LispWorks® User Guide

Version 5.1

Copyright and Trademarks

LispWorks User Guide

Version 5.1

March 2008

Address

LispWorks Ltd
St. John’s Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839 (toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 305 468 5262

From elsewhere: +44 870 2206189

www.lispworks.com

Contents

1

Introduction

1.1 About this manual

This manual describes the features and tools available in LispWorks, and how to use them.

The first group of chapters in this book describes some of the central programming tools and features in LispWorks:

- Chapter 2, “Starting LispWorks” describes how to start LispWorks and supply command line arguments.
- Chapter 3, “The Listener” describes the read-eval-print loop (REPL) listener.
- Chapter 4, “The Debugger” describes the REPL debugger.
- Chapter 5, “The REPL Inspector” describes the REPL inspector.
- Chapter 6, “The Trace Facility”.
- Chapter 7, “The Advice Facility”.
- Chapter 8, “Dspecs: Tools for Handling Definitions” describes the naming system for Lisp definitions, and in particular how to locate these.
- Chapter 9, “Action Lists” describes how you can run code at various hook points.

- Chapter 10, “The Compiler”.
- Chapter 11, “Storage Management”, covers the use of the garbage collector.
- Chapter 12, “The Profiler”.

The next chapter, Chapter 13, “Customization of LispWorks”, explains how to perform some commonly required customizations, such as controlling start-up appearance of LispWorks.

The next group of chapters describe features of specialist interest.

- Chapter 15, “The Metaobject Protocol”.
- Chapter 16, “Multiprocessing”, including locks.
- Chapter 17, “Common Defsystem” describes how to use **`defsystem`** to combine a series of source files into a manageable project.
- Chapter 18, “The Parser Generator”.
- Chapter 19, “Dynamic Data Exchange” describes how to implement DDE functionality in your Microsoft Windows applications.
- Chapter 20, “Common SQL” explains how to use LispWorks to communicate with databases using SQL.
- Chapter 21, “User Defined Streams” provides an illustrative example showing how to define and implement your own streams.
- Chapter 22, “Socket Stream SSL interface” describes the use of Secure Sockets Layer (SSL) with socket streams.
- Chapter 23, “Internationalization” provides an overview of using international characters.
- Chapter 24, “LispWorks’ Operating Environment” explains how to find information about the Operating System and how LispWorks was started.
- Chapter 25, “64-bit LispWorks” outlines differences between 64-bit LispWorks and 32-bit LispWorks.

Please note that documentation for Graphics Ports has been moved to the *LispWorks CAPI User Guide* and *LispWorks CAPI Reference Manual.*

1.2 The LispWorks manuals

The LispWorks manual set comprises the following books:

- The *LispWorks User Guide*—this book—describes the features and tools available in LispWorks.
- The *LispWorks Reference Manual* contains detailed information on all functions, macros, variables and classes available in LispWorks, in alphabetical order.
- The *Common LispWorks User Guide* describes Common LispWorks, the user interface for LispWorks. Common LispWorks is a set of windowing tools that let you develop and test Common Lisp code more easily and quickly.
- The *LispWorks Editor User Guide* describes the keyboard commands and programming interface to the Common LispWorks editor tool.
- The *LispWorks CAPI User Guide* and the *LispWorks CAPI Reference Manual* describe the CAPI. This is a library of classes, functions, and macros for developing graphical user interfaces for your applications. The *LispWorks CAPI User Guide* is a tutorial guide to the CAPI, and the *LispWorks CAPI Reference Manual* is an in-depth reference text.
- The *LispWorks Foreign Language Interface User Guide and Reference Manual* explains how you can use C source code in applications developed using LispWorks.
- The *LispWorks Delivery User Guide* describes how you can deliver working, standalone versions of your LispWorks applications for distribution to your customers.
- *Developing Component Software with CORBA* describes how LispWorks can interoperate with other CORBA-compliant systems.
- The *LispWorks COM/Automation User Guide and Reference Manual* describes a toolkit for using Microsoft COM and Automation in LispWorks for Windows.
- The *LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual* describes APIs for interfacing to Objective-C and Cocoa in LispWorks for Macintosh.

- The *KnowledgeWorks and Prolog User Guide* describes the LispWorks toolkit for building knowledge-based systems. Common Prolog is a logic programming system written in Common Lisp.

The books above are available in HTML format. Commands in the **Help** menu of any of the Common LispWorks tools give you direct access to the online documentation in HTML format, using the HTML browser that is supplied with LispWorks. Details of how to use these commands can be found in the *Common LispWorks User Guide*.

- The *LispWorks Release Notes and Installation Guide* explains how to install LispWorks and start it running. It also contains Release Notes describing the new features in this release and any issues that could not be included in the other manuals.

The LispWorks manuals are all available in Portable Documentation Format (PDF). You can use Adobe Reader to browse the PDF documentation online or to print it. Adobe Reader is available for free download from Adobe's web site, `http://www.adobe.com/`.

Please let us know if you find any mistakes in the LispWorks documentation, or if you have any suggestions for improvements.

1.3 Other documentation

The LispWorks manuals do not attempt to describe Lisp itself. For definitive information on Common Lisp, including CLOS, consult the American National Standard X3.226 for Common Lisp. An HTML version of this document is supplied with LispWorks and can be accessed from the **Help** menu.

For information on CLOS, Sonya E. Keene's book *Object-Oriented Programming in Common Lisp: A Programmers' Guide* is very helpful. This book is published by Addison-Wesley.

For an account of Metaobject protocols as well as a detailed study of an implementation of CLOS see Kiczales, Rivieres and Bobrow, *The Art of the Meta-Object Protocol*, published by MIT Press. The LispWorks MOP mostly conforms to the MOP described in chapters 5 & 6 of this book.

1.4 Notation and conventions

Throughout this manual you will find references such as "... the LispWorks file **`foo/bar.lisp`** ...". This means a file **`bar.lisp`** in a subdirectory **`foo`** of the LispWorks library directory. You can obtain the full path of such a file by evaluating this form in your LispWorks image:

```
(sys:lispworks-file "foo/bar.lisp")
```

The LispWorks manuals follow the notation used in *Common Lisp: the Language* (2nd Edition).

Please note that your windows may differ in some respects from the illustrations given in the LispWorks manuals. This is because some details are controlled by the window manager that you are using, not by LispWorks itself.

2

Starting LispWorks

Firstly you need LispWorks installed as described in the Release Notes and Installation Guide.

2.1 The usual way to start LispWorks

On Windows and Mac OS X the simplest way to run LispWorks is that provided in the desktop environment. On Windows you can run LispWorks from the Start menu. On Mac OS X you can run LispWorks by clicking on the ringed "LW" icon in the Dock. On both these platforms you can create a shortcut to LispWorks and place it somewhere that is convenient for you, such as the Quick Start toolbar in Windows XP.

On Linux, FreeBSD and UNIX systems you start LispWorks by entering the name of the LispWorks executable at a shell prompt.

2.2 Passing arguments to LispWorks

Occasionally you may need to start LispWorks with certain arguments. This section describes the most frequent of these occasions.

2.2.1 Saving a new image

To save a new image, create a suitable file **save-config.lisp** as described in the section "Saving and testing the configured image" in the *LispWorks Release Notes and Installation Guide*. Such a file should call **(load-all-patches)** and then load any desired configuration, modules and application code, and lastly call **save-image**.

Then you run LispWorks with a command line which passes your file as an build script.

On Mac OS X, run Terminal.app to get a shell, and enter a line like this at the prompt:

```
% lispworks-5-1-0-macos-universal -build /tmp/save-config.lisp
```

On Windows, run Command Prompt to get a DOS shell, and enter a line like this:

```
C:\Program Files\LispWorks>lispworks-5-1-0-x86-win32.exe -build
C:\temp\save-config.lisp
```

On Linux, get a shell and enter a line like this:

```
% lispworks-5-1-0-x86-linux -build /tmp/save-config.lisp
```

On UNIX, get a shell and enter a line like this:

```
% lispworks-5-1-0-sparc-solaris -build /tmp/save-config.lisp
```

When the command exits, a new image has been saved. You can run this new image directly from the command line, or create a shortcut or symbolic link to make it convenient to run.

With all the command lines above, if you perform the task frequently, make a script or a shortcut containing the command line, and run that.

2.2.2 Saving a console mode image

To save a LispWorks image which does not start the graphical IDE by default, make a script similar to **save-config.lisp** above, but where you call

```
(save-image "my-console-lispworks" :environment nil)
```

The resulting new image, **my-console-lispworks**, can be made to start the graphical IDE either by calling **env:start-environment** or by passing **-env** or **-environment** on the command line.

2.2.3 Bypassing initialization files

If you do not want to load your personal initialization file, for example to discover if behavior of LispWorks is due to some setting of yours, pass **-init -** on the command line.

To start LispWorks without loading either the personal or site initialization files, start it like this:

```
lispworks -init - -siteinit -
```

2.3 Other command line options

Other less commonly-used LispWorks command line arguments are described in “The Command Line” on page 286

2.4 Starting the Common LispWorks Graphical IDE

In LispWorks images shipped on the Windows, Mac OS X, Linux and FreeBSD platforms, the IDE starts automatically by default.

If you have an image saved such that the IDE does not start by default, you can start the IDE by calling the function **env:start-environment**. Such an image is shipped for UNIX platforms.

2.5 Using LispWorks with SLIME

To use LispWorks with SLIME you need an image which does not start the LispWorks IDE automatically. To create this image, make a **save-image** script, for example in **/tmp/resave.lisp**, containing:

```
(in-package "CL-USER")
(load-all-patches)
(save-image "~/lw-console"
            :console t
            :environment nil
            :multiprocessing t)
```

Run LispWorks like this to create the new image **~/lw-console**:

```
lispworks-5-1-0-x86-linux -build /tmp/resave.lisp
```

Download SLIME from **http://common-lisp.net/project/slime/** and configure Emacs to use **"~/lw-console"** as the value of **inferior-lisp-program** as shown in the SLIME README.

Note: Use of LispWorks Personal Edition with SLIME is not supported.

2.6 Quitting LispWorks

To quit LispWorks from the graphical IDE, use one of the following:

- The menu command **File > Exit** all platforms except Mac OS X.
- The menu command **LispWorks > Quit LispWorks** on Mac OS X.
- The key **Command+Q** on Mac OS X
- The key sequence **Ctrl+X Ctrl+C** in an editor-based tool such as the Editor or Listener
- A platform / window-manager-specific exit gesture such as clicking a close button on the Podium window
- Call the function **quit**.

To quit LispWorks when running in console mode or via SLIME, simply call **quit**.

3

The Listener

The listener is another name for the read-eval-print loop (REPL) which allows you to interactively evaluate Lisp forms and see their output and return values. Lisp programmers typically do incremental development and testing in a listener before saving the working code to disk.

This chapter describes the basic use of a LispWorks listener. You might access this in a terminal (Unix shell) or MS-DOS command window. The Common LispWorks IDE contains a graphical Listener tool which runs a REPL and supports all the functionality described in this chapter, as well as its own graphical features. Please refer to the *Common LispWorks User Guide* for details specific to the graphical Listener tool.

3.1 First use of the listener

LispWorks runs a top-level REPL on startup. The listener by default appears with a prompt. The name of the current package (that is, the value of **`cl:*package*`**) is printed followed by a positive integer, like this:

```
CL-USER 1 >
```

Enter a Lisp form after the prompt and press **`Return`**:

```
CL-USER 1 > (print 42)

42
42

CL-USER 2 >
```

The first '42' printed is the output of the call to **print**. You see it here because output sent to ***standard-output*** is written to the listener.

The second '42' printed is the return value of the call to print.

After the return value a new prompt appears. Notice that it contains '2' after the package name: your successive inputs are numbered. You can now proceed to develop and test pieces of your application code:

```
CL-USER 2 > (defstruct animal species name weight)
ANIMAL

CL-USER 3 > (make-animal :species "Hippopotamus" :name "Hilda"
:weight 42)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)
```

3.2 Standard listener commands

Generally the listener simply evaluates Lisp forms that you enter. However a few keywords, described in the this section, are specially recognized as shortcut for common listener operations.

3.2.1 Standard top-level loop commands

:redo *Listener command*

:redo &optional *command-identifier*

This option repeats a previous input. The *command-identifier* is either a number in the listener's history list or a symbol or subform in the input to repeat. If *command-identifier* is not supplied, the last input is repeated.

:get *Listener command*

:get *name command-identifier*

:get retrieves a previously-entered input from the listener's history and places it in the variable *name*. The *command-identifier* is the history list number of the input to be retrieved.

:use

Listener command

:use ***new old*** **&optional** ***command-identifier***

:use does a variant of a previous input. *old* matches a symbol or subform in the previous input, and is replaced with *new* to construct the new input. If supplied, *command-identifier* is the history list number of the input you want to modify.

:his

Listener command

:his &optional ***n m***

:his produces a list of the input history. If *n* is supplied it should be a positive integer: the last *n* inputs are shown. If *m* is also supplied it should be a positive integer greater than *n*, when inputs numbered *n* through *m* in the history are shown.

:bug-form

Listener command

:bug-form ***subject*** **&key** ***filename***

:bug-form prints a template bug report suitable for sending to Lisp Support. Supply a string *subject*. If you also supply *filename*, the report is printed to the file.

:help

Listener command

:help

:help prints a brief listing of the available listener commands.

:?

Listener command

:?

:? is a synonym for **:help**.

3.2.2 Examples

```
CL-USER 4 > :redo
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)

CL-USER 5 > :his

1: (PRINT 42)
2: (DEFSTRUCT ANIMAL SPECIES NAME ...)
3: (MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
4: (MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)

CL-USER 5 > :get make-hilda 3

CL-USER 5 > make-hilda
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME "Hilda" :WEIGHT 42)

CL-USER 6 > :use "Henry" "Hilda"
(MAKE-ANIMAL :SPECIES "Hippopotamus" :NAME ...)
#S(ANIMAL :SPECIES "Hippopotamus" :NAME "Henry" :WEIGHT 42)

CL-USER 7 > :bug-form "Too many hippos..." :filename "bug-
report.txt"
```

3.3 The listener prompt

The variable ***prompt*** controls the appearance of the listener prompt. See its manual page in the *LispWorks Reference Manual* if you want to alter this.

If the default prompt contains a colon followed by a second positive integer then you are no longer in the top-level loop, but have entered the REPL debugger, as described in “The Debugger” on page 15.

4

The Debugger

The debugger is an interactive tool for examining and manipulating the Lisp environment. Within the debugger you have access to not only the interpreter, but also to a variety of debugging tools. The default behavior when any error occurs is to enter the debugger. Users can then trace backwards through the history of function calls to determine how the error arose. They may inspect and alter local variables of the functions on the execution stack, and possibly continue execution by invoking a pre-defined restart (if available) or by forcing any function invocation on the stack to return user-specified values.

When writing an application it is possible to prevent entry to the debugger when an error occurs, by creating condition handlers to take some appropriate action to recover without user intervention. It is also possible to use restarts to specify some default methods of error recovery. The debugger is entered whenever an error is signalled (via a call to **`error`** or **`cerror`**) and not handled by an error handler, or it can be explicitly invoked via a call to **`break`**.

You can use the debugger in REPL mode (that is, in the listener read-eval-print loop) or using the graphical Debugger tool in the Common LispWorks IDE. This chapter describes the REPL debugger; please refer to the *Common LispWorks User Guide* for details about the graphical Debugger tool.

The compiler generates information necessary for the use of the debugger during compilation. You can opt for faster compilation, at the expense of

reducing the information available to the debugger. See under **toggle-source-debugging** in the *LispWorks Reference Manual*.

4.1 Entering the REPL debugger

The following is a simple example.

```
CL-USER 1 > (defun make-a-hippo (name weight)
              (if (numberp weight)
                  (make-animal 'hippo name weight)
                (error "Argument to make-a-hippo not a number")))
MAKE-A-HIPPO

CL-USER 2 > (make-a-hippo "Hilda" nil)

Error: Argument to make-a-hippo not a number
  1 (abort) return to level 0.
  2 return to top loop level 0.
  3 Destroy process.

Type :c followed by a number to proceed
CL-USER 3 : 1 >
```

The call to **error** causes entry into the debugger. The final prompt in the example contains a 1 to indicate that the top level of the debugger has been entered. The debugger can be entered recursively, and the prompt shows the current level. Once inside the debugger, you may use all the facilities available at the top-level in addition to the debugger commands.

The debugger may also be invoked by using the trace facility to force a break at entry to or exit from a particular function.

The debugger can also be entered by a keyboard interrupt. Keyboard interrupts are generated by **Ctrl+Break** on Microsoft Windows and **Meta-Ctrl-c** on Motif (there's no corresponding gesture on Mac OS X).

4.2 Simple use of the REPL debugger

Upon entering the debugger as a result of an error, a message describing the error is printed and a number of options to continue (called restarts) are presented. Thus:

```
CL-USER 6 > (/ 3 0)

Error: Division-by-zero caused by / of (3 0)
  1 (continue) Return a value to use
  2 Supply new arguments to use
  3 (abort) return to level 0.
  4 return to top loop level 0.
  5 Destroy process.

Type :c followed by a number to proceed

CL-USER 7 : 1 >
```

To select one of these restarts, enter **:c** (continue) followed by the number of the restart. So in the above example you could continue as follows:

```
CL-USER 7 : 1 > :c 2

Supply first number: 33

Supply second number: 11
3

CL-USER 8 >
```

There are two special restarts, a continue restart and an abort restart. These are indicated by the bracketed word continue or abort at their start. The continue restart can be invoked by typing **:c** alone. Similarly, the abort restart can be invoked by entering **:a**. So an alternative continuation of the division example would be:

```
CL-USER 7 : 1 > :c

Supply a form to be evaluated and used: (+ 4 5)
9
```

4.3 The stack in the debugger

The debugger allows you to examine the state of the execution stack. This consists of a sequence of frames representing active function invocations, special variable bindings, restarts, active catchers, active handlers and system-related code. In particular the execution stack has a call frame for each active function call (that is for each function that has been entered but from which control has not yet returned). The top of the stack contains the most recently created frames (and so the innermost calls), and the bottom of the stack contains the oldest frames (and so the outermost calls). You can examine a call frame to find the function's name, and the names and values of its arguments.

The function call frames displayed are affected by any **`hcl:alias`** and **`hcl:invisible-frame`** declarations.

Catch frames are established by using the special form catch, and exist to receive throws to the matching tag. Restart frames correspond to restarts that have been set up, and handler frames correspond to the error handlers currently active. Binding frames are formed when special variables are bound. Open frames are established by the system. By default only the catch frames and the call frames are displayed. However the remaining types of frame are displayed if you set the appropriate variables (see Section 4.5 on page 27).

Within the debugger there are commands to examine a stack frame, and to move around the stack. These are explained in the following section. Typing **`:help`** in the debugger also produces a command listing.

One of the most useful features is that you can access a local variable in the current frame simply by entering its name as shown in the backtrace. See step 7 in "Example debugging session" on page 25.

4.4 REPL debugger commands

This section describes commands specific to the debugger. In the debugger, you can also do anything that you can do in the top-level loop including evaluation of forms and the standard listener commands.

Upon entry to the debugger the implicit current stack frame is set to the top of the execution stack. The debugger commands allow you to move around the stack, to examine the current frame, and to leave the debugger. The com-

mands are all keywords, and as such case-insensitive, but are shown here in lower case for clarity.

You can get brief help listing these commands by entering **:?** at the debugger prompt.

4.4.1 Backtracing

A backtrace is a list of the stack frames starting at the current frame and continuing down the stack. The backtrace thus displays the sequence by which the functions were invoked, starting with the most recent. For instance:

```
CL-USER 10 > (defun function-1 (a b c)
               (function-2 (+ a b) c))
FUNCTION-1

CL-USER 11 > (defun function-2 (a b)
               (function-3 (+ a b)))
FUNCTION-2

CL-USER 12 > (defun function-3 (a) (/ 3 (- 111 a)))
FUNCTION-3

CL-USER 13 > (function-1 1 10 100)

Error: Division-by-zero caused by / of (3 0)
  1 (continue) Return a value to use
  2 Supply new arguments to use
  3 (abort) return to level 0.
  4 return to top loop level 0.
  5 Destroy process.

Type :c followed by a number to proceed

CL-USER 14 : 1 > :bq 10

SYSTEM::DIVISION-BY-ZERO-ERROR <- / <- FUNCTION-3
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-2
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- FUNCTION-1
<- SYSTEM::*%APPLY-INTERPRETED-FUNCTION <- SYSTEM::%INVOKE <-
SYSTEM::%EVAL

CL-USER 15 : 1 >
```

In the above example the command to show a quick backtrace was used (**`:bq`**). Instead of showing each stack frame fully, this only shows the function name associated with each of the call frames. The number 10 following **`:bq`** specifies that only the next ten frames should be displayed rather than continuing to the bottom of the stack.

:b *Debugger command*

`:b &optional` ***verbose m***

This is the command to obtain a backtrace from the current frame. It may optionally be followed by **`:verbose`**, in which case a fuller description of each frame is given that includes the values of the arguments to the function calls. It may also be followed by a number (*m*), specifying that only that number of frames should be displayed.

:bq *Debugger command*

`:bq` ***m***

This produces a quick backtrace from the current position. Only the call frames are included, and only the names of the associated functions are shown. If the command is followed by a number then only that many frames are displayed.

4.4.2 Moving around the stack

On entry to the debugger the current frame is the one at the top of the execution stack. There are commands to move to the top and bottom of the stack, to move up or down the stack by a certain number of frames, and to move to the frame representing an invocation of a particular function.

:> *Debugger command*

This sets the current frame to the one at the bottom of the stack.

:< *Debugger command*

This sets the current frame to the one at the top of the stack.

:p *Debugger command*

`:p [m|fn-name|fn-name-substring]`

By default this takes you to the previous frame on the stack. If it is followed by a number then it moves that number of frames up the stack. If it is followed by a function name then it moves to the previous call frame for that function. If it is followed by a string then it moves to the previous call frame whose function name contains that string.

:n *Debugger command*

`:n [m|fn-name|fn-name-substring]`

Similar to the above, this goes to the next frame down the stack, or *m* frames down the stack, or to the next call frame for the function *fn-name,* or to the next call frame whose function name contains *fn-name-substring.*

4.4.3 Miscellaneous commands

:v *Debugger command*

This displays information about the current stack frame. In the case of a call frame corresponding to a compiled function the names and values of the function's arguments are shown. Closure variables (either from an outer scope or used by an inner scope) and special variables are indicated by **`{Closing}`** or **`{Special}`** as in this session:

```
CL-USER 40 > (compile (defun foo (*zero* one two) (declare
(special *zero*)) (divider one *zero*) (list #'(lambda () one)
two)))
FOO
NIL
NIL

CL-USER 41 > (foo 0 1 2)

Error: Division-by-zero caused by / of (1 0).
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,  or :? for
other options

CL-USER 42 : 1 > :v
Call to FOO (offset 87)
  *ZERO*  {Special} : 0
  ONE     {Closing} : 1
  TWO               : 2

CL-USER 43 : 1 >
```

For an interpreted function the names and values of local variables are also shown.

If the value of an argument is not known (perhaps because the code has been compiled for speed rather than other considerations), then it is printed as the keyword **:dont-know**.

:l *Debugger Command*

:l [*m*|*var-name*|*var-name-substring*]

By default this prints a list of the values of all the local variables in the current frame. If the command is followed by a number then it prints the value of the ***m***'th local variables (counting from 0, in the order shown by the **:v** command). If it is followed by a variable name *var-name* then it prints the value of that variable (note that the same effect can be achieved by just entering the name of the variable into the Listener). If it

is followed by a string *var-name-substring* then it prints the value of the first variable whose name contains *var-name-substring*.

In all cases, `*` is set to the printed value.

:error

Debugger command

This reprints the message which was displayed upon entry to the current level of the debugger. This is typically an error message and includes several continuation options.

:cc

Debugger command

`:cc &optional` ***`var`***

This returns the current condition object which caused entry to this level of the debugger. If an optional *var* is supplied then this must be a symbol, whose symbol-value is set to the value of the condition object.

:ed

Debugger command

This allows you to edit the function associated with the current frame. If you are using TAGS, you are prompted for a TAGS file.

:all

Debugger command

`:all &optional` ***`flag`***

This option enables you to set the debugger option to show all frames (if *flag* is non-`nil`), or back to the default (if *flag* is `nil`). By default, *flag* is `t`.

:lambda

Debugger command

This returns the lambda expression for an anonymous interpreted frame. If the expression is not known, then it is printed as the keyword `:dont-know`

:lf *Debugger command*

This command prints symbols from other packages corresponding to the symbol that was called, but could not be found, in the current package. Any such symbols are also offered as restarts when you first enter the debugger.

```
NEW 21 > (initialize-graphics-port)

Error: Undefined function INITIALIZE-GRAPHICS-PORT called with
arguments ().
  1 (continue) Try invoking INITIALIZE-GRAPHICS-PORT again.
  2 Return some values from the call to INITIALIZE-GRAPHICS-PORT.
  3 Try invoking GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT with the
same arguments.
  4 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to the
symbol-function of GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT.
  5 Try invoking something other than INITIALIZE-GRAPHICS-PORT
with the same arguments.
  6 Set the symbol-function of INITIALIZE-GRAPHICS-PORT to
another function.
  7 (abort) Return to level 0.
  8 Return to top loop level 0.

Type :c followed by a number to proceed or type :? for other
options

NEW 22 : 1 > :lf
Possible candidates are (GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT)
GRAPHICS-PORTS:INITIALIZE-GRAPHICS-PORT

NEW 23 : 1 >
```

4.4.4 Leaving the debugger

You may leave the debugger either by taking one of the continuation options initially presented, or by explicitly specifying values to return from one of the frames on the stack.

:a *Debugger command*

This selects the **`:abort`** option from the various continuation options that are displayed when you enter the current level of the debugger.

:c *Debugger command*

:c &optional *m*

If this is followed by a number then it selects the option with that number, otherwise it selects the **:continue** option.

:ret *Debugger command*

:ret *value*

This causes *value* to be returned from the current frame. It is only possible to use this command when the current frame is a call frame. Multiple values may be returned by using the **values** function. So to return the values 1 and 2 from the current call frame, you could type

```
:ret (values 1 2)
```

:res *Debugger command*

:res *m*

Restarts the current frame. If *m* is **nil**, you are prompted for new arguments which should be entered on one line, separated by whitespace. If *m* is true or is not supplied, the original arguments to the frame are used.

:top *Debugger command*

Aborts to the top level of the debugger. A synonym is **:a :t**.

4.4.5 Example debugging session

This section presents a short interactive debugging session. It starts by defining a routine to calculate Fibonacci Numbers, and then erroneously calls it with a string.

1. First, define the **fibonacci** function shown below in a listener.

```
(defun fibonacci (m)
  (let ((fib-n-1 1)
        (fib-n-2 1)
        (index 2))
    (loop
     (if (= index m) (return fib-n-1))
     (incf index)
     (psetq fib-n-1 (+ fib-n-1 fib-n-2)
                fib-n-2 fib-n-1))))
```

2. Next, call the function as follows.

   ```
   (fibonacci "turtle")
   ```

   The system generates an error, since **=** expects its arguments to be numbers, and displays several continuation options, so that you can try to find out how the problem arose.

3. Type **:bq** at the debugger prompt to perform a quick backtrace. Notice that the problem is in the call to **fibonacci**.

   Note that the calls to ***%apply-interpreted-function** in the backtrace occur because **fibonacci** is being interpreted.

   You should have passed the length of the string as an argument to **fibonacci**, rather than the string itself.

4. Do this now, by typing the following form at the debugger prompt.

   ```
   (legnth "turtle")
   ```

   You intended to call **fibonacci** with the length of the string, but typed in **length** incorrectly. This takes you into the second level of the debugger. Note that the continuation options from your entry into the top level of the debugger are still displayed, and are listed after the new options. You can select any of these options.

5. Type **:a** to return to the top level of the debugger.

6. Type **:v** to display variable information about the current stack frame in the debugger.

   The following output is displayed:

```
M : "turtle"
INDEX : 2
FIB-N-2 : 1
FIB-N-1 : 1
```

You need to set the value of the variable **m** to be the length of the string "turtle", rather than the string itself.

7. Type in the form below.

```
(setq m (length "turtle"))
```

In order to get the original computation to resume using the new value of **m**, you still need to handle the original error.

8. Type **:error** to remind yourself of the original error condition you need to handle.

You can handle this error by returning **nil** from the call to **=**, which is the result that would have been obtained if **m** had been correctly set.

9. Type **:c** to invoke the continue restart, which in this case requires you to return a value to use.

10. When prompted for a form to be evaluated, type **nil**.

This causes execution to continue as desired, and you can obtain the final result with no further problems.

4.5 Debugger control variables

common-lisp:*debug-io* *Variable*

The value of this variable is the stream which the debugger uses for its input and output.

dbg:*debug-print-length* *Variable*

The value to which **common-lisp:*print-length*** is bound during output from the debugger.

dbg:*debug-print-level* *Variable*

The value to which **common-lisp:*print-level*** is bound during output from the debugger.

dbg:*hidden-packages* *Variable*

This variable should be bound to a list of packages. The debugger suppresses symbols from these packages (so, for example, it does not display call frames for functions in these packages).

dbg:*print-binding-frames*

dbg:*print-catch-frames*

dbg:*print-handler-frames*

dbg:*print-open-frames*

dbg:*print-restart-frames*

dbg:*print-non-symbol-frames* *Variables*

These six variables control whether or not the corresponding types of frame are displayed by the debugger. For each variable, if the variable is non-**nil** then that type of frame is shown. Initially only **dbg:*print-catch-frames*** is non-**nil**. Note that the call frames are always displayed.

The following function is used in conjunction with these variables.

dbg:set-debugger-options *Function*

dbg:set-debugger-options &key ***all bindings catchers hidden non-symbol handler restarts invisible***

A call to **set-debugger-options** allows you to set the above variables without having the inconvenience of setting each variable individually with a call to **setq** and without having to remember the precise names for each of the variables.

The keywords in the function refer to the different system variables as described below:

`:all` — affects the state of the **`:all`** command.

`:bindings` — **`dbg:*print-binding-frames*`**

`:catchers` — **`dbg:*print-catch-frames*`**

`:hidden` — **`dbg:*hidden-packages*`**

`:non-symbol` — **`dbg:*print-non-symbol-frames*`**

`:handler` — **`dbg:*print-handler-frames*`**

`:restarts` — **`dbg:*print-restart-frames*`**

`:invisible` — **`dbg:*print-invisible-frames*`**

5

The REPL Inspector

LispWorks provides two inspectors. One is for use with the Common LispWorks environment, and is described in the *Common LispWorks User Guide*. The other is the REPL inspector, which uses a stream interface, and can be used on any terminal (in particular within a Common LispWorks listener). Both inspectors allow you to traverse complex data structures interactively and to destructively modify components of these structures. However, the two inspectors are quite different. No attempt has been made to make their usage compatible and instead each inspector is designed to exploit to the full the particular environment facilities available.

The REPL inspector provides a simple inspector facility which can be used on a stream providing line breaks as the only type of formatting. It is built on top of the **`describe`** function which is briefly described below and modifies the top level loop in a similar way to the debugger (see Chapter 4, "The Debugger").

5.1 Describe

The function **`describe`** displays the slots of composite data structures in a manner dependent on the type of the object. The slots are labeled with a name where appropriate, or otherwise with a number.

The example below shows the result of calling **`describe`** on a simple list.

```
USER 7 > (setq countries '("Chile" "Peru" "Paraguay"
                           "Brazil"))
("Chile" "Peru" "Paraguay" "Brazil")

USER 8 > (describe countries)
("Chile" "Peru" "Paraguay" "Brazil") is a CONS
[0] : "Chile"

[1] : "Peru"

[2] : "Paraguay"

[3] : "Brazil"
```

describe describes slots recursively up to a limit set by the special variable ***describe-level***. Note that only arrays, structures and conses are printed recursively. The slots of all other object types are only printed when at the top level of **describe**.

describe-level has an initial value of 1.

The symbols ***describe-print-level*** and ***describe-print-length*** are similar in effect to ***trace-print-level*** and ***trace-print-length***. They control, respectively, the depth to which nested objects are printed (initial value 10), and the number of components of an object which are printed (initial value 10).

To customize **describe**, define new methods on the generic function **describe-object**.

5.2 Inspect

The function **inspect** is an interactive version of **describe**. It displays objects in a similar way to **describe**. Entering the inspector causes a new level of the top loop to be entered with a special prompt indicating that the inspector has been entered and showing the current inspector level.

In the modified top loop, if you enter a slot name, that slot is inspected and the current object is pushed onto an internal stack of previously inspected objects. The special variables **$**, **$$**, and **$$$** are bound to the top three objects on the inspector stack.

The following keywords are treated specially as comands by the inspector.

Table 5.1 Inspector keywords

:cv	Display current values of control variables.
:d	Display current object.
:dm	Display more of current object.
:dr	Display rest of current object.
:h	Display help on inspector commands.
:i *m*	Recursively invoke a new inspector. *m* is an object to inspect.
:m	Change the inspection mode — see Section 5.3 on page 34.
:q	Quit current inspector.
:s *n v*	Sets slot *n* to value *v*.
:sh	Show inspector stack.
:u *int*	Undo last inspection. If you supply an optional integer argument, *int*, then the last *int* inspections are undone.
:ud	Undo last inspection and redisplay current object.

You can get brief help listing these commands by entering **:?** at the inspector prompt.

The control variables ***inspect-print-level*** and ***inspect-print-length*** are similar to ***describe-print-level*** and ***describe-print-length*** (see above).

:dm displays more slots of the current object. If the object has more than ***describe-length*** slots, then the first ***describe-length*** will be printed, followed by an ellipsis and then

```
(:dm or :dr for more)
```

If you enter the command **:dm** at the prompt it displays the next ***describe-length*** slots, and if you enter **:dr** it displays all the remaining slots. This only works on the last inspected object, so if you recursively inspect a slot and

come back, **:dm** does not do anything useful. Typing **:d** lets you view the object again.

:ud is equivalent to typing **:u** followed by **:d**.

5.3 Inspection modes

The **:m** command displays and changes the current inspection mode for an inspected value. The session below demonstrates how it works:

```
CL-USER 128 > (inspect "a
string with
newlines in it")

"a
string with
newlines in it" is a SIMPLE-BASE-STRING
0       #\a
1       #\Newline
2       #\s
3       #\t
4       #\r
5       #\i
6       #\n
7       #\g
8       #\Space
9       #\w
10      #\i
11      #\t
12      #\h
13      #\Newline
14      #\n
15      #\e
16      #\w
17      #\l
18      #\i
19      #\n ........ (:dm or :dr for more)


CL-USER 129 : Inspect 1 > :m
* 1. SIMPLE-STRING
  2. LINES
```

The **:m** produces an enumerated list of inspection modes for this value.

The asterisk next to

```
* 1. SIMPLE-STRING
```

means that SIMPLE-STRING is the current inspection mode.

You can change mode by typing `:m` followed by the name or number of another mode. To change to LINES mode:

```
CL-USER 130 : Inspect 1 > :m 2

"a
string with
newlines in it" is a SIMPLE-BASE-STRING
0      a
1      string with
2      newlines in it
```

5.3.1 Hash table inspection modes

There are five hash table inspection modes. They can be accessed in either the Common LispWorks Inspector tool or the REPL inspector.

A brief introduction to the representation of hash tables is necessary so that you can fully understand what you gain from the new modes.

Internally, a hash table is a structure containing, among other things,

- a big vector
- size and growth information
- accessing functions.

When keys and values are added to the table, sufficiently similar keys are converted into the same index in the vector. When this happens, the similar keys and values are kept together in a chain that hangs off this place in the vector.

The different inspection modes provide views of different pieces of this structure:

HASH-TABLE This mode is the “normal” view of a hash table; as a table of keys and values. When you inspect an item you inspect the value of the item.

STRUCTURE This mode provides a raw view of the whole hash table structure. When you inspect an item you are inspecting the value of that slot in the hash table structure.

ENUMERATED-HASH-TABLE

This mode is a variation of the normal view, where a hash table is viewed simply as a list of lists. When you inspect an item you are inspecting a list containing a key and a value.

HASH-TABLE-STATISTICS

This mode shows how long the chains in the hash table are, so that you can tell how efficiently it is being used. For example, if all chains contained fewer than two items the hash table would be being used well.

HASH-TABLE-HISTOGRAM

This mode shows the statistical information from **HASH-TABLE-STATISTICS** as a histogram.

Here is an example of hash table inspection.

```
CL-USER 1 > (defvar *hash* (make-hash-table))
*HASH*

CL-USER 2 > (setf (gethash 'lisp *hash*) 'programming
                  (gethash 'java *hash*) 'programming
                  (gethash 'c *hash*) 'programming
                  (gethash 'c++ *hash*) 'programming
                  (gethash 'english *hash*) 'natural
                  (gethash 'german *hash*) 'natural)
NATURAL

CL-USER 3 > (inspect *hash*)

#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE
C++          PROGRAMMING
JAVA         PROGRAMMING
ENGLISH      NATURAL
C            PROGRAMMING
GERMAN       NATURAL
LISP         PROGRAMMING

CL-USER 4 : Inspect 1 > :m
* 1. HASH-TABLE
  2. STRUCTURE
  3. ENUMERATED-HASH-TABLE
  4. HASH-TABLE-STATISTICS
  5. HASH-TABLE-HISTOGRAM
```

STRUCTURE mode displays the raw representation of the hash table:

```
CL-USER 5 : Inspect 1 > :m 2

#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE
KIND                      EQL
SIZE                      37
REHASH-SIZE               2.0
REHASH-THRESHOLD          1.0
THRESHOLD                 37
COUNTER                   525
NUMBER-ENTRIES            6
TABLE                     #(#%((LISP . PROGRAMMING) NIL) NIL NIL
NIL NIL ...)
NO-DESTRUCT-REHASH        NIL
POWER2                    NIL
HASH-REM                  SYSTEM::DIVIDE-GENERAL
HASH-FN                   SYSTEM::EQL-HASHFN
GETHASH-FN                SYSTEM::GETHASH-EQL
PUTHASH-FN                SYSTEM::PUTHASH-EQL
REMHASH-FN                SYSTEM::REMHASH-EQL
GET-TLATTER-FN            SYSTEM::GET-TLATTER-EQL
WEAK-KIND                 NIL
USER-STUFF                NIL
MODIFICATION-COUNTER      0
FAST-LOCK-SLOT            0
```

In **ENUMERATED-HASH-TABLE** mode you can recursively inspect keys and values by entering the index. This is especially useful in cases where the key or value is unreadable and so cannot be entered into the REPL:

```
CL-USER 6 : Inspect 1 > :m 3

#<EQL Hash Table{6} 21C15D97> is an Enumerated HASH TABLE
0      (C++ PROGRAMMING)
1      (JAVA PROGRAMMING)
2      (ENGLISH NATURAL)
3      (C PROGRAMMING)
4      (GERMAN NATURAL)
5      (LISP PROGRAMMING)

CL-USER 7 : Inspect 1 > 5

(LISP PROGRAMMING) is a LIST
0      LISP
1      PROGRAMMING

CL-USER 8 : Inspect 2 > :u
```

The **HASH-TABLE-STATISTICS** mode shows that ***hash*** has 31 chains, of which 25 are empty and 6 have one entry::

```
CL-USER 9 : Inspect 1 > :m 4

#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE (statistical view)
chain of length 0 :      31
chain of length 1 :      6
```

In **HASH-TABLE-HISTOGRAM** mode the same information is represented as a histogram:

```
CL-USER 10 : Inspect 1 > :m 5

#<EQL Hash Table{6} 21C15D97> is a HASH-TABLE (histogram view)
chain of length 0 :      "*******************************"
chain of length 1 :      "******"

CL-USER 11 : Inspect 1 > :q
#<EQL Hash Table{6} 21C15D97>
```

6

The Trace Facility

The trace facility is a debugging aid enabling you to follow the execution of particular functions. At any time there are a set of functions (and macros and methods) which are being monitored in this way. The normal behavior when a call is made to one of these functions is for the function's name, arguments and results to be printed out by the system. More generally you can specify that particular forms should be executed before or after entering a function, or that certain calls to the function should cause it to enter the main debugger. Tracing of a function continues even if the function is redefined; however the tracing of some structure accessors and so forth may be lost if the compiler is set to optimize the code for efficiency (so that these calls are inlined).

The standard way of getting functions to be traced in this way is to call the macro **`trace`** with the symbols of the functions (or macros or generic functions) concerned. In addition it is possible to restrict tracing to a particular method (rather than a generic function), by specifying the requisite classes for the arguments in the call to trace. The trace facility handles recursive and nested calls to the functions concerned.

6.1 Simple tracing

This section shows you how to perform simple traces.

1. Type this definition of the factorial function **`fac`** into the listener:

```
(defun fac (n)
 (if (= n 1) 1
   (* n (fac (- n 1)))))
```

2. Now trace the function by typing the following into the listener.

```
(trace fac)
```

3. Call the function **fac** as follows:

```
(fac 3)
```

The following trace output appears in the listener.

```
0 FAC > (3)
  1 FAC > (2)
    2 FAC > (1)
    2 FAC < (1)
  1 FAC < (2)
0 FAC < (6)
```

Upon entry to each traced function call, **trace** prints the following information:

- The *level* of tracing, that is, the number of recursive entries to **trace** (starting at 0).
- The function name.
- The argument for the current call.

Each call is indented according to the level of tracing for the call.

Upon exit from each call, the same information is produced: The **>** symbol denotes entry to a function, and the **<** symbol denotes exit from it.

Output produced in this way is always sent to a special stream, ***trace-output***, which is either associated with the listener, or with background output. You can give other expressions to be sent to this stream, in addition to the arguments and results of a function.

Calling **trace** with no arguments produces a list of all the functions currently being traced. In order to cease tracing a function the macro **untrace** should be called with commands. All tracing can be removed by calling **untrace** with no arguments.

```
CL-USER 5 > (untrace fac)
NIL
CL-USER 6 > (fac 4)
24

CL-USER 7 >
```

6.2 Tracing options

There are a number of options available when using the trace facilities, which allow you both to restrict or expand upon the information printed during a trace. For instance, you can restrict tracing of a function to a particular process, or specify additional actions to be taken on function call entry and exit.

Note that the options and values available only apply to a particular traced function. Each traced function has its own, independent, set of options.

This section describes the options that are available. Each option can be set as described above.

6.2.1 Evaluating forms on entry to and exit from a traced function

:before *Trace keyword*

`:before` ***list of forms***

If non-**`nil`**, the list of forms is evaluated on entry to the function being traced. The forms are evaluated and the results printed after the arguments to the function.

Here is an example of its use. **`*traced-arglist*`** is bound to the list of arguments given to the function being traced. In this example, it is used to accumulate a list of all the arguments to **`fac`** across all iterations.

1. In the listener, initialize the variable **`args-in-reverse`** as follows:

   ```
   (setq args-in-reverse ())
   ```

2. For the **`fac`** function used earlier, set the value of **`:before`** to the following list:

   ```
   ((push (car *traced-arglist*) args-in-reverse))
   ```

3. In the listener, evaluate the following form:

```
(fac 3)
```

After evaluating this form, **`args-in-reverse`** has the value **`(1 2 3)`**, that is, it lists the arguments which **`fac`** was called with, in the reverse order they were called in.

:after

Trace keyword

`:after` ***list of forms***

If non-**`nil`**, this option evaluates a list of forms upon return from the function to be traced. The forms are evaluated and the results printed after the results of a call to the function.

This option is used in exactly the same way as **`:before`**. For instance, using the example for **`:before`** as a basis, create a list called **`results-in-reverse`**, and set the value of **`:after`** so that **`(car *traced-results*)`** is pushed onto this list. After calling **`fac`**, **`results-in-reverse`** contains the results returned from **`fac`**, in the reverse order they were called in.

Note: **`*traced-arglist*`** is still bound as well.

6.2.2 Evaluating forms without printing results

:eval-before

Trace keyword

`:eval-before` ***list-of-forms***

This option allows you to supply a list of forms for evaluation upon entering the traced function. The forms are evaluated after printing out the arguments to the function, but unlike **`:before`** their results are not printed.

:eval-after

Trace keyword

`:eval-after` ***list-of-forms***

This option allows you to supply a list of forms for evaluation upon leaving the traced function. The forms are evaluated after printing out

the results of the function call, but unlike **:after** their results are not printed.

6.2.3 Using the debugger when tracing

:break *Trace keyword*

:break ***form***

If *form* evaluates to non-**nil**, the debugger is entered directly from **trace**. If it returns **nil**, tracing continues as normal. This option lets you force entry to the debugger by supplying a *form* as simple as **t**.

Upon entry to the traced function, the standard trace information is printed, any supplied **:before** forms are executed, and then *form* is evaluated.

:break-on-exit *Trace keyword*

:break-on-exit ***form***

Like **:break**, this option allows you to enter the debugger from **trace**. It differs in that the debugger is entered *after* the function call is complete.

Upon exit from the traced function, the standard trace information is printed, and then *form* is evaluated. Finally, any supplied **:after** forms are executed.

:backtrace *Trace keyword*

:backtrace ***backtrace***

Generates a backtrace on each call to the traced function. *backtrace* can be any of the following values:

:quick	Like the **:bq** debugger command.
t	Like the **:b** debugger command.
:verbose	Like the **:b :verbose** debugger command.
:bug-form	Like the **:bug-form** debugger command.

6.2.4 Entering stepping mode

:step *Trace keyword*

`:step` ***form***

When non-**`nil`**, this option puts the trace facility into stepper mode, where interpreted code is printed out one step of execution at a time.

6.2.5 Configuring function entry and exit information

:entrycond *Trace keyword*

`:entrycond` ***form***

This option controls the printing of information on entry to a traced function. *form* is evaluated upon entry to the function, and information is printed if and only if *form* evaluates to **`t`**. This allows you to turn off printing of function entry information by supplying a *form* of **`nil`**, as in the example below.

:exitcond *Trace keyword*

`:exitcond` ***form***

This option controls the printing of information on exit from a traced function. *form* is evaluated upon exit from the function, and, like **`:entrycond`**, information is printed if and only if *form* evaluates to **`t`**. This allows you to turn off printing of function exit information by supplying a *form* of **`nil`**.

An example of using **`:exitcond`** and **`:entrycond`** is shown below:

1. For the **`fac`** function, set the values of **`:entrycond`** and **`:exitcond`** as follows.

```
:entrycond  => (evenp (car *traced-arglist*))
:exitcond   => (oddp (car *traced-arglist*))
```

Information is only printed on entry to **fac** if the argument passed to **fac** is even. Conversely, information is only printed on exit from **fac** if the argument passed to **fac** is odd.

2. Type the following call to fac in a listener:

```
CL-USER 12 > (fac 10)
```

The tracing information printed is as follows:

```
0 FAC > (10)
    2 FAC > (8)
        4 FAC > (6)
            6 FAC > (4)
                8 FAC > (2)
                  9 FAC < (1)
              7 FAC < (6)
          5 FAC < (120)
      3 FAC < (5040)
  1 FAC < (362880)
```

6.2.6 Directing trace output

:trace-output *Trace keyword*

:trace-output ***stream***

This option allows you to direct trace output to a stream other than the listener in which the original function call was made. By using this you can arrange to dispatch traced output from different functions to different places.

Consider the following example:

1. In the listener, create a file stream as follows:

```
CL-USER 129 > (setq str (open "trace.txt" :direction :output))
Warning: Setting unbound variable STR
#<File stream "/u/neald/trace.txt">
```

2. Set the value of the **:trace-output** option for the function **fac** to **str**.

3. Call the **fac** function, and then close the file stream as follows:

```
CL-USER 138 > (fac 8)
40320

CL-USER 139 > (close str)
T
```

Inspect the file **trace.txt** in order to see the trace output for the call of **(fac 8)**.

6.2.7 Restricting tracing

:process *Trace keyword*

:process ***process***

This lets you restrict tracing of a function to a particular process. If *process* evaluates to **t**, then the function is traced from within all processes (this is the default). Otherwise, the function is only traced from within the process that *process* evaluates to.

:when *Trace keyword*

:when ***form***

This lets you invoke the tracing facilities on a traced function selectively. Before each call to the function, *form* is evaluated. If *form* evaluates to **nil**, no tracing is done. The contents of **hcl:*traced-arglist*** can be examined by *form* to find the arguments given to trace.

6.2.8 Storing the memory allocation made during a function call

:allocation *Trace keyword*

:allocation ***form***

If *form* is non-**nil**, this prints the memory allocation, in bytes, made during a function call. The symbol that *form* evaluates to is used to accumulate the amount of memory allocated between entering and exiting the traced function.

Note that this symbol continues to be used as an accumulator on subsequent calls to the traced function; the value is compounded, rather than over-written.

Consider the example below:

1. For the **fac** function, set the value of **:allocation** to **$$fac-alloc**.
2. In the listener, call **fac**, and then examine the value of **$$fac-alloc**.

```
CL-USER 152 > $$fac-alloc
744
```

6.2.9 Tracing functions from inside other functions

:inside *Trace keyword*

:inside ***list-of-functions***

The functions given in the argument to **:inside** should reference the traced function in their implementation. The traced function is then only traced in calls to any function in the list of functions, rather than in direct calls to itself.

For example:

1. Define the function **fac2**, which calls **fac**, as follows:

```
(defun fac2 (x)
   (fac x))
```

2. For the **fac** function, set the value of **:inside** to **fac2**.
3. Call **fac**, and notice that no tracing information is produced.

```
CL-USER 154 > (fac 3)
6
```

4. Call **fac2**, and notice the tracing information.

```
CL-USER 155 > (fac2 3)
0 FAC > (3)
  1 FAC > (2)
    2 FAC > (1)
    2 FAC < (1)
  1 FAC < (2)
0 FAC < (6)
```

6.3 Example

The following example illustrates how **trace** may be used as a debugging tool. Suppose that you have defined a function **f**, and intend its first argument to be a non-negative number. You can trap calls to **f** where this is not true, providing an entry into the main debugger in these cases. It is then possible for you to investigate how the problem arose.

To do this, you specify a **:break** option for **f** using **trace**. If the form following this option evaluates to a non-**nil** value upon calling the function, then the debugger is entered. In order to inspect the first argument to the function **f**, you have access to the variable ***traced-arglist***. This variable is bound to a list of the arguments with which the function was called, so the first member of the list corresponds to the first argument of **f** when tracing **f**.

```
CL-USER 12 > (defun f (a1 a2) (+ (sqrt a1) a2))
F

CL-USER 13 > (trace (f :break (< (car *traced-arglist*) 0)))
F

CL-USER 14 > (f 9.0 3)
0 F > (9.0 3)
0 F < (6.0)
6.0

CL-USER 15 > (f -16.0 3)
0 F > (-16.0 3)

Break on entry to F
  1 (continue) return from break.
  2 (abort) return to level 0.
  3 return to top loop level 0.
  4 Destroy process.

Type :c followed by a number to proceed
```

6.4 Tracing methods

You can also trace methods (primary and auxiliary) within a generic function. The following example shows how to specify any qualifiers and specializers.

1. Type the following methods into the listener:

```
(defmethod foo (x)
  (print 'there))

(defmethod foo :before ((x integer))
  (print 'hello))
```

2. Next, trace only the second of these methods by typing the following definition spec.

```
(trace (method foo :before (integer)))
```

3. Test that the trace has worked by calling the methods in the listener:

```
CL-USER 226 > (foo 'x)

THERE
THERE

CL-USER 227 > (foo 4)
0 (METHOD FOO :BEFORE (INTEGER)) > (4)

HELLO
0 (METHOD FOO :BEFORE (INTEGER)) < (HELLO)

THERE
THERE

CL-USER 228 >
```

6.5 Trace variables

hcl:*max-trace-indent* *Variable*

The maximum indentation used during output from **trace**.

hcl:*trace-indent-width* *Variable*

The additional amount by which tracing output is indented upon entering a deeper level of nesting.

hcl:*trace-level* *Variable*

The current depth of tracing.

cl:*trace-output* *Variable*

The stream to which tracing sends its output by default.

hcl:*traced-arglist* *Variable*

The variable that holds the arguments given to the traced function.

hcl:*traced-results* *Variable*

The variable that holds the results from the traced function.

The following four variables allow the output produced by tracing to be printed in a style that is controlled separately from normal printing:

hcl:*trace-print-circle* *Variable*

The value to which **`*print-circle*`** is bound during output from **`trace`**.

hcl:*trace-print-length* *Variable*

The value to which **`*print-length*`** is bound during output from **`trace`**.

hcl:*trace-print-level* *Variable*

The value to which **`*print-level*`** is bound during output from **`trace`**.

hcl:*trace-print-pretty* *Variable*

The value to which `*print-pretty*` is bound during output from `trace`.

7

The Advice Facility

The advice facility provides a mechanism for altering the behavior of existing functions. As a simple application of this, you may supplement the original function definition by supplying additional actions to be performed before or after the function is called. Alternatively, you may replace the function with a new piece of code that has access to the original definition, but which is free to ignore it altogether and to process the arguments to the function and return the results from the function in any way you decide. The advice facility allows you to alter the behavior of functions in a very flexible manner, and may be used to engineer anything from a minor addition of a message, to a major modification of the interface to a function, to a complete change in the behavior of a function. This facility can be helpful when debugging, or when experimenting with new versions of functions, or when you wish to locally change some functionality without affecting the original definition.

Note: It can be very dangerous to put advice on functions defined by the system.

Each change that is required should be specified using the **`defadvice`** macro. This defines a new body of code to be used when the function is called; this piece of code is called a piece of advice. Consider the following example:

```
CL-USER 71 > (defadvice
             (reverse print-advice :before)
             (the-list)
             (format t
               "~%** Calling reverse on ~S **"
               the-list))
NIL

CL-USER 72 > (reverse '(l a m i n a))

** Calling reverse on (L A M I N A) **
(A N I M A L)
```

In the above example you decided to print a message each time **reverse** is called. You called **defadvice** with a description of the function you wanted to alter, a name for the piece of advice, and the keyword **:before** to indicate that you want the code carried out before **reverse** is called. The rest of the call to **defadvice** specifies the additional behavior required, and consists of the lambda-list for the new piece of advice and its body (the lambda-list may specify keyword parameters and so forth). The advice facility arranges that **print-advice** is invoked whenever **reverse** is called, and that it receives the arguments to **reverse**, and that directly after this the original definition of **reverse** is called.

Pieces of advice may be given to be executed after the call by specifying **:after** instead of **:before** in the call to **defadvice**. So if you wished to add further code to be performed after **reverse** you could continue the session above as follows:

```
CL-USER 73 > (defadvice
               (reverse after-advice :after)
               (the-list)
               (format t
             "~%** After calling reverse on ~S **"
              the-list))
NIL

CL-USER 74 > (reverse '("which" "way" "round"))


** Calling reverse on ("which" "way" "round") **

** After calling reverse on ("which" "way" "round") ** ("round"
"way" "which")
```

```
CL-USER 75 >
```

Note that **after-advice** also receives the arguments to **reverse** when it is called.

7.1 Combining the advice

We have already seen how a before and an after piece of advice may be combined, and this section describes the general algorithm. There are three types of advice: *before*, *after* and *around*. These resemble before, after and around methods in CLOS. There may be several pieces of each type of advice present for a particular function.

The first step in working out how the combination is done is to order the pieces of advice. All the around advice comes first, then all the before advice, then the original definition, and lastly the after advice. The order within each of the around, before and after sections defaults to the order in which the pieces of advice were defined (that is most recent first). See the description of **defadvice** in the *LispWorks Reference Manual* for details of how to control the ordering of advice within each section.

The remainder of this section discusses what happens when a function that has advice is called.

7.1.1 :before and :after advice

First we deal with the case when there is no around advice present. Here each of the pieces of before advice are called in turn, with the same arguments that were given to the function, next the original definition is called with these arguments, and finally each of the pieces of after advice is called in reverse order with the same arguments (so that by default the most recently added piece of after advice is invoked last). The results returned by the function call are the values produced by the last piece of after advice to be called (if there is one), or by the original definition (if there is no after advice).

Note that none of these bits of code should destructively modify the arguments that they receive. Adding a piece of before advice thus provides a simple way of specifying some additional action to be performed before the original definition, and before any older bits of before advice. Adding a piece

of after advice allows you to specify extra actions to be performed after the original definition, and after any older bits of after advice. The advice facility automatically links together these bits of advice with the original function definition.

7.1.2 :around advice

Next we shall discuss the use of around advice, which provides you with greater control than do before and after advice. Let us suppose that a function that has some around advice is called. The arguments to the function are passed to the code associated with the first piece of around advice in the ordering, and the values returned by that piece of advice are the results of the function. There is no requirement for the advice to invoke any other pieces of advice, nor to call the original definition of the function.

However the code for any piece of around advice has access to the next member of the ordering, which it may invoke any number of times by calling **`call-next-advice`**. So it is possible for each piece of around advice to call its successor in the ordering if this is desired, and then the bits of around advice are called in turn in a similar fashion to our earlier description for before and after advice. However in the case of around advice the decision whether or not to call the next piece of advice is directly under your control, and you are free to modify the arguments received by the piece of advice, and to choose the arguments to be given to the next piece of advice if it is called.

If the last piece of around advice in the ordering calls **`call-next-advice`**, then it invokes the combination of before and after advice and the original definition that was discussed earlier. That is, the arguments to the call are given in the sequence described above to each of the before pieces of advice, then to the original definition and then to the after pieces of advice. The call to **`call-next-advice`** returns with the values produced by the last of these subsidiary calls, and the around advice may use these values in any way.

7.2 Removing advice

The macro **`delete-advice`** (or the function **`remove-advice`**) may be used to remove a named piece of advice. Since several pieces of advice may be

attached to a single functional definition, the name must be supplied to indicate which one is to be removed.

```
CL-USER 40 > (delete-advice reverse after-advice)
NIL

CL-USER 41 > (delete-advice reverse print-advice)
NIL
```

7.3 Advice for macros and methods

As well as attaching advice to ordinary functions, it may also be attached to macros and methods.

In the case of a macro, advice is linked to the macro's expansion function, and so any before or after advice receives a copy of the arguments given to this expansion function (normally the macro call form and an environment). A simple example:

```
CL-USER 45 > (defmacro twice (b) '(+ ,b ,b))
TWICE

CL-USER 46 > (defadvice
         (twice before-twice :before)
         (call-form env)
         (format t
         "~%Twice with environment ~A and call-form  ~A"
              env call-form))
NIL

CL-USER 47 > (twice 3)
Twice with environment NIL and call-form (TWICE 3)
6
```

Note that the advice is invoked when the macro's expansion function is used. So if the macro is present within a function that is being compiled, then the advice is invoked during compilation of that function (and not when that function is finally used).

In the case of a method, the call to **defadvice** must also specify precisely to which method the advice belongs. A generic function may have several methods, so the call to **defadvice** includes a list of classes. This must correspond exactly to the parameter specializers of one of the methods for that generic function, and it is to that method that the advice is attached. For example:

```
CL-USER 45 > (progn
             (defclass animal ()
              (genus habitat description
              (food-type :accessor eats)
              (happiness :accessor how-happy)
              (eaten :accessor eaten :initform nil)))
             (defclass cat (animal)
              ((food-type :initform 'fish)))
             (defclass elephant (animal)
              (memory (food-type :initform 'hay)))
             (defmethod feed ((animal animal))
              (let ((food (eats animal)))
             (push food (eaten animal))
             (format t "~%Feeding ~A with ~A" animal
               food)))
            (defmethod feed ((animal cat))
             (let ((food (eats animal)))
            (push food (eaten animal))
            (push 'milk (eaten animal))
            (format t "~%Feeding cat ~A with ~A and ~A"
                 animal food 'milk)))
            (defvar *cat* (make-instance 'cat))
            (defvar *nellie* (make-instance 'elephant)))
*NELLIE*

CL-USER 46 > (feed *cat*)
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL

CL-USER 47 > (feed *nellie*)
Feeding #<ELEPHANT 71e7bc> with HAY
NIL

CL-USER 48 > (defadvice
             ((method feed (animal))
              after-feed :after)
             (animal)
             (format t "~%~A has eaten ~A"
                       animal (eaten animal)))
NIL
CL-USER 49 > (defadvice
             ((method feed (cat))
              before-feed :before)
             (animal)
             (format t "~%Stroking ~A" animal)
             (setf (how-happy animal) 'high))
NIL
```

```
CL-USER 50 > (feed *cat*)
Stroking #<CAT 6f35d4>
Feeding cat #<CAT 6f35d4> with FISH and MILK
NIL

CL-USER 51 > (feed *nellie*)
Feeding #<ELEPHANT 71eb7c> with HAY
#<ELEPHANT 71eb7c> has eaten (HAY HAY)
```

7.4 Examples

So far you have only seen examples of before and after pieces of advice. This section contains some further examples. Suppose that you define a function **alpha** that squares a number, and then decide that you intended to return the reciprocal of the square instead. You might proceed as follows.

```
CL-USER 30 > (defun alpha (x) (* x x))
ALPHA

CL-USER 31 > (defadvice
             (alpha reciprocal :around)
             (num)
             (/ (call-next-advice num)))
NIL

CL-USER 32 > (alpha -5)
1/25
```

First you change **alpha** to return the reciprocal of the square. Do this by defining an around method to take the reciprocal of the result produced by the next piece of advice (which initially is the original definition). Now suppose that you later decide that you would like **alpha** to return the sum of the squares of the reciprocals in a certain range. You can achieve this by adding an extra layer of around advice. This must iterate over the range required, summing the results obtained by the calls to the next piece of advice (which currently yields the reciprocal of the square of its argument).

```
CL-USER 36 > (defadvice
             (alpha sum-over-range :around)
             (start end)
             (loop for i from start upto end
               summing (call-next-advice i)))
NIL

CL-USER 37 > (alpha 2 5)
1669/3600
```

Note that **alpha** now behaves as a function requiring two arguments; the outer piece of around advice determines the external interface to the function, and uses the inner pieces of advice as it needs - in this case invoking the inner advice a variable number of times depending on the range specified. The use of the words “outer” and “inner” corresponds to earlier and later pieces of around advice in the ordering discussed above, but is more descriptive of their behavior.

You now realize that taking the reciprocal of zero gives an error. You decide that you wish to generate an error if **alpha** is called in such a way as to cause this, but that you want to generate the error yourself. You also decide to add a warning message for negative arguments. As you want these actions to be performed as the last (that is innermost) in the chain of around advice, you specify this in the call to **defadvice** by giving it a **:where** keyword with value **:end**.

```
CL-USER 41 > (defadvice
             (alpha zero-or-negative
               :around :where :end)
             (x)
             (unless (plusp x)
             (format t
            "~%**Warning: alpha is being called with ~A"
                x))
             (if (zerop x)
              (error "Alpha cannot be called with zero")
             (call-next-advice x)))
NIL

CL-USER 42 > (alpha -5 -2)

**Warning: alpha is being called with -5
**Warning: alpha is being called with -4
**Warning: alpha is being called with -3
**Warning: alpha is being called with -2
1669/3600

CL-USER 43 > (alpha 0 3)

**Warning: alpha is being called with 0
Error: alpha cannot be called with zero
 1 (abort) return to level 0.
 2 return to top loop level 0

Type :c followed by a number to proceed

CL-USER 44 : 1 > :a
```

Finally you decide to alter **alpha** yet again, this time to produce approximations to π. $\pi^2/6$ is the sum of the reciprocals of the squares of all the positive integers. So you can generate an approximation to π using the sum of the reciprocals of the squares of the integers from one to some limit. (In fact this is not an efficient way of calculating π, but it could be of interest.)

```
CL-USER 51 > (defadvice
                 (alpha pi-approximation :around)
                 (limit)
               (sqrt
                (* 6
                   (call-next-advice 1.0 limit))))
NIL
```

Next, try calling the following in turn:

```
(alpha 10.0)
(alpha 100.0)
(alpha 1000.0)
pi
```

Lastly, here is a simple example showing a use of advice with an **&rest** lambda list:

```
(defun foo (a b c)
  (print (list a b c)))

(defadvice (foo and-rest-advice :around) (&rest args)
  (format t "advice called with args ~S" args)
  (apply #'call-next-advice args))
```

7.5 Advice functions and macros

The main functions used for advice are introduced below. See the *LispWorks Reference Manual* for full details.

The main macro used to define new pieces of advice is **defadvice**.

Pieces of around advice should use the macro **call-next-advice** to invoke the next piece of advice. As explained earlier this either calls the next piece of around advice (if one exists), or calls the combination of before advice, the original definition, and after advice. It may only be called from within the body of the around advice.

To remove a piece of advice, use the macro **delete-advice** or the function **remove-advice**.

8

Dspecs: Tools for Handling Definitions

The dspec system is the machinery underlying the way definitions are named in LispWorks. It supports program development by tracking the locations of definitions, and is also used in tracing and advising functions.

This chapter explains the concepts underlying dspecs and their use in tracking locations of definitions. For full details of the programming interface, see the **dspec** chapter in the *LispWorks Reference Manual*.

8.1 Dspecs

Definition specifications, or *dspecs*, are a systematic way of naming definitions.

Most named definitions are global, but local functions can have names, and some of the operations described here can be applied to them as well.

Here are three examples of dspecs:

```
car
(setf car)
(defclass standard-object)
```

A dspec is simply a name: you can operate on it even if the thing named does not currently exist.

8.2 Forms of dspecs

A dspec is one of:

- A symbol
- A **setf** function name
- A list starting with a symbol naming the class of definition (**method** or **defstruct** for example).

A symbol which is used as a dspec always names a function or a macro.

(setf foo) is a name for a setf function.

Note: **nil** is not a legal dspec, because it cannot have a function definition. Therefore when a dspec API returns **nil**, this should be interpreted in the usual way as "not found" or "not applicable".

8.2.1 Canonical dspecs

Internally, dspecs are handled in the canonical form:

```
(dspec-class primary-name . qualifiers)
```

where *dspec-class* in the canonical name of the class, and *qualifiers* is a proper list. *primary-name* is typically a symbol, but can be a list (in the case of a **setf** function) or a string (in the case of a package). The equality for canonical dspecs is **equal**.

As an example the general form of a **defmethod** dspec is:

```
(defmethod name qualifiers (specializer*))

name        := symbol | (setf symbol)
qualifiers  := qualifier | (qualifier qualifier*)
qualifier   := symbol
specializer := symbol | (eql object)
```

Functions in the dspec API accept non-canonical dspecs. All dspec functions, except **dspec:prettify-dspec**, **dspec:find-dspec-locations**, **dspec:name-definition-locations**, **dspec:dspec-definition-locations** and **dspec:find-name-locations** return canonical dspecs.

8.3 Dspec namespaces

Dspec classes are the namespaces for dspecs. Class names are often the same as the name of the defining form, though documentation types as defined for **`documentation`** are also used. See “Details of system dspec classes and aliases” on page 73 for a list of the classes.

8.3.1 Dspec classes

Dspec classes provide a set of handlers, to allow uniform handling of different types of definitions by other parts of the system, such as the editor and various browsers.

The most important handlers are those for testing if a dspec is currently defined and for undefining a dspec. See **`dspec:dspec-defined-p`** and **`dspec:dspec-undefiner`** in the *LispWorks Reference Manual* for details.

New dspec classes are defined using **`dspec:define-dspec-class`**.

Dspec classes can be subclassed. The top-level classes correspond to distinct global namespaces (such as **`variable`** for variables and constants and **`function`** for functions and macros), and at each level, all the subclasses are distinct from each other (but they do not have to form a complete partition of the superclass). See "7.5 Details of dspec classes and aliases" for the full hierarchy of system-provided classes.

You are allowed to define new top-level classes and subclass them, but you cannot add new subclasses to a system-provided class. However, see "7.3.2 Dspec aliases" for how to add new ways of making existing definitions.

8.3.1.1 Complete example of a top-level dspec class

Define a **`saved-value`** object which has a name and a value:

```
(defstruct saved-value
  name
  value)
```

The objects are defined using **`def-saved-value`** and stored on the plist of their name:

```
(defmacro def-saved-value (name value)
  `(dspec:def (def-saved-value ,name)
    (when (record-definition `(def-saved-value ,',name)
                             (dspec:location))
      (setf (get ',name 'saved-value)
            (make-saved-value :name ',name
                              :value ,value))
      ',name)))
```

Define a function to retrieve the **saved-value** object:

```
(defun find-saved-value (name)
  (get name 'saved-value))
```

Define a macro to access a **saved-value** object:

```
(defmacro saved-value (name)
  `(saved-value-value (find-saved-value ',name)))
```

Define a dspec class for **def-saved-value** dspecs:

```
(dspec:define-dspec-class def-saved-value nil
  "Defined saved values"
  :definedp
  #'(lambda (name)
      ;; Find any object that def-saved-value recorded
      (not (null (find-saved-value name))))
  :undefiner
  #'(lambda (dspec)
      ;; Remove what def-saved-value recorded
      `(remprop ,(dspec:dspec-name dspec) 'saved-value))
  :object-dspec
  #'(lambda (obj)
      ;; Given a saved-value object, we can reconstruct its dspec
      (and (saved-value-p obj)
           `(def-saved-value ,(saved-value-name obj)))))
```

For completeness, define a form parser that generates dspecs from forms:

```
(dspec:define-form-parser
    (def-saved-value
     (:parser dspec:single-form-form-parser)))
```

Note: this form parser for **def-saved-value** is not strictly necessary, because the system provides an implicit form parser which recognizes definitions beginning with "def".

8.3.1.2 Example of subclassing

This example is based on that in "Complete example of a top-level dspec class" on page 67.

Define a **computed-saved-value** object has a function to compute the value the first time:

```
(defstruct (computed-saved-value (:include saved-value))
  function)
```

saved-value objects are defined using def-computed-saved-value and stored on the plist of their name:

```
(defmacro def-computed-saved-value (name function)
  `(dspec:def (def-computed-saved-value ,name)
     (when (record-definition `(def-computed-saved-value ,',name)
                              (dspec:location))
       (setf (get ',name 'saved-value)
             (make-computed-saved-value :name ',name
                                        :function ,function))
       ',name)))
```

Define a function to compute a **computed-saved-value**:

```
(defun ensure-saved-value-computed (name)
  (let ((saved-value (find-saved-value name)))
    (or (saved-value-value saved-value)
        (setf (saved-value-value saved-value)
              (funcall
               (computed-saved-value-function saved-value))))))
```

Define a macro to access a **computed-saved-value**:

```
(defmacro computed-saved-value (name)
  `(ensure-saved-value-computed ',name))
```

Define a dspec class for **def-computed-saved-value** dspecs:

```
(dspec:define-dspec-class def-computed-saved-value def-saved-
value
  "Defined computed saved values"
  :definedp
  #'(lambda (name)
      ;; Find any object that def-computed-saved-value recorded
      (computed-saved-value-p (find-saved-value name)))
  ;; The :undefiner is inherited from the superspace.
  :object-dspec
  #'(lambda (obj)
     ;; Given a computed-saved-value object,  we can reconstruct
its dspec
      (and (computed-saved-value-p obj)
           `(def-computed-saved-value ,(saved-value-name obj)))))
```

For completeness, define a form parser that generates dspecs from forms:

```
(dspec:define-form-parser
    (def-computed-saved-value
     (:parser dspec:single-form-form-parser)))
```

Note: this form parser for **`def-computed-saved-value`** is not strictly necessary, because the implicit form parser will recognize definitions beginning with "def".

8.3.2 Dspec aliases

You can add new ways of making existing definitions and use the dspec system to track these definitions. This is what happens when your defining form expands into a system-provided form. The macro **`dspec:define-dspec-alias`** is used to inform the dspec system of this.

8.4 Types of relations between definitions

8.4.1 Functionally equivalent definers

When one definition form simply macroexpands into another, or otherwise has an identical effect as far as the dspec system is concerned, the dspec system should consider them variant forms of the same class.

Use **`dspec:define-dspec-alias`** to convert one definer to the other during canonicalization. A pre-defined example of this in LispWorks is **`defparameter`**

and **defvar**. These cannot be distinguished (other than in the source code), so **defparameter** has been defined as a dspec alias for **defvar**. However, **defvar** and **defconstant** are distinct kinds of variable, since we can easily tell which type of definition is in effect by calling the function **constantp**. To define their dspecs, LispWorks creates a dspec class called **variable** and uses it as the superspace argument when defining the **defvar** and **defconstant** dspec classes.

As an explicit example, suppose you have a defining macro

```
(defmacro parameterdef  (value name)
  `(defparameter ,name ,value))
```

then

```
(dspec:define-dspec-alias parameterdef (value name)
  `(defparameter ,name))
```

would be a suitable appropriate alias definition. This **define-dspec-alias** form defines the dspec.

dspec:define-dspec-alias is like **defmacro** for dspecs, so it could be used to describe complicated conversions, as long as it can be done purely statically and totally in terms of existing dspecs. However, nothing more complicated than **defparameter** has been found necessary.

8.4.2 Grouping subdefinitions together

Some definition forms are macros that expand into a group of other definitions, for example **defstruct**. When the form is associated with a dspec class, the subdefinitions can be automatically recorded as being subforms of the new definition, by use of the **dspec:def** macro.

This means that the dspec system knows that the subdefinitions were inside the main definition (indeed, inside this particular form). Therefore

- Location queries can retrieve this information.
- The source location commands in the LispWorks IDE, when passed a subdefinition, know to search for the main definition given in the **dspec:def**.

Note: to make source location work you will also need a **`dspec:define-form-parser`** definition for the macro that expands into the **`dspec:def`**.

Note: **`dspec:def`** defines a relation between two particular definitions, for example **`(defstruct foo)`** and **`(defun make-foo)`**, not between the two dspec classes.

8.4.3 Distributed definitions

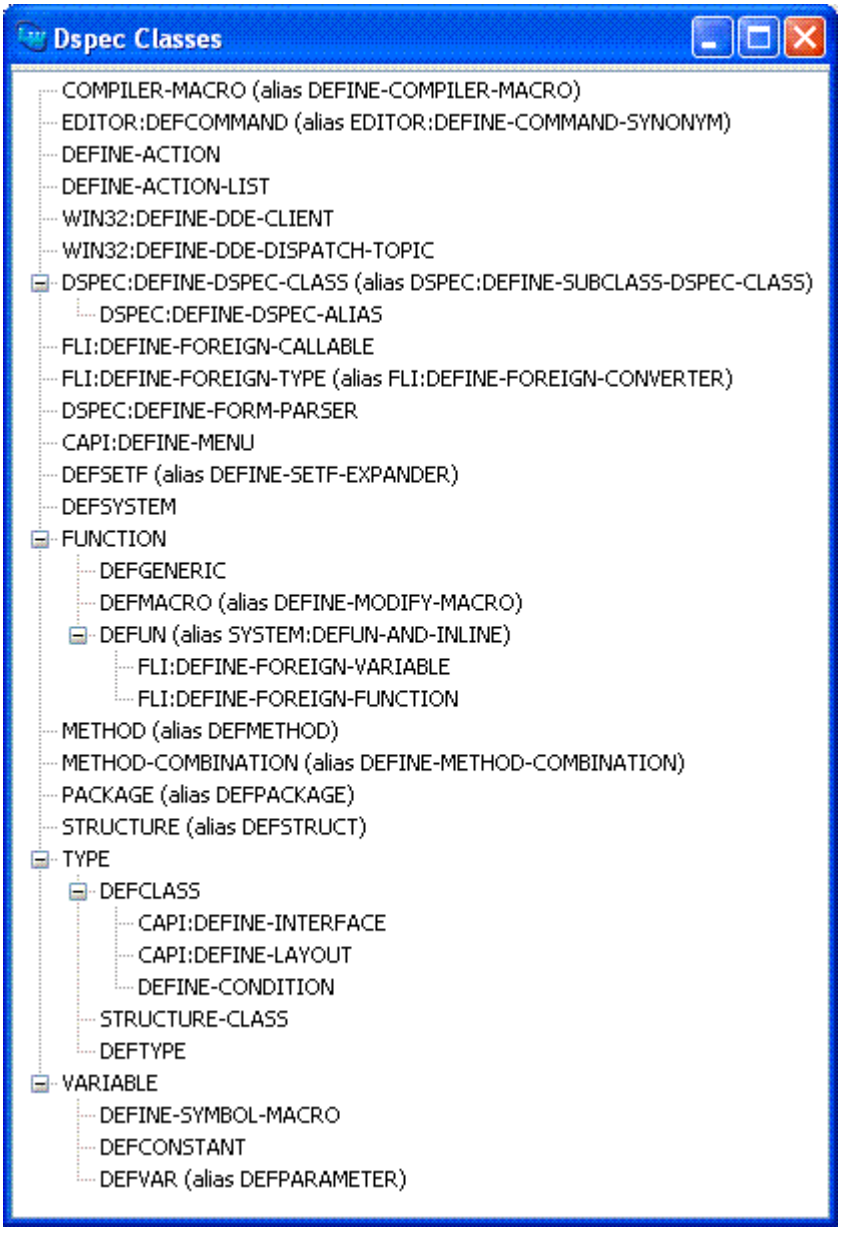
Some definitions are additions to another class of definition, for example methods are additions to generic functions. We call these *distributed definitions*, consisting of "parts" and "the aggregate".

The primary name of a part gives the primary name of the aggregate it is a part of, and the qualifiers distinguish it from the other parts of the same aggregate. Only a part dspec may have qualifiers.

8.5 Details of system dspec classes and aliases

This section shows the dspec classes, subclasses and aliases provided by the system. Following the illustration there are notes about some of these classes. Further dspec classes are provided by modules such as **`kw`** and **`sql`**.

Figure 8.1 Dspec classes and aliases



The canonical form of a symbol dspec is **(function *symbol*)** and the canonical form of a setf function name dspec is **(function (setf *symbol*))**.

8.5.1 CLOS dspec classes

defgeneric and **method** can handle **standard-generic-function** and **standard-method**.

8.5.2 Part Classes

method is a part class for **defgeneric**.

compiler-macro is a part class for **function**.

8.5.3 Foreign callable dspecs

For **fli:define-foreign-callable** the canonical name is the foreign name, with any machine-specific prefixes omitted.

8.6 Subfunction dspecs

For some purposes, we allow dspecs that do not name a global definition, but a local function. These are of the form

(subfunction *name parent*)

where *parent* is another dspec (possibly even a **subfunction** dspec).

name is a symbol, a list, or a number, but it is not used for anything within the dspec system. A **subfunction** dspec can be canonicalized and prettified, and passed as an argument to **dspec:dspec-definition-locations** (which will find where *parent* is defined).

Additionally pseudo-dspecs like this are allowed for top-level forms:

(top-level-form (*location* <#>))

location is a basic location and <#> identifies the top-level form within that location. These are used as parent dspecs in **subfunction** dspecs and **:inside** locations. These dspecs can be canonicalized and prettified, and can be returned as dspecs from the location finders.

8.7 Tracking definitions

The dspec system is used to keep track of global definitions in many ways, and global definition macros usually tell the dspec system when the definition changes.

The main purpose of the system is to keep track of where the definition was located, but it also allows fine-tuned control of redefinitions.

8.7.1 Locations

Locations are mainly something the dspec system just stores and retrieves. **`:inside`** locations are used to describe definitions located as subforms of other definitions.

`:inside` locations are usually not explicitly specified, but arise as a result of having two nested definitions, both of which use the **`dspec:def`** and **`dspec:location`** macros to handle the name and location info.

The types of locations and their meanings are:

A pathname
: A definition existed in the file named or an editor buffer with that name.

The keyword **`:listener`**
: A definition was executed interactively in the listener or an editor buffer not associated with a file.

The keyword **`:unknown`**
: A definition was found in the image (these are entered when a location query does not find any information already in the database).

The keyword **`:implicit`**
: A definition for a part was recorded, but no information exists for the aggregate.

8.7.2 Recording definitions and redefinition checking

The location information is entered into the database when the definition is executed, by the defining function calling **`dspec:record-definition`**.

dspec:record-definition performs various checks, and returns true or false depending on whether the definition was allowed or not. In particular, it checks if the same name has already been defined in a different location and if so a warning or error can be signalled. See the entry in the *LispWorks Reference Manual* for details.

8.7.2.1 Use of record-definition

You should not usually call **dspec:record-definition**, since all the system-provided definers call it.

However, for new classes of definition which you add with **dspec:define-dspec-class**, you should call **dspec:record-definition** for dspecs in their new classes, as shown in "Complete example of a top-level dspec class" on page 67.

8.8 Finding locations

There are two ways of retrieving location information for definitions in the running LispWorks image:

- query for a dspec using **dspec:dspec-definition-locations**, or
- query for a name in a given set of namespaces using **dspec:name-definition-locations**

The difference is that name queries will find the locations of all the part definitions as well as the definition named, whereas dspec queries will only find the locations for the definition named (there might be many if it has been redefined).

To provide for sub-definitions hidden in another definition, such as **defstruct** accessors, all location queries produce a list of pairs of dspecs and locations, each pair naming a definition within the corresponding location that contains the definition looked for. So a query for an accessor called **foo-bar** might produce the pair:

```
((defstruct foo) #P"/usr/users/hacker/hacks/hack.lisp")
```

8.9 Users of location information

To find location information for definitions made in the running image or recorded in a tags database or a tags file:

- query for a dspec using **dspec:find-dspec-locations**, or
- query for a name in a given set of namespaces using **dspec:find-name-locations**

The extent of the search is controlled by the value of the variable **dspec:*active-finders***. See the *LispWorks Reference Manual* for details.

For example, to obtain the locations of the definitions of foo acros all dspec namespaces, call

```
(dspec:find-name-locations dspec:*dspec-classes* 'foo)
```

Another example of the use of **dspec:find-name-locations** is the LispWorks Editor tool's Find Definitions tab.

8.9.1 Finding definitions in the LispWorks editor

Returning to our example definer

```
(defmacro parameterdef  (value name)
  `(defparameter ,name ,value))
```

1. Load a file **foo.lisp** containing

   ```
   (parameterdef 42 *foo*)
   ```

2. Now use **Expression > Find Source** on the symbol ***foo***. Notice that LispWorks knows which file the definition is in, but cannot find the defining top level form.

   Also notice that the Definitions tab of the Editor tool does not display the definition of ***foo***. This is because the Editor does not recognise **parameterdef** as a definer.

3. Now evaluate these forms to associate a parser with **parameterdef** and inform the dspec system that **parameterdef** is another way of naming a **defparameter** dspec:

```
(dspec:define-form-parser parameterdef (value name)
  `(parameterdef ,name))

(dspec:define-dspec-alias parameterdef (name)
  '(defparameter ,name))
```

4. Now use **Expression > Find Source** on the symbol **`*foo*`** again. Notice that the source of the definition of **`*foo*`** is displayed correctly in the text tab of the Editor tool, and that the Definitions tab displays the definition as

   ```
   (parameterdef *foo*)
   ```

8.9.2 The editor's implicit form parser

When testing your form parsers bear in mind that the LispWorks editor has an implicit form parser, independent of explicit parsers defined in the dspec system. It tries to parse a dspec from a top level form which is of length 2 or more and whose car has symbol name beginning with **`"DEF"`**. That is:

```
(defxyz name forms)
```

gets parsed as

```
(defxyz name)
```

which may be a dspec (and thus provides a match for the source location commands). This mechanism operates only when there's no explicit parser defined for **`defxyz`**.

The editor's implicit form parser is useful because it matches a common simple case. However it does not work for the **`parameterdef`** example, because that definer's symbol name does not begin with **`"DEF"`**.

8.9.3 Reusing form parsers

The form parser established above was specifically for **`parameterdef`** forms. However if you have other definers of similar syntax - in this example, definers for which the name is the second subform - then you can define a form parser which can be associated with each of them, as follows:

```
(dspec:define-form-parser (name-second (:anonymous t))
    (value name)
  `(,name-second ,name))
```

Note that the *name-second* variable is evaluated in the body of the parser. Supposing you have another defining macro **constantdef**:

```
(defmacro constantdef (value name)
  `(defconstant ,name ,value))
```

then you can associate the same parser with both this and **parameterdef**:

```
(dspec:define-form-parser (parameterdef
                           (:parser name-second-form-parser)))

(dspec:define-form-parser (constantdef
                           (:parser name-second-form-parser)))
```

9

Action Lists

Action-lists are a unified approach to various different mechanisms for running initializations, or “hook” functions at various points during the life of the system. They provide central gathering points for applications to trigger on system-wide events such as start-up, disk-save, and so on.

An action-list is a tagged list of data, to be executed (in some sense) in sequence whenever the circumstance identified by its tag occurs. It is expected that whatever code detects or causes the circumstance will take care of running the action-list.

An execution-function can be specified for the action-list when it is created. Otherwise, the default behavior is to treat the data of each action as a callable and apply it to any additional arguments specified at execution time. At its simplest, an action-list emulates **`(map nil 'funcall)`**.

Names of action-lists and action-items are general lisp objects, compared with **`equalp`**. This allows strings and other objects to be used as unique identifiers.

Actions can be specified to depend on other actions; when defining an action-item, you can say that it must be before or after other action-items using the **`:before`** and **`:after`** keywords. Aside from that, actions are assumed to have no dependencies, and no order of execution should be counted on for the actions in a list.

You can (and are encouraged to) specify a documentation string for action-lists or action-items.

In addition you can create action-lists that are not registered globally. This allows applications to have disembodied action lists for their own internal purposes. The other action-list functions allow an action-list to be passed in instead of a name, to accommodate this.

9.1 Defining and undefining action lists

Action lists are defined using the **`define-action-list`** macro, and are undefined using the **`undefine-action-list`**. It is also possible to make unnamed, unregistered lists using **`make-unregistered-action-list`**.

define-action-list *Macro*

`define-action-list` ***uid*** **`&key`** ***documentation sort-time dummy-actions default-order execution-function***

The **`define-action-list`** macro defines an action list.

uid is a unique identifier, and is a general lisp object, to be compared by **`equalp`**. It names the list in the global registry of lists. See **`make-unregistered-action-list`** to create unnamed, "unregistered" action-lists. The *uid* may be quoted, but is not required to be. It is possible, but not recommended, to define an action list with unique identifier **`nil`**. If a registered action-list with the *uid* already exists (that is, one which returns **`t`** when compared with **`equalp`**), then notification and subsequent handling is controlled by the value of the **`*handle-existing-action-list*`** variable.

The *documentation* string allows you to provide documentation for the action list.

sort-time is a keyword specifying when added actions are sorted for the given list — either **`:execute`** or **`:define-action`**.

dummy-actions is a list of action-names that specify placeholding actions; they cannot be executed and are constrained to the order specified in this list, for example

'(:beginning :middle :end)

default-order specifies default ordering constraints for subsequently defined action-items where no explicit ordering constraints are specified. An example is

'(:after :beginning :before :end)

execution-function specifies a user-defined function accepting arguments of the form:

(*the-action-list other-args-list* &rest *keyword-value-pairs*)

where the two required arguments are the action-list and a list of additional arguments passed to **execute-actions**, respectively. The remaining arguments are any number of keyword-value pairs that may be specified in the call to **execute-actions**. If no execution function is specified, then the default execution function will be used to execute the action-list.

undefine-action-list *Macro*

undefine-action-list *uid*

The **undefine-action-list** flushes the specified list (and all its action-items). If the action-list specified by *uid* does not exist, then handling is controlled by the value of the ***handle-missing-action-list*** variable.

When defining an action-list, the user may provide an associated execution-function. When executing the action-list, this user-defined execution-function is used instead of the default execution-function, to map over and “execute” the action-list's action-items. The macro **with-action-list-mapping** provides facilities to map over action-items (that is, their corresponding “data”). In addition, the with-action-list-error-handling macro provides a simple mechanism to trap errors and print warnings while executing each action-item.

All execution-functions are required to accept arguments of the form:

```
(action-list other-args &rest keyword-value-pairs)
```

where the two required arguments are the action-list and the list of additional arguments passed to execute-actions (see above), respectively. The remaining arguments are any number of keyword-value pairs that may be specified in

the call to execute-actions. See the LispWorks Reference Manual entries for **`with-action-list-mapping`** and **`with-action-item-error-handling`** for examples of execution-functions.

Actions are added to an action list using **`define-action`**, and are removed using **`undefine-action`**.

define-action *Macro*

`define-action` ***name-or-list action-name data*** **`&rest`** ***specs***

This macro adds a new action to the list specified by *name-or-list*, which will be executed according to the action list's execution function.

undefine-action *Macro*

`undefine-action` ***name-or-list action-name***

This macro removes the action specified by *action-name* from the list specified by *name-or-list*.

9.2 Exception handling variables

The following global variables are used to control the handling of exceptions:

handle-existing-action-list *Variable*

A list containing either **`:warn`** or **`:silent`**, determining whether to notify the user, and either **`:skip`** or **`:redefine`** to determine what to do about an action-list operation when the action-list already exists. The default value is **`'(:warn :skip)`**. It is used by the **`define-action-list`** macro.

handle-existing-action-in-action-list *Variable*

A list containing one of **`:warn`**, or **`:silent`**, determining whether to notify the user, and one of **`:skip`**, or **`:redefine`**, to determine what to do about an action definition when the action already exists in the given action-list. The default value is **`'(:warn :redefine)`**. It is used by **`define-action`**.

handle-missing-action-list *Variable*

A keyword; one of **:warn**, **:error**, or **:ignore**, denoting how to handle an operation on a missing action-list. The default value is **:error**. It is used by **undefine-action-list**, **print-actions**, **execute-actions**, **define-action** and **undefine-action**.

handle-missing-action-in-action-list *Variable*

A keyword; one of **:warn**, **:error** or **:ignore**, denoting how to handle an operation on a missing action. Its default value is **:warn**. It is used by **undefine-action**.

9.3 Other variables

default-action-list-sort-time *Variable*

Contains a keyword that is either **:execute** or **:define-action**, denoting when actions in action-lists are sorted (see **define-action-list** for an explanation of ordering specifiers). Actions are sorted either at time of definition (**:define-action**) or when their action-list is executed (**:execute**). The default sort time is **:execute**.

9.4 Diagnostic utilities

Two diagnostic functions are provided: **print-actions** which prints out the actions on an action list. and **print-action-lists**, which provides a list of all the defined action lists.

print-actions *Function*

print-actions ***name-or-list*** **&optional** ***stream***

Generates a listing of the action items on this action-list, in order. If the action-list specified by name-or-list does not exist, then this is handled according to the value of *handle-missing-action-list*.

print-action-lists *Function*

print-action-lists &optional stream

Generates a listing of all the action lists in the global registry. (The ordering of the action lists here is essentially random.)

9.5 Examples

This example illustrates "typical" use of action lists. The **define-action** forms might be scattered across several files (**mail-utilities.lisp**, **caffeine.lisp**, and so on). Each of the functions, such as **read-mail**, **dont-panic**, and so on, take one argument: **hassled-p**.

```
(in-package "CL-USER")

(define-action-list "On arrival at office"
    :documentation "Things to do in the morning"
    :dummy-actions '("Look busy")
    :default-order '(:before "Look busy"))

(define-action "On arrival at office" "Read mail" 'read-mail)

(define-action "On arrival at office" "Greet co-workers"
                 'say-hello)

(define-action "On arrival at office" "Drink much coffee"
                 'wake-up:after "Locate coffee machine")

(define-action "On arrival at office" "Locate coffee machine"
                 'dont-panic)

(defun my-morning (hassled-p Monday-p)
   (execute-actions ("On arrival at office"
                     :ignore-errors-p Monday-p)
                 hassled-p)
    <rest of my-morning code goes here>)
```

This example illustrates use of execution-functions and post-processing

```
(in-package "CL-USER")
```

Here are the implementation details, which are hidden from the "user".

```
(defstruct (thing (:constructor make-thing (name number)))
  name
  number)
```

```
(defvar *things*
  (make-unregistered-action-list :sort-time :define-action
         :execution-function 'act-on-things))

(defun do-things (function &optional post-process)
  (execute-actions (*things* :post-process post-process)
                   function))

(defun act-on-things (things other-args-list &key post-process)
  (with-action-list-mapping
   (things ignore thing post-process)
   (destructuring-bind
    (function) other-args-list
    (funcall function thing))))
```

The interface is given below. The internals of the mapping mechanism are hidden.

```
(defmacro define-thing (name number)
  (with-unique-names (thing)
    `(let ((,thing (make-thing ,name ,number)))
       (define-action *things* ',name ,thing))))

(defmacro undefine-thing (name)
  `(undefine-action *things* ,name))

(defun find-thing (name)
  (do-things #'(lambda (thing)
                   (and (equal name (thing-name thing))
                        thing))
               :or))

(defun add-things ()
  (reduce '+ (do-things 'thing-number :collect)))
```

9.6 Standard Action Lists

The following action lists are defined in LispWorks as shipped:

"When starting image" - Actions to be executed upon image startup.

"Confirm when quitting image" - Actions to be executed before the image quits. Every action must return non-**nil** as its first value, otherwise the quit will be aborted once the actions are complete.

"When quitting image" - Actions to be executed when the image quits, after success of the "Confirm when quitting image" actions.

"Initialize LispWorks Tools" - Things to do when the LispWorks IDE starts on a screen. You may customise your environment startup by defining actions on it.

"Delivery Actions" - Actions to be executed when doing delivery. Actions on this list are executed in a 'normal' environment. See the *Delivery User Guide* for an example action item.

10

The Compiler

The compiler translates Lisp forms and source files into binary code for the host machine. A compiled Lisp function, for instance, is a sequence of machine instructions that directly execute the actions the evaluator would perform in interpreting an application of the original source lambda expression. Where possible the behaviors of compiled and interpreted versions of the same Lisp function are identical. Unfortunately the definition of the Common Lisp language results in certain unavoidable exceptions to this rule. The compiler, for instance, must macroexpand the source before translating it; any side effects of macro-expansion happen only once, at compile time.

By using declarations, you can advise the compiler of the types of variables local to a function or shared across an application. For example, numeric operations on a variable declared as a **`single-float`** can be compiled as direct floating-point operations, without the need to check the type at execution time. You can also control the relative emphasis the compiler places on efficiency (speed and space), safety (type checking) and support for debugging. By default the compiler produces code that performs all the necessary type checking and includes code to recover from errors. It is especially important that the type declarations be correct when compiling with a safety level less than 3 (see later in this chapter for more details).

When compiling a Lisp source file, the compiler produces its output in a format that is much faster to load than textual Lisp source — the “fasl” or “fast-

load" form. Fasl files contain arbitrary Common Lisp objects in a pre-digested form. They are loaded without needing to use the expensive **read** function. A series of "fasl-loader" routines built into LispWorks interpret the contents of fasl files, building the appropriate objects and structures in such a way that objects that were **eq** before fasl-dumping are created **eq** when fasl-loaded.

Fasl files are given pathname extensions that reflect the target processor they were compiled for; as the fasl files contain processor specific instruction sequences it is essential that the loader be able to distinguish between files compiled for different targets. These pathname extensions always end in "fasl". See **dump-forms-to-file** in the *LispWorks Reference Manual* for details of all the possible fasl file extensions.

10.1 Compiling a function

The function **compile** takes a symbol as its first argument, and an interpreted function definition (a lambda expression) as its second, optional, argument. It compiles the definition and installs the resultant code as the symbol-function of the symbol (unless the symbol was **nil**). If the definition is omitted then the current symbol-function of the symbol is used. Below are some examples:

```
CL-USER 3 >  (compile (defun  fred (a b)

                       (dotimes (n a) (funcall b))))
; FRED
FRED
NIL
NIL

CL-USER 4 > (funcall (compile nil '(lambda (n)
                     (* n n)))  7)
; NIL
49

CL-USER 5 >  (compile  'ident-fun  '(lambda (x) x))
;IDENT-FUN
IDENT-FUN
NIL
NIL
```

10.2 Compiling a source file

The function **`compile-file`** takes a pathname as its argument and compiles all the forms in the file, producing a corresponding fasl file (with pathname derived from the source pathname). Any side effects in the source file are only felt once the compiled file is subsequently loaded. Many proclamations, for example, are not visible at compile time. The **`eval-when`** special form can be used to force such side effects to take effect at the time of compilation, rather than loading.

10.3 How the compiler works

Conceptually the compiler can be viewed as performing a series of separate passes.

- In the first pass the source code is macro expanded in the appropriate macro environment.
- A series of source to source optimizing transformations are performed to simplify the source tree. Type declarations are used to select specialized, efficient versions of low level functions.
- A graph is generated from the source tree. The structure of the graph reflects the flow of control in the tree. The nodes of the graph contain blocks of intermediate code for an abstract machine with byte addressing and an infinite set of registers. Register allocation is performed based on data flow analysis and machine specific rules concerning live ranges across code fragments.
- The blocks of intermediate code are translated into a single linear sequence of target machine code through a process of template matching.
- Finally the relative branch instructions are “fixed up” to point to the correct locations in the code sequence.

The compiler is in fact much more complex than this model might suggest. Machine specific optimizations, for example, can be included in any of the passes. The distinction between passes is also not as simple as that listed above. However, this description is sufficient to allow the programmer to make optimal use of the compiler.

10.4 Compiler control

There are ways to control the nature of compiled code via the **declare** special form and **proclaim** function. See later in this chapter for fuller discussion of these two forms.

In particular there are a set of optimize qualities which take integral values from 0 to 3, in order to control the trade-offs between code size, speed, compilation time, debuggability of the resulting code, and the safety of the code (whether type checks are omitted). For example:

```
(proclaim '(optimize (speed 3) (safety 0) (debug 0)))
```

tells the compiler to concentrate on code speed rather than anything else, and

```
(proclaim '(optimize (safety 3)))
```

ensures that the compiler never takes liberties with Lisp semantics and produces code that checks for every kind of signallable error.

The important declarations to the compiler are type declarations and optimize declarations. To declare that the type of the value of a variable can be relied upon to be unchanging (and hence allow the compiler to omit various checks in the code), say:

```
(declare (type the-type variable * )
```

Optimize declarations have various qualities, and these take values from 0 to 3. The names are **safety**, **fixnum-safety**, **float**, **sys:interruptable**, **debug**, **speed**, **compilation-speed**, and **space**.

Most of the qualities default to 1 (but **safety** and **fixnum-safety** default to 3 and **interruptable** defaults to 0). You can either associate an optimize quality with a new value (with local lexical scope if in **declare**, and global scope if **proclaim**), or just give it by itself, which implies the value 3 (taken to mean "maximum" in some loose sense).

Thus you ensure code is at maximum safety by:

```
(proclaim '(optimize (safety 3)))
```

or

```
(proclaim '(optimize safety))
```

and reduce debugging information to a minimum by

```
(proclaim '(optimize (debug 0)))
```

Normally code is interruptible, but when going for the extreme levels of speed and "undebuggability" this ceases to be the case unless you also ensure it thus:

```
(proclaim '(optimize (debug 0) (safety 0) (speed 3)
interruptable))
```

The levels of **safety** have the following implications:

- 0 implies no type checking upon reading or writing from defstructs, arrays and objects in general, nor any checking of array index bounds.
- 1 implies no type checking upon reading from defstructs, arrays and objects in general, nor any checking of array index bounds when reading. However, array index bounds are checked when writing.
- 2 implies type checking when writing, but not when reading. Other than this the compiler generates generally safe code, but allows **type** and **fixnum-safety** declarations to take effect. Array index bounds are checked for both reading and writing.
- 3 (default) implies complete type and bounds checking, and disallows **fixnum-safety** and **type** declarations from taking any effect.

The levels of **fixnum-safety** have the following implications:

- 0 implies no type checking of arguments to numeric operations, which are assumed to be fixnums. Also the result is assumed, without checking, to not overflow - this level means single machine instructions can be generated for most common integer operations, but risks generating values that may confuse the garbage collector.
- 1 implies that numeric operations do not check their argument types (assumed fixnum), but do signal an error if the result would have been out of range.
- 2 implies that numeric operations signal an error if their arguments are non-fixnum, and also check for overflow.

- 3 (default) implies complete conformance to the semantics of Common Lisp numbers, so that types other than integers are handled in compiled code.

Additionally if the level of **`float`** (really this should be called "float-safety") is 0 then the compiler reduces allocation during float calculations.

The effects of combining these qualities is summarized below:

Table 10.1 Combining debug and safety levels in the compiler

Keyword settings	Operations
`safety=0`	Array access optimizations
`debug>0`	Dumps symbol names for arglist
`debug>=2`	Ensure debugger knows values of args (and variable when source debugging is on)
`debug<1`	Does not generate any debug info at all
`debug=3`	Avoids **`make-instance`** and **`find-class`** optimizations
`debug=3`	Avoids **`gethash`** and **`puthash`** optimizations
`debug=3`	Avoids **`ldb`** and **`dpb`** optimizations
`debug=3`	Avoids an optimization to **`last`**
`safety>1`	Be careful when multiple value counts are wrong
`safety<1`	Do not check array indices during write
`safety<2`	Do not check array indices during read
`speed>space`	Inline map functions (unless **`debug>2`**)
`debug<=2`	Optimize (merge) tail calls
`debug<2` and **`safety<2`**	Self calls
`safety>=2`	Check get special
`safety<2`	Do not check types during write

Table 10.1 Combining debug and safety levels in the compiler

Keyword settings	Operations
safety<3	Do not check types during read
safety>=1	Check structure access
safety<=1	Inline structure readers, with no type check
safety=0	Inline structure writers, with no type check
debug>=1	Call count count
safety>1	Check number of args
safety>=1 or **interruptible>0**	Check stack overflow
safety>1	Ensures the thing being funcalled is a function
safety<3 and **fixnum-safety=2**	Fixnum-only arithmetic with errors for non fixnum arguments.
safety<3 and **fixnum-safety=1**	No fixnum overflow checks
safety<3 and **fixnum-safety=0**	No fixnum arithmetic checks at all
safety>2	**char=** checks for arguments of type character
safety>=2	Ensures symbols in **progv**
debug=3	Avoids “ad hoc” predicate type transforms
compilation-speed=3	Reuse virtual registers in very large functions
debug=3 and **safety=3**	**(declare (type foo x))** and **(the foo x)** ensure a type check
float=0	Optimize floating point calculations

The other optimize qualities are: **speed** — the attention to fast code, **space** — the degree of compactness, **compilation-speed** — speed of compilation, **interruptable** — whether code must be interruptible when unsafe.

Note that if you compile code with a low level of safety, you may get segmentation violations if the code is incorrect (for example, if type checking is turned off and you supply incorrect types). You can check this by interpreting the code rather than compiling it.

For example, the following function, compiled with safety = 2, does not check the type of its argument because it merely reads:

```
(defun foo (x)
  (declare (optimize (safety 2)))
  (car x))
```

However the following function, also compiled with safety = 2, does check the type of its argument because it writes:

```
(defun set-foo (x y)
  (declare (optimize (safety 2)))
  (setf (car x) y))
```

10.5 Declare, proclaim, and declaim

declare *Special form*

declare (*declaration* *)

There are two distinct uses of **declare**, one is to declare Lisp variables as "special" (this affects the semantics of the appropriate bindings of the variables), and the other is to provide advice to help the Common Lisp system (in reality the compiler) run your Lisp code faster, or with more sophisticated debugging options.

The special form **declare** behaves computationally as if it is not present (other than to affect the semantics), and is only allowed in certain contexts, such as after the variable list in a **let**, **do**, **defun**, etc.

(Consult the syntax definition of each special form to see if it takes declare forms and/or documentation strings.)

See the *LispWorks Reference Manual* for more detailed information on **declare**, including some LispWorks extensions to Common Lisp.

proclaim *Function*

proclaim ***declaration-list***

declaration-list must be a list of declaration forms to be put into immediate and pervasive effect.

Unlike **declare**, **proclaim** is a function that parses the declarations in the list (usually a quoted list, note), and puts their semantics and advice into global effect. This can be useful when compiling a file for speedy execution, since a proclamation such as:

```
(proclaim '(optimize (speed 3) (space 0) (debug 0)))
```

means that the rest of the file is compiled with these optimization levels in effect. (The other way of doing this is to make appropriate declarations in every function in the file).

proclaim simply returns **nil**.

declaim *Macro*

This is a macro equivalent to **proclaim**.

Below are some examples:

```
(proclaim '(special *fred*))
(proclaim '(type single-float x y z))
(proclaim '(optimize (safety 0) (speed 3)))
```

As **proclaim** involves parsing a list of lists of symbols and is intended to be used a few times per file, its implementation is not optimized for speed - it makes little sense to use it other than at top level.

Do not forget to quote the argument list if it is a constant list. **(proclaim (special x))** attempts to call function **special**.

10.5.1 Naming conventions

Exercise caution if you **`declare`** or **`proclaim`** variables to be special without regard to the naming convention that surrounds their names with asterisks.

10.6 Optimizing your code

Careful use of the compiler optimize qualities described above may significantly improve the performance of your code. However it is not recommended that you simply experiment with the effect of adding declarations. It is more productive to work systematically:

1. Use the Profiler, described in Chapter 12, "The Profiler", to analyse your application's performance and identify bottlenecks, then
2. Consider whether re-writing of parts of your source code would improve efficiency at the bottlenecks, and
3. Use **`:explain`** declarations to make the compiler generate optimization hints, and
4. Consider adding suitable declarations as described in this chapter to improve efficiency at the bottlenecks.

The remainder of this section describes some specific ways to produce efficient compiled code with LispWorks.

10.6.1 Compiler optimization hints

You can make the compiler print messages which will help you to optimize your code. You add suitable **`:explain`** declarations, recompile the code, and check the output.

The full syntax of the **`:explain`** declaration is documented under **`cl:declare`** in the *LispWorks Reference Manual*.

Various keywords allows you to see information about compiler transformations depending on type information, allocation of floats and bignums, floating point variables, function calls, argument types and so on. Here is a simple example:

```
(defun foo (arg)
  (declare (:explain :variables) (optimize (float 0)))
  (let* ((double-arg (coerce arg 'double-float))
         (next (+ double-arg 1d0))
         (other (* double-arg 1/2)))
    (values next other)))
;;- Variables with non-floating point types:
;;-  ARG OTHER
;;- Variables with floating point types:
;;-  DOUBLE-ARG NEXT
```

Note: the LispWorks IDE allows you to distinguish compiler optimization hints from the other output of compilation, and also helps you to locate quickly the source of each hint. For more information see the chapter "The Output Browser" in the *Common LispWorks User Guide*.

10.6.2 Fast 32-bit arithmetic

The INT32 API provides a way to perform optimal raw 32-bit arithmetic. Note that, unlike Lisp integer types, this is modulo 2^32 like the C int type.

The Lisp type **`sys:int32`** reads 32 bits of memory, like **`(signed-byte 32)`**, but the data is in **`sys:int32`** format for use with the INT32 API.

10.6.2.1 Optimized and unoptimized INT32 code

When optimized correctly, the intermediate **`sys:int32`** objects are not constructed.

In unoptimized code, sequences of operations like

```
(sys:int32+ (sys:int32- a b) (sys:int32- c d))
```

will generate intermediate **`sys:int32`** objects for the results of the subtraction, but the compiler can optimize these away because it knows that the function **`sys:int32+`** consumes **`sys:int32`** objects.

Note: the INT32 API is not designed to optimize **`sys:int32`** objects passed as arguments.

10.6.2.2 The INT32 API

The INT32 API contains the type **sys:int32**, a vector type **sys:simple-int32-vector** and accessor, functions to convert **sys:int32** to and from integer, some constant **sys:int32** values, and a full range of operators for mod 2^32 arithmetic.

You can find all these by evaluating

```
(apropos "INT32" "SYSTEM" t)
```

See the *LispWorks Reference Manual* for details of each.

10.6.2.3 INT32 Optimization

The optimization works safely but without boxing when possible. You need

```
(optimize (float 0))
```

to get the optimization. This **float** level affects whether INT32 operations are optimized. This declaration must be placed at the start of a function (not on an inner **let** or **locally** form).

In this example the **safety** level assures a second optimization in **fli:foreign-typed-aref**:

```
(defun incf-signed-byte-32 (ptr index)
  (declare (optimize (safety 0) (float 0))
           (type fixnum index))
  (setf (fli:foreign-typed-aref 'sys:int32 ptr index)
        (sys:int32-1+ (fli:foreign-typed-aref 'sys:int32
                                              ptr index)))
  ;; return ptr, since otherwise the int32 would
  ;; need to be boxed to return it
  ptr)
```

10.6.3 Floating point optimization

The **float** declaration allows generation of more efficient code using float numbers. It reduces allocation during float calculations. It is best used with safety 0. That is, you declare **(optimize (float 0) (safety 0))** as in this example:

```
(progn
  (setf a
        (make-array 1000
                    :initial-element 1D0
                    :element-type 'double-float))
  nil ; to avoid printing the large array
  )

(compile
 (defun test (a)
   (declare (optimize (speed 3) (safety 0) (float 0)))
   (declare (type (simple-array double-float  (1000))
                  a))
   (let ((sum 0D0))
     (declare (type double-float sum))
     (dotimes (i 1000)
       (incf sum (the double-float (aref a i))))
     sum)))

(time (test a))
=>
Timing the evaluation of (TEST A)

user time    =      0.000
system time  =      0.000
Elapsed time =   0:00:00
Allocation   = 16 bytes standard / 0 bytes conses
0 Page faults
```

Note: calls to `+`, `-` and `*` with more than 4 arguments will not be optimized, even with the declaration described above, so avoid such calls to obtain the best floating point performance

10.6.4 Tail call optimization

In 64-bit LispWorks and on x86 platforms the compiler optimizes tail calls unless

1. The compiler optimize quality `debug` is 3, or
2. There is something with dynamic scope on the stack, such as a special binding, a catch or `dynamic-extent` allocation (so it is not really a tail call)

On all other platforms the compiler optimizes tail calls unless 1.) or 2.) above apply, or

3. The call has more than 4 arguments and this is more than the number of fixed (not **`&optional`**/**`&rest`**/**`&key`**) parameters in the calling function.
4. The call has more than 4 arguments and the calling function has **`&rest`**/**`&key`** parameters.

10.6.5 Stack allocation of objects with dynamic extent

`(declare dynamic-extent)` will optimize these calls so that they allocate in the stack, in all cases:

- **`&rest`** lists
- **`flet`** functions and **`labels`** functions
- **`(cons x y)`**
- **`(list ...)`**
- **`(list* ...)`**
- **`(copy-list x)`**
- **`(make-list x)`**
- **`(vector ...)`**

`(declare dynamic-extent)` will also optimize these specific calls:

- **`(make-array n)`**
- **`(make-array n :initial-element x)`** without any other arguments
- **`(make-foo ...)`** where **`make-foo`** is an inline structure constructor. The default constructor is declared inline automatically when none of the **`defstruct`** slot initforms are calls to functions.
- **`(make-string n :element-type 'base-char)`**

10.6.6 Inlining foreign slot access

Given a structure definition

```
(fli:define-c-struct foo-struct
  (a :int)
  (b :int))
```

you can inline access to a slot by declaring **fli:foreign-slot-value** inline and supplying the *object-type*:

```
(defun foo-a (struct)
  (declare (inline fli:foreign-slot-value))
  (fli:foreign-slot-value struct 'a :object-type 'foo-struct))
```

10.7 Compiler parameters affecting LispWorks

There are six compiler parameters that control the generation of information used by various LispWorks utilities, such as the debugger, and also by various editor commands, such as **Show Paths From**. By default, these parameters are all **t**, which allows you to use all the features of these utilities, at the expense of increasing compilation times.

These variables are initially set to **t** (in the LispWorks file **config/a-dot-lispworks.lisp**). To speed up compilation times, you should set these variables to **nil**. The variables can be controlled as a group by using the function **toggle-source-debugging**. See under this function in the *LispWorks Reference Manual* for details of all the variables that it controls.

11

Storage Management

This chapter introduces some basic ideas of storage management, and then discusses the LispWorks storage management system in more detail. The chapter also introduces the functions and macros needed to control storage management. Full details of all the symbols mentioned here are given in the *LispWorks Reference Manual*.

11.1 Introduction

Automatic memory management is one of the most significant features of a Lisp system. Whenever an object, such as a cons cell, is required to hold an aggregate of values, the system calls the appropriate function to create a new object and fill it with the intended values. The programmer need not be concerned with the low level allocation and management of memory as the Lisp system provides this functionality automatically.

When an object is no longer required (that is, it has become “garbage”), the system must automatically reclaim (“collect”) the space it occupies and reallocate the space to a new object. Whenever the space for new objects is exhausted, a “garbage collector” is run to determine (by a process of elimination) all the existing objects that are still required by the running program. Any other objects still in the image are necessarily garbage, and the space they occupy can be reclaimed.

For a description of how LispWorks uses the address space of different Operating Systems, and factors affecting the maximum image size, see “Address Space and Image Size” on page 289.

Garbage collection with a naive algorithm is extremely inefficient. The time required to scan an entire image, which may occupy many megabytes of memory, is prohibitive; especially when the collector must perform the scan in a small, fixed, workspace.

11.2 Generations and segments

The LispWorks garbage collector works in unison with the storage allocator to arrange allocated objects in a series of “generations”. Each generation contains objects of a particular age. In practice most Lisp data objects are only required for a very short period of time. That is, they are ephemeral. The LispWorks garbage collector concentrates its efforts on repeatedly scanning the most recent generation. Such a scan requires only a fraction of a second and reclaims most of the space allocated since the last collection. Any object in the most recent generation that survives a number of such collections is promoted to the next youngest generation. Eventually this older generation becomes full, and only then is it collected. The generations are numbered from 0 upwards, so that generation 0 is the youngest.

The remainder of this chapter describes the LispWorks garbage collector in more detail. The implementation and the programmatic interface differ between 32-bit and 64-bit LispWorks.

11.3 Memory Management in 32-bit LispWorks

This section describes the garbage collector (GC) in 32-bit LispWorks 5.1.

In LispWorks for UNIX and LispWorks for Macintosh, the implementation is not significantly different to that in LispWorks 4.x or LispWorks 5.0.

In LispWorks for Windows and LispWorks for Linux, the implementation has changed since LispWorks 4.x and you may notice performance improvements relative to those versions.

11.3.1 Generations

In memory, a generation consists of a chain of segments. Each segment is a contiguous block of memory, beginning with a header and followed by the allocation area.

The first generation normally consists of two segments: the first segment is relatively small, and is where most of the allocation takes place. The second segment is called the big-chunk area, and is used for allocating large objects and when overflow occurs (see below for a discussion of overflow).

The second generation (generation 1) is an intermediate generation, for objects that have been promoted from generation 0 (typically for objects that live for some minutes).

Long-lived objects are eventually promoted to generation 2. Note that generation 2 is not scanned automatically. Therefore these objects will not be reclaimed (even if they are not referenced) until an explicit call to a garbage collector function (for example **`mark-and-sweep`** on generation 2, or **`clean-down`**) or when the image is saved. Normally, objects are not promoted from generation 2 to generation 3, except when the image is saved.

Generation 3 normally contains only objects that existed at startup time, that is those were saved in the image. Normally it is not scanned at all, except when an image is saved.

Note that the division between the generations is a result of the promotion mechanism, and is not a property of a piece of code itself. A piece of system software code that is loaded in the system (for example, a patch) is treated the same as any other code. The garbage collection code is explicitly loaded in the static area using the function **`switch-static-allocation`**.

11.3.2 Allocation of objects

Normal allocation is done from a buffer, called the small objects buffer. The Garbage Collector (GC) maintains a pointer to the beginning and end of the buffer, and allocates from it by moving one of the boundaries. When the buffer becomes too small the GC finds another free block and makes that the buffer.

The minimum and maximum size of free block that the GC uses for the small objects buffer can be set by **`set-gc-parameters`**, using the keywords

:minimum-buffer-size and **:maximum-buffer-size**. If the minimum size is too small, the system allocates buffers more frequently, thus slowing the program. Making the minimum too big causes more fragmentation, because small free blocks are not used. There is no easy way to determine the optimal values for the small objects buffer, except by experiment.

When there is an overflow the small object buffer is allocated in the big-chunk area, and then a bigger buffer is allocated (see below).

11.3.2.1 Allocation of static objects

Objects that cannot be moved are allocated in special segments, called static segments. These can be in any generation, but are in generation 2 by default.

Such objects include:

- Code that must not move during garbage collection, in particular the code and data of the garbage collector itself
- Objects allocated explicitly in the static area, by **in-static-area** or by use of **switch-static-allocation**.
- Foreign code loaded from a non-shared library via **link-load:read-foreign-modules**. This applies to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD or Macintosh).
- Objects allocated by **malloc**, **realloc** and **memalign** in foreign code loaded as above.

Because static objects are not allowed to move, the static segments are not allowed to move. This implies that if there is a static segment in a high address the image size cannot be reduced below this size. Applications that use a lot of static area normally allocate additional static segments, and thus grow without being able to shrink again. This can be prevented by enlarging the initial static segment, which is in a low address. Use the function **enlarge-static** to increase the size of the initial static segment. (Use **(room t)** to find its current size.)

11.3.2.2 Allocation in different generations

Objects that are known to have long life can be allocated directly in a higher generation, by using **`allocation-in-gen-num`** and **`set-default-generation`**. Note that both these functions have a global effect, i.e. any object allocated after a call to **`set-default-generation`** or within the body of **`allocation-in-gen-num`** is allocated in the specified generation, unless it is explicitly allocated in a different generation. Therefore careless use of these functions may lead to allocation of ephemeral garbage in high generations, which is very inefficient. Conversely, if a long-lasting object is allocated to a low generation, it has to survive several garbage collections before being automatically promoted to the next generation.

See also "Allocation of interned symbols and packages" on page 118 and "Allocation of stacks" on page 119.

11.3.3 Mark and sweep

Mark and sweep is the basic operation of reclaiming memory, and it is done in two stages:

Mark — All objects that are alive in the generation being garbage collected and in younger generations are marked as alive. (Alive means pointed to by some other live object.)

Sweep — All unmarked objects in the generations being garbage collected are added to the free blocks, and all marked objects are unmarked.

A mark and sweep operation is always on all the generations from 0 to a specific number.

A mark and sweep operation can be caused explicitly by calling **`mark-and-sweep`**.

11.3.4 Promotion

Promotion is the process of moving objects from one generation to the next generation. An object is marked for promotion after surviving a specific number of mark and sweep operations, but may be promoted before that. The

number of survivals is specific to each segment, and can be set by **`set-promotion-count`**.

11.3.5 Garbage collection strategy

When the Garbage Collector runs out of memory, it has to find more memory. Normally (that is, when allocating in generation 0) the first operation is a mark and sweep. Before performing the mark and sweep, the GC compares the amount of memory allocated since the previous mark and sweep with the **`:minimum-for-sweep`** value, which is set by **`set-gc-parameters`**. If the amount allocated is less than **`:minimum-for-sweep`** the GC does not do a mark and sweep, but causes an overflow (described below). This prevents an excessive number of mark and sweep operations in periods when the program allocates a large amount of data which stays alive.

Note that the GC monitor window does not indicate a mark-and-sweep of generation 0, as this operation takes a small amount of time (To change the display would take longer than the mark-and-sweep operation itself.)

If more than **`:minimum-for-sweep`** has been allocated, a mark and sweep operation takes place. After this operation the GC checks that the segment it was trying to allocate to has more free space than the minimum free space for this segment. If the remaining free space is less than **`minimum-free-space`**, the GC tries to create more free space by promoting objects from the segment.

Before promoting, the GC performs two checks. First, it checks that there are enough objects marked for promotion to justify a promotion operation. The minimum free space for a segment is set by **`set-minimum-free-space`**, and can be shown by **`(room t)`**.

Second, the GC checks that there is enough free space in the next generation to accommodate the promoted objects. If there is insufficient space, the GC tries to free some, either by a mark and sweep on the next generation, promoting the next generation, or by enlarging the generation.

The minimum amount of space for promotion is the value **`minimum-for-promote`**, which is set by **`set-gc-parameters`**.

If there is insufficient space, and there are not enough objects marked for promotion, the GC increases the size of the image, by overflow, as described below.

11.3.6 Overflow

If the amount allocated from the previous mark and sweep operation is less than **:minimum-for-sweep**, the GC does not perform a mark and sweep. Instead it allocates a small-objects buffer in the big-chunk area (the second segment in the first generation). The minimum and maximum sizes of this buffer are specified by **:minimum-overflow** and **:maximum-overflow**, which can be set by **set-gc-parameters**. If the GC fails to find a buffer of this size, it looks for a smaller buffer, and if that fails it enlarges the big-chunk area (and the process size) by the amount needed to allocate a buffer of the size of the currently allocated area in generation 0, up to a maximum amount specified by **:maximum-overflow**.

11.3.7 Behavior of generation 1

When objects are promoted from generation 0 to 1, and there is not enough space in generation 1, the GC tries to free space in generation 1. The first step is to check if sufficient space can be freed by promoting the objects marked for promotion. If this is the case the GC promotes these objects from generation 1 to generation 2. (In practice, this rarely happens.) If this check fails the GC marks and sweeps generation 1. If not enough space is freed by this mark-and-sweep, than either all the objects in generation 1 are promoted, or generation 1 is expanded. This is controlled by **expand-generation-1**, which specifies whether expansion or promotion takes place.

If generation 1 is expanded, the amount it tries to expand by is the value **:new-generation-size (**set by **set-gc-parameters**) in words (i.e. multiples of 4 bytes), or the amount of free space needed, whichever is bigger. If **:new-generation-size** is 0, it is not expanded. In this case part of the objects marked for promotion are not promoted.

11.3.8 Behavior of generation 2

Normally generation 2 is not garbage collected. If the system runs out of space in this generation, it expands it, using the value of **:new-generation-size** multiplied by two. Garbage collection of generation 2 can be caused by calling the function **collect-generation-2** with appropriate argument.

11.3.9 Forcing expansion

If you know that a given generation will need to grow, you can save the GC the work by calling **enlarge-generation** to expand the generation in advance.

11.3.10 Controlling Fragmentation

Some applications periodically free (that is, stop using) a substantial amount of data that lived for long enough to reach generation 2 (use **room** or **sys:room-values** and **sys:generation-number** to follow the behaviour of objects). In this case, **mark-and-sweep** should be called on generation 2, to collect these data and re-use the memory. Repeated cycles like this may cause fragmentation, which will slow down promotion into generation 2. This manifests itself in significant pauses, typically of a few seconds. **try-move-in-generation** or **try-compact-in-generation** can be used to reduce the fragmentation, and hence to reduce the pauses. Because these functions themselves take some time, they should be called when such a pause is acceptable.

'Moving' a segment means moving objects out of the segment to another segment, leaving the segment empty. This reduces the fragmentation in the generation, and it is normally much faster than compact. Therefore in almost all cases, **try-move-in-generation** is better than **try-compact-in-generation**.

The actual decision to use these functions will be typically based on the results of **check-fragmentation**. For example, the following function checks if there is more than 10Mb free area in generation 2 in blocks of 4096 bytes or larger (tlb, third return value of **check-fragmentation**). If there is not, and the free area in generation 2 (tf) is more than four times the free area in large blocks, it calls **try-move-in-generation**. Because **try-move-in-generation** gets a time-threshold of 0, it returns after doing at most one segment. (It won't do any segments if none of them looks fragmented.)

```
(defun call-memory-functions()
  (mark-and-sweep 2)          ; first collect all dead objects
  (multiple-value-bind (tf tsb tlb)
      (check-fragmentation 2) ; check the fragmentation
    (when  (and (> 10000000 tlb)
                (> (ash tf -2) tlb))
       (try-move-in-generation 2 0))))
```

A function such as this can be called at times when a pause of a few seconds is acceptable, and it will keep the memory of generation 2 unfragmented.

It is not possible to give definitive guidance here on how to use **`try-move-in-generation`** or **`try-compact-in-generation`**, because it depends on the way the application uses memory. In general, these functions will always improve the behavior of the application. Therefore the main problem is to identify points in the execution of the application where they can be called without causing unacceptably long pauses.

11.3.11 Summary of garbage collection symbols

The remainder of this chapter summarizes which functions are useful in which circumstances. See also "Common Memory Management Features" on page 118. For full details of these functions, see the *LispWorks Reference Manual*.

11.3.11.1 Determining storage usage

To determine storage usage (useful when benchmarking), use the functions **`room`**, **`total-allocation`** and **`find-object-size`**. The function **`sys:room-values`** is suitable for programmatic use: it returns the values that **`room`** prints.

In 32-bit LispWorks, **`memory-growth-margin`** returns the amount by which the Lisp heap can grow, if **`set-maximum-memory`** has been called.

11.3.11.2 Allocating in specific generations

To control the allocation of objects to generations, use **`allocation-in-gen-num`**, **`get-default-generation`**, **`set-default-generation`** and **`*symbol-alloc-gen-num*`**.

11.3.11.3 Controlling a specific generation

To control the behavior of a specific generation, use **`clean-generation-0`**, **`collect-generation-2`**, **`collect-highest-generation`**, **`expand-generation-1`**, **`set-minimum-free-space`** and **`set-promotion-count`**.

11.3.11.4 Controlling the garbage collector

To control the actions of the garbage collector, use the symbols **avoid-gc**, **get-gc-parameters**, **gc-if-needed**, **mark-and-sweep**, **normal-gc**, **set-gc-parameters**, **without-interrupts** and **with-heavy-allocation**.

To control expansion, use **enlarge-generation**.

To control fragmentation, use **check-fragmentation** with **try-move-in-generation** or **try-compact-in-generation**.

11.4 Memory Management in 64-bit LispWorks

This section describes the garbage collector (GC) in 64-bit LispWorks.

11.4.1 General organization of memory

The memory in 64-bit LispWorks is arranged in segments, which belong to generations. Unlike 32-bit LispWorks, segments are sparsely allocated in memory, that is they not contiguous.

Each segment has an allocation type, which defines the type of objects that the segment contains. The system creates and destroys segments as needed. A generation may or may not contain a segment for a specific allocation type, and a generation may contain more than one segment for any particular allocation type. Segments may change in size.

You can see the allocation for each allocation type in the output of:

```
(room t)
```

Additionally you can see the segments of each generation in the output of:

```
(room :full)
```

After the total allocation in each generation, this prints the allocation type for each segment followed by the hexadecimal address range for allocating objects.

11.4.2 Segments and Allocation Types

Some GC interface functions take an allocation type as an argument, which is one of the keywords below. There are two categories of allocation type.

The main allocation types, which can be used as the *what* argument to the function **sys:apply-with-allocation-in-gen-num**, are:

:cons The segment contains only conses.

:symbol The segment contains only symbols (and does not include symbol names or any of the other properties of symbols).

:function The segment contains only function objects.

:non-pointer
The segment contains only objects that do not contain pointers (strings, specialized numeric arrays, double-floats).

:other The segment contain other objects, that is any object that contain pointers, and is not a symbol, cons or a function.

The derived allocation types are:

:mixed The segment contains a mixture of **:other**, **:function** and **:symbol**, but not **:cons** or **:non-pointer**.

:cons-static
The segment contains cons objects that are static.

:non-pointer-static
The segment contains objects that do not contain pointers and are static (currently stacks are also allocated in these segments).

:mixed-static The segment contains a mixture like :mixed, but static.

:weak The segment contains weak objects (arrays, and internals of weak hash tables).

:other-big The segment contains a single very large simple vector. The vector is static.

:non-pointer-big

The segment contains a single very large non-pointer object (a string or a specialized numeric array). The vector is static.

Segments of allocation type **:other-big** or **:non-pointer-big** can be as large as required to hold their object.

For all other allocation types, the size of each single segment is restricted. The implementation limit is currently 256MB, and you can specify a smaller limit using **sys:set-maximum-segment-size**.

11.4.3 Garbage Collection Operations

In 64-bit LispWorks there are two methods of garbage collection: *mark and sweep* (also referred to simply as *mark*) and *copy*. The two methods can be mixed within the same garbage collection operation and generation, but a segment is collected using only one of mark or copy in a given operation.

When a segment is collected using the copying method, the objects within it can either be copied to another segment in the same generation or can be copied to a segment in a higher generation. The latter case is called promotion. The automatic garbage collection copies with promotion until the objects reach the blocking generation, which is collected in a specific way as described in “Generation Management” on page 116.

11.4.4 Generation Management

In general, higher generations contain objects that live longer and are therefore much less likely to die. Each garbage collection only collects the generations up to some number, and never reclaims the objects in higher generations.

Objects move between generations by being promoted. For most allocation types, this means that the GC copies the objects from a segment in one generation to a segment in a higher generation. For allocation types **:other-big** and **:non-pointer-big**, the objects are not actually copied when they are promoted; but instead the whole segment is reattached to the higher generation. The automatic garbage collection promotes objects until they reach the blocking generation.

In the default configuration, there are 8 generations, numbered from 0 to 7. Generation 7 is used to keep objects that survived saving the image. Generations 4, 5 and 6 are not used. Generation 3 is the blocking generation, where long-lived objects accumulate. Generations 0,1, and 2 are ephemeral, and objects that survive a garbage collection in each of these generations are promoted to the next generation.

11.4.5 Tuning the GC

As in LispWorks 4.x, the main tool for tuning the GC is the macro **`extended-time`** and periodical calls to **`room`**, to see how the system behaves.

In the output of **`(room t)`**, the allocation in each generation is presented according to the allocation type, which may be useful to decide on possible tuning.

`(extended-time` ***`forms`*****`)`** outputs the time spent in garbage collection, whether automatic or called explicitly. The time is shown according to the maximum generation number that was collected and to whether it was a standard garbage collection (automatic and calls to **`gc-generation`**) or a marking garbage collection (calls to **`sys:marking-gc`**).

In addition to **`room`** and **`extended-time`**, there are also the functions **`sys:count-gen-num-allocation`**, **`sys:gen-num-segments-fragmentation-state`**, and **`sys:set-automatic-gc-callback`**. These function can be used to collect information about automatic garbage collection operations.

11.4.5.1 Interface for tuning the GC

The main interfaces are those which control the blocking generation.

For generations lower than the blocking generation, objects that survive are promoted, and the system does not automatically promote objects to higher generations. Thus if the application generates long-lived objects, they will accumulate in the blocking generation.

The behavior when the blocking generation grows is controlled by **`sys:set-blocking-gen-num`** and **`sys:set-gen-num-gc-threshold`**. It may also be useful to set the maximum segment size with **`sys:set-maximum-segment-size`**.

Explicit garbage collection can be done by calling **gc-generation** and **sys:marking-gc**.

gc-generation can also be used to promote objects to a higher generation than the blocking generation.

It is normally less important to tune the ephemeral segments, that is the segments below the blocking generation. Functions that may be useful include **sys:set-default-segment-size**, **sys:set-spare-keeping-policy** and **sys:set-delay-promotion**.

11.5 Common Memory Management Features

This section summarises Memory Management functionality common to all LispWorks 5.1 implementations.

11.5.1 Timing the garbage collector

The macro **extended-time** is useful when timing the garbage collector.

11.5.2 Reducing image size

To reduce the size of the whole image, use **clean-down**.

11.5.3 Allocation of interned symbols and packages

Interned symbols (and their symbol names), and packages, are treated in a special way, because they are assumed to have a long life. They are allocated in the generation specified by the variable ***symbol-alloc-gen-num***, which has the initial value 2 in 32-bit LispWorks and 3 in 64-bit LispWorks.

Symbols created with **make-symbol** or **gensym** start out in generation 0.

Symbols will be garbage collected if they are no longer accessible (regardless of property lists) but note that in 32-bit LispWorks, if the symbols are in generation 2 then you might need to invoke **mark-and-sweep** explicitly to collect them in a timely manner.

11.5.4 Allocation of stacks

Stacks are allocated directly in generation 2 because they are relatively expensive to promote. Therefore creating many processes will cause generation 2 to grow, even if these processes are short-lived.

The variable **system:*default-stack-group-list-length*** controls the number of stacks that are cached for reuse. Increase its value if your application repeatedly makes and discards more than 10 processes.

11.5.5 Mapping across all objects

To call a function on all objects in the image, use **sweep-all-objects**.

11.5.6 Special actions

You may want to perform special actions when certain types of object are garbage collected, using the functions **add-special-free-action**, **flag-special-free-action**, **flag-not-special-free-action** and **remove-special-free-action**.

For example, when an open file stream is garbage collected, the file descriptor must be closed. This operation is performed as a special action.

11.5.7 Garbage collection of foreign objects

Users of the Foreign Language Interface may want to specify the allocation of static arrays. The recommended way to do this is to call **make-array** with **:allocation :static**. See for example **:lisp-array** in the *LispWorks Foreign Language Interface User Guide and Reference Manual*.

11.5.8 Freeing of objects by the GC

Weak arrays and weak hash tables can be used to allow the GC to free objects.

Relevant functions are **make-hash-table**, **set-hash-table-weak**, **set-array-weak**, **make-array** and **copy-to-weak-simple-vector**.

For a description of weak vectors see **set-array-weak** in the *LispWorks Reference Manual*.

11.6 Assisting the Garbage Collector

This section describes techniques that may improve the performance of your application by reducing the GC's workload.

11.6.1 Breaking pointers from older objects

This is a technique that can be useful when older objects regularly point to newer objects in a lower generation. In such a case, when the lower generation (only) is collected these newer objects will be promoted even if the older objects are not live. All of these objects will not get collected until the higher generation is collected.

This is a general issue with generational garbage collection and, if it causes poor performance in your application, can be addressed along these lines. It is not necessarily a problem in every case where older objects point to newer objects.

For example, suppose you are popping items from a queue represented as a list of conses (or other structures), then you can set the "next" slot of each popped item to **nil**.

In the code below, if the **queue-head** cons is promoted to generation *n*, then all the other conses will also be promoted to generation *n* eventually, until generation *n* is collected. This happens even after calls to **pop-queue** have removed these conses from the queue.

```
(defstruct queue head tail)

(defun push-queue (item queue)
  (let ((new (cons item nil)))
     (if (queue-head queue)
         (setf (cdr (queue-tail queue)) new)
       (setf (queue-head queue) new))
     (setf (queue-tail queue) new)))

(defun pop-queue (queue)
  (pop (queue-head queue)))
```

The fix is to make **pop-queue** set the "next" slot (in this case the **cdr**) of the discarded **queue-head** cons to **nil**, so that it no longer points from an older object to a newer object. For example:

```
(defun pop-queue (queue)
   (when-let (head (queue-head queue))
     (setf (queue-head queue) (shiftf (cdr head) nil))
     (car head)))
```

12

The Profiler

The LispWorks profiler provides a way of empirically monitoring execution characteristics of Lisp programs. The data obtained can help to improve the efficiency of a Lisp program by highlighting those procedures which are commonly used or particularly slow, and which would therefore benefit from optimization effort.

For complete details of the functions and symbols introduced in this chapter, see the *LispWorks Reference Manual*.

12.1 What the profiler does

With the profiler running, the Lisp process is interrupted regularly at a specified time interval until the profiler is turned off. Having halted the execution of the process the profiler scans the execution stack and records information about it, including the names of all functions found. A special note is made of which function is at the top of the stack. After profiling stops the profiler can present a report containing a call tree and/or a cumulative columnar report.

The columnar report shows aggregated information about each function as follows:

- The number of times the function was called.

- The number of times the function was found on the stack by the profiler, both in absolute terms and as a percentage of the total number of scans of the stack.
- The number of times the function was found on the top of the stack, both in absolute terms and as a percentage of the total number of scans of the stack.

The call tree shows name of a root function and a "tree" of callee functions below it. To the right of each function's name the number of times it was seen on the stack under a particular caller is shown, along with the percentage this represents of the total number of times the function was seen.

The call tree is more computationally expensive to record than the cumulative data. You can choose whether to record and output the call tree, as described in the next section.

12.2 Setting up the profiler

Before a profiling session can start several parameters must be set, using the function **`set-up-profiler`**. This function is introduced here and the full syntax is given in the *LispWorks Reference Manual*. There are four main areas to consider: the symbols to be profiled, the time interval between samples, the kind of profiling required, and the format of the output.

- It is possible to keep track of every function called during a particular computation, but significant effort is involved in determining which symbols are suitable for profiling and in keeping track of the results. To minimize this effort you need to specify which symbols to profile, either by naming the required symbols, or by naming a package, all of whose symbols are profiled. The profiler first checks that these symbols have indeed got function definitions and are therefore suitable for profiling.
- You might want to specify the time interval between interrupts. The resolution of this value is clearly dependent on the operating system. In most cases the default value, 10ms, is adequate. This number is important, because with these statistical methods of program profiling the accuracy of the results increases with the number of samples taken.

- On Unix/Linux/FreeBSD systems the kind of profiling required may be set. This refers to what kind of time is monitored in order to determine when to interrupt the Lisp process. There are three possibilities for how the time interval is measured:

  The time the Lisp process is actually executing plus the time that the system is executing on behalf of the process. This is called *profile time*.

  Just the time that the process is actually executing. This is called *virtual time*.

  The actual elapsed time, called *real time*.

- The output can be presented as a tree of calls seen and a columnar report (*style* **:tree**), or just the columnar report (*style* **:list**). You can restrict the data shown in several ways, helping you to focus on the slowest parts of your program.

Below is an example of setting up the profiler:

```
(set-up-profiler :symbols '(car cdr) :style :list)
```

Here the functions **car** and **cdr** are going to be profiled and the output will be just the columnar report.

The function **set-up-profiler** adds symbols to the ***profile-symbol-list***. The functions **add-symbol-profiler** and **remove-symbol-profiler** can also be used to change the symbols profiled.

The function **set-profiler-threshold** can be used with **reset-profiler** to control the effects of repeated profiler runs.

12.3 Running the profiler

The profiler has two distinct modes. You can use both in the same session, but not at the same time.

To use either mode, you must first call **set-up-profiler** to load the profiler and set its parameters including the output format.

The macro **profile** simply profiles all processes while a body of code is run, as described in “Using the macro profile” on page 126. Start profiling this way if you don’t see a need to use the alternate mode.

Alternatively the functions **start-profiling**, **stop-profiling** and **set-process-profiling** offer programmatic control over when profiling occurs and which processes are profiled. This is described in "Programmatic control of profiling" on page 126.

12.3.1 Using the macro profile

To profile your Lisp forms enter:

```
(profile <forms>)
```

This evaluates the forms as an implicit **progn** and prints the results, according to the parameters established by **set-up-profiler**.

Note: you cannot use **profile** (or the graphical Profiler tool) after a call to **start-profiling** and before a call to **stop-profiling** with *print* **t**, because the two profiling modes are incompatible.

12.3.2 Programmatic control of profiling

Your program can control profiling. This is useful when you want to profile only a part of the program.

In your program, call **start-profiling** start collecting profiling information. Call **stop-profiling** with *print* **nil** to temporarily stop collecting, or call **stop-profiling** with *print* **t** to stop collecting and print the results. At any point you can call **set-process-profiling** to modify the set of processes for which profiling information is being (or will be) collected.

For example:

```
;; start profiling, current process only
(start-profiling :processes :current)
(do-interesting-work)
;; temporarily suspend profiling
(stop-profiling :print nil)
(do-uninteresting-work)
;; resume profiling
(start-profiling :initialize nil)
(do-more-interesting-work)
;; now, all processes are interesting
(set-process-profiling :set :all)
(do-some-more-interesting-work)
;; stop profiling and print the results
(stop-profiling)
```

Note: you cannot call **`start-profiling`** inside the scope of the macro **`profile`** or while the graphical Profiler is profiling, because the two profiling modes are incompatible.

12.4 Profiler output

A typical report would be:

```
profile-stacks called 564 times

Call tree
Symbol                 seen   (%)
    1: MOD               17 (  3)
     2: FLOOR             5 (  1)
    1: EQL                8 (  1)
    1: >=                 7 (  1)
     2: REALP             2 (  0)
    1: +                  6 (  1)
    1: LENGTH             4 (  1)


Cumulative profile summary
Symbol      called  profile   (%)      top   (%)
MOD        1000000       17 (  3)        8 (  1)
EQL        2000117        8 (  1)        8 (  1)
>=         1000001        7 (  1)        5 (  1)
+          1000000        6 (  1)        6 (  1)
FLOOR      1000000        5 (  1)        5 (  1)
LENGTH     2000086        4 (  1)        4 (  1)
REALP      1000001        2 (  0)        2 (  0)

Top of stack not monitored 93% of the time
```

The first line means that Lisp was interrupted 564 times by the profiler.

The call tree shows that in 17 of these interrupts (3% of them) the profiler found the function **mod** on the stack, in 5 of these interrupts it found the function **floor** on the stack, and so on. Moreover, **floor** only appears under the **mod** branch of the tree, which means that each of these times **floor** was called by **mod**.

The cumulative profile summary also shows how many times each symbol was found on the stack. Moreover it shows that the function **mod** was called 1000000 times, the function **eql** was called 2000117 times, and so on. In 17 of these interrupts it found the function **mod** on the stack, and on 8 of these occasions **mod** was on the top of the stack. You can deduce that 526 times the function on the top of the stack was none of those reported.

You can control sort order of the cumulative profile summary with **print-profile-list**.

12.5 Interpretation of profiling results

The most important figures are the number of times a function was called along with the amount of time it was found on top of the stack. Just because a function is found on the stack does not mean that it using up much processing time, but if it is found consistently on the top of the stack then it is likely that this function has a significant execution time. Similarly functions that are called most often are likely to have the most significant effect on the program as a whole.

It must be remembered that the numbers produced are from random samples and thus it is important to be careful in interpreting their meaning. The rate of sampling is always coarse in comparison to the function call rate and so it is possible for strange effects to occur and significant events to be missed. For example, “resonance” may occur when an event always occurs between regular sampling times, though in practice this does not appear to be a problem.

12.6 Profiling pitfalls

Profiling should only be attempted on compiled code. If it is done on interpreted code, the interpreter itself is profiled, and this distorts the results for the actual Lisp program.

Macros cannot be profiled as they are expanded during the compilation process. Similarly some Common Lisp functions may be present in the source code but not in the compiled code as they are transformed by the compiler. For example:

```
(member 'x '(x y z) :test #'eq)
```

is transformed to:

```
(memq 'x '(x y z))
```

by the compiler and therefore the function **member** is never called.

Recursive functions need special attention. A recursive function may well be found on the stack in more than one place during one interrupt. The profiler counts each occurrence of the function. Hence the total number of times a function is found on the stack may be much greater than the number of times the stack is examined.

Care must be taken when profiling structure accessors. Structure accessors compile down into a call to a closure of which there is one for all structure setters and one for all structure getters. Therefore it is not possible to profile individual structure setters or getters by name.

It must be remembered that even though a function is found on the stack this does not mean that it is active or that it is contributing significantly to the execution time. However the function found on the top of the stack is by definition active, and thus this is the more important value.

It is quite possible that the amount of time the top symbol is monitored is significantly less than 100% despite the profiler being set to profile all the known functions of the application. This is because at the time of the interrupt an internal system function may well be on the top of the stack.

It is possible to profile all the symbols in the system by setting up the profiler as follows:

```
(set-up-profiler :package (list-all-packages))
```

12.7 Profiling and garbage collection

The macro **`extended-time`** provides useful information on garbage collection activities. See the *LispWorks Reference Manual* for full details.

The *gc* argument of **`set-up-profiler`** controls whether or not the system's memory management functions are profiled.

13

Customization of LispWorks

This chapter gives examples of how to make changes to LispWorks to make it more suitable for use by you and your colleagues.

13.1 Introduction

13.1.1 Pre-loading code

You can save an image with changes pre-loaded. This is suitable for changes you want to share with other users of that image, and for code which takes some time to load. It cannot be used to alter settings which the system makes automatically on startup.

"Saving a LispWorks image" on page 133 describes how to do this.

13.1.2 Loading code at start up

You can also load changes each time you start LispWorks. This is suitable for code which loads quickly. For changes only you want to see, put the code in your personal initialization file. For changes to share with other users at your site, put the code in your site initialization file.

"Initialization files" on page 132 describes these initialization files.

13.1.3 Specific customizations

The remainder of this chapter describes some customizations, all of which can can be saved in an image or placed in an initialization file, as needed. You can use both techniques: stable code including patches is saved in the image, whilst experimental or fast-loading code is loaded via the initalization file.

13.2 Configuration and initialization files

There are a number of files that contain configuration and initialization information:

13.2.1 Configuration files

- The LispWorks file **`config/configure.lisp`** contains many default configuration settings. You can create a customized copy of this file when you install LispWorks, as described in the *LispWorks Release Notes and Installation Guide*.
- The LispWorks file **`config/key-binds.lisp`** gives the default editor key bindings for Emacs emulation.
- The LispWorks file **`config/mac-key-binds.lisp`** gives the editor key bindings for Mac OS editor emulation, if supported on your platform.
- The LispWorks file **`config/msw-key-binds.lisp`** gives the editor key bindings for Microsoft Windows editor emulation, if supported on your platform.

13.2.2 Initialization files

- The LispWorks file **`config/siteinit.lisp`** is the default site initialization file. The distributed file loads any supplied patches.
- You may also have a personal initialization file which is loaded on startup. By default LispWorks looks for a file called **`.lispworks`** in your home directory, although you can change its name and location (see “Setting global preferences” in the *Common LispWorks User Guide*).

The default location of your home directory varies on Unix systems, but it is typically something like **/home**. On Windows, the directory is constructed from the environment variables HOMEDRIVE and HOMEPATH The directory itself has the same name as your user name, so if you log on as **john**, your home directory might be **/home/john** on Unix systems or something like **C:\Documents and Settings\john** on Windows XP.

A sample personal initialization file, the LispWorks file **config/a-dot-lispworks.lisp**, is supplied. You should created a customized copy of this file when you install LispWorks, as described in the *LispWorks Release Notes and Installation Guide*.

13.3 Saving a LispWorks image

To save an image with changes pre-loaded, first create a file **my-configuration.lisp** containing your settings. You may want to change some of the pre-configured settings shown in **config/configure.lisp**, add customizations from the rest of this chapter, or load your application code.

13.3.1 The save-image script

Now create a **save-image** script which is a file **save-image.lisp** containing something like:

```
(in-package "CL-USER")
(load-all-patches)
(load #-mswindows "/tmp/my-configuration.lisp"
      #+mswindows "C:/temp/my-configuration.lisp")
#+:cocoa
(compile-file-if-needed
 (sys:example-file
  "configuration/macos-application-bundle")
 :load t)
(save-image #+:cocoa
            (write-macos-application-bundle
             "/Applications/LispWorks 5.1/My LispWorks.app")
            #-:cocoa "my-lispworks")
```

The script shown loads **my-configuration.lisp** from a temporary directory. You may need to modify this.

13.3.2 Save your new image

The simplest way to save your new image is to use the Application Builder tool in the LispWorks IDE. Start the Application Builder as described in the *Common LispWorks User Guide*, enter the path of your **save-image** script in the **Build script:** pane, and press the **Build the application using the script** button.

Alternatively you can run LispWorks in a command interpreter and pass your **save-image** script in the command line as shown below.

- On Macintosh, run in Terminal.app:

  ```
  mymac$ "/Applications/LispWorks 5.1/LispWorks.app/Contents/
  MacOS/lispworks-5-1-0-macos-universal" -build save-image.lisp
  ```

  Your new application bundle is saved in **/Applications/LispWorks 5.1/My LispWorks.app**

- On Microsoft Windows, run in a MS-DOS window:

  ```
  C:\temp\>"C:\Program Files\LispWorks\lispworks-5-1-0-x86-
  win32.exe" -build save-image.lisp
  ```

  Your new LispWorks image is saved in **C:\temp\my-lispworks.exe**.

- On Linux, run in a shell:

  ```
  linux:/tmp$ lispworks-5-1-0-x86-linux -build save-image.lisp
  ```

  Your new LispWorks image is saved in **/tmp/my-lispworks**.

For other platforms and for 64-bit LispWorks the image name varies from that shown, but the principle is the same.

13.3.3 Use your new image

Your new LispWorks image contains the settings you specified in **my-configuration.lisp** pre-loaded.

You can add further customizations on start up via the intialization files mentioned in “Initialization files” on page 132.

13.4 Load and open your files on startup

Suppose you always compile and load several files after LispWorks starts. You can arrange for this to happen automatically by adding forms like these in your initialization file:

```
(defvar *my-files*
        '("/path/to/foo1"
          "/path/to/foo2"
          "/path/to/foo3"))

(dolist (file *my-files*)
  (compile-file file :load t))
```

If you also want to open these files in the Editor tool, then you can add this form in your initialization file, after those above:

```
(define-action "Initialize LispWorks Tools"
               "Open My Files"
               #'(lambda (screen)
                   (declare (ignore screen))
                   (dolist (file *my-files*)
                     (ed file))))
```

13.5 Customizing the editor

This section explains some of the customizations you can make to the Editor tool in the LispWorks IDE.

13.5.1 Controlling appearance of found definitions

The commands **`Find Source`**, **`Find Source for Dspec`** and **`Find Tag`** retrieve the file containing a definition and place it in a buffer with the relevant definition visible. By default, the start of the definition is in the middle of the Editor window and is highlighted.

The variable **`editor:*source-found-action*`** controls the position and highlighting of the found definition. The value should be a list of length 2.

The first element controls the positioning of the definition:

> **`t`** means show it at the top of the editor window.
>
> a non-negative fixnum means position it that many lines from the top.

nil means position it at the center of the window.

The second element can be **:highlight**, meaning highlight the definition, or **nil**, meaning don't.

For example, to configure the editor so that found definitions are positioned at the top of the window and are not highlighted, do

```
(setq editor:*source-found-action* '(t nil))
```

This variable is set in the file **a-dot-lispworks.lisp**.

13.5.2 Specifying the number of editor windows

You can specify the maximum number of editor windows that are present at any one time. For example, to set the maximum to 1:

```
(setq editor:*maximum-ordinary-windows* 1)
```

This variable is set in the file **a-dot-lispworks.lisp**.

13.5.3 Binding commands to keystrokes

You can bind existing editor commands to different keystrokes, using **editor:bind-key**.

The LispWorks file **config/key-binds.lisp** is supplied. It shows the standard Emacs key bindings for LispWorks.

The following example shows how to rebind **?** so that it behaves as an ordinary character in the echo area of a Common LispWorks tool — this can be useful if your symbol names include question marks.

```
(editor:bind-key "Self Insert" #\? :mode "Echo Area")
```

Since **?** is then no longer available for help, you may wish to rebind help to **Ctrl+?**.

```
(editor:bind-key "Help on Parse" #\C-? :mode "Echo Area")
```

If you use another editor emulation, then see the LispWorks file **config/msw-key-binds.lisp** or **config/mac-key-binds.lisp** for the corresponding **editor:bind-key**. forms.

13.6 Finding source code

Note: This section does not apply to LispWorks Personal Edition.

To configure LispWorks so that editor commands such as **`Find Source`**, the menu comand **Find Source**, and the dspec system are able to locate definitions in the supplied editor source code:

1. Load the logical host for the editor source code:

   ```
   (load-logical-pathname-translations "EDITOR-SRC")
   ```

2. Configure source finding to know about editor source code:

   ```
   (setf dspec:*active-finders*
         (append dspec:*active-finders*
                 (list "EDITOR-SRC:editor-tags-db")))
   ```

3. Now do (for example) **`Meta+X Find Command Definition`** and enter **`Wfind File`**.

   The definition of the command **`Wfind File`** is displayed in an Editor tool.

See “Controlling appearance of found definitions” on page 135 for information on controlling how the source code is displayed.

13.7 Specifying the initial working directory

The working directory is set on startup and provides the default location for the **File > Open...** dialog. Call **`hcl:change-directory`** in your initialization file (see “Initialization files” on page 132) to control the initial working directory.

13.8 Using ! for :redo

The default way of redoing the previous command from the command history is via **`:redo`**. If you want to use **`!`** (exclamation mark) instead of **`:redo`**, add the following to your **`.lispworks`** file:

```
(set-macro-character #\!
   #'(lambda (stream char)
    ':redo))
```

You may wish during some sessions to reset **!** back to its normal role as a character. To do this, evaluate:

```
(set-syntax-from-char #\! #\@)
```

13.9 Customizing LispWorks for use with your own code

This section contains some information on customizations you can make in order to make developing your own code a little easier.

13.9.1 Preloading selected modules

If you frequently use some code that is normally supplied as separate modules, you can load them at start-up time from your initialization file. This file is called **.lispworks** by default, but can be changed to be any other filename. See "Setting global preferences" in the *Common LispWorks User Guide* for details.

For example, to load the dynamic-completion code every time you start LispWorks, include the following in your initialization file.

```
(require "dynamic-complete")
```

13.9.2 Creating packages

When writing your own code that uses, for instance, the **capi** package, create a package of your own that uses **capi** — do not work directly in the **capi** package. By doing this you can avoid unexpected name clashes.

13.10 Structure printing

By default **defstruct** generates a method on **print-object**. You can avoid this by binding at macroexpansion time the variable **structure:*defstruct-generates-print-object-method***.

13.11 Configuring the printer

This section applies only on Unix/Linux/FreeBSD platforms.

You can configure your LispWorks image for your printer, by selecting **File > Printer Setup** from any tool with printing capacities, for example the editor, and choosing **Add Printer**.

When configuring a printer, the CAPI printing library prompts for a *PostScript Printer Description* file (PPD), which defines such things as the paper size and the printable area of the page, in the form of a standard PostScript language header. The printing code splices this file into the PostScript produced from submitting a CAPI printing request.

The library on the LispWorks CD contains a generic PPD file, called **`generic.ppd`**, that defines these values conservatively to ensure that it should work with most printers. For accurate results, you should use the PPD supplied with your printer.

The PPD files are placed in the **`ppd`** subdirectory of the **`postscript`** directory in the **`lispworks`** library directory. Files added to the **`ppd`** directory are expected to have the extension "**`.ppd`**".

13.11.1 PPD file details

A PPD file contains a description of the attributes and capabilities of a given printer, such as paper sizes supported, the printable area of the page, the number and names of input paper trays, optional features such as additional paper trays or duplex units, and so on, together with the printer-specific PostScript language commands necessary to use the features.

The **`generic.ppd`** file defines a simple generic printer supporting A4, A3, US letter, and US legal paper sizes, and supporting manual feed. It defines conservative margins (1 inch all round), and the documents generated should be compatible with most PostScript printers. It is suitable for producing PostScript files when the destination printer is unknown, and may also be used if the appropriate PPD for the printer is not available.

However, for the best results, we recommend the use of the appropriate PPD for the printer. This allows you to specify which optional features (if any) have been installed on the printer, and ensures that the Print dialog provides access

to appropriate printer capabilities such as multiple input trays and duplex printing. This also ensures that the CAPI uses the correct values for the printable areas of the page.

14

LispWorks as a dynamic library

This chapter describes how to create a dynamic library or DLL from LispWorks and discusses use of the library.

14.1 Introduction

You can use 32-bit LispWorks to build a dynamic library on Microsoft Windows, Intel Macintosh, Linux and FreeBSD, and 64-bit LispWorks on Windows, Intel Macintosh and Linux.

To do this, use **`save-image`** or **`deliver`** and supply a list value for *dll-exports*. On platforms other than Windows passing *dll-added-files* also creates a dynamic library.

The result is a library that cannot be executed on its own, but can be dynamically loaded by another process. On Windows this is done with the Windows APIs **`LoadLibrary`** and then **`GetProcAddress`**. On other platforms the dynamic library can be loaded by **`dlopen`** and then **`dlsym`**.

The dynamic library is usually of file type **`dll`** on Windows, **`dylib`** on Macintosh and and **`so`** on Linux or FreeBSD. The first implementation of this functionality in LispWorks was on Microsoft Windows only, therefore the terminology that is used is sometimes Windows-like. In particular “DLL” refers to any dynamic library.

14.2 Creating a dynamic library

To deliver a LispWorks runtime as a dynamic library supply a list value for *dll-exports* when calling **`deliver`**.

To save a LispWorks image as a dynamic library supply a list value for *dll-exports* when calling **`save-image`**.

Additonally on Linux, Macintosh and FreeBSD platforms, you can supply a list value for *dll-added-files* to deliver or save a dynamic library.

Note: a LispWorks dynamic library is licensed in the same way as a LispWorks executable.

14.2.1 C functions provided by the system

When LispWorks is a dynamic library the functions described in the chapter "Dynamic library C functions" in the *LispWorks Reference Manual* are automatically available. They allow the loading process control over relocation and unloading of the library.

14.2.2 C functions provided by the application

dll-exports specifies application-defined exported functions in a LispWorks dynamic library.

Exports can also be provided in the files named in *dll-added-files*, on Linux, Macintosh and FreeBSD platforms.

14.2.3 Example

This script saves an image **`hello.dll`** which is a Windows DLL:

```
-------------------- hello.lisp -------------------------
(in-package "CL-USER")
(load-all-patches)
;; The signature of this function is suitable for use with
;; rundll32.exe.
(fli:define-foreign-callable ("Hello"
                              :calling-convention :stdcall)
    ((hwnd w:hwnd)
     (hinst w:hinstance)
     (string :pointer)
     (cmd-show :int))
  (capi:display-message "Hello world"))

(save-image "hello"
            :dll-exports '("Hello")
            :environment nil)
---------------------------------------------------------
```

Run the script by

```
lispworks-5-1-0-x86-win32.exe -build hello.lisp
```

on the command line, or use the Application Builder tool.

(See Chapter 13, "Customization of LispWorks" for more information about how to save an image.)

You can test the DLL by running

```
rundll32 hello.dll,Hello
```

on the command line.

To see the dialog, you may need to dismiss the LispWorks splashscreen first.

14.3 Initialization of the dynamic library

Each of the exports specified via *dll-exports* ensure first that LispWorks has finished initializing. If initialization has not yet started, they start the initialization process themselves. This is true regardless of the value of *automatic-init* (see below).

A LispWorks dynamic library is initialized automatically on loading, or not, according to the value of *automatic-init* in the call to **deliver** or **save-image**.

14.3.1 Automatic initialization

On Windows when *automatic-init* was true the initialization finishes before **`LoadLibrary`** returns, and if LispWorks fails for some reason then the call to **`LoadLibrary`** fails too.

On other platforms when *automatic-init* was true, during the automatic initialization **`dlopen`** just causes the initialization to start and returns immediately. The initialization will finish sometime later. The function **`LispWorksState`** can be used to check whether it finished initializing.

Automatic initialization is useful when the dynamic library is something like a server that does not communicate by function calls. On Windows it also allows **`LoadLibrary`** to succeed or fail according to whether the LispWorks dynamic library initialized successfully or not.

14.3.2 Initialization via InitLispWorks

Not using automatic initialization (that is, creating the dynamic library with *automatic-init* **`nil`**) allows using **`InitLispWorks`** to relocate the image if necessary, and do any other initialization that may be required.

14.4 Relocation

LispWorks normally maps its heap on startup in the same place that it was when it was saved, and when it needs more memory it maps this nearby. This applies when LispWorks is a dynamic library as well as for LispWorks executables.

This mapping can cause memory clashes with other software, which may be avoided by relocating LispWorks. Most of the LispWorks implementations are relocatable though the details vary between platforms and between 32-bit LispWorks and 64-bit LispWorks.

On Microsoft Windows and Macintosh, LispWorks detects and avoids memory clashes automatically. On other platforms, you can relocate a LispWorks dynamic library (for all the relocatable implementations) if necessary by a suitable call to **`InitLispWorks`** as described in “Startup relocation” on page 289.

14.5 Multiprocessing in a dynamic library

Multiprocessing is started automatically in a LispWorks dynamic library. Therefore you can arrange for Lisp initialization operations by adding process specifications to **mp:*initial-processes***.

For example, if you have a function like this:

```
(defun my-server ()
  (let ((s (establish-a-socket)))
    (loop (accept-connection s))))
```

you need to do something like:

```
(pushnew '("My server" () my-server) mp:*initial-processes*
         :test 'equalp)
```

before saving or delivering your library.

14.6 Unloading a dynamic library

Before a LispWorks dynamic library is unloaded, LispWorks should be made to ‘quit’ cleanly, allowing it to clean up resources that it uses.

When the LispWorks dynamic library is loaded by a main process which you (the LispWorks programmer) do not control, then use **dll-quit**. If you control the main process, then use **QuitLispWorks** instead. For the details, see the respective manual entries for **dll-quit** and **QuitLispWorks** in the *LispWorks Reference Manual*.

15

The Metaobject Protocol

LispWorks CLOS essentially supports the metaobject protocol described in chapters 5 & 6 of *The Art of the Metaobject Protocol* (Kiczales, des Rivières & Bobrow, The MIT Press, 1991). Throughout the LispWorks documentation, "AMOP" refers to this book. Users might find it helpful to refer to the relevant chapters online at **`http://www.lisp.org/mop/`**.

All the LispWorks MOP symbols are in the **`clos`** package.

There are some discrepancies between LispWorks and AMOP, which are described in this Chapter.

This Chapter also describes some common problems encountered by programmers using the MOP.

15.1 Metaobject features incompatible with AMOP

15.1.1 Instance Structure Protocol

The generic functions implementing slot access are like those described in AMOP, except that each takes a *slot-name* argument rather than a slot definition object, and the primary methods are therefore specialized differently.

For details, see the *LispWorks Reference Manual* pages for **`slot-boundp-using-class`**, **`slot-value-using-class`** and **`slot-makunbound-using-class`**.

Note: by default, standard slot accessors are optimized to not call **`slot-value-using-class`**. This can be overidden with the **`:optimize-slot-access`** class option. See the second definition of **`virtual-metaclass`** below for an example of the use of this.

`standard-instance-access` is not supported as defined in AMOP. Note that there is an internal function of the same name, but this is not optimal. Also, **`funcallable-standard-instance-access`** is not supported. An alternative for fast instance access is to use the **`:optimize-slot-access`** class option.

15.1.2 Method Metaobjects

`standard-reader-method`, **`standard-accessor-method`** and **`standard-writer-method`** all have a required **`:slot-name`** initarg, rather than a **`:slot-definition`** initarg as specified in AMOP.

Note: in LispWorks 4.3 and previous versions, **`accessor-method-slot-definition`** was not implemented. This is implemented in the current version.

15.1.3 Method Lambdas

LispWorks **`make-method-lambda`** is not AMOP-compatible. It takes separate *lambda-list* and *body* arguments, and the returned **`lambda`** form is different to that specified in AMOP (see Chapter 15, "Method Functions" below).

LispWorks does not support user defined methods for the generic function **`make-method-lambda`**.

15.1.4 Method Functions

LispWorks method functions take the same arguments as the method itself, whereas in AMOP they take a list of arguments and a list of next methods.

15.1.5 EQL specializers

eql-specializer, **eql-specializer-object** and **intern-eql-specializer** are not implemented.

eql specializers in LispWorks are lists.

15.1.6 Generic Function Invocation Protocol

compute-applicable-methods-using-classes is not implemented.

compute-discriminating-function is implemented and returns the discriminator but:

- It does not use **compute-applicable-methods-using-classes** since LispWorks does not have that function.
- It does not call **compute-applicable-methods**.

Moreover **add-method** does not call **compute-discriminating-function** because this would be inefficient when doing multiple calls to **add-method**. Instead, **compute-discriminating-function** is called when the generic function is called.

15.1.7 Method combinations

method-combination objects do not contain the arguments, merely the type. There is a single **method-combination** object per type.

Therefore the value returned by **generic-function-method-combination**, and the default value of the **:method-combination** initarg, and the **:method-combination** argument processed by **ensure-generic-function-using-class** are specific only to the type of the method combination.

Also, **find-method-combination** is not implemented.

15.1.8 Inheritance Structure of Metaobject Classes

funcallable-standard-object is implemented as defined in AMOP, except that its class precedence list has direct superclasses

```
(function standard-object)
```

rather than

```
(standard-object function)
```

so that LispWorks is compliant with the ANSI Common Lisp rules.

For details, see the *LispWorks Reference Manual* page for **funcallable-standard-object**.

15.2 Common problems when using the MOP

15.2.1 Inheritance across metaclasses

Usually an inherited class is of the same metaclass as the parent class.

For other kinds of inheritance, you need to define a method on **validate-superclass** which returns true when called with the respective metaclasses. For example:

```
(defclass mclass-1 (standard-class)
  ())

(defclass mclass-2 (standard-class)
  ())

(defclass a ()
  ()
  (:metaclass mclass-1))

(defmethod validate-superclass
           ((class mclass-2)
            (superclass mclass-1))
  t)

(defclass b (a)
  ()
  (:metaclass mclass-2))
```

Without the **validate-superclass** method, the last form signals an error because **mclass-1** is an invalid superclass of **mclass-2**.

15.2.2 Accessors not using structure instance protocol

By default, **`defclass`** creates optimized standard accessors which do not call **`slot-value-using-class`**.

This optimization is controlled by the **`defclass`** option **`:optimize-slot-access`**, which defaults to **`t`**.

There is an illustration of this effect of **`:optimize-slot-access`** in the example below.

15.2.3 The MOP in delivered images

Issues with MOP code that occur only in delivered LispWorks images are documented in the section "Delivery and the MOP" in the *LispWorks Delivery User Guide*.

15.3 Implementation of virtual slots

This is an implementation of virtual slots with readers, writers and which also allow access by **`slot-value`**.

```
;; ----------------------- Virtual Slots --------------------
(in-package "CL-USER")

;; Metaclass of objects that might contain virtual slots.

(defclass virtual-metaclass (standard-class)
  ()
  )

;; Mixin metaclass for virtual slots and methods to make them
;; appear virtual.

(defclass virtual-slot-definition
          (standard-slot-definition)
  ((function :initarg :function
             :accessor virtual-slot-definition-function))
  )

(defmethod slot-definition-allocation
           ((slotd virtual-slot-definition))
  :virtual)

(defmethod (setf slot-definition-allocation)
           (allocation (slotd virtual-slot-definition))
  (unless (eq allocation :virtual)
    (error "Cannot change the allocation of a ~S"
           'virtual-direct-slot-definition))
  allocation)

;; Class of direct virtual slots and methods to construct them
;; when appropriate.

(defclass virtual-direct-slot-definition
          (standard-direct-slot-definition
           virtual-slot-definition)
  ()
  )

;; Called when the class is being made, to choose the metaclass of
;; a given direct slot. It should return the class of slot
;; definition required.

(defmethod clos:direct-slot-definition-class
           ((class virtual-metaclass) &rest initargs)
  ;; Use virtual-direct-slot-definition if appropriate.
  (if (eq (getf initargs :allocation) :virtual)
      (find-class 'virtual-direct-slot-definition)
```

```
    (call-next-method)))

;; Called when the defclass is expanded, to process a slot option.
;; It should return the new list of slot options, based on
;; already-processed-options.

(defmethod clos:process-a-slot-option
           ((class virtual-metaclass) option value
            already-processed-options slot)
  ;; Handle the :function option by adding it to the
  ;; list of processed options.
  (if (eq option :function)
      (list* :function value already-processed-options)
    (call-next-method)))


;; Class of effective virtual slots and methods to construct
;; them when appropriate.

(defclass virtual-effective-slot-definition
          (standard-effective-slot-definition
           virtual-slot-definition)
  ()
  )

;; Called when the class is being finalized, to choose the
;; metaclass of a given effective slot.  It should return the
;; class of slot definition required.

(defmethod clos:effective-slot-definition-class
           ((class virtual-metaclass) &rest initargs)
  ;; Use virtual-effective-slot-definition if appropriate.
  (let ((slot-initargs (getf initargs :initargs)))
    (if (member :virtual-slot slot-initargs)
        (find-class 'virtual-effective-slot-definition)
      (call-next-method))))

(defmethod clos:compute-effective-slot-definition
           ((class virtual-metaclass)
            name
            direct-slot-definitions)
  ;; Copy the function into the effective slot definition
  ;; if appropriate.
  (let ((effective-slotd (call-next-method)))
    (dolist (slotd direct-slot-definitions)
      (when (typep slotd 'virtual-slot-definition)
        (setf (virtual-slot-definition-function effective-slotd)
```

```
                (virtual-slot-definition-function slotd))
            (return)))
        effective-slotd))


;; Underlying access methods for invoking
;; virtual-slot-definition-function.

(defmethod clos:slot-value-using-class
           ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                     :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
        (funcall (virtual-slot-definition-function slotd)
                 :get
                 object)
      (call-next-method))))

(defmethod (setf clos:slot-value-using-class)
           (value (class virtual-metaclass) object slot-name)
  (format t "~% setf slot : ~A" slot-name)
  (let ((slotd (find slot-name (class-slots class)
                     :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
        (funcall (virtual-slot-definition-function slotd)
                 :set
                 object
                 value)
      (call-next-method))))

(defmethod clos:slot-boundp-using-class
           ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                     :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
        (funcall (virtual-slot-definition-function slotd)
                 :is-set
                 object)
      (call-next-method))))

(defmethod clos:slot-makunbound-using-class
           ((class virtual-metaclass) object slot-name)
  (let ((slotd (find slot-name (class-slots class)
                     :key 'slot-definition-name)))
    (if (typep slotd 'virtual-slot-definition)
        (funcall (virtual-slot-definition-function slotd)
                 :unset
```

```
                  object)
      (call-next-method))))

(defmethod clos:slot-exists-p-using-class
           ((class virtual-metaclass) object slot-name)
  (or (call-next-method)
      (and (find slot-name (class-slots class)
                   :key 'slot-definition-name)
           t)))


;; Example virtual slot which depends on a real slot.
;; Compile this separately after the virtual-metaclass etc.

(defclass a-virtual-class ()
  ((real-slot :initarg :real-slot :accessor real-slot
              :initform -1)
   (virtual-slot :accessor virtual-slot
                 :initarg :virtual-slot
                 :allocation :virtual
                 :function
                 'a-virtual-class-virtual-slot-function))
  (:metaclass virtual-metaclass))

(defun a-virtual-class-virtual-slot-function
       (key object &optional value)
  (ecase key
    (:get (let ((real-slot (real-slot object)))
            (if (<= 0 real-slot 100)
                (/ real-slot 100.0)
              (slot-unbound (class-of object)
                            object
                            'virtual-slot))))
    (:set (setf (real-slot object) (* value 100))
          value)
    (:is-set (let ((real-slot (real-slot object)))
               (<= real-slot 100)))
    (:unset (setf (real-slot object) -1))))
;; ----------------------- Virtual Slots --------------------
```

Compile the code above. Then make an object and access the virtual slot:

```
CL-USER 1 > (setf object (make-instance 'a-virtual-class))
#<A-VIRTUAL-CLASS 2067B064>

CL-USER 2 > (setf (virtual-slot object) 0.75)

 setf slot : VIRTUAL-SLOT
0.75

CL-USER 3 > (virtual-slot object)
0.75

CL-USER 4 > (real-slot object)
75.0
```

Note that when you call **(setf real-slot)** there is no output.since **(setf clos:slot-value-using-class)** is not called. Compare with **(setf virtual-slot).**

```
CL-USER 5 > (setf (real-slot object) 42)
42
```

Redefine **a-virtual-class** with **:optimize-slot-access nil**:

```
CL-USER 6 > (defclass a-virtual-class ()
              ((real-slot :initarg :real-slot
                          :accessor real-slot
                          :initform -1)
               (virtual-slot :accessor virtual-slot
                             :initarg :virtual-slot
                             :allocation :virtual
                             :function
                   'a-virtual-class-virtual-slot-function))
              (:metaclass virtual-metaclass)
              (:optimize-slot-access nil))
Warning: (DEFCLASS A-VIRTUAL-CLASS) being redefined in LISTENER
(previously in H:\tmp\vs.lisp).
Warning: (METHOD REAL-SLOT (A-VIRTUAL-CLASS)) being redefined in
LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF REAL-SLOT) (T A-VIRTUAL-CLASS)) being
redefined in LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD VIRTUAL-SLOT (A-VIRTUAL-CLASS)) being redefined
in LISTENER (previously in H:\tmp\vs.lisp).
Warning: (METHOD (SETF VIRTUAL-SLOT) (T A-VIRTUAL-CLASS)) being
redefined in LISTENER (previously in H:\tmp\vs.lisp).
#<VIRTUAL-METACLASS A-VIRTUAL-CLASS 21AD908C>
```

Now the standard accessors call **`clos:slot-value-using-class`**, so we see output when calling **`(setf real-slot)`**

```
CL-USER 7 > (setf (real-slot object) 42)

 setf slot : REAL-SLOT
42
```

16

Multiprocessing

LispWorks supports “lightweight” processes. The programming environment, for example, makes extensive use of this mechanism to create separate processes for the various tools.

On some platforms, LispWorks multiprocessing uses native threads. See Section 16.5 on page 167 for more details.

16.1 Introduction to processes

A process can be in one of three different states: *running*, *waiting*, and *inactive*. When a process is *waiting*, it is still active, but is waiting for the system to wake it up and allow its computation to restart. A process that is *inactive* has stopped, because it has an arrest “reason”.

For a process to be active (that is, running or waiting), it must have at least one run reason and no arrest reasons. If, for example, it was necessary to temporarily stop a process, it could temporarily be given an arrest reason. However the arrest reason mechanism is not commonly used in this manner.

The process that is currently executing is termed “the current process” and is the current value of the variable **`mp:*current-process*`**. The current process continues to be executed until either it becomes a waiting process, by calling **`mp:process-wait`** or **`mp:process-wait-with-timeout`**, or it allows itself to be

interrupted, by calling **mp:process-allow-scheduling** (or its current timeslice expires and it involuntarily relinquishes control).

The system runs the waiting process with the highest priority. If processes have the same priority then the system treats them equally and fairly. This is called round robin scheduling.

The simplest way to create a process is to use **mp:process-run-function**. This creates a process with the specified name which commences by applying the specified function to arguments. **mp:process-run-function** returns immediately and the newly created process runs concurrently.

16.2 The process programming interface

16.2.1 Creating a process

To create a new process, use **mp:process-run-function**.

16.2.2 Finding out about processes

The system initializes a number of processes on startup. These processes are specified by **mp:*initial-processes***.

The current process is specified by **mp:*current-process***. A list of all the current processes is returned by **mp:list-all-processes**. The function **mp:ps** is analogous to the UNIX command **ps**, and returns a list of the processes in the system, ordered by priority.

To find a process when you know its name, use **mp:find-process-from-name**. To find the name, when you have the process, use **mp:process-name**. The variable **mp:*process-initial-bindings*** specifies the variables that are initially bound in a process.

When a process has stopped, you can find a list of reasons why by calling **mp:process-arrest-reasons**. To obtain a list of the reasons why a process is running, call **mp:process-run-reasons**. Both these lists can be changed using **setf**, though it is not normally necessary to add arrest reasons.

16.2.3 Process Priorities

Each process has a priority and can either be runnable, blocked or suspended. If there is a runnable process with priority P, then no processes with priority less than P will run. When there are runnable processes with equal priority, they will be scheduled in a round-robin manner.

If a process with priority P is running and a blocked process with priority greater than P becomes runnable, the second process will run when the scheduler is next invoked (either explicity or at the next preemption tick).

To find the priority of a process, use **mp:process-priority**. This can be changed using **mp:change-process-priority**.

```
(mp:change-process-priority proc-1 10)
```

Another way to specify the priority is to create the process with **mp:process-run-function**, passing the keyword **:priority**:

```
(list
 (mp:process-run-function
  "SORTER-DOT" '(:priority 10) #'sorter #\.)
 (mp:process-run-function
  "SORTER-DASH" () #'sorter #\-))
```

16.2.4 Interrupting a process

To interrupt a running process, use **mp:process-interrupt**, or **mp:process-kill**. To break a process and enter the debugger, use **mp:process-break**.

To suspend a process until a predicate is **t**, use **mp:process-wait** or **mp:process-wait-with-timeout**. The function **mp:process-wait-function** returns a function that specifies a reason for the process waiting.

To control whether or not a process can be interrupted, use **mp:without-preemption** or **mp:without-interrupts**.

16.2.5 Multiprocessing

To start multiprocessing, use **mp:initialize-multiprocessing**. This function does not return until multiprocessing has terminated.

It is not necessary to use **mp:initialize-multiprocessing** when the LispWorks environment is already running. Note that, on Windows, Mac OS X, Linux and FreeBSD, the LispWorks images shipped do start the programming environment. If you create an image which does not start the programming environment, by using the **:environment nil** argument to **save-image**, then multiprocessing can be started in this new image as described below.

16.2.5.1 Starting multiprocessing interactively

You can call **mp:initialize-multiprocessing** from the REPL interface, which generates a default Listener process if no other processes are specified by **mp:*initial-processes***.

16.2.5.2 Multiprocessing on startup

There are three ways to make a LispWorks executable start multiprocessing on startup.

1. Use the **-multiprocessing** command line argument

2. Save an image which starts multiprocessing by doing

```
(save-image "mp-lispworks"
:restart-function 'mp:initialize-multiprocessing)
```

3. Use delivery to create the executable and pass the argument **:multiprocessing t** to **deliver**. The delivery function will be called automatically in a new process. See the *LispWorks Delivery User Guide* for more details.

LispWorks dynamic libraries always start multiprocessing on startup. See "Multiprocessing in a dynamic library" on page 145 for more information.

In all cases, **mp:*initial-processes*** can be used to control which processes are created on startup, as described in "Running your own processes on startup" on page 163.

Note: On Windows, Linux, FreeBSD and Mac OS X you cannot save a LispWorks image with multiprocessing running.

16.2.5.3 Running your own processes on startup

mp:*initial-processes* is a list of lists. Each list is used by the system as a set of arguments to **mp:process-run-function**. During initializing multiprocessing, the system does this:

```
(dolist (x mp:*initial-processes*)
  (apply 'mp:process-run-function x))
```

This script saves a LispWorks image which starts multiprocessing on restart and runs a user-defined process.

```
(load-all-patches)
(load "my-server-code")
(push '("Start Server" () start-my-server)
      mp:*initial-processes*)
(save-image "my-server"
            :remarks "My Server"
            :restart-function 'mp:initialize-multiprocessing
            :environment nil)
```

See **save-image** in the *LispWorks Reference Manual* for a description of how to save an image.

16.2.6 Values across processes

16.2.6.1 Returning a value from another process

Rather than using global variables to pass values between processes, you can use closures instead. For example:

```
(defun send-with-result (process function)
  (let ((remote-result :none))
    (flet ((resultp ()
             (listp remote-result))
           (run-it ()
             (setq remote-result
                   (multiple-value-list (funcall function)))))
      (mp:process-send process (list #'run-it))
      (mp:process-wait "Waiting for result" #'resultp)
      (values-list remote-result))))
```

16.2.6.2 Accessing symbol values across processes

Use **mp:symeval-in-process** to read the value of a dynamically bound symbol in a given process.

(setf mp:symeval-in-process) can set the value of such a symbol.

16.2.7 Stopping and unstopping processes

This section describes a typical way of using **mp:process-stop** and **mp:process-unstop**.

Suppose a pool of "worker" processes is managed by a "manager" process. A process in the worker pool marks itself as available for work, and then calls **mp:process-stop**. The manager process later finds a worker process that is marked as available for work, puts the work in a place known to the worker process, and then calls **mp:process-unstop** on the worker process.

For this scheme to work properly, the check of whether the worker is available needs to include a call to **mp:process-stopped-p**. Otherwise, it is possible for the following sequence of events to occur:

1. A worker marks itself as available.
2. The manager process finds the worker and gives it the work.
3. The manager process calls **mp:process-unstop** on the worker.
4. The worker process proceeds and calls **mp:process-stop**, and never wakes up.

To guard against this possibility, then the manager should call **mp:process-stopped-p** when finding the worker in the second step above. Alternatively, it could check the result of **mp:process-unstop**.

16.2.8 Example

The following example allows two (or more) multiplication tables to be printed out simultaneously.

First, the function to print out a multiplication table.

```
(in-package "USER")

(defun print-table (number total stream)
(do ((i 1 (+ i 1)))
((> i total))
 (format stream "~S X ~S = ~S~%" number i (* i number))
 (mp:process-allow-scheduling)))
```

Note the use of **mp:process-allow-scheduling** to allow the process to be interrupted once during each iteration of the do loop.

Now we define the function that calls **print-table** within multiprocessing:

```
(defun process-print-table (name number total)
  (mp:process-run-function name nil
    #'print-table number total *standard-output*))
```

The **nil** argument is used because no keywords are specified.

process-print-table can now be called from two separate Listener windows to print out different multiplication tables simultaneously, for example:

```
(process-print-table "t1" 5 50)
```

in one Listener and:

```
(process-print-table "t2" 6 50)
```

in another Listener.

16.3 Locks

Locks can be used to control access to shared data by several processes.

The two main symbols used in locking are the function **make-lock**, to create a lock, and the macro **with-lock**, to execute a body of code while holding the specified lock.

A lock has a name (a string) and several other components. The printed representation of a lock shows the name of the lock and whether it is currently locked. Additionally if the lock is locked it shows the name of the process holding the lock, and how many times that process has locked it. For example:

```
#<MP:LOCK "my-lock" Locked 2 times by "My Process" 2008CAD8>
```

The function **lock-owner** returns the process that locked a given lock.

mp:make-lock *Function*

mp:make-lock &key *important-p* &allow-other-keys

Creates a lock object. If **important-p** is **t** the lock is added to the list held in the global variable **mp:*important-locks***. The function **mp:free-important-locks** frees all important locks associated with a given process (or all the important locks if called on **nil**). Other keywords should be names of the lock components.

mp:process-lock *Function*

mp:process-lock *lock* &optional *whostate timeout*

Blocks the current process until the lock is claimed or **timeout** elapses if it has been specified. Returns **t** if lock was claimed, **nil** otherwise.

mp:process-unlock *Function*

mp:process-unlock *lock* &optional *errorp*

Releases the lock. If *errorp* is non-**nil** it signals an error if the current process does not own the lock. The default value of *errorp* is **t**.

mp:with-lock *Macro*

mp:with-lock ((*lock* &rest *lock-args*) &body *body*

Executes the body with lock held. Arguments to pass on to **mp:process-lock** are specified using *lock-args*.

The following accessor is available for locks: **lock-name**.

16.4 Timers

Use timers to run code after a specified time has passed. You can schedule a timer to run once or repeat at regular intervals, and you can unschedule it before it expires. For details see **mp:make-timer** and **mp:schedule-timer** in the *LispWorks Reference Manual*.

16.4.1 Timers and multiprocessing

Timers run in unpredictable threads, therefore it is not safe to run code that interacts with the user directly. The recommended solution is something like

```
(mp:schedule-timer-relative
 (mp:make-timer 'capi:execute-with-interface
                interface
                'capi:display-message "Time's up")
 5)
```

or

```
(mp:schedule-timer
 (mp:make-timer 'capi:execute-with-interface
                interface
                'capi:display-message "Lunchtime")
 (* 4 60 60))
```

where *interface* is an existing CAPI interface on the screen.

Timers actually run in the process that is current when the scheduled time is reached. This is likely to be The Idle Process in cases where LispWorks is sleeping, but it is inherently unpredictable.

16.4.2 Input and output for timer functions

I/O streams default to the standard input and output of the process, which is initially **`*terminal-io*`** in the case of The Idle Process.

16.5 Native threads and foreign code

Support for native threads differs between platforms as described in this section.

16.5.1 Native threads on Windows, Mac OS X, Linux and FreeBSD

Each Lisp **`mp:process`** has a separate native thread. You can have many runnable **`mp:process`** objects/native threads, but Lisp code can only run in one thread at a time and a lock is used to enforce this. This can limit performance on multi-CPU machines.

When a foreign function is called using the FLI, the lock is released until the function returns. This allows other Lisp threads to run, for instance while waiting for a database query to execute.

You can call back into Lisp using **`fli:define-foreign-callable`** in any thread, without any other setup.

Note: In a LispWorks 5.0 executable on Microsoft Windows you must first call **`setup-for-alien-threads`** before calling back into Lisp, but in LispWorks 5.1 and later versions this setup is handled automatically (and **`setup-for-alien-threads`** does not exist).

Threads running Lisp code can be rescheduled preemptively, so if you call into Lisp from more than one thread simultaneously and one request takes a long time then it will not delay the requests in other threads.

16.5.2 Native threads on other platforms

Lisp uses a single native thread and implements user level threads to support **`mp:process`**.

You can only call back into Lisp from its single native thread.

Note: This section applies to LispWorks for UNIX only (not LispWorks for Linux, FreeBSD or Macintosh).

16.6 Example

The following is an informal example of multi-processing with a single process (other then the idle process), namely a top-loop. Once it has started up, try **`(mp:ps)`**.

```
(in-package "CL-USER")

;;; (guarantee-processes) will start up
;;; multiprocessing with a top-level loop
;;; in this example,
;;; use *base-process* to ensure that base
;;; process will only be pushed
;;; onto *initial-processes* once, no matter how
;;; many times guarantee-processes is called

(defvar *base-process*
  '("base-process" nil base-process-function))

;;; the base process consists of a top-level
;;; loops with restarts which allow control of
;;; return in the event of an error -- to see
;;; these in action, evaluate (guarantee-processes)
;;; and then an unbound variable.

;;; Note that starting and stopping multiprocessing is not
;;; relevant if Common LispWorks is already running. This example
;;; is included for illustration only.

(defun base-process-function ()
  (with-simple-restart
   (abort "Return from multiprocessing")
   (loop
    (with-simple-restart
     (abort "Return to top-level-loop")
     (system:%top-level))))
  (mp::stop-multiprocessing))

;;; simple startup of multiprocessing with one
;;; process (apart from the idle process)

(defun guarantee-processes ()
   (unless mp:*multiprocessing*
   (pushnew *base-process*
     mp:*initial-processes*)
   (mp:initialize-multiprocessing)))
```

17

Common Defsystem

17.1 Introduction

When an application becomes large, it is usually prudent to divide its source into separate files. This makes the individual parts of the program easier to find and speeds up editing and compiling. When you make a small change to one file, just recompiling that file may be all that is necessary to bring the whole program up to date.

The drawback of this approach is that it is difficult to keep track of many separate files of source code. If you want to load the whole program from scratch, you need to load several files, which is tedious to do manually, as well as prone to error. Similarly, if you wish to recompile the whole program, you must check every file in the program to see if the source file is out of date with respect to the object file, and if so re-compile it.

To make matters more complicated, files often have interdependencies; files containing macros must be loaded before files that use them are compiled. Similarly, compilation of one file may necessitate the compilation of another file even if its object file is not out of date. Furthermore, one application may consist of files of more than one source code language, for example Lisp files and C files. This means that different compilation and loading mechanisms are required.

The Common LispWorks system tools, and the system browser in particular, are designed to take care of these problems, allowing consistent development and maintenance of large programs spread over many files. A system is basically a collection of files that together constitute a program (or a part of a program), plus rules expressing any interdependencies which exist between these files.

You can define a system in your source code using the **defsystem** macro. Once defined, operations such as loading, compiling and printing can be performed on the system as a whole. The system tools ensure that these operations are carried out completely and consistently, without doing unnecessary work.

A system may itself have other systems as members, allowing a program to consist of a hierarchy of systems. Each system is treated independently of the others, and can be used to collect related pieces of code within the overall program. Operations on higher-level systems are invoked recursively on member systems.

17.2 Defining a system

A system is defined with a **defsystem** form in an ordinary Lisp source file. This form must be loaded into the Lisp image in order to define the system in the environment. Once loaded, operations can be carried out on the system by invoking Lisp functions, or, more conveniently, by using the system browser.

For example, the expression:

```
CL-USER 5 > (compile-system 'debug-app :force t)
```

would compile every file in a system called **debug-app**.

Note: When defining a hierarchy of systems, the leaf systems must be defined first — that is, a system must be declared before any systems that include it.

By convention, system definitions are placed in a file called **defsys.lisp** which usually resides in the same directory as the members of the system.

The full syntax of **defsystem** is given in the *LispWorks Reference Manual*. Below is a brief introduction.

17.2.1 DEFSYSTEM syntax

defsystem *Macro*

defsystem ***system-name options*** **&key** ***members rules***

system-name A symbol used as the name of the system. If a string is given, it is interned in the current package.

options Any of a number of options that can be specified.

members The members of the system. These may be files of Common Lisp source code, foreign source code, or other systems.

rules A set of rules describing the requirements for and order in which compilation and loading of the system members should take place.

See the following sections for more information about these parameters.

17.2.2 DEFSYSTEM options

Options may be specified to **defsystem** which affect the behavior of the system as a whole. For example, **:package** specifies a default package into which files in the system are compiled and loaded if the file itself does not contain its own package declaration. The **:default-pathname** option tells the system tools where to find files which are not expressed as a full pathname.

17.2.3 DEFSYSTEM members

The **:members** keyword to **defsystem** is used to specify the members of a system. The argument given to **:members** is a list of strings. A system member is either a file or another system, identified by a name. If a full pathname is given then the function **pathname-name** is used to identify the name of the member. Thus, for example, the name of a member expressed as **/u/neald/foo.lisp** is **foo**.

The behavior of any member within a system can be constrained by supplying keyword arguments to the member itself. So, for example, specifying the

`:source-only` keyword ensures that only the source file for that member is ever loaded.

17.2.4 DEFSYSTEM rules

Rules may be defined in a system which modify the default behavior of that system, ensuring, for instance, that certain files are always loaded or compiled before others.

When you invoke an action such as compiling a system, the following happens by default:

- Each member of the system is considered in turn, in the order they are given in the system definition.
- If the member is itself a system then the action is performed on that system too, and so on recursively.
- If the member is a file and action-specific constraints are satisfied, the file action is inserted into a *plan*.

  For example, in the case of compiling, a “compile this file” event is put into the plan if the source file is newer than the object file.
- After the plan has been assembled, it can be viewed or executed.

This behavior can be modified by describing dependencies between the members using *rules*. These are specified using the **`:rules`** keyword to **`defsystem`**.

A rule has three components:

The target(s). The action that is performed if the rule executes successfully.

This is an action-member description like **`:compile "foo"`**. The member can be an actual member of the system or **`:all`** (meaning the rule should apply to each member of the system).

The actions that the target(s) are **`:caused-by`**.

The actions that cause the rule to execute successfully.

This is a list of action-member descriptions. The member of each of these descriptions should be either a real system member, or **`:previous`**, which means all members listed before the member of the target in the system description.

If any of these descriptions are already in the current plan (as a result of other rules executing successfully, or as a result of default system behavior), they trigger successful execution of this rule.

The actions that the target(s) **`:requires`**.

The actions that need to be performed before the rule can execute successfully.

This is a list of action-member descriptions that should be planned for before the action on the target(s). Again, each member should either be a real member of the system, or **`:previous`**.

The use of the keyword **`:previous`** means, for example, that you can specify that in order to compile a file in the system, all the members that come before it must be loaded.

When the action and member of a target are matched during the traversal of the list of members, the target is inserted into the plan if either of the following are true:

- any of the action-member descriptions in the **`:caused-by`** clause is already in the plan, or
- any implicit conditions (such as the source file being newer than the object file) are satisfied.

If the target is put into the plan then other targets are inserted beforehand if the action-member description of any **`:requires`** clause is not already in the plan.

17.2.5 Examples

Consider an example system, **demo**, defined as follows:

```
(defsystem demo (:package "USER")
  :members ("parent"
            "child1"
            "child2")
  :rules ((:in-order-to :compile ("child1" "child2")
          (:caused-by (:compile "parent"))
          (:requires (:load "parent")))))
```

This system compiles and loads members into the **USER** package if the members themselves do not specify packages. The system contains three members — **parent**, **child1**, and **child2** — which may themselves be either files or other systems. There is only one explicit rule in the example. If **parent** needs to be compiled (for instance, if it has been changed), then this causes **child1** and **child2** to be compiled as well, irrespective of whether they have themselves changed. In order for them to be compiled, **parent** must first be loaded.

Implicitly, it is always the case that if any member changes, it needs to be compiled when you compile the system. The explicit rule above means that if the changed member happens to be **parent**, then *every* member gets compiled. If the changed member is not **parent**, then **parent** must at least be loaded before compiling takes place.

The next example shows a system consisting of three files:

```
(defsystem my-system
  (:default-pathname "~/junk/")
   :members ("a" "b" "c")
   :rules ((:in-order-to :compile  ("c")
            (:requires (:load "a"))
            (:caused-by (:compile "b")))))
```

What plan is produced when all three files have already been compiled, but the file **b.lisp** has since been changed?

First, file **a.lisp** is considered. This file has already been compiled, so no instructions are added to the plan.

Second, file **b.lisp** is considered. Since this file has changed, the instruction *compile b* is added to the plan.

Finally file **`c.lisp`** is considered. Although this has already been compiled, the clause

```
(:caused-by (:compile "b"))
```

causes the instruction *compile c* to be added to the plan. The compilation of **`c.lisp`** also requires that **`a.lisp`** is loaded, so the instruction *load a* is added to the plan first. This gives us the following plan:

1. Compile **`b.lisp`**.
2. Load **`a.lisp`**.
3. Compile **`c.lisp`**.

18

The Parser Generator

18.1 Introduction

The parser generator generates an LALR parser from a specification of a grammar. The parser generator has a simple facility for the static resolution of ambiguity in the grammar and supports an automatic run-time error correction mechanism as well as user-defined error correction. Semantic actions can be included in the rules for the grammar by specifying Lisp forms to be evaluated when reductions are performed.

For further details on LALR parsing, see *Compilers, Principles Techniques and Tools*, by Aho, Sethi and Ullman, publishers Addison Wesley, 1986.

Load the parser generator by **`(require "parsergen")`**.

18.2 Grammar rules

The parser generator is accessed by the macro **`defparser`**, described below:

defparser *Macro*

`defparser` ***name*** **`{`*rules*`}*`**

name	The name to be used for the parsing function. The remainder of the macro form specifies the reduction rules and semantic actions for the grammar.
rules	The rules specified in a **defparser** form are of two types, *normal rules* and *error rules*, described below.

Each *normal rule* corresponds to one production of the grammar to be parsed:

```
((non-terminal {grammar-symbol}*)  {form}*)
```

The *non-terminal* is the left-hand side of the grammar production and the list of grammar symbols defines the right-hand side of the production. (The right-hand side may be empty.) The list of forms specifies the semantic action to be taken when the reduction is made by the parser. These forms may contain references to the variables **$1 … $n**, where **n** is the length of the right hand side of the production. When the reduction is done, these variables are bound to the semantic values corresponding to the grammar symbols of the rule.

18.2.1 Example

If a grammar contains the production:

```
expression -> expression operator expression
```

with a semantic representation of a list of the individual semantic values, the Lisp grammar would contain the rule:

```
((expression expression operator expression)  (list $1 $2 $3))
```

Error productions of the form:

```
((nt :error)  (some error behavior))
```

are explained in the section below.

The first rule of the grammar should be of the form:

```
((nt nt1)  $1)
```

where the non-terminal *nt* has no other productions and **nt1** serves as the main “top-level” non-terminal.

18.2.2 Resolving ambiguities

If the grammar is ambiguous, there is conflict between rules of the grammar: either between reducing with two different rules or between reducing by a rule and shifting an input symbol. Such a conflict is resolved at parser generation time by selecting the highest priority action, where the priority of a reduce action is determined by the closeness of the rule to the beginning of the grammar. A priority is assigned to a shift by associating it with the rule that results in the shift being performed.

For example, if the grammar contains the two rules:

- Rule a: *statement* -> **`:if`** *expression* **`:then`** *statement* **`:else`** *statement*
- Rule b: *statement* -> **`:if`** *expression* **`:then`** *statement*

this results in a conflict in the parser between a shift of **`:else`**, for rule a, and a reduce by rule b. This conflict may be resolved by listing rule a earlier in the grammar than rule b. This ensures that the shift is always done.

Note that ambiguities cannot always be resolved successfully in this way. In this example, if the ambiguity is resolved the other way around, by listing rule b first, this results in the **`if … then …`** part of an **`if … then … else …`** statement being reduced, and a syntax error is produced for the **`else`** part.

During parser generation, any conflicts between rules are reported, together with information about how the conflict was resolved.

18.3 Functions defined by defparser

The form **`(defparser`** *name grammar***`)`** defines a number of functions. The main function *name* is defined as the parsing function. For example:

```
(defparser my-parser .. grammar .. )
```

defines the function

`my-parser` ***lexer*** **`&optional`** ***symbol-to-string*** **`=>`**

lexer specifies the lexical analyzer function to be used. The optional argument *symbol-to-string* should be a function mapping grammar symbols to strings for printing purposes. The default value of *symbol-to-string* is the function **`identity`**.

defparser also defines functions corresponding to the individual actions of the parser.

Normal actions are named:

name*-action*index

and error actions are named:

name*-error-action*index

where *name* here is the name as given to **defparser** and *index* is the number of the rule or error rule in the grammar.

All function names are interned in the current package when **defparser** is called.

18.4 Error handling

The parser supports automatic error correction of its input. The strategy used involves attempting to either push a new token onto the input, replacing an erroneous symbol, or discarding an erroneous symbol. Such action is only taken if it is guaranteed that the parser can continue parsing and read at least one more symbol from its input.

If the correction strategy fails, then error recovery is invoked.

The parser allows the inclusion of grammar productions of the form:

***non-terminal* -> :error**

This means that the parser accepts an erroneous string of tokens as constituting an occurrence of the non-terminal. Such productions may be used to skip over portions of input when attempting to recover from an error. The action associated with such an error is specified by a form in the same way as for ordinary actions. The action may perform manipulation of the parser state and input.

18.5 Interface to lexical analyzer

The lexical analyzer function that is passed to the parser is expected to be a function of zero arguments that returns two values each time it is called. The first value is the next token on the input and the second value is the semantic

value corresponding to that token. If there is no more input, then the lexical analyzer may return either the token **:eoi** or **nil**.

For example:

```
(defparser my-parser
   ...)

(defun my-lexer (stream)
   .. read next token from stream ..
  (values token value))
(defun my-symbol-to-string (symbol)
   .. returns a string ..)
(defun my-parse-stream (stream)
  (let ((lexer #'(lambda () (my-lexer stream))))
    (my-parser lexer #'my-symbol-to-string)))
```

Note that during error correction, the parser may push extra tokens onto the input, in which case they are given the semantic value **nil**. The semantic actions should therefore be capable of dealing with this situation. Manipulation of the input (e.g. pushing extra tokens) is done within the parser generator and the lexical analyzer need not concern itself with this.

18.6 Example

The following example shows a simple grammar for a very small subset of English.

```
(defpackage "ENGLISH-PARSER")
(in-package "ENGLISH-PARSER")
(use-package '(parsergen))

;;; Define the parser itself.

(defparser english-parser
  ((bs s) $1)
  ((s np vp)
   `(,$1 ,$2))
  ((bnp :adj bnp)
   `(,$1 ,$2))
  ((bnp bnp relp)
   `(,$1 ,$2))
  ((bnp :noun) $1)
  ((relp :rel vp)
   `(,$1 ,$2))
  ((vp :verb np locp)
   `(,$1 ,$2 ,$3))
  ((vp :verb locp)
   `(,$1 ,$2))
  ((vp :verb np)
   `(,$1 ,$2))
  ((vp :verb)
   $1)
  ((np :art bnp locp)
   `(,$1 ,$2 ,$3))
  ((np :art bnp)
   `(,$1 ,$2))
  ((np bnp) $1)
  ((locp :loc np)
   `(,$1 ,$2)))

;;; The lexer function.

;;; The basic lexing function

(defvar *input*)
(defun lex-english ()
  (let ((symbol (pop *input*)))
    (if symbol (get-lex-class symbol)
      nil)))

;;; Getting syntactic categories.

(defparameter *words*
 '((the :art)(a :art)(some :art)(ate :verb)(hit :verb)
```

```
(cat :noun)(rat :noun)(mat :noun)(which :rel)(that :rel)
(who :rel)(man :noun)(big :adj)(small :adj)(brown :adj)
(dog :noun)(on :loc)(with :loc)(behind :loc)(door :noun)
(sat :verb)(floor :noun)))

(defun get-lex-class (word)
   (values
    (or (cadr (assoc word *words*))
        :unknown)
    word))

;;; The main function -- note bindings of globals (these
;;; are exported from the parsergen package).

(defun parse-english (input)
   (let ((*input* input))
     (english-parser #'lex-english)))
```

The following example session shows the parsing of some sentences.

```
ENGLISH-PARSER 34 > (parse-english '(the cat sat on the
                       mat))
((THE CAT) (SAT (ON (THE MAT))))

ENGLISH-PARSER 35 > (parse-english '(the big brown dog
                        behind the door ate the cat
                        which sat on the floor))
((THE (BIG (BROWN DOG)) (BEHIND (THE DOOR)))
 (ATE (THE (CAT (WHICH (SAT (ON (THE FLOOR)))))))
```

19

Dynamic Data Exchange

19.1 Introduction

Dynamic data exchange (DDE) involves passing data and instructions between applications running under the Microsoft Windows operating system. Typically the data is passed in the form of a string, which is interpreted when it is received. One application acts as a *server* and the other as a *client*.

19.1.1 Types of transaction

The server is normally a passive object, which waits for a client object to tell it what to do. The client can communicate with the server in four ways:

- The client can issue a *request transaction* to the server. This means the client is asking for some information about the server application.
- The client can issue a *poke transaction*. This means the client is passing data to be stored by the server application.
- The client can issue an *execute transaction*. This means the client is asking the server to get the server application to run a command.
- The client can ask the service to set up an *advise loop*, or to close an existing advise loop. An advise loop causes the server to communicate with the client whenever a specified change occurs in the server application.

19.1.2 Conversations, servers, topics, and items

For a transaction to take place between a client and a server, a conversation must be established. A conversation is established when a client makes a request by broadcasting a service name and topic name, and a server responds. Transactions can then take place across the conversation. When no more transactions are to be made, the conversation is terminated.

The following list identifies the elements involved with client/server activity:

conversation	A conversation is established when a server responds to a client.
service name	A service name is a string broadcast by a client hoping to establish a conversation with a server that recognizes the service name. The service name is usually clearly related to the server application name.
topic name	The topic name identifies what the conversation between client and server is to be about. For example, it could be the name of a file that is open in the server application. Each topic is attached to one particular server. A server can have many topics.
item name	The item usually identifies an element of the file identified by the topic which should be read (in the case of a request) or written to (in the case of a poke). For example, it might refer to a cell in a spreadsheet document.

19.1.3 Advise loops

An advise loop instructs the server to inform the client when data in the server's application changes. Advise loops are set up across a conversation, and closing the conversation closes the advise loop.

An advise loop is identified by an item and a key. The key is included to allow any number of uniquely identifiable advise loops to be set up on the same server/topic/item combination.

A successfully established advise loop is also known as a link. When a change occurs to item, the link informs the client by causing it to execute a function.

There are two types of link: the warm link which only informs the client that a change to item has occurred, and the hot link which also sends the new data across.

19.1.4 Execute transactions

When a client issues an execute transaction to a server, the command to be executed is transferred as a string. This involves the marshalling of the command and its arguments into a suitable string format. The standard format of such a string is:

```
[command(arg1,arg2,...)]
```

19.2 Client interface

19.2.1 Opening and closing conversations

A LispWorks client can open a conversation by using **dde-connect**, which takes a service designator and a topic designator as its arguments. If successful, a conversation object is returned which can be used to refer to the conversation. Conversations are closed by the LispWorks client at the end of a transaction by using **dde-disconnect**.

dde-connect *Function*

dde-connect ***service topic*** **&key** ***class errorp***

The function **dde-connect** attempts to create a conversation with a DDE server specified by *server*, on the topic given by *topic*.

dde-disconnect *Function*

dde-disconnect ***conversation***

The function **dde-disconnect** disconnects the conversation object specified by *conversation*. The conversation may no longer be used.

Another method for managing conversations uses **with-dde-conversation** to bind a conversation with a server across a body of code. If no conversation is

available for **with-dde-conversation**, then one is automatically opened, *body* is executed, and then the conversation is closed.

with-dde-conversation *Macro*

with-dde-conversation (*conv service topic* &key *errorp new-conversation-p)* &body *body*

The macro **with-dde-conversation** dynamically binds a conversation with a server across the scope of a body of code specified by *body*. The argument *conv* is bound to a conversation with the server specified by *service*, and the topic specified by *topic*.

19.2.2 Automatically managed conversations

There is an alternative to manually establishing a conversation and then disconnecting it once all transactions between server and client are concluded: the automatically managed conversation. Client functions that end with a ***** conduct automatically managed conversations.

A function handling an automatically managed conversation takes a service designator and topic designator as two of its arguments, and either automatically establishes a conversation with a server responding to the service designator / topic designator pair, or uses an existing equivalent conversation. For the purpose of brevity, functions conducting automatically managed conversations will not be explicitly documented in this chapter. See the *LispWorks Reference Manual* for full details.

19.2.3 Advise loops

A LispWorks client can set up an advise loop across a conversation using **dde-advise-start**, which takes a *conversation* (or a *service* designator / *topic* designator pair in the case of an automatically managed conversation using **dde-advise-start***), an *item*, and a *key* as its main arguments. The *key* argument defaults to the conversation name, and can be used to distinguish between multiple advise loops established on the same service / topic / item group.

dde-advise-start *Function*

dde-advise-start ***conversation item*** **&key** ***key function format datap type errorp***

The **dde-advise-start** function sets up an advise loop for the data item specified by *item* on the specified *conversation*.

dde-client-advise-data *Generic Function*

dde-client-advise-data ***key item data*** **&key &allow-other-keys**

The generic function **dde-client-advise-data** is the default function called when an advise loop informs a client that the data monitored by the loop has changed. By default it does nothing, but it may be specialized on the object used as the key in **dde-advise-start** or **dde-advise-start***, or on a client conversation class if the default key is used.

define-dde-client *Macro*

define-dde-client ***name*** **&key** ***service class***

The macro **define-dde-client** defines a mapping from the symbol *name* to the DDE service name with which to establish a conversation, and the conversation class to use for this conversation. The argument *service* is a string which names the DDE service. It defaults to the print-name of *name*. The argument *class* is a subclass of **dde-client-conversation** which is used for all conversations with this service. It defaults to **dde-client-conversation**. Specifying a subclass allows various aspects of the behavior of the conversation to be specialized.

The following is an example of how to set up an advise loop. The first step defines a client conversation class, called **my-conv**.

```
(defclass my-conv (dde-client-conversation)
  ())
```

The function **define-dde-client** can now be used to define a specific instance of the **my-conv** class for referring to a server application that responds to the service name "**FOO**".

```
(win32:define-dde-client :foo :service "FOO" :class my-conv)
```

The next step defines a method on **dde-client-advise-data** which returns a string stating that the item has changed.

```
(defmethod dde-client-advise-data ((self my-conv) item data &key
&allow-other-keys)
  (format t "~&Item ~s changed to ~s~%" item data))
```

Finally, the next command starts the advise loop on the server **foo**, with the topic name "**file1**", to monitor the item "**slot1**".

```
(win32:dde-advise-start* :foo "file1" "slot1")
```

When the value of the item specified by "**slot1**"" changes, the server calls **dde-client-advise-data** which returns a string, as described above.

The *function* argument of **dde-advise-start** and **dde-advise-start*** specifies the function called by the advise loop when it notices a change to the item it is monitoring. The function is **dde-client-advise-start** by default. A different function can be provided, and should have a lambda list similar to the following:

key item data **&key** ***conversation*** **&allow-other-keys**

The arguments *key* and *item* identify the advise loop, or link. The argument *data* contains the new data for hot links; for warm links it is **nil**.

Advise loops are closed using **dde-advise-stop** or **dde-advise-stop***.

dde-advise-stop *Function*

dde-advise-stop ***conversation item*** **&key** ***key format errorp disconnectp no-advise-ok***

The function **dde-advise-stop** removes a particular link from *conversation* specified by *item*, *format* and *key*. If *key* is the last key for the *item* / *format* pair, the advise loop for the pair is terminated.

19.2.4 Request and poke transactions

LispWorks clients can issue request and poke transactions across a conversation using **dde-request** and **dde-poke**, which take a *conversation* (or a *service* designator / *topic* designator pair in the case of an automatically managed con-

versation), and an *item* as their main arguments. In the case of a poke transaction, data to be poked into *item* must also be provided.

In the case of a successful request transaction with **dde-request** or **dde-request***, the data contained in *item* is returned to the LispWorks client by the server.

In the case of a successful poke transaction with **dde-poke** or **dde-poke***, the data provided is poked into *item* by the server.

The accessor **dde-item** (or **dde-item*** for automatically managed conversations) can perform request and poke transactions. See the *LispWorks Reference Manual* for more details.

dde-item *Accessor*

dde-item ***conversation item*** **&key** ***format type errorp***

The accessor **dde-item** performs a request transaction when read. It performs a poke transaction when set.

dde-poke *Function*

dde-poke ***conversation item data*** **&key** ***format type errorp => result***

The function **dde-poke** issues a poke transaction on *conversation* to set the value of the item specified by *item* to the value specified by *data*. The argument *item* should be a string, or a symbol. If it is a symbol its print name is used.

dde-request *Function*

dde-request ***conversation item*** **&key** ***format type errorp***

The function **dde-request** issues a request transaction on *conversation* for the specified *item*. The argument *item* should be a string, or a symbol. If it is a symbol its print name is used.

19.2.5 Execute transactions

A client can issue an execute transaction across a conversation, or in the case of an automatically established conversation, to a recognized server. There is no need to specify a topic, as an execute transaction instructs the server application to execute a command.

The command and its arguments are issued to the server in the form of a string in a standard format (see "Execute transactions" on page 189). LispWorks provides two ways of issuing an execute transaction, namely **dde-execute-string** and **dde-execute-command** (and the corresponding ***** functions that automatically manage conversations).

dde-execute-string *Function*

dde-execute-string ***conversation command*** **&key** ***errorp***

The function **dde-execute-string** takes the command to issue in the form of an appropriately formatted string. The following example shows how **dde-execute-string*** can issue a command to a server designated by **:excel** on the topic **:system**, in order to open a file called **foo.xls**:

```
(win32:dde-execute-string* :excel :system "[open(\"foo.xls\")]")
```

dde-execute-command *Function*

dde-execute-command ***conversation command arg-list*** **&key** ***errorp***

The function **dde-execute-command** takes the command to issue, and its arguments, and marshals these into an appropriate string for you. The following example shows how **dde-execute-command*** can issue the same command as in the previous example:

```
(win32:dde-execute-command* :excel :system ‘open ‘("foo.xls"))
```

19.3 Server interface

19.3.1 Starting a DDE server

To provide a LispWorks application with a DDE server, the following three steps should be followed: define a specialized server class using **define-dde-server**, provide the server class with the functionality it requires by specializing methods on it and/or using **define-dde-server-function**, and finally, start an instance of the server using **start-dde-server**

define-dde-server *Macro*

define-dde-server ***class-name service-name***

The macro **define-dde-server** defines a class for a Lisp DDE server. The class inherits from **dde-server**.

define-dde-server-function *Macro*

define-dde-server-function ***name-and-options transaction* (*binding**) *form****

The macro **define-dde-server-function** is used to define a server function, called *name*, which is called when a specific transaction occurs.

The defined function may either be attached to a server object (possibly only for a particular topic class) or to a dispatching topic object.

start-dde-server *Function*

start-dde-server ***name***

The function **start-dde-server** creates an instance of a server of the class specified by *name* which then starts accepting transactions. If successful the function returns the server, otherwise **nil** is returned.

You need to call **start-dde-server** in a thread that will process Windows messages. This can either be done by using **capi:execute-with-interface** to run it in the thread of an application's main window (if there is one) or by running it in a dedicated thread as in the example in

the *LispWorks Reference Manual*. DDE callbacks will happen in this thread.

The next command line shows how to use **define-dde-server** to define a server class called **foo-server** that has the service name "**FOO**".

```
(win32:define-dde-server foo-server "FOO")
```

It is usual to provide the new server class with some functionality. The next command illustrates how to define a server function called **test**, which takes a string as an argument, and prints this to the standard output. For convenience, the system topic is used, though usually it is better to define your own topic.

```
(win32:define-dde-server-function (bar :topic :system)
    :execute
    ((x string))
  (format t "~&~s~%" x)
  t)
```

Finally, a **foo-server** can be started using **start-dde-server**

```
(win32:start-dde-server 'foo-server)
```

This function returns the server object, which responds to requests for conversations with the service name "**FOO**", and accepts execute transactions for the function **test** in the "System" topic.

19.3.2 Handling poke and request transactions

Poke and request transactions issued to a server object are handled by defining a method on each of the generic functions **dde-server-poke** and **dde-server-request**.

dde-server-poke *Generic Function*

dde-server-poke ***server topic item data*** **&key** ***format*** **&allow-other-keys**

The generic function **dde-server-poke** is called in response to a poke transaction. A method specializing on the classes of *server* and *topic* should poke the data given by *data* into the item specified by *item*.

dde-server-request *Generic Function*

dde-server-request ***server topic item*** **&key** ***format*** **&allow-other-keys**

The generic function **dde-server-request** is called in response to a request transaction. A method specializing on the classes of *server* and *topic* should return the data in *item*.

19.3.3 Topics

DDE servers respond to connection requests containing a service name and a topic name. The service name of a server is the same for any conversation whereas the topic name may vary from conversation to conversation, and identifies the context of the conversation. Typically, valid topics correspond to open documents within the application, so the set of valid topics varies from time to time. In addition, all servers implement a topic called "**System**", which contains a standard set of items that can be read.

The LispWorks DDE interface supports three types of topics:

1. General topics

   A general topic is an instance of a user-defined topic class. The actual set of topics available may vary from time to time as the application is running.

2. Dispatching topics

   A dispatching topic has a fixed name, and is available at all times that the server is running. It supports a fixed set of items, and each of these items has Lisp code associated with it to implement these items.

3. The system topic.

   The system topic is provided automatically by the LispWorks DDE interface. However, a mechanism is provided to extend the functionality of the system topic by handling additional items.

19.3.3.1 General topics

To use general topics, the LispWorks application must define one or more subclasses of **dde-topic**. If an application supports only a single type of document, it will typically require only one topic class. If several different types of

document are supported, it may be convenient to define a different topic class for each type of document.

If the application uses general topics, it should define a method on the **dde-server-topics** generic function, specializing on the application's server class.

dde-server-topics *Generic Function*

dde-server-topics ***server***

The generic function **dde-server-topics** returns a list of the available general topics on a given server. A suitable method specializing on the server class should be defined. Dispatching topics should not be returned, as they are handled automatically by LispWorks. If you do not provide a **dde-server-topics** method, the default method returns **:unknown**, which prevents the DDE server from responding to the topics request.

19.3.3.2 Dispatching topics

A dispatching topic is a topic which has a fixed name and always exists. Dispatching topics provide dispatching capabilities, whereby appropriate application-supplied code is executed for each supported transaction. Dispatch topics are defined using **define-dde-dispatch-topic**.

define-dde-dispatch-topic *Macro*

define-dde-dispatch-topic ***name*** **&key** ***server topic-name***

The macro **define-dde-dispatch-topic** defines a dispatching topic. Note that the server implementation also provides some dispatching capabilities.

19.3.3.3 The system topic

The system topic is implemented as a predefined dispatching topic called **:system**. It is automatically available to all defined DDE servers. Its class is **dde-system-topic**, which is a subclass of **dde-dispatching-topic**.

The following items are implemented by the system topic:

SZDDESYS_ITEM_TOPICS *Constant*

The constant **SZDDESYS_ITEM_TOPICS** has the value "**Topics**". Referring to this item in the system topic calls **dde-server-topics** to obtain a list of topics implemented by the server. The server should define a method on this generic function to return a list of strings naming the topics supported by the server. If this item is not to be implemented, do not define a method on the function, or define a method that returns **:unknown**.

SZDDESYS_ITEM_SYSITEMS *Constant*

The constant **SZDDESYS_ITEM_SYSITEMS** has the value "**SysItems**". Referring to this item in the system topic calls **dde-topic-items** to obtain a list of items implemented by the system topic. If a server implements additional system topic items it should define a method on the generic function specialized on its server class and **dde-system-topic** returning the complete list of supported topics. The server can return **:unknown** if this item is not to be implemented.

SZDDESYS_ITEM_FORMATS *Constant*

The constant **SZDDESYS_ITEM_FORMATS** has the value "**Formats**", and returns **unicodetext** and **text**. Currently only text formats are supported.

The system topic is a single object which is used by all DDE servers running in the Lisp image. You should therefore not under normal circumstances modify it with **define-dde-server-function** by specifying a value of **:system** for the *topic* argument, as this would make the changes to the system topic visible to all users of DDE within the Lisp image.

Instead, specify **:server** *my-server* **:topic :system**, where *my-server* is the name of your DDE server. This makes the additional items available only on the system topic of the specified server.

20

Common SQL

20.1 Introduction

This chapter is applicable to UNIX LispWorks and the Enterprise Edition of LispWorks. It describes Common SQL — the LispWorks interface to SQL. It should be used in conjunction with the relevant chapter of the *LispWorks Reference Manual*, which contains full reference entries for all the symbols in the SQL package.

For a longer introduction to Common SQL, please see the SQL Tutorial available at **`www.lispworks.com`**.

This chapter covers the following areas:

- Initialization and Connection
- The Functional SQL Interface
- The Object-Oriented (CLOS) SQL Interface
- The Symbolic SQL Syntax
- SQL I/O Recording
- SQL Interface Errors

The LispWorks SQL interface uses the following database terminology:

Data Definition Language (DDL)
: The language used to specify and interrogate the structure of the database schema.

Data Manipulation Language (DML)
: The language used for retrieving and modifying data. Also known as *query language*.

table
: A set of records. Also known as *relation*.

attribute
: A field of information in the table. Also known as *column*.

record
: A complete set of attribute values in the table. Also known as *tuple*, or *row*.

view
: A display of a table configured to your own needs. Also known as *virtual table*.

20.1.1 Overview

Common SQL is designed to provide both embedded and transparent access to relational databases from the LispWorks environment. That is, SQL/relational data can be directly manipulated from within Lisp, and also used as necessary when instantiating or accessing particular Lisp objects.

The SQL interface allows the following:

- Direct use of standard SQL statements as strings
- Mixed symbolic SQL and Common Lisp expressions
- Implicit SQL invocation when instantiating or accessing CLOS objects

The SQL interface provides these features through two complementary layers:

- A *functional* SQL interface
- An *object-oriented* SQL interface

The functional interface provides users with Lisp functions which map onto standard SQL DML and DDL commands. Special iteration constructs which utilize these functions are also provided. The object-oriented interface allows users to manipulate database views as CLOS classes via **`def-view-class`**. The

two interfaces may be flexibly combined in accordance with system requirements and user preference. For example, a *select* query can be used to initialize slots in a CLOS instance; conversely, accessing a CLOS slot may trigger an implicit functional query.

20.1.2 Supported databases

Common SQL supports connections to various databases on the platforms indicated below in Table 20.1. Common SQL may work with other platform / ODBC driver combinations and we would be pleased to hear of your experience with these. The keyword shown in the last column is the corresponding value of the *database-type* argument to **`sql:connect`**:

Table 20.1

Platform	Database	Driver / Client library	*database-type*
Windows	ODBC	Microsoft SQL Server	**`:odbc`**
Windows	ODBC	Oracle	**`:odbc`**
Windows	ODBC	Postgres	**`:odbc`**
Linux	ODBC	MySQL	**`:odbc`**
Linux	ODBC	Postgres	**`:odbc`**
Mac OS X	ODBC	MySQL	**`:odbc`**
Mac OS X	ODBC	Postgres	**`:odbc`**
Linux	Oracle	Oracle 9i (r2) or 10g	**`:oracle`**
Mac OS X	Oracle	Oracle 10g	**`:oracle`**
Windows	Oracle	Oracle 9i (r2) or 10g	**`:oracle`**
Solaris	Oracle	Oracle 9i (r2) or 10g	**`:oracle`**
HP-UX	Oracle	Oracle 9i (r2) or 10g	**`:oracle`**
Linux	Oracle	Oracle	**`:oracle8`**

Table 20.1

Platform	Database	Driver / Client library	*database-type*
Solaris	Oracle	Oracle	**`:oracle8`**
HP-UX	Oracle	Oracle	**`:oracle8`**
Linux	PostgreSQL	Postgres	**`:postgresql`**
FreeBSD	PostgreSQL	Postgres	**`:postgresql`**
Mac OS X	PostgreSQL	Postgres	**`:postgresql`**
Windows	PostgreSQL	Postgres	**`:postgresql`**
Linux	MySQL	MySQL	**`:mysql`**
FreeBSD	MySQL	MySQL	**`:mysql`**
Mac OS X	MySQL	MySQL	**`:mysql`**
Windows	MySQL	MySQL	**`:mysql`**
Solaris	MySQL	MySQL	**`:mysql`**

Note: MySQL versions prior to 4.1.1 should be run in ANSI mode to work with Common SQL. That is, **`mysqld`** must be started with **`--ansi`** or the **`ansi`** option must appear in the **`[mysqld]`** section of its configuration file.

Note: Oracle is not currently (July 2006) available for the Intel based Macintosh.

Note: To use PostgreSQL on any non-Microsoft Windows platform, LispWorks / Common SQL requires PostgreSQL version >= 8.x built with **`--enable-thread-safety`**.

20.2 Initialization

The initialization of Common SQL involves three stages. Firstly the SQL interface loaded. Next, the database type (actually class) to be used is initialized. Finally, Common SQL is used to connect to a database. These stages are explained in more detail in this section.

The rest of the definitions in this chapter are exported from the **sql** package. Application packages requiring convenient access to these facilities should therefore use the **sql** package.

20.2.1 SQL interface

The SQL interface itself is loaded by issuing the command **(require "odbc")** or **(require "oracle")** or **(require "postgresql")** or **(require "mysql")**. In an application, this step should be performed at build-time.

Not all of these modules are available on every LispWorks platform. See Table 20.1, page 203 for information about which databases are supported per-platform.

20.2.2 Database classes

A connection to a database is represented by an instance of the CLOS class **sql:database**. This instance holds information about the connected database. The special variable **sql:*default-database*** holds the current connection. The database class is subclassed on both vendor and version to provide the right kind of specialized behavior across database facilities: for example, the transaction model or the "brand" of SQL.

20.2.3 Initialization functions and variables

The initialization of the chosen database type is achieved by calling **sql:initialize-database-type** with the appropriate value of *database-type*. In an application, this step should be done at runtime. Where multiple database types are supported, it is possible to initialize more than one database type if needed (by making multiple calls to **initialize-database-type**).

The following functions and variables are relevant to initialization:

default-database-type specifies the default type of database. The possible values are shown in the *database-type* column of Table 20.1, page 203.

The function **initialize-database-type** initializes a database type according to the value of its *database-type* argument, which defaults to the value of ***default-database-type***.

A sample code sequence for initializing Common SQL to work with an ODBC database, using the above functions and variables, is as follows:

```
(require "odbc")
(setf sql:*default-database-type* :odbc)
(sql:initialize-database-type)
```

You can find which database types have been initialized by the value of the variable **`*initialized-database-types*`**.

20.2.4 Database libraries

Note: This section applies only to Unix/Linux systems.

A database directory environment variable specifies the root of the database directories. This variable is checked by LispWorks when you initialize a database type, and the libraries loaded are dependent on its value. The details of foreign code loading are described in the *LispWorks Foreign Language Interface User Guide and Reference Manual*

Note that most users only need to set the appropriate environment variable for their specific database vendor.

In order to override the default loading of database library code, you may set **`*sql-libraries*`**. To control messages while loading the libraries, set **`*sql-loading-verbose*`**.

In LispWorks for UNIX only (not LispWorks for Linux, FreeBSD or Macintosh), the list of library modules is added to **`link-load:*default-libraries*`** and **`link-load:read-foreign-modules`** is called to do the loading. If you need to load a different set of library modules, add your list of library modules to **`link-load:*default-libraries*`** before loading.

20.2.5 General database connection and disconnection

Once the database type has been initialized a connection can be established by calling **`connect`** with an appropriate *connection-spec*. A call to **`connect`** sets **`*default-database*`** to the database instance which represents the connection. All the other database functions described take a **`:database`** argument that can be either a database or a database name, and which defaults to the value of **`*default-database*`**.

Database connections can be named by passing the **:name** argument to **connect**, allowing you to have more than one connection to a given database. If this is omitted, then a unique database name is constructed from *connection-spec* and a counter. Connection names are compared with **equalp**.

To find all the database connection instances, call the function **connected-databases**. To retrieve the *name* for a connection instance, call **database-name**, and to find a connection instance with a given name use **find-database**. To print status information about the existing connections, call **status**.

To close a connection to a database, use **disconnect**.

To reestablish a connection to a database, use **reconnect**.

20.2.5.1 Connection example

The following example assumes that the **sql** package has been loaded, and that the **:odbc** database type has been initialized. It connects to two databases, **scott** and **personnel**, and then prints out the connected databases.

```
(setf *default-database-type* :odbc)
(connect "scott")
(connect "personnel" :database-type :odbc)
(print *connected-databases*)
```

20.2.6 Connecting to Oracle

For *database-type* **:oracle**, *connection-spec* conforms to the canonical form described for **connect** in the *LispWorks Reference Manual*. The *connection* part is the string used to establish the connection. When connecting to a local server, it may be the SID, otherwise it is an alias recognized by the names server, or in the **tnsnames.ora** file.

To connect to Oracle via SQL*Net, *connection-spec* is of the form ***username/password@host*** where *host* is an Oracle hostname.

Common SQL uses the Oracle Call Interface internally where this is available. For Oracle version 8, Common SQL automatically uses the same API as in LispWorks 4.4. On some platforms, this can also be obtained by using *database-type* **:oracle8**. Note that the **:oracle8** database type is restricted

because it cannot access or manipulate LOBs and all connections must use the same character set.

20.2.7 Connecting to ODBC

For *database-type* **:odbc** or **:odbc-driver**, *connection-spec* may take the canonical form described for **connect** in the *LispWorks Reference Manual*, but an addtional syntax is also allowed.

connect keyword arguments **:encoding**, **:signal-rollback-errors** and **:date-string-format** are all ignored.

20.2.7.1 Connecting to ODBC using a string

connection-spec should have one of the forms:

username/password@dsn The general form.

dsn/username/password For backward compatibility.

The two forms of strings are distinguished by the presence (or absence) of the '@' character. In both forms, *password* can be omitted along with the preceding '/'. Also, *username* can simply be omitted.

Note that this means that **"xyz"** and **"@xyz"** are both interpreted to give the same values (*username* is null, *password* is null, *dsn* is **"xyz"**).

20.2.7.2 Connecting to ODBC using a plist

In the plist, the acceptable keywords are **:username**, **:password**, **:dsn** and **:connection**.

:connection is a synonym of **:dsn**.

20.2.8 Connecting to MySQL

For *database-type* **:mysql**, *connection-spec* may be in the canonical form described for **connect** in the *LispWorks Reference Manual*, but it may also have the extensions described in this section.

In both the string and plist forms of *connection-spec* described below, any part that is omitted defaults to the MySQL default:

username	anonymous user
password	No password
dbname	No default database
hostname	localhost
port	3306 (unless using *unix-socket*).

20.2.8.1 Connecting to MySQL using a string

connection-spec can be a string of the form:

username/password/dbname@hostname:port

where *port* is a decimal number specifying the port number to use. *port* can be omitted along with the preceding '**:**'.

hostname can be omitted. If *port* is omitted too, the '**@**' can be omitted as well. If *port* is supplied and *hostname* is not supplied, then both the '**@**' and the '**:**' are required, for example:

```
me/my-password/my-db@:3307
```

hostname may also specify a unix socket name, which must start with the character '**/**'.

dbname may be omitted along with the preceding '**/**'.

password may be omitted. If *dbname* is also omitted, the preceding '**/**' can be omitted too.

username may be omitted.

20.2.8.2 Connecting to MySQL using a plist

connection-spec can be a plist containing (some of) the keywords **:username**, **:password**, **:dbname**, **:hostname**, **:port**, **:connection**, **:unix-socket**.

Each of these keywords may be omitted.

If **:unix-socket** is specified, then none of **:hostname**, **:port** and **:connection** can be specified. If **:hostname** is specified then **:connection** must not be specified.The value supplied for **:hostname** can be a raw hostname, or a string of

the form ***hostname*:*port***. If **:connection** is specified then it can a string conforming to one of these patterns:

hostname

hostname*:*port

:port

unix-socket Must start with '/'

That is, the value *connection* supplied in a plist *connection-spec* is interpreted jsust like the part of a string *connection-spec* following the '@' character.

20.2.8.3 Locating the MySQL client library

The MySQL interface to initialize, it must find the appropriate MySQL client library. The special variables **sql:*mysql-library-path*** and **sql:*mysql-library-directories*** give you control over this, as described in the *LispWorks Reference Manual*.

20.2.8.4 Special instructions for MySQL on Mac OS X

Download the 32-bit or 64-bit MySQL package to match your LispWorks image.

The downloadable packages from the MySQL web site contain only static client libraries, but LispWorks needs a dynamic library. You need to create the dynamic library, for example by using the following shell command.

To build the 32-bit dynamic library:

```
gcc -dynamiclib -fno-common \
  -o /usr/local/mysql/lib/libmysqlclient_r.dylib \
  -all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

To build the 64-bit dynamic library:

```
gcc -m64 -dynamiclib -fno-common \
  -o /usr/local/mysql/lib/libmysqlclient_r.dylib \
  -all_load /usr/local/mysql/lib/libmysqlclient_r.a -lz
```

This command should be executed as the root user, or some other user with write permission to the **/usr/local/mysql/lib** directory and assumes that

MySQL was installed in **/usr/local/mysql**, which is the location used by the prepackaged downloads.

An alternate way to create a dynamic library is to build MySQL from its source code with the **--enable-shared** flag.

By default, LispWorks expects to find the library either in **/usr/local/mysql/lib** or on the shared library path. This can be overridden by setting the special variable **sql:*mysql-library-directories***, as described in the *LispWorks Reference Manual*.

By default, LispWorks expects the library to be called **libmysqlclient.*.dylib** and it searches for a library that matches that pattern, where ***** is any version number. This search can be avoided by setting **sql:*mysql-library-path*** to something other than the default (**"-lmysqlclient"**), for example, it is possible to force LispWorks to look for version 12 by evaluating

```
(setq sql:*mysql-library-path* "libmysqlclient.12")
```

You can also set **sql:*mysql-library-path*** to a full path, which avoids the need to set **sql:*mysql-library-directories***.

If the environment variable **LW_MYSQL_LIBRARY.** is set, then its value is used instead of the value of **sql:*mysql-library-path***.

20.2.9 Connecting to PostgreSQL

For *database-type* **:posytgresql**, *connection-spec* must be either a string in the format specified by the PostgreSQL libraries or a plist.

20.2.9.1 Connecting to PostgreSQL using a string

If *connection-spec* is a string then it should be in the format specified by

www.postgresql.org/docs/7.4/static/libpq.html#LIBPQ-CONNECT

For example,

```
dbname=test user=scott password=tiger host=scandium
```

20.2.9.2 Connecting to PostgreSQL using a plist

connection-spec can be a plist containing (some of) the keywords **:username** (or **:user**), **:password**, **:dbname**, **:hostname** (or **:host**), **:port**, **:connection**. Each of these keywords may be omitte, but if **:connection** is specified, then **:hostname** and **:port** must not be specified.

The value supplied for **:hostname** can be a raw hostname or a string of the form ***hostname*:*port***. The value supplied for **:post** can be an integer or a string naming a service.

If **:connection** is specified then it can a string conforming to one of these patterns:

hostname

hostname*:*port

The values should not be escaped or quoted: LispWorks will esacpe and quote it as needed before passing it to the PostgreSQL library.

20.3 Functional interface

The functional interface provides a full set of Data Manipulation and Data Definition functions. The interface provides an SQL-compatible means of querying and updating the database from Lisp. In particular, the values returned from the database are Lisp values — thus smoothly integrating user applications with database transactions. An embedded syntax is provided for dynamically constructing sophisticated queries through **select**. Iteration is also provided via a mapping function and an extension to the **loop** macro. If necessary, the basic functions **query** and **execute-command** can be called with SQL statements expressed as strings. It is also possible to update or query the data dictionary.

20.3.1 Functional Data Manipulation Language (FDML)

The functions available for Data Manipulation and Data Definition are described below.

20.3.1.1 Querying

The function **select** returns data from a database matching the constraints specified. The data is returned, by default, as a list of records in which each record is represented as a list of attribute values. See the *LispWorks Reference Manual* for a full specification.

Database identifiers used in **select** are conveniently specified using the symbolic SQL **[]** syntax. This syntax is enabled by calling **enable-sql-reader-syntax**.

The square bracket syntax assumes that sql symbols are visible. Therefore when using the **[]** syntax, ensure that the current package either is **sql**, or is a pacakge which has the **sql** package on its package-use-list.

For a description of the symbolic SQL syntax see Section 20.5 on page 224. For example, the following is a potential query and its result:

```
(select [person_id] [person surname] :from [person])
=>
((111 "Brown") (222 "Jones") (333 "Smith"))
("PERSON_ID" "SURNAME")
```

In this example, **[person_id]**, **[person surname**] and **[person**] are database-identifiers and evaluate to literal SQL. The result is a list of lists of attribute values. Conversely, consider

```
(select [surname] :from [person] :flatp t)
=>
("Brown" "Jones" "Smith")
("SURNAME")
```

In this case the result is a simple list of surname values because of the use of the *flatp* keyword. The *flatp* keyword only works when there is one column of data to return.

In this example we use * to match all fields in the table, and then we use the *result-types* keyword to specify the types to return:

```
(select [*] :from [person])
=>
((2 111 "Brown") (3 222 "Jones") (4 333 "Smith"))
("ID" "Person_ID" "Surname")

(select [*] :from [person] :result-types '(:integer :string
:string))
=>
((2 "111" "Brown") (3 "222" "Jones") (4 "333" "Smith"))
("ID" "Person_ID" "Surname")
```

If you want to affect the result type for a specified field, use a type-modified database identifier. As an example:

```
(sql:select [Person_ID :string][Surname] :from [person])
=>
(("111" "Brown") ("222" "Jones") ("333" "Smith"))
("PERSON_ID" "SURNAME")
```

With *database-type* **:mysql**, further control over the values returned from queries is possible as described in "Types of values returned from queries" on page 238.

In this final example the **:where** keyword is used to specify a condition for returning selected values from the database.

```
(select [surname] :from [person] :where [= [person_id] 222])
=>
(("Jones"))
("SURNAME")
```

To output the results of a query in a more easily readable tabulated way, use the function **print-query**. For example the following call prints two even columns of names and salaries:

```
(print-query [select [surname] [income] :from [employee]]
             :titles '("NAME" "SALARY"))

NAME   SALARY
Brown  22000
Jones  45000
Smith  35000
```

20.3.1.2 Modification

Modifications to the database can be done using the following functions; **insert-records**, **delete-records** and **update-records**. The functions **commit**, **rollback** and the macro **with-transaction** are used to control transactions. Although **commit** or **rollback** may be used in isolation it is advisable to do any updates inside a **with-transaction** form instead. This provides consistency across different database transaction models. For example, some database systems do not provide an explicit "start-transaction" command while others do. **with-transaction** allows user code to ignore database-specific transaction models.

The function **insert-records** creates records in a specified table. The values can be either specified directly with the argument *values* or in the argument *av-pairs*, or they can be the result of a query specified in the *query* argument. The attributes can be specified with the argument *attributes* or in the argument *av-pairs*.

If *attributes* is supplied then *values* must be a corresponding list of values for each of the listed attribute names. For example, both:

```
(insert-records :into [person]
    :attributes '(person_id income surname occupation)
    :values '(115 11000 "Johnson" "plumber"))
```

and:

```
(insert-records :into [person]
               :av-pairs '((person_id 115)
                           (income 11000)
                           (surname "Johnson")
                           (occupation "plumber")))
```

are equivalent to the following SQL:

```
INSERT INTO PERSON
  (PERSON_ID,INCOME,SURNAME,OCCUPATION)
  VALUES (115,11000,'Johnson','plumber')
```

If *query* is provided, then neither *values* nor *av-pairs* should be. In this case the attribute names in the query expression must also exist in the insertion table. For example:

```
(insert-records :into [person]
     :query [select [id] [firstname] [surname]
               :from [manager]]
     :attributes '(person_id firstname surname))
```

To delete or alter those records in a table which match some condition, use **delete-records** or **update-records**. See the *LispWorks Reference Manual* for a full specification.

20.3.1.3 Caching of table queries

Operations which add or modify records sometimes need to perform an internal query to obtain type information for the relevant attributes. In principle it is possible for the database schema to change between update operations, and hence this query is run for each update operation. This can be a significant overhead.

For tables which are guaranteed to have a constant schema, you can optimize performance by adding a cache of these internal query results, using the function **cache-table-queries**. This can also be used to reset the cache if the table schema is actually altered. To control the default caching behaviour throughout every database connection, you can set the variable ***cache-table-queries-default***.

20.3.1.4 Transaction handling

A transaction in SQL is defined as *starting from* the **connect**, or from a **commit**, **rollback** or data-dictionary update and *lasting until* a **commit**, **rollback**, data-dictionary update or a **disconnect** command.

The macro **with-transaction** executes a body of code and then does a commit, unless the body failed in which case it does a rollback. Using this macro allows your code to cope with the fact that transactions may be handled differently in the different vendor implementations. Any differences are transparent if the update is done within a **with-transaction** form.

Note: Common SQL opens an ODBC database in manual commit mode, so that **with-transaction** and **rollback** take effect.

Applications should perform all database update operations in a **with-transaction** form (or follow them with **commit** or **rollback**) in order to safely com-

mit or discard their changes. This applies to operations that modify either the data or the schema.

The following example shows a series of updates to an employee table within a transaction. This example would commit the changes to the database on exit from **with-transaction**. This example inserts a new record into the **emp** table, then changes those employees whose department number is 40 to 50 and finally removes those employees whose salary is more than 300,000.

```
(connect "personnel")

(with-transaction
 (insert-records :into [emp]
                 :attributes '(empno ename job deptno)
                 :values '(7100 "ANDERSON" "SALESMAN" 30))
 (update-records [emp]
                :attributes [deptno]
                :values 50
                :where [= [deptno] 40])
 (delete-records :from [emp]
                 :where [> [sal] 300000]))
```

To commit or roll back all changes made since the last commit, use the functions **commit** or **rollback**.

20.3.1.5 Iteration

Common SQL has three iteration constructs: a **do** loop, a mapping function, and an extension to the Common Lisp **loop** macro.

The macrso **do-query** and **simple-do-query** repeatedly execute a piece of code within the scope of variables bound to the attributes of each record resulting from a query.

The function **map-query** maps a function across the results of a query and returns its result in a sequence of a specified type, like the Common Lisp **map** function.

Common SQL provides an extension to the ANSI Common Lisp macro **loop** which is a clause for iterating over query results. The syntax of the clause is:

```
{for|as} var [type-spec] being
             {the|each}{tuples|tuple}
             {in|of} query-expression
```

The more general word **tuple** is used so that it can also be applied to the object-oriented case. In the functional case, **tuple** is synonymous with **record**.

Each iteration of the loop assigns the next record of the table to the variable *var*. The record is represented in Lisp as a list. Destructuring can be used in *var* to bind variables to specific attributes of the records resulting from *query-expression*. In conjunction with the panoply of existing clauses available from the **loop** macro, the new iteration clause provides an integrated report generation facility.

Suppose the name of everyone in an employee table is required. This simple query is shown below using the different iteration method. The function **map-query** requires *flatp* to be specified; otherwise each name would be wrapped in a list.

```
(do-query ((name)[select [ename] :from [emp]])
          (print name))

(map-query
     nil
     #'(lambda (name) (print name))
     [select [ename] :from [emp] :flatp t])

(loop for (name)
          being each tuple in
             [select [ename] :from [emp]]
          do
(print name))
```

The following extended **loop** example binds, on each record returned as a result of the query, **name** and **salary**, accumulates the salary, and for salaries greater than 2750 increments a count, and prints the details. Finally, the average salary is printed.

```
(loop for (name salary) being each record in
   [select [ename] [sal] :from [emp]]
   initially (format t "~&~20A~10D" 'name 'salary)
   when (and salary (> salary 2750))
     count salary into salaries
     and sum salary into total
     and do (format t "~&~20A~10D" name salary)
   else
     do (format t "~&~20A~10D" name "N/A")
   finally
   (format t "~2&Av Salary:  ~10D" (/ total salaries)))
```

20.3.1.6 Specifying SQL directly

Sometimes it is necessary to execute vendor-specific SQL statements and queries. For these occasions Common SQL provides the functions **query** and **execute-command**. They can also be used when the exact SQL string is known in advance and thus the square bracket syntax is not needed.

The function **query** runs a SQL query on a database and returns a list of values like **select** (see "Querying" on page 213). It also returns a list of the field names selected.

execute-command is the basic function which executes any SQL statement other than a query. A stored procedure can be run using **execute-command** - see the *LispWorks Reference Manual* for details.

20.3.1.7 Building vendor-specific SQL

Common SQL does not provide a general interface to vendor-specific syntax.

There are two approaches you can take with SQL such as this:

```
SELECT B.PARTY_CODE_ALIAS, A.VALUE FROM CODES A, CODE_ALIASES B
     WHERE A.DOMAIN=B.CODE_DOMAIN(+) AND A.VALUE=B.CODE_VALUE(+)
           AND B.PARTY_ID(+)=<party_id>
```

1. Construct the string as above and then call **query** as described in "Specifying SQL directly" on page 219.
2. Use **sql-expression** to construct the vendor-specific pieces of the SQL. The above expression can be written like this:

```
(sql:select [b party_code_alias] [a value]
            :from '([codes a] [codes_aliases b])
            :where [and [= [a domain]
                           (sql:sql-expression
                            :string "B.CODE_DOMAIN(+)")]
                        [= (sql:sql-expression
                            :string "B.PARTY_ID(+)") PARTY-ID]])
```

20.3.2 Functional Data Definition Language (FDDL)

Functions in the FDDL may be used to change or query the structure of the database.

20.3.2.1 Querying the schema

The functions **list-tables**, **list-attributes**, **attribute-type** and **list-attribute-types** return information about the structure of a database.

20.3.2.2 FDDL Querying example

This example shows you how to query the type of the **ename** attribute of the **emp** table.

```
(attribute-type [ename] [emp]) -> :char
```

20.3.2.3 Modification

You may create or drop (delete) tables using the functions **create-table** and **drop-table.**

Create or drop indexes using the functions **create-index** and **drop-index**.

To create or drop a view (that is, a derived table based on a query) use the functions **create-view** and **drop-view**.

20.4 Object oriented interface

This section describes the object-oriented interface to SQL databases using specialized CLOS classes. These classes have **standard-db-object** as one of their superclasses and have **standard-db-class** as their metaclass. The **standard-db-class** metaclass provides the specialized behavior for mapping subclasses of **standard-db-object** onto records in the database. A class of this kind is created using **def-view-class**.

20.4.1 Object oriented/relational model

In the simple case, a class maps onto a database table, an instance of the class maps onto a record in the table, and a slot in the class maps onto an attribute in the table.

In general, however, a class maps onto a database view, an instance of the class maps onto a collection of records in the view, and a slot in the class is either:

- A *base slot* that maps onto an attribute in the view
- A *join slot* that points to a list of other view-class instances

If an instance maps onto more than one record in the view then for each record, all the key attributes from each table in the view are the same.

20.4.2 Object-Oriented Data Definition Language (OODDL)

The OODDL lets you define a mapping between the relational and object-oriented worlds to be defined. Through the mapping a CLOS object can effectively denote a collection of records in a database view, and can contain pointers to other view-based CLOS objects. The CLOS object makes explicit an object implicitly described by the flat relational values.

The mapping is defined using the macro **`def-view-class`**. This extends the syntax of **`defclass`** to allow special *base slots* to be mapped onto the attributes of database views (presently single tables). When you submit a select query that names a View Class (that is, a class defined by **`def-view-class`**), then the corresponding database view is queried, and the slots in the resulting instances are filled with attribute values from the database.

It is also possible to create *join slots* and *virtual* (ordinary) *slots*.

All the special slots are distinguished by a modified set of class and slot options. The special slots and their options are described in more detail under **`def-view-class`** in the *LispWorks Reference Manual*.

Note: **`def-view-class`** defines a Lisp view of an underlying database table. It is a similar concept to that of SQL VIEWs, but does not interact with them.

You can create a table based on a View Class using the function **`create-view-from-class`** and delete it using the function **`drop-view-from-class`**.

20.4.2.1 Example View Class definition

The following example shows a View Class corresponding to the traditional employees table, with the employee's department given by a join with the departments table. See **`def-view-class`** in the *LispWorks Reference Manual* for a description of the slot options.

```
(def-view-class employee (standard-db-object)
   ((employee-number :db-kind :key
                     :column empno
                     :type integer)
    (employee-name :db-kind :base
                   :column ename
                   :type (string 20)
                   :accessor employee-name)
    (employee-department :db-kind :base
                         :column deptno
                         :type integer
                         :accessor employee-department)
    (employee-job :db-kind :base
                  :column job
                  :type (string 9))
    (employee-manager :db-kind :base
                      :column mgr
                      :type integer)
    (employee-location :db-kind :join
                       :db-info (:join-class department
                                 :retrieval :deferred
                                 :set nil
                                 :home-key employee-department
                                 :foreign-key department-number
                                 :target-slot department-loc)
                       :accessor employee-location))
                 (:base-table emp))
```

The **`def-view-class`** macro allows elements or lists of elements to follow **`:home-key`** and **`:foreign-key`**. The elements can be symbols, **`nil`**, strings, integers or floats.

This syntax means that an object from the join class is only included in the join slot if the values from *home-key* are **`equal`** to the values in *foreign-key*, in order. These values are calculated as follows:

- If the element in the list is a symbol it is taken to be a slot name and the value of the slot is used
- Otherwise the element is taken to be the value

Note that some database vendors may have short maximum identifier lengths. The CLOS interface uses constructed alias names for tables in its SQL queries, and long table names or long class names may cause the constructed aliases to exceed the maximum identifier length for a particular vendor.

20.4.3 Object-Oriented Data Manipulation Language (OODML)

The OODML is designed to be powerful and expressive, while remaining familiar to users of the FDML. To achieve this aim, some of the functions and macros in the SQL interface have been *overloaded* — particularly the **`select`** function and the iteration constructs.

The function **`select`** is common across the both the functional and object-oriented SQL interfaces. If its first argument, *selections*, refers to a View Class by supplying its symbolic name then the select operation becomes object-oriented and it returns a list of instances instead of a list of attributes.

A subsequent equivalent **`select`** call will return the same (**`eql`**) instances. The **`:refresh`** argument can be used to ensure that existing instances get updated with any changed data. If such an update requires action by your application, then add methods on the generic function **`instance-refreshed`**.

In a View Class **`select`** call, the symbol **`slot-value`** is a valid SQL operator for use within the **`:where`** argument.

To find the View Classes for a particular database, use the function **`list-classes`**.

To manipulate data via a View Class, that is to modify the records corresponding to instances of the View Class, using the generic functions **`update-records-from-instance`**, and **`update-record-from-slot`**.

To delete records corresponding to instances of the View Class, use the generic function **`delete-instance-records`**.

To update existing instances of a View Class when data is known to have changed, use the generic functions **`update-slot-from-record`** and **`update-instance-from-records`**.

20.4.3.1 Examples

```
[select 'employee]
-> #<SQL-OBJECT-QUERY (EMPLOYEE)>
```

```
(select 'employee
        :where [= [slot-value 'employee 'employee-job]
                  "SALESMAN"])
((#<db-instance EMPLOYEE 8067092>)
 (#<db-instance EMPLOYEE 8069536>)
 (#<db-instance EMPLOYEE 8069176>))

(list-classes)
(#<db-class EMPLOYEE> #<db-class DEPARTMENT>)
```

20.4.3.2 Iteration

The object-oriented SQL interface has the same three iteration constructs as the functional interface (see Section 20.3.1.5 on page 217): a **do**-loop, a mapping function, and an extension to the Common Lisp **loop** macro. However, in this case, the iteration focus is not a tuple of attributes (that is, a record), but a tuple of instances. For example:

```
(loop for (jones company) being the tuples in
      [select 'person 'organisation
      :where [= [slot-value 'person 'surname] "Jones"]]
      do (format t "~A ~A ~%"
                 (slot-value jones 'forename)
                 (slot-value company 'short-name)))
```

Note: Instances may denote many database records, and hence the effective iteration focus in this case is a tuple of sets of tuples of attributes.

20.4.3.3 Garbage collection of view instances

View instance objects are not released for garbage collection (GC) until the connection is closed because they are pointed to by the **sql:database** object. This is to ensure that they can reliably be compared by **eq**.

20.5 Symbolic SQL syntax

Common SQL supports a symbolic query syntax across both the functional and object-oriented interface layers. It allows SQL and Common Lisp expressions to be mixed together — with as much processing as possible done at compile-time. Symbolic SQL expressions are read as square-bracketed lists to distinguish them from Lisp expressions. However, each can be nested within the other to achieve the desired result.

By default, this reader syntax is turned off. To turn it on see Section 20.5.3 on page 231.

20.5.1 The "[...]" Syntax

The square bracket syntax for the SQL interface is heavily overloaded to provide the most intuitive behavior in all situations. There are three uses of square brackets:

1. To enclose a database identifier
2. To construct an SQL string representing a symbolic expression
3. To enclose literal SQL

Each of these uses is demonstrated below.

20.5.1.1 Enclosing database identifiers

Database identifiers can be enclosed in the square bracket syntax as shown in the following examples.

`[foo]` => **`#<SQL-IDENT "FOO">`**

This case corresponds to an unqualified SQL identifier as in: **`SELECT FOO FROM BAR`**.

`[foo bar]` => **`#<SQL-IDENT "FOO.BAR">`**

This corresponds to a qualified SQL identifier as in: **`SELECT FOO.BAR FROM FOO`**

`["foo" bar]` => **`#<SQL-IDENT "\"foo\".BAR">`**

This corresponds to a qualified SQL identifier with an aliased table name containing special characters as in: **`SELECT "foo".BAR FROM BAZ "foo"`**.

`[foo "bar"]` => **`#<SQL-IDENT FOO \"bar\">`**

This corresponds to an alias definition as in: **`SELECT "bar".* FROM FOO "bar".`**

[foo :integer] => **#<SQL-IDENT "FOO" :INTEGER>**

As above, but including a type coercion component.

[foo bar :integer] -> **#<SQL-IDENT "FOO.BAR" :INTEGER>**

As above, but includes a type coercion component.

["foo" bar :integer] -> **#<SQL-IDENT "\"foo\".BAR" :INTEGER>**

As above, but includes a type coercion component.

20.5.1.2 SQL strings representing symbolic expressions

There are some SQL operators which may take a single argument (for example **any**, **some**, **all**, **not**, **union**, **intersect**, **except**, and **minus**). These are read as calls to the appropriate SQL operator. For example:

```
[any '(3 4)] -> #<SQL-VALUE-EXP "(ANY (3,4))">
```

This causes no conflict, however, as it is illegal to use these reserved words as identifiers in SQL. Similarly with two argument operators:

```
[> [baz] [beep]]
     -> #<SQL-RELATIONAL-EXP "(BAZ > BEEP)">
```

The **select** statement itself may be prepared for later query execution using the **[]** syntax. For example:

```
[select [person_id] [surname] :from [person]]
```

This form results in an SQL expression, which could be bound to a Lisp variable and later given to **query** to execute. For example:

```
[select [foo] [bar *]
        :from '([baz] [bar])
        :where [or [= [foo] 3]
                   [> [baz.quux] 10]]]
->
#<SQL-QUERY
    "(SELECT FOO,BAR.* FROM BAZ,BAR
                      WHERE ((FOO = 3)
                             OR (BAZ.QUUX > 10)))">
```

Strings can be inserted in place of database identifiers within a **select**:

```
[select [foo bar] [baz]
        :from '([foo] [quux])
        :where [or [> [baz] 3]
                   [like [foo bar] "SU%"]]]
->
#<SQL-QUERY:
   "(SELECT FOO.BAR,BAZ
         FROM FOO,QUUX
         WHERE ((BAZ > 3)
                OR (FOO.BAR LIKE 'SU%')))">
```

Any non-constant included gets filled in at runtime, for example:

```
[> [foo] x]
```

when macroexpanded reads as

```
(SQL-> #<SQL-IDENT "FOO"> X)
```

which constructs the actual SQL string at runtime.

Any arguments to an SQL operator that are Lisp constants are translated to the matching SQL construct at compile-time, for example:

```
"foo" -> "'foo'"
3 -> "3"
'("this" 5 "that") -> "('this', 5, 'that')"
'xyz -> "XYZ"
```

SQL operators which are supported are **`null`**, **`exists`**, **`*`**, **`+`**, **`/`**, **`-`**, **`like`**, **`substr`**, **`and`**, **`or`**, **`not`**, **`in`**, **`all`**, **`any`**, **`some`**, **`||`**, **`=`**, **`<`**, **`>`** ,**`>=`**, **`<=`**, **`<>`**, **`order-by`**, **`count`**, **`max`**, **`min`**, **`avg`**, **`sum`**, **`minus`**, **`nvl`**, **`distinct`**, **`except`**, **`intersect`**, **`union`**, **`slot-value`**, **`between`** and **`userenv`**. There are also pseudo operators for calling database functions (see “Calling database functions” on page 228).

The general syntax is: **`[<operator> <operand> ...]`**, for instance:

```
(select [count [*]] :from [emp])
```

The operand can itself be a SQL expression, as in the following example:

```
(sql:create-table [company]
                  '(([name] (varchar 20) not-null)))

(loop for company in '("LispWorks Ltd"
                       "Harlequin"
                       "Oracle"
                       "Rover"
                       "Microsoft")
      do
      (sql:insert-records :into [company]
                          :av-pairs `(([name] ,company))))

(sql:create-table [person]
                  '(([surname] (varchar 20) not-null)
                    ([firstname] (varchar 20) not-null)))

(loop for person in '(("Joe" "Bloggs")
                      ("Fred" "Smith")
                      ("Rover" "the Dog")
                      ("Fido" "the Dog"))
      do (sql:insert-records :into [person]
                             :av-pairs
                             `(([firstname] ,(car person))
                               ([surname] ,(second person)))))

(sql:select [name]
            :from [company]
            :where [= [name]
                      [any [select [surname]
                                   :from [person]]]])

(sql:select [surname]
            :from [person]
            :set-operation [union [select [firstname]
                                          :from [person]]])
```

20.5.1.3 Calling database functions

An arbitrary function can be included in the SQL using the pseudo operator **`sql-function`**. The first argument is the function name and the rest are its arguments, for example:

```
(select [sql-function "COS" [age]] :from [EMPLOYEES])
```

```
(insert-records
 :into [atable]
 :attributes '(a b)
 :values
 (list 1 [sql-function "TO_DATE" "02/06/99" "mm/DD/RR"]))
```

Also you can call SQL infix operators using the pseudo operators **sql-boolean-operator** and **sql-operator**.

20.5.1.4 Enclosing literal SQL

Literal SQL statements can simply be enclosed in the square bracket syntax, as shown below.

```
["SELECT FOO, BAR FROM BAZ"]
-> #<SQL "SELECT FOO, BAR FROM BAZ">

[select [*] :from [tbl]]
-> #<SQL-QUERY "(SELECT * FROM TABLE)">

[person surname]
->#<SQL-IDENT "PERSON.SURNAME">

[> [foo] 37]
->#<SQL-RELATIONAL-EXP "(FOO > 37)">
```

20.5.2 Programmatic interface

In some cases it is necessary to build SQL-expressions dynamically under program control.

The function **sql-operation** returns the SQL expression for an operator applied to its arguments. It also supports building SQL expressions which contain arbitrary SQL functions using the pseudo operators **sql-function**, **sql-operator** and **sql-boolean-operator**. For examples see **sql-operation** in the *LispWorks Reference Manual*.

The function **sql-expression** makes an SQL expression from the given keywords. This is equivalent to the first and third uses of the **[]** syntax as discussed in Section 20.5.1 on page 225.

The function **sql-operator** returns the Lisp symbol for an SQL operator.

The function **sql** makes SQL out of the arguments supplied. Each argument to **sql** is turned into SQL and then the *args* are concatenated with a single space

between each pair. A Lisp string maps to the same characters enclosed between single quotes (this corresponds to an SQL string constant). **nil** maps to **"NULL"**, that is, an SQL null value. Symbols and numbers map to strings. A list maps to a parenthesised, comma-separated expression. A vector maps to a comma-separated expression, which allows the easy generation of SQL lists that require no parentheses such as table lists in select statements.

The rules for the conversion via **sql** are fully specified in the *LispWorks Reference Manual*.

20.5.2.1 Examples

The following example function, taken from the object-oriented SQL interface layer, makes an SQL query fragment that finds the records corresponding a CLOS object (using the slots as attributes), when built into the *where*-clause of an updating form.

```
(let* ((class (class-of object))
          (key-slots (db-class-keyfields class)))
   (loop
     for key in key-slots
     for slot-name = (slot-definition-name key)
     for slot-type = (db-slot-definition-type key)
     collect
     [= (make-field-name class key)
        (lisp-to-sql-format
           (slot-value object slot-name)
           (if (listp slot-type)
               (car slot-type)
               slot-type))]
     into cols
     finally (apply (sql-operator 'and) cols)))
->
#<SQL-RELATIONAL-EXP "(EMP.EMPNO = 7369">
```

Here is another example that produces an SQL **select** statement:

```
(sql-operation 'select
    (sql-expression :table 'foo
                    :attribute 'bar)
    (sql-expression :attribute 'baz)
  :from (list
          (sql-expression :table 'foo)
          (sql-expression :table 'quux))
  :where (sql-operation 'or
            (sql-operation '>
               (sql-expression :attribute 'baz)
             3)
            (sql-operation 'like
               (sql-expression :table 'foo
                               :attribute 'bar)
             "SU%")))
->
#<SQL-QUERY "SELECT FOO.BAR,BAZ FROM FOO,QUUX
  WHERE ((BAZ > 3) OR (FOO.BAR LIKE 'SU%'))">
```

20.5.3 Utilities

The function **enable-sql-reader-syntax** switches square bracket syntax on and sets the state so that **restore-sql-reader-syntax-state** restores the syntax again if it is subsequently disabled. The function **disable-sql-reader-syntax** switches square bracket syntax off and sets the state so that **restore-sql-reader-syntax-state** disables the syntax again if it is subsequently enabled.

The functions **locally-enable-sql-reader-syntax** and **locally-disable-sql-reader-syntax** switch square bracket syntax on and off, but do not change the state restored by **restore-sql-reader-syntax-state**. The intended use of these is in a file:

```
#.(locally-enable-sql-reader-syntax)
    <code using [...]>
#.(restore-sql-reader-syntax-state)
```

20.6 Working with date fields

This section describes particular issues around using datetime database fields via Common SQL.

See also “Types of values returned from queries” on page 238 for information specifically about returning datetime values from MySQL.

20.6.1 Testing date values

Compare DATE values by formatting the date as a string in a date format that the database can parse. For example:

```
(sql:select * :from [Table] :where [= [Date] "25-Dec-2005"])
```

Note that it is not possible to lookup date values in the database using numeric values. This is because:

1. Common SQL cannot know that the field will be a date field until the results are returned, and
2. the database probably does not know about Common Lisp universal time.

20.6.2 DATE returned as universal time

By default Common SQL converts DATE values to Common Lisp universal times. Therefore code like this returns Common Lisp universal times (that is, integers) where **MyDate** is a DATE field type:

```
(sql:select [MyDate] :from [MyTable] :where [= [id] 1])
```

20.6.2.1 Timezone of returned DATEs

Common SQL creates universal time values from DATE fields assuming that the database contains times in Coordinated Universal Time (UTC). That is, as if by passing *time-zone* 0 to **encode-universal-time**. To decode the values consistently with this encoding, pass *time-zone* 0 to **decode-universal-time**.

If the database contains times in a different timezone, then the integer *time-zone* needs to be adjusted by adding an appropriate multiple of 3600 before calling **decode-universal-time**.

20.6.3 DATE returned as string

Instead of universal time integers, you can obtain strings formatted by the database by modifying the **MyDate** database identifier, adding **:string** like this:

```
(sql:select [MyDate :string] :from [MyTable] :where [= [id] 1])
```

This avoids the overhead of converting DATEs to universal times and so may improve performance of your application.

See **select** in the *LispWorks Reference Manual* for details.

20.6.4 Using universal time format

If the database is only accessed via Common SQL and you want to use the universal time date format, then you might consider using an INTEGER column containing universal time values instead of a DATE column.

20.7 SQL I/O recording

It is sometimes convenient to simply monitor the flow of commands to, and results from, a database. A number of functions are provided for this purpose.

The functions operate on two stream collections (*broadcast streams*) — one each for commands and results. They allow the recording to be started and stopped, checked, or recorded on further individual streams. By default, both commands and results recording is printed only to ***standard-output*.**

For details, see the *LispWorks Reference Manual* pages for **start-sql-recording**, **stop-sql-recording**, **sql-recording-p**, **list-sql-streams**, **sql-stream**, **add-sql-stream** and **delete-sql-stream**.

20.8 Error handling in Common SQL

All errors generated by Common SQL are of type **sql-user-error** or **sql-database-error**. You can test for these conditions and their subtypes in your error handlers.

20.8.1 SQL condition classes

An **sql-user-error** is an error inside Lisp.

An **sql-database-error** is an error inside the database interface that Lisp uses.

The following are subclasses of **sql-database-error**:

sql-database-data-error
: An error with the data given. It signifies an error that must be fixed for the code to work.

sql-temporary-error
: Signifies an error that is a result of other users using the same database. It means the code can work without change, once the other users stop using the database.

sql-connection-error
: An error with the connection to the RDBMS.

The following are subclasses of **sql-connection-error**:

sql-timeout-error
: A timeout with some operation.

sql-fatal-error
: An error which means that the connection is no longer usable.

Note: In general, the documentation for the various supported databases make it difficult to decide which error code should be made into which of the above condition class, and we probably get many of these wrong. If you find errors that seem to be signalled with the wrong condition class, please report them to Lisp Support, including the full printout of the condition, and we will fix it.

20.8.2 Database error accessors

Three functions are provided which access slots of **sql-database-error**, allowing you to discover more about the actual error that occurred.

sql-error-error-id and **sql-error-secondary-error-id** return primnary and secondary error identifiers. If you use these, please read the detailed description in the *LispWorks Reference Manual*.

sql-error-database-message is a string (maybe **nil**) returned by the foreign code.

20.9 Using MySQL

This section describes particular issues in Common SQL with MySQL databases.

20.9.1 Connection specification

See "Connecting to MySQL" on page 208 for information about MySQL specific extensions for the *connection-spec* passed to **connect**.

20.9.2 Case of table names and and database names

MySQL is case sensitive on table names and database names when the server is on a Unix machine. MySQL does not automatically change raw names to uppercase as specified by the SQL standard. However, Common SQL is geared towards uppercasing all names, so this may cause some mismatches. In general, Common SQL uppercases strings, and uses symbol names, which are normally uppercase, as-is.

One solution, possible only if you control the naming of tables and databases, is to make them all have the same case. If this is uppercase, that suffices. If it is lowercase, you need to set the variable **lower_case_table_names** in the configuration of the server.

If you cannot make all the names the same case, you have to get the case right. This can be achieved in several ways:

1. Specify tables names using strings, for example:

   ```
   (sql:select [*] :from ["TableNAMEwithVARIABLEcase"])
   ```

   Note that this does not work in LispWorks 4.4 and previous versions.

2. Pass the Lisp string directly:

   ```
   (sql:select [*] :from "TableNAMEwithVARIABLEcase")
   ```

Note that in this case the table name is passed to the database inside double quotes. That works only when the mode of the Common SQL connection contains **ANSI_QUOTES** (which is the default, see "SQL mode" on page 236 for details).

3. Specify table names as escaped symbols:

   ```
   (sql:select [*] :from [|TableNAMEwithVARIABLEcase|])
   ```

4. Construct the whole query string and pass it to **query** rather than using **select**:

   ```
   (sql:query "select * from TableNAMEwithVARIABLEcase")
   ```

20.9.3 Encoding (character sets in MySQL).

You can specify the encoding to be used by passing the **:encoding** argument to **connect**. Common SQL supports various encodings for MySQL as documented in the *LispWorks Reference Manual*.

The default is to use the default for the particular MySQL installation.

20.9.4 SQL mode

Because Common SQL is geared towards ANSI SQL, by default it connects in ANSI mode. If another mode is required, it can be set at connection time.

For example, to make MySQL treat quotes as in ANSI without setting other ANSI features, do:

```
(sql:connect "me/mypassword/mydb"
             :sql-mode "ANSI_QUOTES")
```

See the description of the **:sql-mode** argument to **connect** in the *LispWorks Reference Manual* for details.

20.9.5 Meaning of the :owner argument to select

In the Common SQL MySQL interface, the value of the **select** keyword argument **:owner** is interpreted to select a database name.

20.9.6 Special considerations for iteration functions and macros

This section describes particular issues when fetching multiple records using Common SQL with MySQL databases.

20.9.6.1 Fetching multiple records

The function **map-query** and the macros **do-query**, **simple-do-query** and **loop** with **each record** use internally **mysql-use-query**, which means that the underlying MySQL code brings the data from the server one record at a time. With a small number of records, it may be preferable to bring all the data immediately instead. This can be done by passing the argument ***get-all***, as follows:

```
(sql:map-query nil 'print
               "select forname,surname from people"
               :get-all t)

(sql:do-query
 ((forname surname) "select forname,surname from people"
  :get-all t)
 body)

(sql:simple-do-query
 (list "select forname,surname from people"
       :get-all t)
 body)

(loop for (forname surname) being each record
      "select forname,surname from people"
      get-all t
      body)
```

20.9.6.2 Aborting queries which fetch many records

In the MySQL interface there is no way to abort a query when part way through it. When any of the iterations above stops before reaching its end, the underlying code retrieves all the records to the end of the query (though without converting them to Lisp objects). If the query found many records, that may be an expensive (that is, time consuming) operation.

It is possible to avoid this inefficiency by passing the argument *not-inside-transaction*. If *not-inside-transaction* is true then when a query is aborted, then

LispWorks closes the database connection and reopens it, rather than retrieving all the remaining records.

```
(sql:map-query nil 'print
               "select forname,surname from people"
               :get-all t
               :not-inside-transaction t)
```

Note that this will lose any state associated with the connection, and so *not-inside-transaction* should only be used with care.

20.9.7 Table types

By default, **create-table** creates tables of the default type. This behaviour can be overridden by the **connect** keyword arguments **:default-table-type** and **:default-table-extra-options**, and the **:type** and **:extra-options** keyowrd arguments to **create-table**.

If *type* is passed to **create-table** or *default-table-type* was passed to **connect**, it is used as the argument to the "keyword" **TYPE** in the SQL statement:

create table MyTable (*column-specs*) TYPE = *type-value*

If *extra-options* is passed to **create-table** or *default-table-extra-options* was passed to **connect**, it is appended in the end of the SQL statement above.

connect with *default-table-type* and **create-table** with *type* also accept the keyword argument **:support-transactions**. When *support-transactions* is true, these functions will attempt to make tables that support transactions. It does this by using the type **innodb**.

20.9.8 Rollback errors

The default value of the **connect** keyword argument **:signal-rollback-errors** is determined by the value of the **:default-table-type** argument. If *default-table-type* is **:support-transactions** or **"innodb"** or **"bdb"**, then the default value for **:signal-rollback-errors** is **t**, otherwise the default value is **nil**.

20.9.9 Types of values returned from queries

Common SQL uses the MySQL mechanism in that returns values as strings.

By default, Common SQL converts these strings to the appropriate Lisp type corresponding to the column type (or more accurately, the type of the field in the query) according to Table 20.2

Table 20.2

MySQL column type	Lisp Type	Meaning
All integer types	**`integer`**	
Double	**`double-float`**	
Single	**`single-float`**	
Decimal	**`rational`**	
All String types	**`string`**	
All Binary types	**`(array (unsigned-byte 8) (*))`**	
Date	**`integer`**	Universal time
Datetime	**`integer`**	Universal time
Timestamp	**`integer`**	Universal time
Time	**`integer`**	Number of seconds
Year	**`integer`**	Number of years

However, if you specify the result type as **`:string`**, this eliminates the conversion and the return value is simply the string retrieved by MySQL. For information about specifying the result type for a column (or multiple columns) in a query, see "Querying" on page 213.

Each of the five date-like types (that is, Date, Datetime, Timestamp, Time and Year) can have result type **`:date`**, **`:date-string`** or **`:datetime-string`** with the following effects:

:date This result type means a Universal time. This is the default except for Year.

:date-string
A string with the format that MySQL uses for Date columns.

:datetime-string
A string with the format that MySQL uses for Datetime columns.

All the numeric types can have result type **:int**, **:single-float** or **:double-float**, causing the appropriate conversion. No check is made on whether the result is actually useful.

String types can have result type **:binary**, which returns an array.

20.9.10 Autocommit

Common SQL sets **autocommit** to 0 when it opens a MySQL connection.

20.10 Using Oracle

This section describes particular issues in Common SQL with Oracle databases, apart from the LOB interface, which is described in "Oracle LOB interface" on page 241.

20.10.1 Connection specification

See "Connecting to Oracle" on page 207 for information about Oracle-specific interpretation of the *connection-spec* passed to **connect**.

20.10.2 Setting connection parameters

Oracle database connections have prefetch values which you can control via Common SQL. Alternatively you can allow the database default prefetch values to take effect.

You can set the default prefetch values for a connection by passing **:prefetch-rows-number** and **:prefetch-memory** keyword arguments to **con-**

nect. The default value of *prefetch-rows-number* is 100 and the default value of *prefetch-memory* is #x100000 (meaning 1MB of data).

You can also pass the value **:default** for either of these arguments. This means that Common SQL does not set the default. This is useful if Oracle itself provides a suitable default.

20.11 Oracle LOB interface

20.11.1 Introduction

The Common SQL Oracle LOB interface allows you to retrieve LOB locators and then perform operations on them. It is also possible to insert new empty LOBs.

20.11.1.1 Retrieving LOB locators

This is done by normal **select** or **query** calls where the *selections* list names one or more columns that are of a LOB type. The LOB types are BLOB, CLOB, NCLOB, BFILE and CFILE.

The returned value is a LOB locator: an opaque Lisp object on which the **ora-lob-*** APIs (that is, those functions with names beginning with "ora-lob-") can be used. This LOB locator contains a pointer to an Oracle descriptor of type OCILobLocator*. Note that there can be multiple LOB locator objects associated with the same LOB in the server, but a LOB locator uniquely identifies a LOB object.

It is possible to specify that the result object should be a stream either for input or output. Then the resulting stream (which will be of type **lob-stream**) can be used as a normal Lisp stream.

20.11.1.2 Operating on LOB locators

This is done using the **ora-lob-*** functions. Most of these functions map directly to the underlying OCILob* functions.

Note that when modifying a LOB locator, the corresponding record must be locked. See “Retrieving Lob Locators” on page 242 for details.

20.11.1.3 Inserting empty LOBs

To add a new LOB object to the database, you must insert an empty LOB. The preferred way of doing this is to use the Oracle SQL functions EMPTY_BLOB and EMPTY_CLOB, which can called by using the pseudo operator **sql-function**, like this:

```
(sql:insert-records :into [mytable]
                    :values
                    (list "name" [sql-function 'empty_blob]))
```

This code inserts a record with "name" and an empty BLOB. It is also possible to make an empty LOB by calling **ora-lob-create-empty**, and passing the empty LOB as a value to **insert-records** or **update-records**.

20.11.2 Retrieving Lob Locators

When the selections list of a query that is used in **select**, **query**, **do-query**, **map-query**, **simple-do-query** or **loop .... for *x* being each record** contains a column of a LOB type, the results are LOB locator objects. For example, if the table definition is:

```
create table mytable {
    name varchar(200),
    image  blob
    }
```

Then doing

```
(sql:select [image] :from [mytable] :flatp t)
```

returns a list of LOB locators.

This example lists the size of the images in the table mytable:

```
(dolist (pair (sql:select [name][image] :from [mytable]))
  (format t "~a has an image of size ~a~%"
          (first pair) (sql:ora-lob-get-length (second pair)))
  (sql:ora-lob-free (second pair)))
```

or more efficiently

```
(sql:do-query ((name lob-locator)
               [sql:select [name][image] :from [mytable]])
              (format t "~a has an image of size ~a~%"
                      name (sql:ora-lob-get-length lob-locator)))
```

Note: The lifetime of the LOB locator objects differs between the functions that return a list of objects (**select** and **query**) and the iterative functions and macros (**do-query**, **simple-do-query**, **loop** and **map-query**). The iteration functions and macros free the LOB locators that they retrieve before proceeding to the next iteration. **select** and **query** do not free the LOB locators. Each LOB locator stays alive until the application makes an explicit call to **ora-lob-free**, or until the database is closed by a call to **disconnect**.

20.11.3 Locking

When the LOB or its contents need to modified, the corresponding record must be locked (Oracle enforces this). The best way to lock a record is to pass **:for-update** when calling **select**. See the documentation for **select** in the *LispWorks Reference Manual* for details. For example, writing a line in the end of the log file of station number 573:

```
create table logfiles (stationid integer, logiles clob)
  .. insert records ..

(sql:do-query ((log-stream)
               [select [log :output-stream] :from [logfiles]
               :where [= [stationid]  573] :for-update t])
              (file-position log-stream :end)
              (write-line "Add this line to the log" log-stream)
              (close log-stream)   ; forces the output
              )
(sql:commit)
```

Note that any call to **commit** or **rollback** on the same connection removes the lock. If you want to modify the LOB later, you must lock it again. An efficient way to achieve this is to use the special token ROWID, which returns the ROWID in the database, because this does not involve searching on the server side. For example:

```
(let ((lobs-list
       (sql:select [lob-field][rowid] ; get pairs of LOB
                   :from  [mytable]   ;  locators and ROWIDs
                   :where [some-condition])))
  ... do something ...
  ... reach a point when we want to modify one
  ... of the LOBS above and have bound one of the
  ... pairs in the variable pair.
  (sql:select [1]
              :from [mytable]            ; retrieve a constant
              :where
              [= [rowid] (second pair)]  ; get the right record
              :for-update t)             ; lock it
  (sql:ora-lob-write-buffer (car pair)   ; modify the lob
                            offset
                            amount
                            buffer)
  (sql:commit)                           ; also unlock everything
  )
```

20.11.4 Retrieving LOB Locators as streams

To retrieve LOB locators as streams, specify the type of retrieved object as **:input-stream** or **:output-stream** in the query. For example:

```
(sql:select [image :input-stream] :from [mytable] :flatp t)
```

returns a list of streams.

For example, to print the name of all images that start with some "magic number", that is a sequence of 4 specific bytes (#xf5 #x12 #x4e #x23):

```
(let ((array (make-array 4 :element-type '(unsigned-byte 8))))
  (sql:do-query ((name lob-stream)
                 [sql:select [name][image :input-stream]
                             :from [mytable]])
                (when (and (eq (read-sequence array lob-stream ) 4)
                           (eq (aref array 0) #xf5)
                           (eq (aref array 0) #x12)
                           (eq (aref array 0) #x4e)
                           (eq (aref array 0) #x23))
                  (print name))))
```

Closing the stream also frees the LOB object.

When using **:output-stream**, it is important to call **force-output** before trying to commit the changes, because the stream is buffered.

20.11.5 Attaching a stream to a LOB locator

It is possible to attach a stream to a LOB locator, passing the LOB locator as a **`:lob-locator`** argument to **`(make-instance 'sql:lob-stream ...)`**. The value of the **`:direction`** argument must be **`:input`** or **`:output`**. By default, if the stream is closed the LOB locator is freed, unless the value of the initarg **`:free-lob-locator-on-close`** is passed as **`nil`**.

Operations via the stream can be mixed with direct operations on the LOB. However, because of the buffering, accessing the LOB contents will give non-obvious results, as other operations may not see something that was written to the stream because it is still in the stream buffer, or the stream may have already read some contents before they were overwritten. Use **`force-output`** or **`clear-input`** before accessing the LOB in other ways to avoid these problems.

It is possible to attach more than one stream to the same LOB locator, in both directions. Apart from the issue of the buffering described above, the streams can be used independently of each other. Note that if you want to close one of the streams and to continue to use the others or the LOB locator itself, you must pass **`:free-lob-locator-on-close nil`** when you make the stream.

The LOB locator to which a stream is attached can be found by using the reader **`lob-stream-lob-locator`**.

20.11.6 Interactions with foreign calls

You can define your own foreign calls and use them on the underlying OCI descriptors. For this, you need to access the OCI handles using **`ora-lob-lob-locator`**, and maybe **`ora-lob-env-handle`** and **`ora-lob-svc-ctx-handle`**. These accessors return foreign pointers that can be passed to foreign functions in the usual way.

When the foreign functions deal only with the data, rather than with LOB objects, use the functions **`ora-lob-read-foreign-buffer`**, **`ora-lob-write-foreign-buffer`** and **`ora-lob-get-buffer`**.

For example:

```
;;; You have a C function my_lob_processor
;;; int my_lob_processor(OCILobLocator *lob,
;;;                      OCISvcCtx *Context,
;;;                      int other_arg)

(fli:define-foreign-function my-lob-processor
    ((lob sql:p-oci-lob-locator)
     (env sql:p-oci-svc-ctx)
     (other-arg :int))
   :result-type :int)
```

Assuming you have the LOB locator in the variable *lob*, call the foreign function on it:

```
(my-lob-processor (sql:ora-lob-lob lob)
                  (sql:ora-lob-svc-ctx-handle lob)
                  36)
```

There are three handles in the LOB: the LOB descriptor itself, the environment and the context. The pointer types, the reader and the corresponding C type for each handle are shown in Table 20.3 below..

Table 20.3 Handles in the LOB locator

OCI handle	Reader	Pointer type	C type
LOB descriptor	**ora-lob-lob-locator**	**p-oci-lob-locator** or **p-oci-file**	OCILobLocator*
context	**ora-lob-svc-ctx-handle**	**p-oci-svc-ctx**	OCISvcCtx*
environment	**ora-lob-env-handle**	**p-oci-env**	OCIEnv*

The **p-oci-lob-locator** pointer type is used for internal LOBs (that is, BLOB, CLOB and NCLOB). The **p-oci-file** pointer type is used for file LOBs (CFILE and BFILE). For functions that take both, the type **p-oci-lob-or-file** is defined as the union of these two types

20.11.7 Determining the type of a LOB

The function **ora-lob-internal-lob-p** returns whether it is internal (that is BLOB, CLOB or NCLOB) or not (that is BFILE or CFILE). The function **ora-lob-element-type** returns the LISP element type that best corresponds to the

LOB locator. This will be one of **(unsigned-byte 8)** for BLOB and BFILE, or **base-char** or **simple-char** for CLOB, NCLOB and CFILE, depending on the charset of the LOB object.

It is possible to distinguish between CLOB and NCLOB by looking at the result of **ora-lob-char-set-form**. It returns 2 for NCLOB and 1 for CLOB.

20.11.8 Reading and writing from and to LOBs

One way of reading and writing is to use streams as described in the section "Retrieving LOB Locators as streams" on page 244. When large amounts of data are written (read) to (from) the LOB the direct interface may be useful. The direct interface is implemented by **ora-lob-read-foreign-buffer**, **ora-lob-read-buffer**, **ora-lob-write-foreign-buffer**, and **ora-lob-write-buffer**.

All the direct interfaces are more efficient if the buffer that is passed is static. That is always true for the ***-foreign-buffer** functions, but normally not true for Lisp objects. See the documentation for **make-array** in the *LispWorks Reference Manual*. See also **ora-lob-get-buffer**.

The direct reading and writing methods can be used for "random" access, but they can also be used conveniently for efficient linear access, simply by passing **nil** as the *offset* parameter.

20.11.9 The LOB functions

Most of the LOB functions take an *errorp* argument, which is a boolean controlling what happens if an error occurs inside an OCI function. If *errorp* is true, an error is signaled. If *errorp* is false, the function returns an error object (of type **sql-database-error**).

All the LOB functions signal an error if the *lob-locator* argument given is not a LOB locator object as returned by **select** or **query**.

Many of the functions basically perform a call to the underlying OCI function. When the match is direct, this is mentioned in the *LispWorks Reference Manual*.

20.11.9.1 Querying functions

You can test whether a LOB locator is initialized, open or temporary with **ora-lob-locator-is-init**, **ora-lob-is-open** or **ora-lob-is-temporary**.

The predicate for internal LOBs is **ora-lob-internal-lob-p**.

ora-lob-element-type returns a Lisp element type corresponding to the LOB locator as described "Determining the type of a LOB" on page 246.

ora-lob-lob-locator, **ora-lob-env-handle** and **ora-lob-svc-ctx-handle** return foreign pointers to the various handles in the LOB mentioned in "Interactions with foreign calls" on page 245. To determine the best value for the size of a buffer use **ora-lob-get-chunk-size**.

ora-lob-char-set-form and **ora-lob-char-set-id** query the charset of a *lob-locator*.

The querying functions specifically for file LOBs are **ora-lob-file-exists**, **ora-lob-file-is-open** and **ora-lob-file-get-name**

You can obtain the current length of the LOB with **ora-lob-get-length**.

You can test two LOB locators for whether they point to the same LOB object with **ora-lob-is-equal**.

20.11.9.2 LOB management functions

You can create a LOB object with **ora-lob-create-empty**.

You can assign a LOB to another LOB locator with **ora-lob-assign**.

You can free a LOB locator with **ora-lob-free**.

20.11.9.3 Modifying LOBs

All the functions mentioned in this section are applicable to internal LOBs only, except **ora-lob-load-from-file**.

Before modifying a LOB, the corresponding record must be locked. See the discussion in "Locking" on page 243.

If you make several modifications to a LOB which has functional or domain indexes, it is useful to wrap several calls of modifying functions in a pair of

ora-lob-open and **ora-lob-close**. That means that the indexes will be updated once (when **ora-lob-close** is called), which saves work. Note that after a call to **ora-lob-open**, **ora-lob-close** must be called before any call to **commit**.

To append the contents of one LOB to another, use **ora-lob-append**.

You can copy all or part of a LOB into another LOB using **ora-lob-copy**.

ora-lob-load-from-file loads the data from a file LOB into an (internal) LOB.

You can erase (that is, fill with the 0 byte or with Space character) all or part of a LOB using **ora-lob-erase**.

You can reduce the size of a LOB using **ora-lob-trim**.

If you need to make multiple updates to a LOB you can optionally create a transaction using **ora-lob-open** and **ora-lob-close** call. This may save work on the server side.

20.11.9.4 File operations

These functions are used to modify the properties of file LOBs.

Open and close the file associated with a file LOB using **ora-lob-file-open** and **ora-lob-file-close**.

You can close all the files associated with a file LOB locator that have been opened through the database connection with **ora-lob-file-close-all**.

You can alter the directory and/or the file name for a file LOB locator by calling **ora-lob-file-set-name**

20.11.9.5 Direct I/O

The direct I/O functions perform input or output directly on the OCI handle, without the intervening layer of a stream. If you move large amounts of data to or from the LOB, and in particular if you pass the data to or from foreign functions, the direct calls can be more efficient, and in some cases also more convenient to use. Note, however, that if you make many small modifications to the data, the **lob-stream** interface may be more efficient.

Note also that the difference in efficiency between the direct calls and the **lob-stream** interface is likely to be quite small compared to the time spent on network traffic.

If you make many modifications to a LOB, you should also consider wrapping the operations in a transaction created by a pair of calls to **ora-lob-open** and **ora-lob-close**.

You can read data from the LOB locator into a Lisp buffer or foreign buffer using **ora-lob-read-buffer** and **ora-lob-read-foreign-buffer** respectively.

Similarly **ora-lob-write-buffer** and **ora-lob-write-foreign-buffer** can be used to write buffer to a LOB.

You can obtain a buffer suitable for efficient I/O with foreign functions via **ora-lob-get-buffer**.

ora-lob-read-into-plain-file writes the contents of a LOB into a file.

ora-lob-write-from-plain-file writes the contents of a file into a LOB.

20.11.9.6 Temporary LOBs

You can create a temporary LOB with **ora-lob-create-temporary**.

You can test whether a LOB is temporary with **ora-lob-is-temporary**.

You can free a temporary LOB locator if necessary with **ora-lob-free-temporary**, though temporary LOB locators are freed automatically when the database connection is closed by **disconnect**.

20.11.9.7 Control of buffering

These functions control the internal buffering by the Oracle client: **ora-lob-enable-buffering**, **ora-lob-disable-buffering**, and **ora-lob-flush-buffer**. They have no interaction with any of the other functions above.

21

User Defined Streams

21.1 Introduction

A number of classes and functions are provided in the **stream** package that allow you to define your own input and output streams. The user defined streams can use the standard I/O functions, and methods specializing on the relevant user defined stream class can also be defined to provide specific implementations of other I/O functions. Note that some changes have been made to the standard I/O functions to allow for this. For example, **stream-element-type** is now a generic function. See the **common-lisp** package reference entries in the *LispWorks Reference Manual* for alterations to standard functions, and the **stream** package entries for more details on user defined streams.

21.2 An illustrative example of user defined streams

In this chapter an example is provided to illustrate the main features of the **stream** package. In this example a stream class is defined to provide a wrapper for **file-stream** which uses the Unicode Line Separator instead of the usual ASCII CR/LF combination to mark the end of lines in the file. Methods are then defined, specializing on the user defined stream class to ensure that it handles reading from and writing to a file correctly.

21.2.1 Defining a new stream class

Streams can be capable of input or output (or both), and may deal with characters or with binary elements. The **stream** package provides a number of stream classes with different capabilities from which user defined streams can inherit. In our example the stream must be capable of input and output, and must read characters. The following code defines our stream class appropriately:

```
(defclass unicode-ls-stream
          (stream:fundamental-character-input-stream
           stream:fundamental-character-output-stream)
  ((file-stream :initform nil
                :initarg :file-stream
                :accessor ls-stream-file-stream)))
```

The new class, **unicode-ls-stream**, has **fundamental-character-input-stream** and **fundamental-character-output-stream** as its superclasses, which means it inherits the relevant default character I/O methods. We shall be overriding some of these with more relevant and efficient implementations later.

Note that we have also provided a slot, called **file-stream**. This slot is a place holder for a Common Lisp file stream. When making an instance of **unicode-ls-stream** we can create an instance of a file stream in this slot. This allows us to use the Common Lisp file stream functionality for reading from and writing to a file.

fundamental-character-input-stream *Class*

fundamental-character-output-stream *Class*

Package: **stream**

The classes **fundamental-character-input-stream** and **fundamental-character-output-stream** provide default methods for generic functions used for character input and character output respectively, and should therefore be included by stream classes concerned with these. The user can provide methods for these generic functions specialized on the user-defined classes.

21.2.2 Recognizing the stream element type

We know that the stream will read from a file using **file-stream** functionality and that the stream element type will be **simple-char**. The following defines a method on **stream-element-type** to return the correct element type.

```
(defmethod stream-element-type ((stream unicode-ls-stream))
  'simple-char)
```

stream-element-type *Generic Function*

stream-element-type ***stream***

Package: **common-lisp**

The function **stream-element-type** is implemented as a generic function. Depending on the stream, a method should be defined for this generic function that takes a *stream* as its argument and returns the element type of the stream.

21.2.3 Stream directionality

Streams can be defined for input only, output only, or both. In our example, the **unicode-ls-stream** class needs to be able to read from a file and write to a file, and we therefore defined it to inherit from an input and an output stream class. We could have defined disjoint classes instead, one inheriting from **fundamental-character-input-stream** and the other from **fundamental-character-output-stream**. This would have allowed us to rely on the default methods for the direction predicates. However, given that we have defined one bi-directional stream class, we must define our own methods for the direction predicates.

```
(defmethod input-stream-p ((stream unicode-ls-stream))
  (input-stream-p (ls-stream-file-stream stream)))

(defmethod output-stream-p ((stream unicode-ls-stream))
  (output-stream-p (ls-stream-file-stream stream)))
```

The above code allows us to “trampoline” the correct direction predicate functionality from **file-stream**, using the **ls-stream-file-stream** accessor we defined previously.

input-stream-p *Generic Function*

output-stream-p *Generic Function*

input-stream-p *stream*

output-stream-p *stream*

Package: **common-lisp**

The predicates **input-stream-p** and **output-stream-p** are implemented as generic functions. Their default methods return **t** if *stream* is respectively an input or output stream. If the user wants to implement a stream with no inherent directionality (and thus does not include **fundamental-input-stream** or **fundamental-output-stream**) but for which the directionality depends on the instance, then suitable methods should be provided.

21.2.4 Stream input

The following definition for the **stream-read-char** reads a character from the stream. If the character read is a **#\Line-Separator**, then the method returns **#\Newline**, otherwise the character read is returned. It also returns **:eof** at the end of the file.

```
(defmethod stream:stream-read-char ((stream unicode-ls-stream))
  (let ((char (read-char (ls-stream-file-stream stream)
              nil :eof)))
    (if (eq char #\Line-Separator)
        #\Newline
      char)))
```

There is no need to define a new method for **stream-read-line** as the default method uses **stream-read-char** repeatedly to read a line, and our implementation of **stream-read-char** ensures that this will work.

stream-read-char *Generic Function*

stream-read-char *stream*

Package: **stream**

The generic function **stream-read-char** reads one item from *stream*. The item read is either a character or the end of file symbol **:eof** if the stream is at the end of a file. Every subclass of **fundamental-character-input-stream** must define a method for this function.

We also need to make sure that if a **#\Newline** is unread, it is unread as a **#\Line-Separator**. The following code uses the Common Lisp file stream function **unread-char** to achieve this.

```
(defmethod stream:stream-unread-char ((stream unicode-ls-stream)
                                      char)
  (unread-char (if (eq char #\Newline) #\Line-Separator char)
               (ls-stream-file-stream stream)))
```

stream-unread-char *Generic Function*

stream-unread-char ***stream***

Package: **stream**

The generic function **stream-unread-char** undoes the last call to **stream-read-char**, as in **unread-char**. Every subclass of **fundamental-character-input-stream** must define a method for this function.

Finally, although the default methods for **stream-listen** and **stream-clear-input** would work for our stream, it is faster to use the functions provided by **file-stream**, using our accessor, **ls-stream-file-stream**.

```
(defmethod stream:stream-listen ((stream unicode-ls-stream))
  (listen (ls-stream-file-stream stream)))

(defmethod stream:stream-clear-input ((stream unicode-ls-stream))
  (clear-input (ls-stream-file-stream stream)))
```

stream-clear-input *Generic Function*

stream-clear-input ***stream***

Package: **stream**

The generic function **stream-clear-input** implements **clear-input**. The default method is defined on **fundamental-input-stream** and does nothing.

stream-listen *Generic Function*

stream-listen ***stream***

Package: **stream**

The generic function **stream-listen** is used by **listen** and returns **t** if there is input available. The default method uses **stream-read-char-no-hang** and **stream-unread-char**. Most streams should define their own method as this is usually trivial and more efficient than the method provided.

21.2.5 Stream output

The following definition for the **stream-write-char** uses **write-char** to write a character to the stream. If the character written to **unicode-ls-stream** is a **#\Newline**, then the method writes a **#\Line-Separator** to the file stream.

```
(defmethod stream:stream-write-char ((stream unicode-ls-stream)
                                     char)
  (write-char (if (eq char #\Newline)
                  #\Line-Separator
                char)
              (ls-stream-file-stream stream)))
```

stream-write-char *Generic Function*

stream-write-char ***stream character***

Package: **stream**

The generic function **stream-write-char** writes *character* to *stream*. Every subclass of **fundamental-character-output-stream** must have a method defined for this function.

The default method for **stream-write-string** calls the above generic function and successfully write a string to the stream. However, the following is a more efficient implementation for our stream.

```
(defmethod stream:stream-write-string ((stream unicode-ls-stream)
                                       string &optional (start 0)
                                        (end (length string)))
  (loop with i = start
        until (>= i end)
        do (let* ((newline (position #\Newline
                            string :start i :end end))
                  (this-end (or newline end)))
             (write-string string (ls-stream-file-stream stream)
                           :start i :end this-end)
             (incf i this-end)
             (when newline
                   (stream:stream-terpri stream)
                   (incf i)))
        finally (return string)))
```

We do not need to define our own method for **stream-terpri**, as the default uses **stream-write-char**, and therefore works appropriately

To be useful, the **stream-line-column** and **stream-line-start-p** generic functions need to know the number of characters preceding a **#\Line-Separator**. However, since the LispWorks file stream records line position only by **#\Newline** characters, this information is not available. Hence we define the two generic functions to return **nil**:

```
(defmethod stream:stream-line-column
  ((stream unicode-ls-stream))
  nil)

(defmethod stream:stream-start-line-p
  ((stream unicode-ls-stream))
  nil)
```

stream-line-column *Generic Function*

stream-line-column ***stream***

Package: **stream**

The generic function **stream-line-column** returns the column number where the next character will be written from *stream*, or **nil** if this is not meaningful for the stream. This function is used in the implementation of **print** and the **format ~t** directive. A method for this function must be defined for every character output stream class that is defined, although at its simplest it may be defined to always return **nil**.

stream-start-line-p *Generic Function*

stream-start-line-p ***stream***

Package: **stream**

The generic function **stream-start-line-p** returns **t** if *stream* is positioned at the beginning of a line, and **nil** otherwise. It is permissible to define a method that always returns **nil**.

Finally, the methods for **stream-force-output**, **stream-finish-output** and **stream-clear-output** are "trampolined" from the standard **force-output**, **finish-output** and **clear-output** functions.

```
(defmethod stream:stream-force-output ((stream
                                        unicode-ls-stream))
  (force-output (ls-stream-file-stream stream)))

(defmethod stream:stream-finish-output ((stream
                                         unicode-ls-stream))
  (finish-output (ls-stream-file-stream stream)))

(defmethod stream:stream-clear-output ((stream
                                        unicode-ls-stream))
  (clear-output (ls-stream-file-stream stream)))
```

21.2.6 Instantiating the stream

Now that the stream class has been defined, and all the methods relevant to it have been set up, we can create an instance of our user defined stream to test it. The following function takes a filename and optionally a stream direction as its arguments and makes an instance of **unicode-ls-stream**. It ensures that the **file-stream** slot of the stream contains a Common Lisp **file-stream** capable of reading from or writing to a file given by the filename argument.

```
(defun open-unicode-ls-file (filename &key (direction :input))
  (make-instance 'unicode-ls-stream :file-stream
    (open filename
          :direction direction
          :external-format :unicode
          :element-type 'simple-char)))
```

The following macro uses **open-unicode-ls-stream** in a similar manner to the Common Lisp macro **with-open-file**:

```
(defmacro with-open-unicode-ls-file ((var filename
                                        &key (direction :input))
                                        &body body)
  `(let ((,var (open-unicode-ls-file ,filename
                 :direction ,direction)))
     (unwind-protect
         (progn ,@body)
       (close ,var)))
```

We now have the required functions and macros to test our user defined stream. The following code uses **config.sys** as a source of input to an instance of our stream, and outputs it to the file **unicode-ls.out**, changing all occurrences of **#\Newline** to **#\Line-Separator** in the process.

```
(with-open-unicode-ls-file (ss "C:\\unicode-ls.out"
                               :direction :output)
   (write-line "-*- Encoding: Unicode; -*-" ss)
   (with-open-file (ii "C:\\config.sys")   ; Don't edit this file!
     (loop with line = nil
           while (setf line (read-line ii nil nil))
           do (write-line line ss))))
```

After running the above code, if your load the file **C:\unicode-ls.out** into an editor (for example, a LispWorks editor), you can see the line separator used instead of CR/LF. Most editors do not yet recognize the Unicode Line Separator character yet. In some editors it appears as a blank glyph, whereas in the LispWorks editor it appears as **<2028>**. In LispWorks you can use **Alt+X What Cursor Position** or **Ctrl+X =** to identify the unprintable characters.

You can also use the follow code to print out the contents of the new file line by line.

```
(with-open-unicode-ls-file (ss "C:\\unicode-ls.out")
   (loop while (when-let (line (read-line ss nil nil))
                 (write-line line))))
```

22

Socket Stream SSL interface

The Socket Stream SSL interface allows you to use Secure Socket Layer (SSL) with **`comm:socket-stream`**s.

The interface is based on the OpenSSL code, and most of it is simply an FLI interface to OpenSSL functions. The main LispWorks specific code is the way OpenSSL is integrated with **`socket-stream`**.

Note: to load the Socket Stream SSL interface, evaluate

```
(require "comm")
```

Note: Below we assume that the current package uses the **`comm`** package. That is, **`comm`** package symbols may not be qualified explicitly.

22.1 Creating a stream with SSL

There are three ways to make a **`socket-stream`** with SSL processing:

- Call **`(make-instance 'socket-stream :ssl-ctx ...)`**
- Call **`(open-tcp-stream ... :ssl-ctx ...)`**
- Call **`attach-ssl`** on a **`socket-stream`**.

For example:

```
(open-tcp-stream some-url 443 :ssl-ctx t)
```

22.2 SSL-CTX and SSL objects

When the value of the **:ssl-ctx** argument is a symbol, LispWorks automatically creates an SSL_CTX object and an SSL object and uses them. If you need to configure these objects, you can access them by the following methods:

- When passing **:ssl-ctx** or when calling **attach-ssl** (as described above) also pass **:ctx-configure-callback** and **:ssl-configure-callback**.
- Use the accessors **socket-stream-ssl** and **socket-stream-ctx**.
- Make your own SSL-CTX or SSL objects and pass them as the *ssl-ctx* argument.

22.3 OpenSSL interface

The configuration interface contains mostly FLI function definitions that map directly to OpenSSL calls. See below for a list of those provided.

There are also some functions to make common cases simpler. These are **read-dhparms**, **pem-read**, **set-ssl-ctx-options**, **set-ssl-ctx-password-callback**, and **set-ssl-ctx-dh**. See the *LispWorks Reference Manual* for details of these.

22.3.1 OpenSSL constants

The Lisp constants **SSL_FILETYPE_ASN1** and **SSL_FILETYPE_PEM** representing file types are provided.

22.3.2 Naming conventions for direct OpenSSL calls

This section describes the mapping between OpenSSL function names and the corresponding Lisp names.

22.3.2.1 Mapping C names to Lisp names

For functions that map directly to OpenSSL calls, the convention is to create the LISP name from the C name by replacing underscores by hyphens.

22.3.2.2 Mapping Lisp names to C names

To find the C name from the LISP function name:

1. the hyphens need to be replaced by underscores, and
2. the initial SSL or SSL_CTX has to be in uppercase, and
3. the rest has to be lowercase, except that
4. the following phrases are cased specially, like this: "RSAPrivateKey", "DSH ", "ASN1", "CA", "PrivateKey"

22.3.3 Direct calls to OpenSSL

The following functions map directly to the OpenSSL functions. Check the OpenSSL documentation for details.

Where an OpenSSL function takes an SSL* or SSL_CTX*, the Lisp function's argument must be a foreign pointer of type **`ssl-pointer`**, **`ssl-ctx-pointer`** or **`ssl-cipher-pointer`**. Where an OpenSSL function takes a **`char*`** or **`int`**, the Lisp function's argument must be a string or integer. Where an OpenSSL function takes other kinds of pointers, the Lisp function's argument must be a foreign pointer. The return values are integers or foreign pointers unless stated otherwise.

If an error occurs in one of these functions, an error code is returned. They do not signal any Common Lisp conditions and so you should check the return value carefully.

Table 22.1

Direct calls to OpenSSL	Return values
`ssl-add-client-ca`	

Table 22.1

Direct calls to OpenSSL	Return values
ssl-cipher-get-bits	First value is number of bits the cipher actually uses. Second value is number of bits the algorithm of the cipher can use (which may be bigger).
ssl-cipher-get-name	string. e.g. "DHE-RSA-AES256-SHA"
ssl-cipher-get-version	string. e.g. "TLSv1/SSLv3"
ssl-clear-num-renegotiations	
ssl-ctrl	
ssl-ctx-add-client-ca	
ssl-ctx-add-extra-chain-cert	
ssl-ctx-ctrl	
ssl-ctx-get-max-cert-list	
ssl-ctx-get-mode	
ssl-ctx-get-options	
ssl-ctx-get-read-ahead	
ssl-ctx-get-verify-mode	integer
ssl-ctx-load-verify-locations	
ssl-ctx-need-tmp-rsa	
ssl-ctx-sess-set-cache-size	
ssl-ctx-sess-get-cache-size	
ssl-ctx-sess-set-cache-mode	
ssl-ctx-sess-get-cache-mode	

Table 22.1

Direct calls to OpenSSL	Return values
ssl-ctx-set-client-ca-list	
ssl-ctx-set-max-cert-list	
ssl-ctx-set-mode	
ssl-ctx-set-options	
ssl-ctx-set-read-ahead	
ssl-ctx-set-tmp-rsa	
ssl-ctx-set-tmp-dh	
ssl-ctx-use-certificate-chain-file	
ssl-ctx-use-certificate-file	
ssl-ctx-use-privatekey-file	
ssl-ctx-use-rsaprivatekey-file	
ssl-get-current-cipher	**ssl-cipher-pointer** Can be a null pointer.
ssl-get-max-cert-list	
ssl-get-mode	
ssl-get-options	
ssl-get-verify-mode	integer
ssl-get-version	string "TLSv1", "SSLv2" or "SSLv3"
ssl-load-client-ca-file	
ssl-need-tmp-rsa	
ssl-num-renegotiations	
ssl-session-reused	
ssl-set-client-ca-list	

Table 22.1

Direct calls to OpenSSL	Return values
ssl-set-max-cert-list	
ssl-set-mode	
ssl-set-options	
ssl-set-tmp-rsa	
ssl-set-tmp-dh	
ssl-total-renegotiations	
ssl-use-certificate-file	
ssl-use-rsaprivatekey-file	
ssl-use-privatekey-file	

If you need OpenSSL functionality that is not provided here, you can define your own foreign functions via the LispWorks Foreign Language Interface.

If you do this, an important point to note is that on Microsoft Windows, the **:calling-convention** must be **:cdecl** (it defaults to **:stdcall**). If using OpenSSL suddenly causes mysterious crashes, the *calling-convention* in your foreign function definitions is the first thing to check.

22.4 Socket Stream SSL keyword arguments

The keyword arguments **:ssl-ctx**, **:ssl-side**, **:ctx-configure-callback** and **:ssl-configure-callback** can be be passed to create and configure socket streams with SSL processing. The various methods for creating and configuring SSL streams accept these keyword arguments as shown in Table 22.2, page 267.

Table 22.2

	:ssl-ctx	**:ssl-side**	**:ctx-configure-callback**	**:ssl-configure-callback**
socket-stream make-instance	Yes	Yes	Yes	Yes
open-tcp-stream	Yes	No	Yes	Yes
attach-ssl	Yes	Yes	Yes	Yes
make-ssl-ctx	Yes	Yes	No	No

(make-instance 'socket-stream ...) and **open-tcp-stream**, when *ssl-ctx* is non-**nil**, call **attach-ssl** and pass it all the arguments.

:ssl-ctx specifies that SSL should be used, and also specifies the SSL_CTX object to use. See the OpenSSL manual entry for SSL_CTX_new for details of making a SSL_CTX. The value of *ssl-ctx* can be:

A symbol — Together with *ssl-side*, this symbol specifies which protocol to use. *ssl-ctx* can be one of:

1) **t** or **:default**, meaning use the default. Currently this is the same as **:v23**.

2) One of **:v2**, **:v3**, **:v23** or **:tls-v1**. These are mapped to the SSLv2_*, SSLv3_*, SSLv23_*, TLSv1_* methods.

LispWorks makes a new SSL_CTX object and uses it and frees it when the stream is closed. **make-instance**, **attach-ssl** and **open-tcp-stream** also make an SSL object, use it and free it when the stream is closed.

A foreign pointer of type **ssl-ctx-pointer**

This corresponds to the C type SSL_CTX*. This is used and is not freed when the stream is closed. **make-instance**, **attach-ssl** and **open-tcp-stream** also make an SSL object, use it and free it when the stream is closed. The foreign pointer maybe a result of a call to **make-ssl-ctx**, but it can also be a result of your code, provided that it points to a valid SSL_CTX and has the type **ssl-ctx-pointer**.

A foreign pointer of type **ssl-pointer**

This corresponds to the C type SSL*. This specifies the SSL to use in **make-instance**, **attach-ssl** and **open-tcp-stream**. This maybe a result of a call to **ssl-new**. but can also be a result of your code, provided that it points to a valid SSL object and has the type **ssl-pointer**. The SSL is used and is not freed when the stream is closed.

When you pass a **ssl-ctx-pointer** or a **ssl-pointer** foreign pointer, these must have already been set up correctly.

:ssl-side specifies which side the **socket-stream** is. The value *ssl-side* can be one of **:client**, **:server** or **:both**. **open-tcp-stream** does not take this keyword and always uses **:client**. For the other calls this argument defaults to **:server**. The value of *ssl-side* is used in two cases:

When a new SSL_CTX object is created, it is used to select the method:

:client => *_client_method

:server => *_server_method

:both => *_method

When a new SSL object is created, when *ssl-side* is either **:client** or **:server**, LispWorks calls **ssl-set-connect-state** or **ssl-set-accept-state** respectively.

If the value of *ssl-ctx* is a **ssl-pointer**, *ssl-side* is ignored.

:ctx-configure-callback specifies a callback, a function which takes a foreign pointer of type **ssl-ctx-pointer**. This is called immediately after a new

SSL_CTX is created. If the value of *ssl-ctx* is not a symbol, *ctx-configure-callback* is ignored.

:ssl-configure-callback specifies a callback, a function which taks a foreign pointer of type **ssl-pointer**. This is called immediately after a new SSL is created. If the value of *ssl-ctx* is not a *ssl-pointer*, *ssl-configure-callback* is ignored.

22.5 Attaching SSL to an existing socket-stream

You can attach SSL to an existing **socket-stream** by calling **attach-ssl** on the stream. **attach-ssl** ensures the OpenSSL library is loaded and seeds the Pseudo Random Number Generator (PRNG). The **socket-stream** SSL keyword arguments are processed by **attach-ssl** as described in "Socket Stream SSL keyword arguments" on page 266.

Detach SSL from a **socket-stream** and shut down the SSL with **detach-ssl**.

For full descriptions of **attach-ssl** and **detach-ssl** see the entries in the COMM chapter of the *LispWorks Reference Manual*.

22.6 Using SSL objects directly

The C objects SSL and SSL_CTX are represented in LispWorks by foreign pointers with type **ssl-pointer** and **ssl-ctx-pointer**, which correspond to the C types SSL* and SSL_CTX*. These foreign types should be used for any foreign function that takes or returns these C types, and must be used when passing a foreign pointer as the value of the **:ssl-ctx** argument.

Making SSL objects is a way of getting access to them to perform configuration, but, especially in the case of the SSL_CTX, it is a useful way to avoid repeated calls to the configuration routines which may be time consuming. For example, if we have defined a function **configure-a-ctx**, and we want to read once every 60 seconds from some URL, we can write:

```
(loop (with-open-stream
          (str (comm:open-tcp-stream some-url 443 :ssl-ctx t
                                     :ctx-configure-callback
'configure-a-ctx))
        (read-something str))
      (sleep 60))
```

This will cause **configure-a-ctx** to be called each time. If it is expensive, we can call it only once by changing the code to:

```
(let ((ctx (comm:make-ssl-ctx :ssl-side :client)))
  (configure-a-ctx ctx)
  (loop (with-open-stream
    (str (comm:open-tcp-stream some-url 443 :ssl-ctx ctx))
          (read-something str))
        (sleep 60))
  (ssl-ctx-free ctx))
```

The SSL objects could be made either by **make-ssl-ctx** or **ssl-new** or by user code that calls the C functions SSL_CTX_new and SSL_new. **destroy-ssl-ctx** frees the SSL_CTX object. To free an SSL object you would call **destroy-ssl**. See the entries in the *LispWorks Reference Manual* for full descriptions of these functions.

22.7 Initialization

All the functions that make a SSL_CTX first call **ensure-ssl**, so normally you do not need to initialize the library. If your code makes a SSL_CTX itself (that is, not by calling any of the LispWorks interface functions), it needs to initialize the library first. Normally that should be done by an explicit call to **ensure-ssl**, which loads the SSL library and calls SSL_library_init and SSL_load_error_strings, and also does some LispWorks specific initializations. If your code must do the initialization, **ensure-ssl** should still be called with the argument **:already-done t**, which tells it that the library is already loaded and initialized.

22.8 Obtaining and installing the OpenSSL library

At the time of writing, OpenSSL is available as shown in Table 22.3:

Table 22.3

Operating System	Availability of OpenSSL
Linux	Installed by default on most 32-bit and 64-bit distributions
Window (32-bit)	Available at **www.slproweb.com/products/Win32OpenSSL.html** for 32-bit.
Windows (64-bit)	Not currently supported. Version of 0.9.8b of the library can be built from the source at **www.openssl.org/source** and is classed as "initial".
Mac OS X (32-bit)	Installed by default.
Mac OS X (64-bit)	Not currently supported. The development branch of the library at **www.openssl.org/source** has support for this platform.
FreeBSD	Installed by default
Solaris	In the freeware from Sun at **sunfreeware.com** for both 32-bit and 64-bit.

22.8.1 Installing the OpenSSL library on Solaris

After installing (with pkgadd) you need to put the shared libraries **libcrypto.so** and **libssl.o** on the loader path. By default these are installed in **/usr/local/ssl/lib**.

To add the libraries to the loader path, either

- Add **/usr/local/ssl/lib** to the environment variable **LD_LIBRARY_PATH**, or

- Create links from **/usr/lib**.

22.8.2 Loading the OpenSSL libraries

Since OpenSSL is not a standard on all machines yet, the location of the library or libraries varies. By default, **ensure-ssl** loads libraries as shown in Table 22.4, page 272.

Table 22.4

Operating System	Libraries
Linux	**-lssl**
Windows	**libeay32.dll libssl32.dll**
Solaris	**-lssl**
Mac OS X	**-lssl**
Others	**nil**

On machines where the path is unknown or is incorrect, you must set the path. Do this by calling **set-ssl-library-path**, or by passing the path as the *library-path* argument to **ensure-ssl**.

22.9 Errors in SSL

If there are errors inside SSL, LispWorks will signal an error of type **ssl-condition**, which is a subclass of **socket-error**.

The condition can be one of the types **ssl-x509-lookup**, **ssl-closed**, **ssl-error** and **ssl-failure**. See the *LispWorks Reference Manual* for details of the condition classes.

23

Internationalization

23.1 Introduction

LispWorks uses Unicode (UCS-2 encoding) internally in its representation of **character** objects. All Unicode characters can be represented in strings, though 8-bit string types are also provided for efficiency when characters beyond the Latin-1 range are not needed. Character and string data can be input and output in various encodings (external formats).

23.2 Character and String types

23.2.1 Character types

The following subtypes of character are defined:

base-char simple characters with **char-code** less than **base-char-code-limit** (**256**).

lw:simple-char simple characters with **char-code** less than **char-code-limit** (**65536**).

character All characters including non-simple characters (that is, with non-null bits attributes).

23.2.2 Character Syntax

All simple characters have names that consist of **U+** followed by the code of the character in hexadecimal, for example **#\U+764F** is **(code-char #x764F)**.

Additionally, Latin-1 characters have names derived from the ISO10646 name, for example:

```
(char-name (code-char 190))
=>
"Vulgar-Fraction-Three-Quarters"
```

Names are also provided for space characters:

```
(name-char "Ideographic-Space")
=>
#\Ideographic-Space
```

If required, the bits attributes names can be preprended as usual:

```
#\ctrl-ideographic-space
=>
#\Control-Ideographic-Space
```

23.2.3 String types

String types are supplied which are capable of holding each of the character types mentioned above. The following string types are defined:

base-string holds any **base-char**

lw:text-string holds any **simple-char**

sys:augmented-string
holds any character.

In particular, **text-string** is the type that can hold all characters used in texts. The types above include non-simple strings - those which are displaced, adjustable or with a fill-pointer.

The Common Lisp type **string** itself is dependent on the value of **lw:*default-character-element-type*** according to the rules for string construction described in “String Construction” on page 276. For example:

```
CL-USER 1 > (set-default-character-element-type 'base-char)
BASE-CHAR

CL-USER 2 > (coerce (list #\Ideographic-Space) 'string)

Error: In a call to SEQ::%SET-ACCESS-ARRAY: #\Ideographic-Space
is not of type BASE-CHAR.
  1 (abort) Return to level 0.
  2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed,  or :? for
other options

CL-USER 3 : 1 > :a

CL-USER 4 > (set-default-character-element-type 'simple-char)
SIMPLE-CHAR

CL-USER 5 > (coerce (list #\Ideographic-Space) 'string)
" "
```

The following types are subtypes of **simple-string.** Note that in the names of the string types, 'simple' refers to the string object and does not mean that the string's elements are **simple-char**s.

lw:simple-base-string
: holds any **base-char**

lw:simple-text-string
: holds any **simple-char**

sys:simple-augmented-string
: holds any character.

The Common Lisp type **simple-string** itself is dependent on the value of ***default-character-element-type*** according to the rules for string construction described in "String Construction" on page 276.

23.2.3.1 String types at runtime

The type **string** (and hence **simple-string**) is defined by ANSI Common Lisp to be a union of all the character array types. This makes a call like

```
(coerce s 'simple-string)
```

ambiguous because it needs to select a concrete type (such as **simple-base-string** or **simple-text-string**).

When LispWorks is running with ***default-character-element-type*** set to **base-char**, it expects that you will want strings with element type **base-char**, so functions like **coerce** treat references to **simple-string** as if they were **(simple-array base-char (*))**.

If you call **set-default-character-element-type** with a larger character type, then **simple-string** becomes a union of the array types that are subtypes of that character type.

23.2.3.2 String types at compile time

The compiler always does type inferencing for **simple-string** as if ***default-character-element-type*** was set to **character**.

For example, when you declare something to be of type **simple-string**, the compiler will never treat it as **simple-base-string**. Therefore calls like

```
(schar (the simple-string x) 0)
```

will work whether *x* is a **simple-base-string**, **simple-text-string** or **sys:simple-augmented-string**.

23.3 String accessors

schar works on any simple string object. However, for efficient string access when a simple string type is known, the following specialised accessors are provided:

lw:sbchar for **simple-base-string**.

lw:stchar for **simple-text-string**.

23.4 String Construction

LispWorks constructs strings of a suitable type where sufficient information is available. Failing that, strings are constructed of type according to the value of ***default-character-element-type***.

23.4.1 Default string construction

If the value is **base-char** then:

```
(make-string 3)
```

returns a **simple-base-string**. If the value is **simple-char** then the same form returns a **simple-text-string**.

If the value of ***default-character-element-type*** is **base-char**, then

```
(coerce sequence 'simple-string)
```

attempts to construct a **simple-base-string**. This will signal an error if any element of *sequence* is not a **base-char**.

If the value of ***default-character-element-type*** is **simple-char**, then

```
(coerce sequence 'simple-string)
```

attempts to construct a **simple-text-string**. This will signal an error if any element of *sequence* is not a **simple-char**.

Other string constructors also take their default from ***default-character-element-type***. For instance, the string reader will always construct a string of type determined by this variable, unless it sees a character of a larger type, in which case a suitable string is constructed. Also **with-output-to-string** and **make-string-output-stream** will construct a stream with element type determined by this variable and generate a string of the same element type.

23.4.2 String construction with known type

The variable ***default-character-element-type*** merely provides the default behavior. If enough information is supplied, then a string of suitable type is constructed. For instance, the form:

```
(make-string 3 :initial-element #\Ideographic-Space)
```

constructs a string of a type that can hold its elements, regardless of the value of ***default-character-element-type***.

23.4.3 Controlling string construction

The initial value of ***default-character-element-type*** is **base-char,** to avoid programs that only require 8-bit strings needlessly creating larger string objects. If your application uses Unicode characters beyond the Latin-1 range (characters of type **extended-char**) then you should consider which of the following two approaches to use:

- Ensure that all strings which may hold **extended-char**s are constructed explicitly with the appropriate type. This is the conservative approach, allowing you to avoid allocation of 16-bit strings where these are not required. Note that you can use the specialised accessors such as **stchar** for strings of type **simple-text-string**.
- Change the default so that by default 16-bit strings are allocated. Do this by:
  (set-default-character-element-type 'simple-char)
  Bear in mind that this is a global setting which affects default string construction for the entire system. It could be called from a user interface, depending on whether the user needs to handle **extended-char**s.

Note: Do not attempt to bind or set directly the variable ***default-character-element-type***.

23.4.4 String construction on Windows systems

When LispWorks for Windows starts up on a OS with a non-Latin-1 code page, it calls

```
(set-default-character-element-type 'simple-char)
```

so that by default, newly constructed strings can contain the data likely to be returned from the OS or user input.

If you know your string only needs to contain 8-bit data, then you can create it explicitly with element type **base-char**.

Conversely if you know that a string may need to contain 16-bit data even on a Latin-1 code page system, then you should create it explicitly with element-type **simple-char**.

23.5 External Formats

External formats are two-way translations from Lisp's internal encoding to an external encoding. They can be used in file I/O, and in passing and receiving string data in foreign function calls.

An external format is named in LispWorks by an *external format specification* (*ef-spec*). An ef-spec is a symbol naming the external format, or a list with such a name as its first element followed by parameter/value pairs.

LispWorks has a number of predefined external formats:

win32:code-page
: The Windows code page with identifier given by the **:id** parameter. Implemented only on Windows.

:latin-1
: ISO8859-1.

:latin-1-terminal
: As Latin-1, except that if a non-Latin-1 character is output, it is written as **<xxxx>** where **xxxx** is the hexdecimal character code and does not signal error.

:latin-1-safe
: As Latin-1, except that if a non-Latin-1 character is output, it is written as **?** and does not signal error.

:macos-roman
: The Mac OS Roman encoding.

:ascii
: ASCII.

:unicode
: The UCS-2 encoding of Unicode. The parameter **:little-endian** defaults to the endianness of the platform.

:utf-8
: The UTF-8 encoding of Unicode.

:jis
: JIS. The encoding data is read from a file **Uni2JIS** and is pre-built into LispWorks.

:euc-jp
: EUC-JP. The encoding data is read from a file **Uni2JIS** and is pre-built into LispWorks.

:sjis
: Shift JIS.

:windows-cp936 Windows code page 936. The encoding data is read from a file **windows-936-2000.ucm** and is pre-built into LispWorks.

:gbk A synonym for **:windows-cp936**.

Note: **windows-936-2000.ucm** is provided by way of documentation in the directory **lib/5-1-0-0/etc/**. It is not read at runtime.

Note: **Uni2JIS** is provided by way of documentation in the directory **lib/5-1-0-0/etc/**. It is also used at runtime by the function **char-name**.

23.6 External Formats and File Streams

The **:external-format** argument of **open** and related functions should be an ef-spec, where the name can be **:default**. The symbol **:default** is the default value.

In this section, an ef-spec is "complete" if and only if the name is not **:default** and the parameters include **:eol-style**.

All external formats have an **:eol-style** parameter and if it is not explicit in an ef-spec a default is used. The allowed values are

:lf This is the default on Unix/Linux/FreeBSD/Mac OS X systems, meaning that lines are terminated by Linefeed.

:crlf This is the default on Windows, meaning that lines are terminated by Carriage-Return followed by Linefeed.

:cr Lines are terminated by Carriage-Return.

23.6.1 Using complete external formats

If **open** or **with-open-file** gets a complete **:external-format** argument then, it is used as is. For example, this form opens an ASCII linefeed-terminated stream:

```
(with-open-file (ss "C:/temp/ascii-lf"
                    :direction :output
                    :external-format
                    '(:ascii :eol-style :lf))
  (stream-external-format ss))
=>
(:ASCII :EOL-STYLE :LF)
```

If you know the encoding of a file you are opening, then you should pass the appropriate **:external-format** argument.

23.6.2 Guessing the external format

If **open** or **with-open-file** gets a non-complete **:external-format** argument *ef-spec* then the system decides which external format to use by calling the function **sys:guess-external-format**.

The default behaviour of **sys:guess-external-format** is as follows:

1. When *ef-spec*'s name is **:default**, this finds a match based on the file-name; or (if that fails), looks in the Emacs-style (-*-) attribute line for an option called ENCODING or EXTERNAL-FORMAT; or (if that fails), chooses from amongst likely encodings by analysing the bytes near the start of the file, or (if that fails) uses a default encoding. Otherwise *ef-spec*'s name is assumed to name an encoding and this encoding is used.
2. When *ef-spec* does not include the **:eol-style** parameter, it then also analyses the start of the file for byte patterns indicating the end-of-line style, and uses a default end-of-line style if no such pattern is found.

The file in this example was written by a Windows program which writes the Byte Order Mark at the start of the file, indicating that it is Unicode (UCS-2) encoded. The routine in step 1 above detects this:

```
(set-default-character-element-type 'simple-char)
=>
SIMPLE-CHAR

(with-open-file (ss "C:/temp/unicode-notepad.txt")
  (stream-external-format ss))
=>
(:UNICODE :LITTLE-ENDIAN T :EOL-STYLE :CRLF)
```

The behaviour of **sys:guess-external-format** is configurable via the variables **sys:*file-encoding-detection-algorithm*** and **sys:*file-eol-style-detection-algorithm***. See the *LispWorks Reference Manual* for details.

23.6.3 External formats and stream-element-type

The **:element-type** argument in **open** and **with-open-file** defaults to the value of ***default-character-element-type***.

If *element-type* is not **:default**, checks are made to ensure that the resulting stream's **stream-element-type** is compatible with its external format:

1. If *direction* is **:input** or **:io**, the *element-type* argument must be a supertype of the type of characters produced by the external format.
2. If *direction* is **:output** or **:io**, the *element-type* argument must be a subtype of the type of characters accepted by the external format

If the *element-type* argument doesn't satisfy these requirements, an error is signalled.

If *element-type* is **:default** the system chooses the **stream-element-type** on the basis of the external format.

23.6.4 External formats and the LispWorks Editor

The LispWorks Editor uses **open** with :**element-type :default** to read and write files. On reading a file, the external format is remembered and used when saving the file. On writing a Unicode (UCS-2) file, the Byte Order Mark is written.

It is possible to insert characters in the Editor (for example by pasting clipboard text) which are not supported by the chosen external format. This will lead to errors on attempt to save the buffer. You can handle this by setting the external format appropriately.

See the *LispWorks Editor User Guide* for more details.

23.6.5 Byte Order Mark

The Unicode Byte Order Mark (BOM) is treated as whitespace in the default readtable. This allows the Lisp reader to read a Unicode (UCS-2 encoded, *external-format* `:unicode`) file regardless of whether the BOM is present.

Some editors including Microsoft Notepad and the LispWorks editor write the BOM when writing a file with Unicode (UCS-2) encoding.

23.7 External Formats and the Foreign Language Interface

External formats can be used to pass and receive string data via the FLI. See the section on string types in the *LispWorks Foreign Language Interface User Guide and Reference Manual*.

24

LispWorks' Operating Environment

This chapter describes the interfaces which provide information about the environment in whch LispWorks is running. This includes the operating system, the physical location of the LispWorks executable, and the arguments it was passed on startup.

24.1 The Operating System

The Common Lisp function **`software-type`** returns a generic name for the Operating System. The Common Lisp function **`software-version`** returns information about the the version of the Operating System.

In particular **`software-type`** can be used to distinguish between systems based on Windows 95 and those based on Windows NT. **`software-version`** allows you to identify variants such as Windows Millennium Edition, Windows 2000, Windows XP, Windows Vista and so on. See the *LispWorks Reference Manual* for details.

24.2 Site Name

The Common Lisp functions **`short-site-name`** and **`long-site-name`** can be configured using **`setf`**:

```
(setf (long-site-name) "LispWorks Ltd"
      (short-site-name) "LW")
```

24.3 The Lisp Image

The function **lw:lisp-image-name** returns the namestring of the full path of the LispWorks executable or dynamic library (DLL). For example, the directory of the image can be found using:

```
(pathname-location (lisp-image-name))
```

To create a new executable or DLL, typically after loading patches, modules and application code, use **save-image** or **deliver**.

24.4 The Command Line

The command line used to run LispWorks can be found using the variable **system:*line-arguments-list***. The value is a list of strings containing the executable name followed by any other command line arguments, in the order they were passed.

For example, if your application needs to behave differently when passed an argument **-foo**, use the following test:

```
(member "-foo" sys:*line-arguments-list* :test 'string=)
```

24.4.1 Command Line Arguments

The following command line options are supported by the system.

-siteinit ***siteinit-file***

siteinit-file names a file to be loaded on startup. The file is the LispWorks site initialization file, containing code that by default is loaded when LispWorks is started by any user in that installation. The default is to load the file that is the result of evaluating **(sys:lispworks-file "config/siteinit.lisp")**.

If *siteinit-file* is not found, an error is signalled. To suppress loading of a site initialization file, pass **-siteinit -**.

-init ***init-file***

init-file names a file to be loaded on startup after *siteinit-file*. The file is user's own LispWorks initialization file, containing code that by default is loaded when LispWorks is started. It is useful for loading initializations that should not be done for all users.

Initially the default is to load the file **"~/.lispworks"** where ~ expands to the user's home directory as described in "Configuration and initialization files" on page 132.

Your default initialization file can be set in the Common LispWorks IDE. See "Setting global preferences" in the *Common LispWorks User Guide* for details.

If *init-file* is not found, an error is signalled. To suppress loading of a user initialization file, pass **-init -**.

-build *build-script*

build-script names a file to be loaded on startup, typically for the purpose of building another image. LispWorks quits after loading the file. If an error is signalled while loading the file, a backtrace is displayed and LispWorks quits.

Note: *init-file* and *siteinit-file* are not loaded when using **-build**, so your build script file must include **(load-all-patches)**.

-environment Start the Common LispWorks development environment automatically, even in an image saved with **(save-image ... :environment nil)**

-env A synonym for **-environment**.

-display ***display***

Sets the X display to use when starting a LispWorks GUI on X Windows.

-multiprocessing

Initializes multiprocessing on startup. See "Multiprocessing" on page 159.

-no-restart-function

Suppresses the execution of a restart function on startup. Restart functions can be supplied when saving an image to automatically invoke application code. This argument suppresses that behavior. See **save-image** in the *LispWorks Reference Manual*.

-ORBport ***orbport***

orbport specifies a port number for the LispWorks ORB. The special value 0 allows the system to pick a port.

-IIOPhost *host*

Controls the host name in placed in IORs. See *Developing Component Software with CORBA* for details.

-IIOPnumeric

IORs contain a host name which is the numeric IP address obtained by reverse lookup of the machine name. See *Developing Component Software with CORBA* for details.

--relocate-image ***BaseAddress***

Causes the image to relocate at *BaseAddress* on supported platforms, as described in "Startup relocation" on page 289. This can be useful on a system where libraries are mapped in address space that LispWorks would otherwise use as it grows. If the image is saved, then on restart without **--relocate-image**, it will locate itself automatically at *BaseAddress*.

Compatibility Note: In LispWorks 5.0 and earlier versions, to be effective, **--relocate-image** must be the first argument on the LispWorks command line. This restriction does not apply in LispWorks 5.1.

--reserve-size ***ReserveSize***

Specifies the reserve size on supported platforms, as described in "Startup relocation" on page 289. New in LispWorks 5.1.

24.5 Address Space and Image Size

There are two factors that affect the maximum size of the Lisp image: the size of real memory, and the layout of memory. On most platforms you can relocate LispWorks to avoid clashes with other software as described in "Startup relocation" on page 289.

24.5.1 Size of real memory

If LispWorks becomes significantly larger than the size of the real memory, then paging will be the main activity and LispWorks will not function effectively.

24.5.2 Layout of memory

This is Operating System-dependent:

On Solaris, 32-bit LispWorks is mapped at **`#x10000000`**. In principle it can grow to almost **`#x80000000`** (the libraries are at higher addresses).

On HP-UX, 32-bit LispWorks is mapped at **`#x50000000`**, because it cannot use the first quadrant. The libraries are also mapped at the same quadrant, at around **`#x7a000000`**, so the total size can be a little more than 0.5GB.

For the other platforms and for 64-bit LispWorks, see the discussion in "Startup relocation" on page 289.

24.5.3 Reporting current allocation

The simplest way to see the current Lisp allocation is to call **`(room t)`**.

To obtain values representing the current total allocation, call **`sys:room-values`**.

24.6 Startup relocation

On startup, LispWorks normally maps its heap at the address where it was mapped when the image was saved. It maps more memory close to this when needed. This may cause memory clashes with other software, but such clashes may be avoided by relocating LispWorks.

32-bit LispWorks is relocatable on Microsoft Windows, Intel Macintosh, Linux and FreeBSD. The 32-bit LispWorks implementations on non-x86 platforms are not relocatable. 64-bit LispWorks is relocatable on all supported platforms. The discussion in this section is applicable to all relocatable implementations.

On Microsoft Windows and Macintosh, LispWorks detects memory clashes and avoids them automatically. On these platforms there is no need to explicitly relocate LispWorks. The other relocatable implementations - LispWorks (32-bit) for Linux, LispWorks (64-bit) for Linux, LispWorks (32-bit) for FreeBSD, and LispWorks (64-bit) for Solaris - cannot safely detect memory clashes. Relocation may therefore be useful in these implementations.

24.6.1 How to relocate LispWorks

Relocate LispWorks by passing two parameters: the base address and the reserve amount. Both are optional. The interpretation of these parameters is very different between 64-bit LispWorks and 32-bit LispWorks.

To relocate a LispWorks executable on non-Windows platforms, pass one or both of these command line arguments:

`--relocate-image` ***BaseAddress***

The base address, interpreted as a hexadecimal number by calling **`strtol(`*BaseAddress*`,NULL,16))`**

`--reserve-size` ***ReserveSize***

The reserve size, interpreted as a hexadecimal number by calling **`strtol(`*ReserveSize*`,NULL,16))`**

There is currently no way to control the relocation of a LispWorks for Windows executable.

On all relocatable platforms, a LispWorks dynamic library or Windows DLL can be relocated by calling **`InitLispWorks`** with second and/or third argument non-zero.

On non-Windows platforms, you can add the appropriate call to **`InitLispWorks`** in wrappers written in C and added to the dynamic library by passing *dll-added-files* to **`save-image`** or **`deliver`**. There is no such option in LispWorks for Windows.

The startup relocation takes some time, normally less than 0.1 seconds on a modern machine. If the relocation address is fixed and known, this startup overhead can be eliminated by relocating the image before calling **save-image** or **deliver**.

24.6.2 Startup relocation of 32-bit LispWorks

32-bit LispWorks on x86 platforms maps its heap in one continuous block, and then grows upwards from the top. When it reaches a region that it cannot use, it can skip it. On Windows and Macintosh this skipping is safe, because LispWorks can safely detect regions of memory that it cannot use. On other x86 platforms, both the initial mapping and the furthur growing cannot safely detect when they overwrite some other code.

BaseAddress (passed on command line with **--relocate-image** or as the second argument to **InitLispWorks**) tells LispWorks where to map the heap. On Windows and Macintosh, if the address is already used the heap will be mapped elsewhere. On other platforms, the mapping always works, and may destroy what is already mapped at that address.

ReserveSize (passed on command line with **--reserve-size** or as the third argument to **InitLispWorks**) tells LispWorks how much additional memory to reserve. Reservation is properly supported on Windows and Macintosh, though the actual reserved size can be smaller if it fails to reserve as much as was requested. On platforms that do not support reservation (that is, not Windows or Macintosh), the reservation is done by using **mmap** with protection PROT_NONE.

24.6.2.1 Linux

On Linux, the default initial heap is mapped at address **#x20000000** (0.5GB). LispWorks then tries to locate the location of dynamic libraries, and marks a region around these libraries that should not be used (by default 64MB from the bottom). In most cases this suffices to avoid clashes.

Problems can arise if the memory at **#x20000000** or above is already used by another part of the software. If that memory gets used before LispWorks is mapped, LispWorks must be relocated elsewhere, typically to a higher address.

If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by supplying *ReserveSize*.

The location of dynamic libraries differs between Linux configurations, and that needs to be taken into account. For most cases, including the cases where the libraries are mapped at **#x40000000** or somewhere above **#x28000000**, the mechanism for detecting libraries works and no action is required.

In principle LispWorks (32-bit) for Linux can grow up to some distance below **#xBF000000** (almost 2.5GB), though this depends on the OS kernel allowing this size.

Note: In LispWorks 5.0 and previous, we told some customers to relocate above the libraries, for example at **#x50000000** or **#x48000000**, but this should not be needed in LispWorks 5.1.

24.6.2.2 FreeBSD

By default, LispWorks is mapped at **#x30000000**.

Problems may arise if something uses memory above **#x30000000**. If this memory is used before LispWorks is mapped, LispWorks must be relocated elsewhere, typically to a higher address.

If the memory above LispWorks gets used by other parts of the software after LispWorks was mapped, it may be possible to avoid the problem by reserving some memory above LispWorks by using *ReserveSize*.

Normally the dynamic libraries are mapped at **#x28000000**, and therefore LispWorks can grow without a problem.

In principle LispWorks can grow up to some distance below **#xC0000000** (almost 2.25GB), though this depends on the OS kernel allowing this size and how many threads you have running.

24.6.2.3 Windows and Macintosh

LispWorks (32-bit) for Windows and LispWorks (32-bit) for Macintosh both map by default at **#x20000000**. Since these platforms support reservation, normally you will not need to do anything special about this.

Problems may however arise if LispWorks operates in conjunction with non-relocatable software which insists on using addresses at **#x20000000** or some distance above, in which case you will need to relocate LispWorks.

LispWorks (32-bit) for Windows can in principle grow up to some distance below **#x80000000** (almost 1.5GB) but there is always the possibility that some DLL will be mapped in this region. On startup, it reserves 0.5GB above its location, so that much is guaranteed.

LispWorks (32-bit) for Macintosh can grow to around 2.7GB. You can relocate it only on the Intel architecture.

24.6.3 Startup relocation of 64-bit LispWorks

The size of address space that 64-bit LispWorks can use is limited by the size of internal tables to a "span" of 2^44 (16TB). The span always starts at 0.

Inside this span LispWorks can use any address. However, to avoid clashes with other software, it uses memory only in some defined range.

Startup relocation means changing this range. *BaseAddress* (passed on command line with **--relocate-image** or as the second argument to **InitLispWorks**, rounded up to 2^28) is the start of the range. *ReserveSize* (passed on command line with **--reserve-size** or as the third argument to **InitLispWorks**) is the size of the range. The default of the size of the range is 2^40.

If the entire heap is within the new range, nothing else is done. If some part of the heap is outside the new range, the heap is relocated.

The range in each 64-bit LispWorks implementation starts at **#x4000000000** (256GB).

24.6.3.1 Linux

In LispWorks (64-bit) for Linux the range is 192GB, ending at **#x7000000000**, because Linux cannot map above **#x8000000000** and puts the dynamic libraries just below that limit (at least in some configurations). Since LispWorks uses the address space sparsely, it will run out of memory with less virtual memory, probably around 150GB to 160GB. If more memory is required, the range can be extended downwards, and possibly some distance upwards too. If other software uses memory in the range from **#x4000000000** to

#x7000000000, LispWorks should be relocated (potentially just by decreasing the range) to avoid memory clashes.

24.6.3.2 Solaris

In LispWorks (64-bit) for Solaris the default range is 768GB, ending at **#x10000000000**. If other software uses memory in this range, the range for LispWorks should be decreased to avoid memory clashes.

24.6.3.3 Windows and Macintosh

In LispWorks (64-bit) for Windows and LispWorks (64-bit) for Macintosh the size of the range is **#x3c000000000** (3.75TB). Since these platforms properly support reservation, there should not be any reason to change the range. The only time when this is needed is when other software insists on using some address in this range and does not relocate automatically.

24.7 Exit status

You can return a process exit status to the Operating System when LispWorks or a delivered LispWorks application quits.

Do this by passing a *status* value to the function **quit**. For example:

```
(quit :status 42)
```

24.8 Creating a new executable with code preloaded

There are two ways to create a new executable with your code preloaded.

- **save-image** writes a copy of the currently running image to disk. For more details see the *LispWorks Reference Manual*. This image requires a development license key to run.
- **deliver** creates a runtime image, removing unused code to make the image smaller. For more details see the *LispWorks Delivery User Guide*.

For example of how to use **save-image**, see the section "Saving and testing the configured image" in the *LispWorks Release Notes and Installation Guide*.

See Section 24.9 for information about universal binaries on Mac OS X.

24.9 Universal binaries on Mac OS X

The supplied 32-bit LispWorks for Macintosh images are universal binaries, which run the correct native architecture on PowerPC and Intel-based Macintosh computers by default.

A running Lisp image only supports one architecture, chosen when the image was started. On a PowerPC based Macintosh, this is always the PowerPC architecture. On an Intel-based Macintosh, it can be either the native Intel architecture or the PowerPC architecture (using Rosetta).

Functions such as **`save-image`** and **`deliver`** mentioned in Section 24.8 create an image containing only the running architecture and functions that operate on fasl files such as **`compile-file`** and **`load`** only support the running architecture.

To build a universal binary application from LispWorks for Macintosh 5.x, you will need to install LispWorks on an Intel-based Macintosh computer.

Building a new universal binary requires three steps:

1. Build the application for PowerPC.

   This can be done on your Intel machine using Rosetta

2. Build the application for Intel.
3. Combine the two applications to make a universal binary.

These steps can be automated on a single Intel-based Macintosh by creating a script that compiles and loads the application and then saves the image. Loading this by running LispWorks with the **`-build`** command line argument would save an image containing a single architecture, but you can use the same script to save a universal binary by calling **`save-universal-from-script`** as described in the *LispWorks Reference Manual*.

Note: You may install LispWorks on multiple machines for use at the same time only if you own multiple LispWorks licenses.

24.10 User Preferences

LispWorks provides an API for setting and querying persistant per-user settings in a platform-dependent registry.

24.10.1 Location of persistant settings

On Microsoft Windows the preferences are stored in the HKEY_CURRENT_USER branch of the Windows registry. (LispWorks also offers a general Windows registry API, described in "Accessing the Windows registry" on page 296.)

On non-Windows the preferences are stored in subdirectories of the user's home directory.

To implement preferences for your Lispworks application, you will need to define a registry path using **(setf sys:product-registry-path)**.

24.10.2 Accessing persistant settings

Get and set preferences under the product path at runtime with **lw:user-preference** and **(setf lw:user-preference)**.

24.10.3 Example using user preferences

Define a registry path:

```
(setf (sys:product-registry-path :deep-thought)
      '("Software" "My Company" "Deep Thought"))
```

Store a preference for the current user:

```
(setf (user-preference "Answers"
                       "Ultimate Question"
                       :product :deep-thought)
      42)
```

Retrieve a preference for the current user, potentially in a subsequent session:

```
(user-preference "Answers" "Ultimate Question"
                 :product :deep-thought)
```

24.11 Accessing the Windows registry

There is an API for accessing the registry on Microsoft Windows. It is available only in LispWorks for Windows. All of its symbols are in the **win32** package.

Create and delete keys with the functions **create-registry-key** and **delete-registry-key**. Open a key for reading and/or writing with **open-registry-**

key and close it with **close-registry-key**, or wrap your registry operation inside the macro **with-registry-key**.

Query the registry with **registry-key-exists-p**, **enum-registry-value**, **collect-registry-values**, **collect-registry-subkeys**, **query-registry-key-info**, **query-registry-value**, and **registry-value**. Write to the registry with **set-registry-value** or **(setf registry-value)**.

For example, this function returns the name, progid and filename for each of the installed ActiveX controls:

```
(defun collect-control-names (&key insertable
                                   (max-name-size 256)
                                 (max-names most-positive-fixnum))
  (win32:collect-registry-subkeys
   "CLSID"
   :root :root
   :max-name-size max-name-size
   :max-names max-names
   :value-function
   #'(lambda (hKeyClsid ClassidName)
       (win32:with-registry-key
           (hkeyX ClassidName :root hKeyClsid :errorp nil)
         (when (and
                (win32:registry-key-exists-p "Control"
                                             :root hkeyX)
                (if insertable
                    (win32:registry-key-exists-p "Insertable"
                                                 :root hkeyX)
                  t))
           (when-let
               (progid (win32:query-registry-value "ProgID" nil
                                                   :root hkeyX
                                                   :errorp nil))
             (values
              (list
               (win32:query-registry-value nil nil
                                           :root hkeyX)
               progid
               (win32:query-registry-value "InprocServer32" nil
                                           :root hkeyX
                                           :errorp nil))
              t)))))))
```

24.12 The home directory

This section describes the implementation of the Common Lisp function **user-homedir-pathname**.

On Unix-based systems, the home directory is looked up using the C function getpwuid.

On Microsoft Windows systems, **user-homedir-pathname** uses the environment to construct its result. It uses the values of the environment variables HOMEDRIVE and HOMEPATH, if both are defined. If at least one of environment variables HOMEDRIVE and HOMEPATH is not defined, then a pathname **#P"C:/users/*login-name*"** is returned. These environment variables should be correctly set before LispWorks starts. However it is possible to change the values in Lisp using

```
(setf lw:environment-variable)
```

24.13 Special Folders

On Microsoft Windows and Mac OS X there are various special folders used for application data and user data. Here are some examples of the folder for application data which is shared between all users.

Windows Vista:

C:\ProgramData

Windows XP:

C:\Documents and Settings\All Users.WINDOWS\Application Data

Windows 2000:

C:\Documents and Settings\All Users\Application Data

Mac OS X:

/Library/Application Support

The locations and folder names can differ between versions of the operating system, therefore it is useful to have a system-independent way to get the path at runtime. The function **system:get-folder-path** can be used to retrieve the path to special folders. Directory pathnames corresponding to each of the examples above can be obtained by calling:

```
(sys:get-folder-path :common-appdata)
```

Here is another example of differences between operating systems. On Windows Vista:

```
(sys:get-folder-path :my-documents)
=>
#P"C:/Users/dubya/Documents/"
```

On Windows 98 SE:

```
(sys:get-folder-path :my-documents)
=>
#P"C:/My Documents/"
```

On Mac OS X:

```
(sys:get-folder-path :my-documents)
=>
#P"/u/ldisk/dubya/Documents/"
```

See the *LispWorks Reference Manual* for more details of **`sys:get-folder-path`**.

On Windows NT-based systems there is a profile folder for each user. You can find the profile path for the current user with the function **`sys:get-user-profile-directory`**. See the *LispWorks Reference Manual* for more details.

25

64-bit LispWorks

This chapter summarises the technical differences between 64-bit LispWorks and 32-bit LispWorks. Both are ANSI Common Lisp implementations and support the language same extensions and libraries so in many ways they behave the same. However the programmer should be aware of the differences mentioned here.

25.1 Introduction

64-bit LispWorks has a larger address space, subject to physical memory. The maximum heap sizes are shown in Table 25.1.

You can make larger arrays and the **`fixnum`** type is much larger than in 32-bit LispWorks. The values of various Common Lisp architectural constants reflect this, as shown in Table 25.2.

Other differences in 64-bit LispWorks are noted in the remaining sections of this chapter.

25.2 Heap size

In principle 64-bit LispWorks can grow to almost 16TB but it is intentionally limited to a defined range in order to avoid clashes with other software as shown in Table 25.1.

Table 25.1 Default range for 64-bit LispWorks heap

Platform	Default range	Notes
Intel-based Macintosh	`#x4000000000` to `#x40000000000` (3.75TB)	
PowerPC Macintosh	`#x4000000000` to `#x40000000000` (3.75TB)	
Linux	`#x4000000000` to `#x7000000000` (192GB)	Effective limit around 160GB.
Windows	`#x4000000000` to `#x40000000000` (3.75TB)	
Solaris	`#x4000000000` to `#x10000000000` (768GB)	

64-bit LispWorks is relocatable on all supported platforms as described in “Startup relocation of 64-bit LispWorks” on page 293.

In contrast, 32-bit LispWorks has a maximum heap size of 1.5-3.0GB depending on platform and is relocatable on non-Windows platforms only, as described in “Startup relocation of 32-bit LispWorks” on page 291.

25.3 Architectural constants

Common Lisp constants have the values shown in Table 25.2

Table 25.2 Architectural constants

Constant	32-bit LispWorks	64-bit LispWorks
most-positive-fixnum	2^29 - 1	2^60 - 1
array-dimension-limit	67108337 (almost 2^26)	2^29 - 1
array-total-size-limit	2^26	2^29 - 1

Note: In 32-bit LispWorks 5.0, **array-total-size-limit** is 2^29 - 1, which is wrong.

25.4 Speed

64-bit LispWorks is generally faster than 32-bit LispWorks.

We would be interested to see comparitive performance data from your application if it runs on both 32-bit and 64-bit LispWorks.

25.5 Memory Management

Memory layout and the garbage collector (GC) differs significantly between the two implementions.

For the details see “Memory Management in 32-bit LispWorks” on page 106 and “Memory Management in 64-bit LispWorks” on page 114.

25.6 Float types

In 64-bit LispWorks **single-float**s are immediate objects, and **short-float** is the same type as **single-float**.

In 32-bit LispWorks **single-float**s are boxed objects, and **short-float** is disjoint from other float types.

25.7 External libraries

Third party libraries loaded into 64-bit LispWorks must be 64-bit. Availability of a suitable library is therefore a possible issue when porting your LispWorks application to 64-bit.

Third party libraries loaded into 32-bit LispWorks must be 32-bit.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W