achieve the same effect by supplying *pathlist* as the `:shadows` argument to either **load-module** or **compile-module**. Note that you can have both. First the `:shadows` pathnames are tried, followed by any module shadows, and finally, by the regular files.

describe-module *name* `&Key` *compile-plan in-module sub-modules* *[function]*

Describes the *name* module. If *compile-plan* is `non-nil`, it will show any actions that would take place if **compile-module** were called. If *in-module* is `non-nil`, it will show the **in-module** expansion for this module. If *sub-modules* is `non-nil`, it also describes any sub-modules within *name*.

module-files *name* `&Key` *source sub-modules* *[function]*

Returns all the module filenames for *name*. It does not include any shadowed filenames. If *source* is `nil`, it returns compiled filenames. If *sub-modules* is `nil`, it does not include files from sub-modules.

find-module *name* `&Optional` *error-p look-up* *[function]*

Tries to return the module definition of *name*. If it is currently defined, it returns the module. If ***auto-load-modules*** is `non-nil`, it will try to load in the modules definition file. If this is unknown, and *look-up* is supplied, that file is loaded and should define the module, or an error is signaled (regardless of *error-p*).
If ***auto-load-modules*** is a list, then it should be a list of pathnames. For each pathname, the name of the module is merged with the three file types ***module-file-type***, ***compiled-file-type*** and ***source-file-type***, in that order. If any of these files exist and define the module, the module is returned. If all files have been tried and no module is defined, **find-module** will return `nilif` *error-p* is `non-nil`, otherwise an error is signalled.

module-definition-file *name* *[function]*

Returns the file where the module definition of *name* is located, if known. This is `setf`-able, so you can set the module-definition-file of *name*. **find-module** will load in this file to define the module.

auto-load-modules *[variable]*

If `non-nil`, module definitions are loaded when referenced. It can be a list of pathnames to look up module definitions as described in **find-module**.

defined-modules [*variable*]

Bound to the list of all defined modules.

8.4 File Types

These variables are used as the default values of the file types.

module-file-type [*variable*]

The file type used for looking up module definitions.

source-file-type [*variable*]

The default file type used for source files. It defaults to “lisp”.

compiled-file-type [*variable*]

The default file type used for compiled files. It defaults to “zoom”.

8.5 Using Parallelism for Parallel Operations

These variables control whether the module program will use parallelism when compiling and loading modules.

parallel-compile-modules [*variable*]

If `non-nil`, parallel module specifications will use **process** calls to perform compilations.

parallel-load-modules [*variable*]

If `non-nil`, parallel module specifications will use **thread** calls to perform loads. Since it uses **thread** calls, some number of needles must be executing for parallel loads to occur.

Chapter 9

Lisp Worlds

Top Level Common Lisp should be booted using the Unix command "`topcl`." The exact method depends on how the system was installed.

9.1 Booting a Lisp World

When booting TopCL, a number of parameters can be specified.

`-core` *corefile*	The core file or Lisp world to boot. Defaults to "`/usr/topcl/topcl.core`", or to the value of the Unix environment variable `LISPCORE`.
`-nolistener`	Disables bringing the system up with the lisp listener. This Useful if you are running Lisp within an Emacs buffer, from a hardcopy device, or as a batch job.
`-noinit`	Disables the loading of your "lisp-init" file in your home directory before running the top-level loop.
`-boot` *number*	*number* is set to `sys::*boot-arg*` before the initializations are run.
`-args`	Allows unknown command line arguments to be supplied, which otherwise would abort the booting. It can be used to supply additional arguments that can be looked at by an initialization function.
`-gcpages` *pages*	Specifies how many 4K pages of memory will be allocated before GC is invoked. GC's will occur less often with a larger value, but will require a corresponding larger amount of swap space.
`-max` *address*	Specifies the largest virtual address TopCL will use. This essentially limits the amount of swap space that Lisp will consume. Note that the system will terminate without warning when its address space is exhausted.
`-nogc`	Disable GC, using all memory up to `-max` as Newspace memory. This can be used to run experiments that can complete without

	requiring a GC, and therefore produce more stable timings. However, note that the system will crash without warning when its address space is exhausted.
`-verbose`	Turns on printing of certain boot parameters.
`-debug`	Turns on printing of certain run-time messages.
`-syspages` *pages*	Specifies the number of 4K pages to leave for any C code memory allocation.
`-site` *sitefile*	Specifies the Site.Authorization file to a valid use of Lisp. Defaults to "`/usr/topcl/Site.Authorization`", or to the value of the Unix environment variable `LISPSITE`.
`-netdebug` *hostname*	Establishes *hostname* as the network debugging host. The hostname must have the netdebug server installed. If not specifed, the value of the Unix environment variable `LISPHOST` will be used. If this variable is not defined, then no network debugging will be set up.
`-netport` *portnumber*	Uses *portnumber* as the TCP port number to connect to the network debugger.

In addition, the following options are supported on the Sequent Symmetry:

`-copy` *corefile*	Specifies a core file copy to use. It defaults to `/usr/topcl/tmp/topcl.core-copy` or the value of a defined `LISPCORECOPY` environment variable. These defaults will be overridden by an explicit `-core` option to the boot command. In this case, a core file with the same name, but with `-copy` appended will be used as the actual core file. If a core copy file can be accessed, this core copy will be renamed and booted. The real core file will then be copied back into the copy core file. This allows a boot to occur immediately without waiting for the copy to complete.
`-nocopy`	Disables the use of a core file copy.
`-tmp` *directory*	Specifies a directory to keep temporary files. It defaults to `/usr/topcl/tmp` or a defined `LISPTMP` environment variable. This directory is used for maintaining oldspace for garbage collection, and other system files.

unix-command-line [*function*]

Returns a list of strings for each item on the command line. For example with a boot command of `% topcl -boot 10 -args fe fi fo`, it returns `("topcl" "-boot" "10" "-args" "fe" "fi" "fo")`

unix-command-line [*variable*]

This global variable is bound to the Unix-command-line that was used to boot Lisp. It is set before any initializations are performed.

boot-arg [*variable*]

This global variable is bound to the value of the `-boot` argument, as an integer, in the command line. It is set before any initializations are performed.

user-initializations [*variable*]

Each function on this list is called before **%top-level** is called.

9.2 User Login

When Lisp is booted, it will "log" you in to the Lisp system by looking up your user-id and obtaining your user information from "`/etc/passwd`". The variable ***user-info*** is bound to a `user` structure that contains each item from the "`/etc/passwd`" file. The following functions can be used to extract the information.

user-name *user* [*function*]
user-password *user* [*function*]
user-id *user* [*function*]
user-group *user* [*function*]
user-full-name *user* [*function*]
user-directory *user* [*function*]
user-shell *user* [*function*]

After Lisp obtains this information, it will look for the file "lisp-init.lisp", or a compiled version of it, in your home directory. If the file is found, Lisp will silently load this file into the Lisp system, unless a `-noinit` was specified on the command line. If an error occurs during the loading of this file, the load is aborted and the message "Error occured while loading lisp-init: " is printed, followed by a message for the error found.

9.3 Creating a Lisp World

The function **save-world** is provided to create a copy of the currently executing Lisp world. This can be used to create a core image that will contain user customizations without requiring loading in the code each time the system is booted.

save-world *unix-filename* `&Key` *gc top-level* [*function*]

Copies the current state of the system into *unix-filename*. *Unix-filename* should be a string that is passed directly to the operating system as a filename. If *gc* is `:full` (the default), then a **full-gc** is performed before writing out the world. If *gc* is `t`, it will perform a garbage collection before writing the world. If *gc* is `nil`, it will not perform a GC unless it has to. To save the world, Newspace must be in Dynamic-0, so a **gc** may be invoked regardless of the *gc* option.
If *top-level* is non-`nil`(the default), the current world is saved, so that when it is booted it will perform all initializations and then invoke the top-level **%initial-loop** function. This works by exiting the system, but saving state before the exit. Therefore, the system will exit when using this option.
If *top-level* is `nil`, the current state of execution is saved. When the saved core file is booted, the saved system state is resumed, and the call to **save-world** returns `nil`. Since the booted system resumes this computation, no system initializations are performed, such as the user login. Also be aware that any files open when the save occurs will not be properly closed, and will incorrectly appear open when the core file is resumed.

Chapter 10

Networking

Top Level Common Lisp provides access to network services and the creation of network servers using TCP. All communication through the network is done using network-streams, which are essentially byte-streams. Network-streams in Top Level Common Lisp can also be used as ASCII character-streams, so functions such as `read-char`, or `read-line` can be used on them. However, no attempt is made to perform character translation if the foreign host sends characters that are not ASCII.

10.1 Connecting to Servers

network-connect *hostname portnumber* [*function*]

Connects to host *hostname* using TCP port *portnumber.* It returns a network-stream.

10.2 Creating network servers

network-server *function portnumber* [*function*]

Establishes a server for TCP port *portnumber.* When a connection is made for this service, *function* is called, with one argument being the network-stream. This function is executed in its own process. Because of limitations in Unix, the stream object cannot be used outside of this process. You should be careful to close the stream before returning from *function.* If a server for *portnumber* has already been defined, **network-server** will replace the function invoked for the server with *function.* The first time **network-server** is called it will create a process to service all connection requests. This process should never complete, but if it does, the next call to **network-server** will create a new process to handle server connections.

tcp-services [*variable*]

A list of all services that have been defined via **network-server.**

Chapter 11

Memory Management

11.1 Garbage Collection

`gc &Optional` *notify verbose* *[function]*

Invokes the garbage collector, reclaiming address space occupied by inaccessible Lisp objects. *notify* indicates whether a message should be printing to indicate that the system is garbage collecting. It defaults to the value of ***gc-notify***. If *verbose* is `non-nil`, other information may be printed about the garbage collection activity. It defaults to the value ***gc-verbose***.

full-gc *[function]*

Garbage collects the entire Lisp world, reverting any static areas back to dynamic before the collection. After the collection is completed, all remaining Lisp objects are considered static, and will not be moved or collected until another **full-gc**. When this returns, newspace will be dynamic-0, which is required for a **save-world**. **Full-gc** may actually require three garbage collections to perform its job.

gc-notify *[variable]*

If `non-nil`, when GC is invoked a message will be printed. Defaults to `t`.

gc-verbose *[variable]*

If `non-nil`, GC will print out stages of gc activity. Defaults to `nil`.

gc-count *[variable]*

This variable is incremented every time a GC is performed.

11.2 Resources

This section describe the *resource* facility in Top Level Common Lisp. Resources give a programmer explicit control over allocation and deallocation of Lisp object.

11.2.1 Introduction to Resources

One of the many features of Lisp is its ability to recycle address space. You can simply allocate objects, use them, and let the garbage collector reclaim their address space when they are no longer accessible. In addition to speeding program development, this capability is essential for complex programs in which the usage of objects is not well defined. When an object does have a well-defined usage and lifetime, it may be faster in some cases to do your own explicit memory management. A resource is appropriate when the rate of allocation-deallocation is fairly high. In such cases a resource will let you continually reuse the same object instead of quickly creating a large amount of garbage. For example, since **format** is typically used heavily, it uses a resource of string-output-streams to avoid creating a new one that will immediately become garbage when the call returns. Typically, serial programs deal with this by creating a single copy that is reused for each call, but this creates non-reentrant code that cannot be used for a multiprocessor system.

Another case occurs when an object is large. Even if its rate of allocation-deallocation is not high, creating a small number of objects will still consume a large amount of address space. Bitmaps are an example of this type of usage. While they may not be allocated-deallocated at a high rate, their large size consumes lots of address space, which will cause frequent garbage collections if new bitmaps are always created.

A third case occurs when the time needed to create an object is very long compared with its initialization and use. Tasks are an example of this type. They can take a long time to create, but may not be used for a long period of time. Thus, it is much faster to keep them around to reuse than to create new ones every time.

The drawback to using resources is that objects on the free-list take up address space. Following a burst of activity that creates and frees many objects, a program may move to a new "phase" that will never use those saved objects. The address space they occupy will never be reclaimed, but the objects are essentially garbage. The function **clear-resource** can be used to clear out free objects from a resource if you know you won't be needing them soon.

11.2.2 Defresource

The macro **defresource** can be used to define a named resource. The macro **with-resource** will execute the macro body with an object allocated. The **defresource** macro also allows you to specify your own "wrapper" macro that will be more efficient than the general **with-resource** form. In particular, since Top Level Common Lisp can have many processes attempting to access a resource simultaneously, access to the resource must be protected by locks. The wrapper macro defined by **defresource** is built to minimize the time spent locking out other processes from the resource.

The functions **allocate-resource** and **deallocate-resource** can also be used to explicitly allocate and deallocate data structures, but the **with-resource** wrapper macros should be used when possible. They allow an object to be "stack" allocated, whereas explicit allocate-deallocate functions preclude that possibility since the dynamic extent of an allocated object is not known.

defresource *name parameters* `&Optional` *docstring* `&Key` [*macro*]

Defines the resource *name*. *Docstring* is set as the **resource** documentation type that is accessible by the function `documentation`.
Parameters should be a lambda-list that specifies the parameters for allocating and manipulating a resource object.
The keyword arguments are as follows:

`:with-resource`	Defines a specialized *with-resource* macro that will evaluate its body with a resource object allocated. See the documentation for the general **with-resource** macro, below.
`:constructor`	This required argument is a function (symbol) that is called with the supplied *parameters* when a new object is needed. It can also be a *form* that is evaluated to produce the new object. The *form* can refer to *parameters*.
`:initializer`	If this is supplied, it should be a function (symbol) that is called to initialize the contents of the object. It is passed to the object to be initialized followed by any parameters. It can also be a *form* that is evaluated to produce the new object. The object is referred to by the symbol `RESOURCE-OBJECT` in the current package. The *form* can also refer to any *parameters*.
`:matcher`	If this is supplied, it should be a function (symbol) that is called to determine whether a free object meets the requirements of the supplied *parameters*. If it returns non-nil, the object is used. This function is called without the resource being locked. It can be called more than once on the same object. It can also be a *form* that is evaluated to test the object. The object is referred to by the symbol `RESOURCE-OBJECT` in the current package. The *form* can also refer to any *parameters*.
`:deallocator`	If this is supplied, it is a function or form that is called or evaluated to clean up any pointers in the object after it has been deallocated. It is called with one argument, which is the object to be deallocated.

with-resource *(var resource-name* `&Rest` *parameters)* `&Body` *body* [*macro*]

Evaluates *body* with *var* bound to an object allocated from *resource-name* with the optional *parameters*. It is more efficent to supply a `with-resource` argument to **defresource** to define a specialized version of this macro. The specialized version will put inline all the code to do allocation and initialization.

The following is an example using string-output-streams:

```
;; Define a resource of string-output-streams.
(DefResource string-output-stream-resource ()
  "String output streams"
  :constructor
    make-string-output-stream
  :initializer
    (progn
      (setf (string-output-stream-index RESOURCE-OBJECT) 0)
      RESOURCE-OBJECT) ;; Return the resource object.
  :WITH-RESOURCE With-String-Output-Stream
  )

;; An example use:

(Defun binary-string-of-number (n)
  (With-String-Output-Stream (string-stream)
    (let ((*print-base* 2.))
      (princ n string-stream)
      (get-output-stream-string string-stream))))
```

The following functions are also provided:

allocate-resource *resource-name* `&Rest` *parameters* [*function*]

Allocates an object from the free list of *resource-name* with the optional *parameters.*

deallocate-resource *resource-name object* [*function*]

Returns *object* back to the free list for the resource *resource-name.*

clear-resource *resource-name* [*function*]

Clears all objects currently stored on the free list for the resource *resource-name.*

find-resource *resource* `&Optional` *error-p error-value* [*function*]

Returns `non-nil` if *resource* is a resource or names a defined resource. If *error-p* is `nil`, then *error-value* is returned if *resource* is not the name of a resource.

all-resources [*variable*]

A lists of all resources that have been defined using **defresource.**

Chapter 12

Metering and Profiling

Top Level Common Lisp provides a simple mechanism to help determine where your Lisp program is spending its time, and thus where optimization activity should be focused. The meter facility currently provides information that is statistical in nature. It samples the stack-state at specified intervals, recording which functions were running at each sample. Functions that are called often or that run for longer periods of time are more likely to appear in the samples. Functions that are called less frequently or that execute relatively quickly will appear less often. Thus, the frequency distribution of function calls provided by the meter facility provides information on which functions are important to optimize for improvements in execution-time performance.

The time between sampling has a critical impact on the validity and performance of the meter information provided. The shorter the interval time, the more accurate the results will be, but there is a corresponding increase in run time to perform the metering. However, even with the shortest possible interval, there are many factors that affect the information provided.

To use the meter facility most effectively, you should perform numerous meter runs. By comparing the results from each run, a more accurate picture of run-time behavior is possible.

The meter facility uses global data structures, and thus can be used only for recording information in a single process. Thus, meter will not work when using multiple processes.

12.1 Meter

meter-on `&Key` *clear depth micro seconds* [*function*]

Begins metering of function calls. If *clear* is `nil`(the default), then new call information will be combined with any previous metering info. If `non-nil`, then previous meter information is cleared.
depth indicates the depth of the function calls that are recorded. With a larger depth, more calls are seen, but the amount of information is less precise. A call that exists on the stack that repeatedly calls a number of other functions will be given more weight

than the functions it calls. Just the opposite occurs with a smaller depth. "Outer-loop" calls are given less weight, with more weight provided to "inner-loop" calls. The default value is four.
micro indicates the number of microseconds between samplings. The system will round up to the finest grain size supported, which is typically 1-10 milliseconds in Unix. The default is 100000.
seconds indicates the number of seconds between samplings. This value is added to the microsecond value to produce the total sampling interval. Intervals of more than one second are not likely to be very useful, but in very long-running programs, reducing the sampling time reduces possible bignum allocations for data collection.

meter-off *[function]*

Stops recording of meter info.

show-meters `&Optional` *top-n* *[function]*

Shows the current information about function call frequencies. The *top-n* argument will restrict the display to the *top-n* most frequently called functions. It defaults to ten.

12.2 System Timers

The Sequent and Encore systems have a microsecond clock that can be accessed with very low overhead to provide fine-grained, real-time timing measurements. The Common Lisp function `time` reports on the time required to execute a Lisp form, and includes microsecond time. This works well for "top level" calls, but is not sufficient for writing your own timing code. The Common Lisp function `get-internal-real-time` provides this functionality, but has a relatively large overhead. Top Level Common Lisp provides direct access to the microsecond clock with the function **unix-get-uclock**, but using this interface forces garbage to be created since timer values are 32-bit quantities (bignums) and memory must be allocated to store them accurately.

Top Level Common Lisp provides a more efficient mechanism that reduces bignum consing by maintaining a small number (100) of statically allocated slots to hold timer values. This interface returns only elapsed time, so allocation of a bignum can be avoided when the elapsed time is shorter than 8 seconds.

These timer values are global, so be careful when using timers for parallel programs.

start-timer *nth* [*function*]

Records a start-time in the *nth* system timer. There are 100 timer slots, 0 to 99. It returns the clock value as a fixnum, which retains the lowest 23 bits only. In general, the return value should not be used.

end-timer *nth* [*function*]

Returns the elapsed time for the *nth* system timer since the last call to **start-timer**. It handles timer "wrap", but this works only if the total elapsed time is less than about 42 hours. A fixnum can hold a value of only about 8 million, so it will return a bignum for an elapsed time of more than 8 seconds.
Calling **end-timer** does not reset the start-time. Thus repeatedly calling **end-timer** will yield multiple elapsed times.

unix-get-uclock [*function*]

Returns the current unsigned 32-bit value of the uclock.

Chapter 13

Parallel Programming

13.1 Introduction

Top Level Common Lisp includes constructs for parallel programming. The system supports efficient use of multiprocessor computers by providing multiple grain sizes of parallel operators. It defines a small set of primitives that provide a powerful and natural way of expressing parallelism in Lisp programs. By using the extensibility of Lisp, higher-level or special-purpose parallel facilities can be defined using the primitives provided in Top Level Common Lisp.

13.2 Programming Using Parallelism

Solving a problem using parallelism requires that the problem be partitioned into smaller units that can be executed in parallel. These units must be provided with the neccessary resources in order to complete their work before being combined back together to produce the final solution to the problem. Some systems can force the programmer to create all these components, i.e., specifying the break-up of activities, which resources must be available to each, and providing the mechanism for combining their results. This approach is taken in languages that provide the fork-join construct. For some systems, it is performed entirely by the compiler, as when FORTRAN do-loops are automatically partitioned for parallel execution.

The parallel facilities offered by Top Level Common Lisp provide for programmer-specified parallelism. Thus, a programmer must specify which pieces of code can be performed in parallel. However, Top Level Common Lisp eliminates the need to explicitly specify when and how the results of parallel computation are combined by using the *future* construct.

13.2.1 Future Objects

The future construct provides *implicit* synchronization of parallel activity by *using* the *results* of forked computation. When a forked computation is created it returns a future object that represents the forked computation. Until the computation has completed, the future object is *undetermined.* When the computation completes, the future object transforms into the value returned by the computation. The future object is now *determined* and indistinguishable from this value.

If a *strict* operation attempts to operate on a future object, it implicitly *forces* the future object to be determined, and will wait for its value before continuing. Strict operations involve looking at the data type or value of an object, such as the function **car**, which requires that its argument be a list. Non-strict operations, such as **cons**, only *reference* their arguments, but do not require them to be any particular type.

Note that the determined value of a future object can itself be another future object. A strict operation will continue to force future objects until a non-future object is finally obtained.

EQ with Future Objects

The function eq is defined to be a strict operation. An undetermined future object will be eq to itself regardless of its determined value, but in all other cases a determined value is necessary. Therefore, to maintain consistent semantics, eq will force its arguments to be determined.

force *object* [*function*]

Forces *object* to be determined and returns its value. If *object* is a future object, **force** will either wait for its value or will compute it. If *object* is not a future object, it is simply returned. **force** will return the direct representation of an object, so unsafe type-specific code can be applied to the returned object.
Note that you do not need to call this function to force a future object to be determined. Any strict operation is sufficient.

futurep *object* [*function*]

Returns `non-nil` if *object* is a future object that is not yet determined.

delay *function* `&Rest` *arguments* [*function*]

Immediately returns a future object, delaying the application of *function* to *arguments.* Note that **delay** is a function, and therefore *its* arguments *are* evaluated. *Function* will be applied to *arguments* only if a strict operation forces the future object returned to be determined.
The **delay** operator allows *lazy* or *demand-driven* evaluation.
The future object returned by **delay** has the same properties as one returned by **thread**, discussed in Section 13.3. However, it is also possible to have a delayed computation return *another* future object that represents a task or process.

An example using **delay** is the function `all-prime-numbers` below. This will create an infinite list of *all* prime numbers. However, it will only compute the numbers that are actually needed.

```
(Defun all-prime-numbers (previous)
  (let ((next (next-prime (1+ previous))))
    (cons next (delay 'all-prime-numbers next))))
```

Delay can also be used to simplify problems with "forward references". You can define the data structures that refer to results that would not be possible to compute at the time the structure was built, but would be possible after some initialization or other processing. This is typically dealt with in Lisp by using symbols and `symbol-value` to get the value at run time. The initialization code would determine the value of the symbols. By using a **delay**, you can place the code that needs to be run for the value right where the value is needed, instead of in the initialization code. This also eliminates the `symbol-value` calls at run time, so using delays is more efficient as well. This shows how future objects are useful even for serial programs.

13.2.2 Multiple Grain Sizes

To effectively use parallelism, the associated overhead must not be prohibitive. If it takes more than 2 seconds to perform two 1-second computations in parallel, there is clearly no advantage gained. To avoid this possibility, the overhead should be as small as possible when using parallel computation. However, if a pair of computations takes a long time, proportionately more overhead can be incurred to enable them to run them in parallel. This overhead can be used both to improve the performance of each parallel piece and to improve overall throughput by improving the utilization of available resources.

Top Level Common Lisp currently provides three different levels of parallel operators, each having different characteristics and overheads. Both fine as well as large grained parallelism can be expressed, allowing maximum utilization of possible parallelism in an application. The following sections discuss the characteristics and overheads for each parallel operator available in Top Level Common Lisp.

13.3 Threads

The finest grain-size of parallelism in TopCL is the *thread*. It is intended to be a computation that will be performed quickly, and thus any time spent dealing with it should be as small as possible. Threads consist only of a function and its arguments; they do not have a control or binding stack of their own.

Threads can be either *queued* or *opportunistic*. The first argument to the function **thread** indicates whether **thread** *must* immediately return a future object. When its value is `non-nil`, a future object is always returned, and the computation may be placed on a queue for execution. If its value is `nil`, then the system has the option of immediately applying the function to its arguments and thereby not actually perform the computation in parallel.

The short-lived property of threads also defines the strategy used when another computation forces their future objects. A threaded future object can be in one of two states. Either a task is currently computing its value, or it has not yet received any computing resources. In the first case, any computation which forces its value will simply "spin" until the value is produced, which is the lowest overhead mechanism possible. This strategy is used since it is assumed that no useful computation can occur between the time of reference and the time of value delivery. In the second case the referencing computation will perform the execution itself, changing the future object's state into the first case.

As with any function, a thread function can bind dynamic variables. However, since any task can potentially perform the execution of a thread, references to dynamic variables that are *not* bound by the thread function are undefined. In particular, there is no guarantee that a reference to an unbound dynamic variable in a thread will access its global value.

Top Level Common Lisp has a number of "needle" tasks that will execute all threads. When there are not any needles running, a **thread** operates as either a **delay** or a simple function call. If a thread is queued, it operates the same as a **delay**, otherwise **thread** will call the function immediately. The function **needles** will either create or remove needles.

Note there is always a "root" process running in TopCL, so the total number of processes executing threads is the number of needles running plus at least one. Therefore, to evaluate timings against the number of processors, the number of needles created should be one less. For example, running a program with 7 needles will actually be using 8 processes.

It is possible to create more needles than actual physical processors in a machine. However, this results in severe performance degradation since the system assumes a needle is actually running and making progress computing any threads assigned to it.

needles `&Optional` *n* *[function]*

Adjusts the number of needles running to be *n* and returns *n*. If *n* is not supplied, it returns the number of needles currently running. If *n* indicates a reduction in needles, **needles** will wait until the needles being removed become idle before returning.

thread *queue? function* `&Rest` *arguments* *[function]*

The *queue?* argument determines whether the call to **thread** must return immediately or not. If *queue?* is `non-nil`, a future object is immediately returned that will eventually receive the value of *function* applied to *arguments*. If *queue?* is `nil`, **thread** will call the function immediately if no needle is free, and a future object will not be returned. The computing agent that actually performs the thread computation is arbitrary and therefore the dynamic binding context in which *function* executes is not defined. References to dynamic variables that are not bound by *function* are not guaranteed to refer to their global values.
The overhead to create a thread is about 4 to 12 function calls.

Waiting for Threads

While threads are a low-overhead mechanism, the time spent waiting for their values by spinning can be wasteful. In particular, a task waiting for a thread is not able to apply its computing resources towards completion of the computation it is waiting for. However, this is only a problem if the computation being waited for can actually use the computing resources. In these cases, a **task** may be a more appropriate parallel operator.

13.4 Tasks

The medium grain form of parallelism is the *task*. It has its own control-stack, binding-stack, as well as other information. Because a task is assumed to be a computation of some duration, a call to **Task** will *always* immediately return a future object and be executed in parallel. It is undefined exactly when the execution will actually begin, but it will eventually be executed. This property is important in some situations where deadlock would occur if two activities did not proceed in parallel.

A task's longer-lived property also defines the strategy used when its future object is forced. Since it could be a fair amount of time before the value is delivered, waiting for its value by spinning would be wasting resources. Instead, a task will *cycle* until the value is delivered. A task cycles by returning control back to the task scheduler, allowing other tasks to run. Each time it is resumed by the scheduler it will again check for a delivered value. Once the value has been delivered it continues with its computation. This mechanism allows tasks to be managed using a very simple scheduling algorithm, making their use more efficient.

Because a task maintains its own binding stack, references to dynamic variables that are *not* bound by the task are defined to access the global value of the variable.

Tasks provide the basic capability for more than one flow of control. With their own control stack and binding stack, their control flow can be suspended and switched to another task. They can also be executed in parallel.

The task scheduler in Lisp has a number of "worker" tasks that will execute all scheduled tasks. Similiar to the function **needles**, the function **workers** will adjust the number of workers able to run tasks. At least one worker must be running before the function **task** can be called.

workers `&Optional` *n* [*function*]

Adjusts the number of workers running to be *n* and returns *n*. If *n* not supplied, it returns the number of workers currently running. If *n* indicates a reduction in workers, **workers** will wait until the workers become idle before returning. **Workers** will fail to remove workers if there are tasks queued locally to a worker being removed, or if *n* is zero and the scheduling queue is not empty.

The function **task** is used to create a task that will apply a function to its arguments.

task *function* `&Rest` *arguments* [*function*]

task immediately returns a future object. The result of *function* applied to *arguments* becomes the determined value of the future object.

The computational environment in which a task executes is defined to be the global environment. Therefore, references to dynamic variables that are not bound by *function* are references to their global values.

The overhead to create a task is about 300 function calls.

current-task *[function]*

Returns the task that is currently executing.

next-task *[function]*

Suspends control of the current task, and returns control to the scheduler, giving other tasks a chance to run.

show-tasks *[function]*

Shows the state of all active tasks.

13.4.1 Task Control Stacks

The following variables can be used to modify how control stacks are allocated and how stack overflow is handled.

control-stack-words *[variable]*

Specifies the size, in words, of control stacks allocated by **task**. It must be at least 500. Defaults to 1000.

grow-control-stack *[variable]*

Specifies the number of words to extend a control stack that overflows. It must be at least 500. Defaults to 1000.

control-stack-limit *[variable]*

Specifies the number of words to extend a control stack before a stack overflow error is signaled. Defaults to 3000.

stack-room *[function]*

Returns the number of free stack words remaining in the current stack.

13.4.2 Task Waiting

wait-until *form* `&Optional` *why* [*macro*]

Waits until *form* returns `non-nil` and returns the `non-nil` value. If *why* is supplied it should be a string or symbol indicating why the task is waiting. Defaults to "Wait".

task-wait *why function* `&Rest` *arguments* [*function*]

Task-wait is a functional version of **wait-until**. It waits until *function* applied to *arguments* returns `non-nil`, and returns that value.

13.4.3 Other Task Functions

This section documents other system functions that may be of interest. In general, only the function **task** should be used to create tasks. These function should be used with caution, since many do not check their arguments for validity to make them as fast as possible. They will cause system failure if used improperly. They are documented here for experimental purposes, and may be changed or renamed in future system releases.

create-task *function* `&Rest` *arguments* [*function*]

Creates a task that will apply *function* to *arguments* when executed.

taskp *task* [*function*]

Returns `non-nil` if *task* is a task.

next-task [*function*]

Switches control over to the task that last switched control to the current one. This is usually a worker task.

switch-task *task* [*function*]

Switches control over to *task*. When this task calls **next-task** or its initial function returns, this call will return.

task-caller [*function*]

Returns which task switched execution to the current task. It returns which task will be resumed if the current task calls **next-task**. If this returns `nil`, then the task is a *process*, and calling **next-task** will simply return.

task-function *task* [*function*]

Returns the initial function for *task*.

task-arglist *task* [*function*]

Returns the argument list for *task*'s initial function.

task-state *task* [*function*]

Returns the current state of the task. Note that this state may have changed by the time this function returns. It can be in one of the following states:

`INITIALIZED`	The initial function is set up, but has not yet been executed.
`FORKING`	It is getting ready to run in its own Unix process.
a fixnum	It is running on the Unix process with the specified PID.
`NEXT-TASK`	It is not currently executing.
`DEALLOCATED`	The initial function has returned, and it is about to become `COMPLETED`. Its contents are garbage and should not be examined.
`COMPLETED`	The initial function has returned. Its contents are garbage and should not be examined.

13.5 Processes

In the present implementation, a Process is essentially a task that is executed by its own Unix process, and is therefore managed by the Unix operating system. By running in its own Unix process, a process can perform callouts to C code or make system calls without preventing other activities from running. This allows processes to deal with signals, since they can safely perform `wait` calls in order to wait for a signal. System `wait` calls are used when a process references a future object that is owned by a task. It will put itself on the waiting list for the value, and then wait for a wake-up signal. Processes also maintain their own allocation pointers, so they do not need to inhibit interrupts while performing allocations.

These features come at a hefty price, however. Creating a Unix process is a very expensive operation. The overhead to create a process is about 40,000 function calls, which is two orders of magnitude more than a task.

process *function* `&Rest` *arguments* [*function*]

Process immediately returns a future object. The result of *function* applied to *arguments* becomes the determined value of the future object.
The computational environment in which a process executes is defined to be the global environment. Therefore, references to dynamic variables that are not bound by *function* are references to their global values.

fork-task *task* [*function*]

Forks a new Unix process to execute *task*. *Task* cannot be running or managed by the task scheduler. Use **process** to create a process. This function may go away in future releases.

13.6 Future Groups

When writing parallel algorithms it is often necessary to wait for the completion of parallel activity before moving on to the next step of the algorithm. This style of parallelism can be implemented using future objects in a number of different ways. One common method is to use recursion to fork off the parallelism, which then waits for the completion of the activity as the recursion unwinds. For example, the function **map-and-wait**, below, will fork a thread to call a function (which presumably has side effects) on each item in a list, and will not return until each thread has completed:

```
(Defun Map-and-wait (function list)
  (If (null list) nil
      (let ((future (Thread t function (car list))))
        (map-and-wait function (cdr list))
        (force future))))
```

The **thread** call will create the parallelism for the current element of the list, saving the pending value in the local variable. Once the end of the list is found and **map-and-wait** returns, **force** is called on each future created as the recursion unwinds, This insures all the computation is completed before the initial call to **map-and-wait** returns.

This mechanism works well when you have direct access to all future objects created. However, there are other cases in which you need to continue computing after forking without having to return back to the fork-point to wait for the completion. In this situation the future objects must be stored somewhere so they can be referred to later. Future groups provide this functionality.

with-future-group *tag* &Body *body* [*macro*]

Establishes a future group identified by *tag*, which is evaluated, and then executes *body*. It will wait until all members of this group complete their computation and then return the value returned by *body*. During the evaluation of *body*, the macro **group** can be called to add members to the *tag* group. A **group** call is *not* restricted to the lexical environment of *body*. It must only be within the dynamic scope of the **with-future-group**.

group *tag operator* &Rest *arguments* [*macro*]

Adds a member to the future group identified by *tag*. *Operator* is either **thread**, **task**, or **process**. *Arguments* are the arguments for the fork operator, which should be a function and the arguments for this function. This will return a future object whose determined value will be the result of the function applied to its arguments.

Chapter 14

Locks and Atomic Operations

14.1 Locks

The future mechanism provides all the synchronization necessary for programs that do not have side effects. When side effects are present it is neccessary to provide locking primitives to enable atomic and other synchronization operations. This chapter presents the locking functions in Top Level Common Lisp and other related facilites.

14.1.1 Using Future Objects as Locks

The synchronization provided by future objects can also be used as a type of locking mechanism for programs with side effects. For example, if a set of tasks could not continue until some activity with side-effects is completed, this activity can be represented by a future object that all other tasks force to be determined before continuing. The actual value of the future object is ignored; only the synchronization properties are used.

14.1.2 Symbol Locks

Every symbol in Top Level Common Lisp has an associated lock state. It can be either locked or unlocked. Three simple lock primitives are provided for lock states: **symbol-lock**, **symbol-getlock**, and **symbol-unlock**.

symbol-lock *symbol* [*function*]

Returns `non-nil` if it acquired *symbol*'s lock, or `nil` if it was already locked.

symbol-getlock *symbol* [*function*]

Does not return until it acquires *symbol*'s lock. **Symbol-getlock** is implemented by spinning until the lock is available, and therefore should be used *only* when you are sure the lock will become available almost immediately. It is optimized to reduce contention when repeated requests for a lock are being performed, and should be used instead of a Lisp-level loop using **symbol-lock**.

symbol-unlock *symbol* [*function*]

Unlocks the lock associated with *symbol*. It returns *symbol*.

with-symbol-lock *(symbol* `&Rest` *options)* `&Body` *body* [*macro*]

Executes *body* with the lock for *symbol* locked. The *symbol* argument is *not* evaluated. It returns the values of *body*. The possible *options* are as follows:

`:sleep` *seconds*	If the `:sleep` option is supplied, it must be followed by the number of *seconds* to sleep between failed attempts to get the lock.
`:next-task`	If the `:next-task` option is supplied, the current task will call **next-task** between failed attempts to get the lock.
`:eval`	Causes the *symbol* argument to be evaluated.

If `:sleep` or `:next-task` is not supplied, **with-symbol-lock** will use **symbol-getlock** to acquire the lock.

14.1.3 Atomic Operations Using Symbol Locks

Top Level Common Lisp uses certain conventions when using symbols as locks.

The macro **defun-atomic** will define a function that will have only one copy executing at any one time. It is implemented by first acquiring the lock for the name of the function being defined before executing the body of the function.

defun-atomic *name lambda-list* `&Body` *body* [*macro*]

Defines the function *name* as with `defun`, but guarantees that all calls to function will be executed serially.

A number of atomic macros are defined that perform read-modify-write operations atomically. When the *place* argument is a symbol, the operations acquire the symbol's lock before performing the operation. If the *place* argument is a generalized variable, these operations lock on the symbol **atomic**, which is shared by *all* atomic operations performed on generalized variables. This can cause unnecessary serialization of parallel programs.

An atomic operation CANNOT execute another atomic operation that must acquire the same lock it already owns. Therefore EMBEDDED ATOMIC OPERATIONS WILL RESULT IN DEADLOCK.

It should also be emphasized that atomic operations do not prevent non-atomic operations from reading or writing the *place* argument to these operations.

atomic-push *value place* [*macro*]

Pushes *value* onto *place* as an atomic operation.

atomic-pop *place* [*macro*]

Pops an object from *place* as an atomic operation.

atomic-incf *place* `&Optional` *delta* [*macro*]

Increments *place* by *delta* as an atomic operation.

atomic-decf *place* `&Optional` *delta* [*macro*]

Decrements *place* by *delta* as an atomic operation.

define-atomic-modify-macro *name lambda-list* `&Body` *body* [*macro*]

Defines the read-modify-write macro *name* as with `define-modify-macro`, but guarantees that only one atomic operation at a time will be performed on the *place* argument to the macro.

14.1.4 General Locks

Associating locks with symbols has a number of advantages.

- Symbols are interned, and thus references to a symbol are references to the same shared object.
- Symbols can be referenced directly as constants. Using a separate lock object requires they be referenced indirectly through a global variable or some other accessor.
- Symbols Reduce the need to allocate additional memory when using locks since symbols are already present.

Associating locks with symbols has some disadvantages. It can force an implemention to allocate more space for each symbol, thus wasting memory for the large number of symbols that are never used as locks. However, symbols in Top Level Common Lisp have unused bits that otherwise would be unused.

The more important difficulty, however, is using symbols for locks violates the abstraction of a lock as a distinct object with its own behavior. Therefore Top Level Common Lisp also provides general lock objects. They are currently implemented using symbols, but this may change in future releases.

make-lock [*function*]

Returns a lock object that is initially unlocked.

lock *lock* *[function]*

Returns `non-nil` if it acquired the lock, or `nil` if it was already locked.

getlock *lock* *[function]*

Does not return until it acquires *lock*. This is implemented by spinning until the lock is available, and therefore should be used *only* when you are sure the lock will become available almost immediately. It is optimized to reduce contention when repeated requests for a lock are being performed, and should be used instead of a Lisp-level loop using **lock**.

unlock *lock* *[function]*

Unlocks the lock. It returns *lock*.

lock-owner *lock* *[function]*

Returns the task that owns *lock* if it is locked, or returns `nil` if the lock is not currently locked. This represents only the state of the lock at the moment this call is made, which can change before this function returns. In particular, this function can return `nil` for a locked lock.

with-lock *(lock* `&Rest` *options)* `&Body` *body* *[macro]*

Executes *body* with the *lock* locked. The *lock* argument is evaluated. It returns the values of *body*. The possible *options* are as follows:

`:sleep` *seconds*	If the `:sleep` option is supplied, it must be followed by the number of *seconds* to sleep between failed attempts to get the lock.
`:relock`	If the `:relock` is supplied, a task may relock a lock it already owns. In this case the lock will not be unlocked when this form returns.
`:next-task`	If the `:next-task` option is supplied, the current task will call **next-task** between failed attempts to get the lock.

If `:sleep` or `:next-task` is not supplied, **with-lock** will use **getlock** to acquire the lock.

Chapter 15

Foreign Function Interface

NOTE: The foreign interface is not completed for the Encore Multimax implementation. System calls can still be defined and invoked, but user-defined C code cannot be loaded into the Lisp system in Release 1.0. NOTE: You must compile all foreign function definitions and foreign defstructs.

Top Level Common Lisp allows routines written in C to be called from within Lisp. Data structures can be passed by reference to these routines, allowing data sharing between C and Lisp. Both Lisp objects and C-based data structures can be shared between the different languages. However, C objects that are created dynamically using `malloc` cannot be passed back to Lisp since they will not be within Lisp's address space. C routines and data structures are referred to as "foreign" since they can be understood only through an interface with a "translator".

When using foreign routines care must be taken to ensure that the Lisp system is not corrupted. Without the type-checking mechanisms provided by Lisp, foreign code can read and write to the entire Lisp address space without any safeguards. Therefore, bugs in foreign routines can damage critical system data structures and cause the system to immediately crash; or even worse leave "time bombs" in the system that only later will result in a system crash. Be careful.

A foreign routine must be both loaded and defined in order to be called. The **load-foreign** function will read the foreign function into memory. The **defun-foreign** function specifies the necessary argument and return information that must be present before the foreign call.

15.1 Defining A Foreign Function

Defun-foreign is used to supply the information needed to perform a foreign call. It defines a Lisp function that is called to invoke the foreign function. The value returned by the foreign function will become the Lisp function's return value. When a foreign function has "out" arguments (discussed in detail below), the Lisp function defined by **defun-foreign** will return multiple values. **Defmacro-foreign** can be used to define a macro that will side-effect the actual "out" arguments in the call form.

defun-foreign *name-and-options [documentation]* `&Rest` *arg-specs* [*macro*]

Defun-foreign defines a Lisp function that will invoke a foreign function. The defined Lisp function will convert the Lisp arguments into foreign data types before the call. Upon return, the foreign return value and any "out" parameters are converted back to Lisp objects.
When type information is required, the Lisp type must be specified. If the corresponding foreign type is not specified, it will default to a reasonable type. Table 15.1 indicates the foreign type defaults that correspond to specified Lisp types.
Name-and-options specifies the name of the foreign function, the return information, and where the foreign routine is located.

Name-and-options has the following form:

function-name |
(*function-name result-type*) |
(*function-name* `&Key` *result-type entry-name object-file*)

Function-name identifies the name of the Lisp function to be defined. *Result-type* specifies the data type of the return value. It can be either a symbol, specifying only the Lisp result type, or a list of the form (*lisp-result-type foreign-result-type*).
The *entry-name* will default to the lowercase name of *function-name* with "_" prepended. The *object-file* defaults to the value of ***foreign-path***. *Entry-name* must be an entry in the symbol table of the specified object file.
The *arg-specs* define each argument, its name and per-argument call-discipline and conversion information. By specifying each argument, the number of arguments is implicitly defined.
Each *arg-spec* has the following form:

(*argument-name* `&Optional` *type-spec mechanism*)

The *type-spec* is either the *lisp-type* or (*lisp-type foreign-type*). Note that a type specification must be supplied if the call mechanism differs from the default specified in Table 15.1.
Argument-name names the argument, and is used for internal purposes. *Lisp-type* specifies the type of the Lisp actual parameter. *Foreign-type* specifies the type of the foreign parameter. *Mechanism* specifies the call-discipline and argument conversion. It defaults to `:in` for value-result parameters, and `:ref` for reference parameters.

defmacro-foreign *name-and-options [documentation]* `&Rest` *arg-specs* [*macro*]

Defmacro-foreign is analogous to **defun-foreign**, with a couple of exceptions. First, a Lisp macro is defined instead of a function. Second, all out-type parameters (both `:out` and `:inout`) will be side-effected. Thus, all actual out-type parameters must be forms acceptable to `setf`.

15.1.1 Calling Mechanism

The *mechanism* parameter of **defun-foreign** specifies the calling convention used with each parameter.

Lisp Data Type	*Foreign Data Type*	*Calling Convention*
`integer`	`int, unsigned, double`	`:in,:inout,:out`
`fixnum`	`int, unsigned`	`:in,:inout,:out`
`single-float`	`double`	`:in,:inout,:out`
`short-float`	`double`	`:in,:inout,:out`
`string`	`*char`	`:ref`
`vector`	`*int`	`:ref`
`array`	`*int`	`:ref`
`foreign`	`*int`	`:ref`

Table 15.1: Foreign Function Argument Attributes

15.1.1.1 Call By Value, Value-Result and Result

Many Lisp data types do not have identical machine formats as their corresponding C data types. For these data types, the foreign function interface provides call by value, call by result, and call by value-result. The corresponding mechanism specifiers are `:in`, `:out`, and `:inout` respectively. Any argument that has `:inout` or `:out` as its mechanism will be destructively modified before returning. Any `:out` or `:inout` parameter must be a form acceptable to `setf`.

All `:in` and `:inout` parameters are converted from their Lisp representation to their C representation when passed to the C function. All `:out` and `:inout` parameters are converted from their C representation back to their Lisp representation when returning from C. Table 15.1 enumerates the possible combinations of C and Lisp data types.

When there are `:out` or `:inout` arguments, the function returns them as multiple values; the first value is the return value, and the rest of them are returned in *reverse* order, so the *last* `:out` argument is the second value, etc. This happens because C takes arguments backwards.

15.1.1.2 Call by Reference

Lisp data types that do have identical machine formats as the corresponding C data types are passed by reference. The *mechanism* argument is `:ref`. Call-by-reference data types are not required to be `setf`'able.

Table 15.1 summarizes the allowable combinations of data type and calling convention specifications. The first entry in the *Foreign Data Type* column is the default. For example, if one specifies *lisp-type* to be `integer`, the *foreign-type* will default to `int`.

15.1.2 Data Types and Parameter Passing

When passing parameters to and returning them from routines, the C compiler treats a `short` the same as an `int`, and a `float` as identical to a `double`. The data types `int` and `long` are equivalent. For instance, if one passes a `float` to a C function, the float will be converted to a `double` before being inserted into the activation record. Also, a `long` is identical to a C `int`. The syntax of **defun-foreign** allows `short` to be a synonym for `int`, and `float` to be a synonym for `double`. These synonyms will not change the semantics of the foreign call.

The *lisp-type* `integer` represents either a bignum or a fixnum. A *lisp-type* `fixnum` represents a fixnum. The Dynix C `int` represents values from -2^{31} to $2^{31} - 1$, while an `unsigned int` represents integers from 0 to $2^{32}-1$. In Top Level Common Lisp, the integers from -2^{23} to $2^{23} - 1$ are represented as `fixnum`; integers out of this range are bignums.

15.1.3 Run-time Range and Type Checking

The variable ***foreign-runtime-safety*** specifies run-time range and type checking. If ***foreign-runtime-safety*** is not 0, each actual parameter will be checked for a type match with the type specified in the **defun-foreign**.

Run-time range checking will verify that the Lisp type that was passed in to the C routine is within the range of the foreign type. The conversion of bignums to a C int or a C unsigned, and of Lisp integers to a C short, are the only cases in which this run-time range checking is necessary. When range checking is turned off, a bignum larger than 32 bits will have its lowest 32 bits passed to the C function.

If the *lisp-type* is bignum and the *unix-type* is double, range checking will not be performed. Range checking is not performed on floating point values since all floats representable in Lisp are representable in C. Range checking is not performed on any call by Reference parameters; only an address is passed.

If ***foreign-runtime-safety*** is greater than 0, the called foreign function will check to verify that the foreign function is both defined and loaded.

15.1.4 Overflow Within C

Arithmetic operations causing integer overflows are not flagged as exceptions by Unix. The resulting value may either overflow into the sign bit, yielding a large negative value, or overflow out of the 32-bit word, yielding 0 as the result. For example, a C routine containing the expression $x = 1024 * 1024 * 1024 * 4$; will return 0.

15.2 Loading Foreign Functions

The foreign loading mechanism performs the actual reading of a foreign function into the address space of the Lisp process.

The syntax of foreign loading is as follows:

load-foreign *object-file* &Key *link-entries link-options reload temp-file boot-image* [*function*]

Load-foreign will load *object-file* into the Lisp address space if it hasn't been loaded already. If the keyword argument *reload* is non-nil, it forces a reloading of the object-file into memory, even if the file has already been loaded.

It will look up the entry points for all entries specified by the *link-entries* argument. If *link-entries* is nil, all foreign routines defined in Lisp as located in this object file are linked.

Note that the entire object file must be loaded, whether or not all the entries are linked.

15.3 A Simple Example

In this example, we define a C function that will multiply two double-floats together. The following C function implements this:

```
double cmultiply(arg1,arg2)
double arg1; double arg2;
  { return(arg1 * arg2);}
```

We use the -c option to obtain an object file.

```
% cc -c cfuncts.c
```

We must define, compile, and load the foreign function prior to a call. In the file cmult.lisp we have:

```
(defun-foreign (cmultiply :RESULT-TYPE single-float
                          :OBJECT-FILE "cfuncts.o"
                          :ENTRY-NAME "_cmultiply")
  "CMULTIPLY ARG1 ARG2 multiply two floats; returning a float."
  (arg1 single-float :in)
  (arg2 single-float :in))
```

The **defun-foreign** will define the function cmultiply,. It specifies the entry _cmultiply in the object file cfuncts.o. Table 15.1 shows that by specifying the *lisp-result-type* to be single-float, the unspecified *foreign-result-type* defaults to double. The *foreign-type* of both arguments also defaults to double.

```
USER: (compile-file "cmult.lisp") ...   ;; Compile Lisp file
Error output from "cmult.lisp#>",6-Nov-1989 21:24:59
Compiled on 6-Nov-1989 21:25:21 by Compiler Version 1.0.

CMULTIPLY compiled.

Finished Compilation of "cmult.lisp#>"
0 Errors, 0 Warnings
Elapsed Time 0:00:10, run time 0:00:04.

#.(pathname "cmult.zoom")
USER: (load "cmult.zoom") ...           ;; Load Lisp file.
Loading #<File stream #.(pathname "cmult.zoom#>")>

CMULTIPLY defined.
T
USER: (load-foreign "cfile.o") ...      ;; Load the C program.
Foreign Loading #<File stream #.(pathname "cmult.o#>")>

_cmultiply linked.

T
USER: (cmultiply 365.25 24.0) ...       ;; Call the C function.
8766.0
```

The foreign function `cmultiply` is called with two Lisp single-floats. One Lisp single-float is returned.

15.4 Foreign Structures

Foreign structures facilitate interfacing to structured data of non-lisp programs. The foreign structure allows the combination of various data types into one data structure, which is passed by reference to the foreign function.

The macro defstruct-foreign is used to define the foreign structure. The user supplies a name and field descriptions. This macro translates these field descriptions into a Lisp structure.

defstruct-foreign is similar in syntax to `defstruct`, providing only the `:conc-name`, `:constructor`, and `:copier` options.

defstruct-foreign *name-and-options [documentation]* `&Rest` *arg-specs* [*macro*]

The *name-and-options* are of the following form:
(*function-name* `&Key :conc-name :constructor :copier`)
Arg-specs is a list of the form:
(*field-name initial-value specifier*)
The *field-name* provides a name for future references to this field. If the *initial-value* is `nil`, no initial value will be supplied. The *specifier* will denote both the Lisp type and the Unix type.

Name	*Lisp Description*	*Unix Description*
`:float`	`single-float`	`double`
`:string`	`string`	`*char`
`:char`	`char`	`char`
`:long`	`32-bit integer`	`long`
`:byte`	`8-bit integer`	`char`

Table 15.2: Foreign Structure Field Specifications

The *specifier* could be one of the following: An integer specifier followed by `:signed` or `:unsigned`, i.e., `:long :signed`, `:long :unsigned`, `:byte :signed`, or `:byte :unsigned`. The default sign of `:long` and `:byte` is `:signed`. The type `:float` specifies a Lisp `short-float`, and a Unix `double` (the Sequent C compiler passes all float parameters as doubles). A Lisp string is specified by `:string` *n*, where *n* is the length of the string. A character is specified by `:char`. If no specifier is supplied, it defaults to `:long`. Table 15.2 above describes the foreign structure field specifiers.

The following foreign structure illustrates the use of all the basic data types.

```
(Defstruct-Foreign Foo
  (a 0    :long :unsigned)
  (b 0    :byte :unsigned)
  (c 0    :char)
  (d nil  :string 4)
  (e nil  :float))
```

The **defstruct-foreign** of Foo is translated into the following form:

```
(defstruct (foo (:type (:foreign 18))
                (:predicate nil))
  (a 0    :type (:unsigned-integer 0 4))
  (b 0    :type (:unsigned-integer 4 5))
  (c 0    :type (:char 5 6))
  (d nil  :type (:string 6 10))
  (e nil  :type (:float 10 18)))
```

For those who wish more control over field attributes (perhaps to create overlapping fields), direct coding of the defstruct is possible.

15.5 System Calls

Those routines documented in Section II of the Unix programmer's manual may also be called from Lisp. These Unix system calls are part of the kernel, and thus are always linked into the Lisp system. Thus, the **load-foreign** call is unnecessary for system calls. Instead, one must associate a Unix system call number (an integer) with the foreign function name. System call numbers are available in `/usr/include/syscall.h`.

define-system-call *name-and-options index [documentation]* `&Rest` *arg-specs* [*macro*]

Define-system-call is similiar to **defun-foreign**, except that an index is specified instead of loader information. All system calls return a fixnum.

15.5.1 System Call Example

The following example defines a function to invoke the system call `setpriority` from Lisp. System call numbers are obtained in `/usr/include/syscall.h`.

```
;;
;; Define call to return our process id.
;;
(Define-system-call unix-getpid 20)
;;
;; Define call to change our priority.
;;
(Define-system-call unix-set-priority 96
  (which fixnum) ;; This is one of process(0), group(1), or user(2).
  (who fixnum)   ;; more specific spec for the type of priority.
  (priority fixnum)) ;; The new priority.
;;
;; This will boost the priority of the current process, and
;; all processes forked henceforth. 0 is the default priority.
;; (You must be super-user to boost your priority).
;;
(Defun hog-processors ()
  (unix-set-priority 0 (unix-getpid) -10))
```

15.5.2 Foreign Example Using Structures

The following example illustrates the use of foreign structures. The Unix system call `gettimeofday(2)` expects two structures, both containing two integers.

```
/*
    The Unix programmers manual provides the
    calling information.
*/

gettimeofday(tp,tzp)
struct timeval *tp;
struct timezone *tzp;

struct timeval {
      long tv_sec;
      long tv_usec;
}

struct timezone {
      int tz_minuteswest;
      int tz_dsttime;
}

;;;
;;; These are the corresponding Lisp definitions.
;;;

(Defstruct-foreign timeval
  (secs  0 :long :unsigned)
  (usecs 0 :long :unsigned))

(Defstruct-foreign timezone
  (mwest 0 :long :unsigned)
  (dst   0 :long :unsigned))

(Define-system-Call unix-gettimeofday 76
  (arg1 timeval ref)
  (arg2 timezone ref))

(Defun Seconds-Since-1970 ()
  (let ((timeval (make-timeval))
        (timezone (make-timezone)))
    ;; Call UNIX to fill out the structures.
    (unix-gettimeofday timeval timezone)
    ;; Return just the seconds from the timeval.
    (timeval-secs timeval)))
```