sun® microsystems

Sun™ 2.1 Common Lisp Release Notes

Part Number: 800-2069-10
Revision: A of 1 September 1987

Release Notes
Sun Common Lisp 2.1

Introduction

Release 2.1 of Sun Common Lisp is a maintenance release that supercedes Release 2.0. This document provides the following information about Release 2.1:

- functional improvements to the software
- performance enhancements
- extensions to the user interface
- other extensions
- restrictions
- additional examples of the Foreign Function Interface
- corrections to existing documentation

Functional Improvements

You can now load foreign code without multiple invocations of the **ld** command or use of temporary file space; thus, loading is much faster. Foreign code can be dynamically reloaded; Lisp maintains the relocation information required for updating references to the reloaded code.

The **arglist** function provides more useful information about macros and functions that have special variables in their argument lists.

The Compiler generates safe memory accessors when the safety optimization class is set to 2 or higher. Code compiled with safe memory accessors signals an error when an illegal reference, such as `(cdr 1)`, is attempted.

Performance Enhancements

The Lisp image is much smaller. We have "shaken" the image to remove inaccessible data structures and code. The **disksave** function has been modified to create images with a bss of zero size. Both of these changes improve Lisp's performance in environments with limited physical memory and small swapping volumes.

The Compiler conses less; there are fewer garbage collections, and there is less paging activity. This improvement is important for machines with limited physical memory and/or slow swapping devices.

The Compiler can now generate in-line code for the 68881. With proper declarations, it maintains floating-point numbers on the stack rather than allocating them in the dynamic heap. These two changes have improved the speed of some floating-point benchmarks by a factor of 50.

The Compiler can allocate the **&rest** arguments of some functions on the stack rather than on the dynamic heap. Thus, certain calls cons less.

Internal changes to the Lisp memory model reduce the amount of memory scanned during garbage collections. Thus, garbage collection is faster.

Improvements in the generic arithmetic interfaces reduce the overhead for bignum operations and provide a substantial speed-up of bignum-intensive computations. For example, 1000! is computed about five times faster than in the previous release.

The **send** function in the Flavor System does not cons when it has four or fewer arguments.

Extensions to the User Interface

The **defadvice** macro, similar to that found in Symbolics™ ZetaLisp®, is available to users. The Advice Facility allows you to modify the behavior of an existing function by attaching named pieces of advice. You can attach multiple pieces of advice to the same function in a specified order.

A Source File Recording Facility is available to record definitions of functions, macros, and structures. It can be extended to record definitions of objects of other types.

New functions provide interaction with the UNIX environment: **cd**, **pwd**, **working-directory**, **environment-variable**, **command-line-argument**, **run-program**, and **shell**. Note that the function **run-unix-program** has been renamed **run-program**; however, **run-unix-program** is included in this release for backward compatibility.

The Editor provides two new functions. The *meta-.* command displays the source code associated with a symbol. The **Ctrl-C Ctrl-A** command displays the argument list and documentation associated with a symbol.

Some useful functions, including **def-compiler-macro**, **destructuring-bind**, **defsubst**, and **gc-size** have been exported to the user package.

The macro **with-static-area** causes objects to be allocated in the static heap rather than in the dynamic heap during a cons.

The variable ***gc-silence*** may be bound to a function that is called at certain times during each garbage collection.

The **load** function is more flexible. It uses new variables to control what file extensions to consider when looking for a source file and when looking for a binary file. In addition, new keywords control the behavior of **load** when it finds both a source and a binary file or when it finds only a source file.

The initial value of the variable ***load-verbose*** is now **t**; thus, by default **load** prints information about its progress on the standard output.

Other Extensions

A file named RELEASE-NOTES is included on the release tape. This file contains documentation for several functions, macros, variables, and constants that are internal to the Lucid system and that may be useful to developers and advanced users of Sun Common Lisp. Users of these constructs should take note of the following points:

- The constructs documented in the file are not supported. They may change or may not be included in future releases.
- The only documentation for these constructs appears in the RELEASE-NOTES file.
- Many functions listed in the file do not check their arguments. It is an error to pass inconsistent arguments to these internal functions.

The following Lucid extensions to Common Lisp are documented in the RELEASE-NOTES file:

lucid::add-debugger-binding
lucid::debugger-bindings
lucid::remove-debugger-binding
ed::catching-editor-errors
ed::*debug-switch-to-lisp-buffer*
ed::*real-editor-input*
ed::*scrollbars-p*
lucid::code-ref
lucid::code-length
lucid::map-objects
lucid::*bx-alloc-segments*
windows::root-reshaped-actions
windows::leave-windows-actions
windows::bitblt-bitmap-to-bitmap
windows::bitblt-bitmap-to-screen
windows::bitblt-screen-to-bitmap
windows::bitblt-screen-to-screen
windows::bitblt-bitmap-to-bitmap-with-replication
windows::bitblt-bitmap-to-screen-with-replication

lucid::with-error-trapping
lucid::top-level
lucid::procedurep
lucid::procedure-arglist
lucid::procedure-code
lucid::procedure-flags
lucid::procedure-length
lucid::procedure-literals
lucid::procedure-ref
lucid::procedure-symbol
lucid::structurep
lucid::structure-length
lucid::structure-ref
lucid::structure-type
lucid::structure-type-p
lucid::%pointer

Restrictions

Compiled files that contain calls to **define-c-function** and **load-foreign-files** must be recompiled in 2.1 because these macros have been changed.

The previously undocumented internal function **lucid::getenv** no longer exists; it has been replaced by the function **environment-variable**, which is described in the function pages of this document.

Using the New Foreign Loader

The following rules apply to loading and reloading foreign code:

1. Writers of foreign code cannot assume that text, data, and bss address spaces have a particular layout in memory; with the dynamic loader, pieces of text and data are intermixed. If a given module requires contiguous text and data segments, use the following command to make a single object file that follows the UNIX model:

   ```
   ld -r file-names -o temp.o
   ```

The text segment is never contiguous to the data segment, and it is unsafe to assume that text will lie below data. The information returned by `sbrk(0)` is not useful.

Note: Files compiled using the `-g` option to the `cc` command cannot be used in the new foreign loader.

2. Use **malloc** to manage memory, if possible; avoid using **sbrk**.
3. Define global variables in files separate from those containing the definitions of functions referring to these variables. If this is not possible and it is necessary to redefine a function in one of these files, reload new definitions for all of the variables and functions in the file that changed.
4. Avoid functions with multiple entry points. For example, do not write assembly code or C code with references such as the following:

```
jsr _funct +3                   ; Assembly code to avoid.

temp = &function +10;           ; C code to avoid.
(*temp)(19);
```

Foreign Function Interface Examples

p. 7-9 define-fortran-function

```
; This example defines a function that calls the FORTRAN library function
; "getpid" to return the pid number of this process.

> (define-fortran-function getpid() :result-type :integer)
GETPID
> (load-foreign-libraries nil '("-lF77" "-lU77"))
T
> (getpid)
2385

; This example assumes that you have created a FORTRAN file called
; "bar.f" with the following contents (leading blanks are significant):
;
;         function baz (x,y)
;         real baz
;         real x,y
;         baz = x+y
;         return
;         end
;
; You then compile this file with the command 'f77 -c bar.f' to produce
; the binary object file "bar.o".

> (define-fortran-function baz (x y) :result-type :single)
BAZ
> (load-foreign-files '("bar.o"))
T
> (baz 1.2 3.3)
4.5
```

p. 7-12 define-foreign-symbol

```
; This example shows how to get the address of a foreign symbol.
; The symbol atol contains the address of the C library function "_atol".

> (define-foreign-symbol atol)
ATOL
> (load-foreign-libraries nil '("-lc"))
T
> atol
2810524
```

Note: The symbol that is defined contains the address of the foreign symbol and not the value. You cannot use the macro **setf** to change the value of the symbol inside the foreign code.

p. 7-13 extract-stream-handles

```
; This example shows how stream handles can be used.  The first value
; of the UNIX file descriptor that is returned represents the input
; direction, and the second value is the output direction.

> (extract-stream-handles (open "any-file" :direction :output
                                           :if-exists :supersede))
NIL
3
> (extract-stream-handles (open "old-file" :direction :input))
4
NIL
> (multiple-value-setq (in out) (extract-stream-handles *terminal-io*))
0
> (define-c-function (write-c "_write") (d s n))
WRITE-C
> (load-foreign-libraries nil '("-lc"))
T
> (progn (write-c out "hello" 5) (terpri) t)
hello
T
```

p. 7-14 foreign-address-of

```
; This example assumes that the file cegl.c has been compiled from the
; following source code with cc -c cegl.c.
;
;       call_c_func(fun,a,b)
;       int (*fun)();
;       char *a,*b;
;       {
;         return((*fun)(a,b));
;       }
;

> (define-c-function strcmp((a :string) (b :string))
          :result-type :integer)
STRCMP
> (load-foreign-libraries nil '("-lc"))
T
> (strcmp "aba" "abc")
-2
```

```
> (foreign-address-of 'strcmp)
2752648
> (define-c-function call-c-func((f :pointer)(a :string)(b :string))
          :result-type :integer)
CALL-C-FUNC
> (load-foreign-files '("cegl.o") '("-lc"))
T
> (call-c-func (foreign-address-of 'strcmp) "aba" "abc")
-2
```

Note: The function **foreign-address-of** applies to foreign functions that have been defined with **define-c-function** or **define-fortran-function**; the function **register-lisp-function** applies to Lisp functions that have been defined with the macro **define-c-callable.**

p. 7-20 **register-lisp-function**

Refer to the example for **define-c-callable** on page **7-8** of the *Sun Common Lisp User's Guide.*

Corrections to the Documentation

This section lists corrections to the text of the *Sun Common Lisp User's Guide* and the *Sun Common Lisp Reference Manual.* Corrections are listed by page number.

Sun Common Lisp User's Guide

p. 2-4, 5 In the sections "Customizing the Lisp Environment" and "Using the Display Facilities," the references to the extension `.2bin` should be ignored. This extension is not used by default when a source file is compiled with a **:target** option of `68020`.

p. 7-6 In the "Table of Data Types for FORTRAN Programs," **logical*1 array** should be **character*1 array**.

p. 7-8 The expression `(load-foreign-files '("callmeback.o"))` should return `T`, not `NIL`.

p. 7-9 The following expression should be added to the "Purpose" statement for **define-c-function, define-fortran-function**:

> The functions created by **define-c-function** and **define-fortran-function** are interpreted Lisp functions that can be compiled to reduce consing.

p. 7-10 In the description of "type-checking syntax" for **define-c-function, define-fortran-function**, the sentence "Possible argument types include all of the types that the keyword **:result-type** accepts, as well as the keyword **:string**" should be replaced with the following:

> Possible argument types are **:fixnum**, **:integer**, **:pointer**, **:single**, and **:string**.

p. 7-11 The expression `(load-foreign-libraries '("_printf"))` should return `T`.

The expression `(printf "It's %d %s" 3 "in the morning.")` should return 22, not 21.

p. 7-15 The following paragraph should be added to the "Remarks" section for ***foreign-temporary-directory***:

> This variable is provided for backward compatibility; you should not use it in ordinary circumstances.

p. 7-17 In the "Remarks" section for **load-foreign-libraries**, the first sentence should read as follows:

The *symbols* argument is a list of Lisp strings that name the foreign functions as they appear in a foreign object file: `"_fn"` for C functions named `fn`, and `"_fn_"` for FORTRAN functions named `fn`.

In the "Examples" section, the line that reads

```
(define-c-function hypot (x y) :result-type :single)
```

should be replaced with the following lines:

```
(define-c-function hypot (x y) :result-type
                               :coerce-double-to-single)
```

The line that reads

```
(load-foreign-libraries '("_mktemp" "_hypot") '("-lm"))
```

should be replaced with the following line:

```
(load-foreign-libraries nil '("-lm" "-lc"))
```

pp. 10-5, 6 The output for the expression `(room t)` should be as follows:

```
> (room t)
;;; 49156 words [196624 bytes] of dynamic storage in use.
;;; 32512 words [130048 bytes] of free storage available before a GC.
;;; 114180 words [456720 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 320K bytes [5 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 192K bytes [3 segments]
;;; Memory Growth Limit: 15360K bytes [240 segments], total
;;; Memory Growth Rate: 768K bytes [12 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                       Size [used/allocated]
;;; ----                       ----
;;; Foreign-Area               41K/64K bytes,     1/1 segment
;;; Readonly-Pointer-Area      359K/384K bytes,   6/6 segments
;;; Readonly-Non-Pointer-Area  2584K/2624K bytes, 41/41 segments
;;; Dynamic-0-Area             193K/320K bytes,   4/5 segments
;;; Dynamic-1-Area             0K/320K bytes,     0/5 segments
;;; Static-Area                1779K/1792K bytes, 28/28 segments
NIL
```

p. 10-8 The description of the keyword **:expand** for **change-memory-management** should read as follows:

> This keyword argument forces an immediate expansion of each semi-space of dynamic memory by the specified number of segments. The expansion cannot exceed the maximum size specified by the **:growth-limit** keyword argument.

p. 10-10, 11 The last sentence of paragraph three in the "Remarks" section for **change-memory-management** should read as follows:

> The expansion cannot exceed the maximum size specified by the **:growth-limit** keyword argument.

The "Examples" section should change as follows:

```
> (room t)
;;; 45318 words [181272 bytes] of dynamic storage in use.
;;; 626174 words [2504696 bytes] of free storage available before a GC.
;;; 1297666 words [5190664 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 15360K bytes [240 segments], total
;;; Memory Growth Rate: 768K bytes [12 segments]
;;; Reclamation Ratio: 33% desired free after garbage collection
;;; Area Information:
;;; Name                       Size [used/allocated]
;;; ----                       ----
;;; Foreign-Area               25K/64K bytes,     1/1 segment
;;; Readonly-Non-Pointer-Area  2265K/2304K bytes, 36/36 segments
;;; Readonly-Pointer-Area      257K/320K bytes,   5/5 segments
;;; Read-Write-Area            456K/512K bytes,   8/8 segments
;;; Static-Area                609K/640K bytes,   10/10 segments
;;; Dynamic-0-Area             178K/2624K bytes,  3/41 segments
;;; Dynamic-1-Area             0K/2624K bytes,    0/41 segments
NIL
> (change-memory-management :growth-limit 202 :growth-rate 16
                            :reclamation-ratio 0.25)
T
> (room t)
;;; 45572 words [182288 bytes] of dynamic storage in use.
;;; 625920 words [2503680 bytes] of free storage available before a GC.
;;; 1297412 words [5189648 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                       Size [used/allocated]
;;; ----                       ----
;;; Foreign-Area               25K/64K bytes,     1/1 segment
;;; Readonly-Non-Pointer-Area  2265K/2304K bytes, 36/36 segments
;;; Readonly-Pointer-Area      257K/320K bytes,   5/5 segments
;;; Read-Write-Area            456K/512K bytes,   8/8 segments
;;; Static-Area                609K/640K bytes,   10/10 segments
;;; Dynamic-0-Area             179K/2624K bytes,  3/41 segments
;;; Dynamic-1-Area             0K/2624K bytes,    0/41 segments
NIL
```

p. 10-13, 14 The "Examples" section should change as follows:

```
> (room t)
;;; 5068 words [20272 bytes] of dynamic storage in use.
;;; 666424 words [2665696 bytes] of free storage available before a GC.
;;; 1337916 words [5351664 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-1-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                       Size [used/allocated]
;;; ----                       ----
;;; Foreign-Area               25K/64K bytes,     1/1 segment
;;; Readonly-Non-Pointer-Area  2265K/2304K bytes, 36/36 segments
;;; Readonly-Pointer-Area      257K/320K bytes,   5/5 segments
;;; Read-Write-Area            456K/512K bytes,   8/8 segments
;;; Static-Area                609K/640K bytes,   10/10 segments
;;; Dynamic-0-Area             0K/2624K bytes,    0/41 segments
;;; Dynamic-1-Area             20K/2624K bytes,   1/41 segments
NIL
> (gc-off)
;;; GC: 5018 words [20072 bytes] of dynamic storage in use.
;;; 666474 words [2665896 bytes] of free storage available before a GC.
;;; 1337966 words [5351864 bytes] of free storage available if GC is disabled.
T
> (room t)
;;; 5030 words [20120 bytes] of dynamic storage in use.
;;; 1338206 words [5352824 bytes] of free storage available before a GC.
;;; 1338206 words [5352824 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Disabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                         Size [used/allocated]
;;; ----                         ----
;;; Foreign-Area                 25K/64K bytes,      1/1 segment
;;; Readonly-Non-Pointer-Area    2265K/2304K bytes,  36/36 segments
;;; Readonly-Pointer-Area        257K/320K bytes,    5/5 segments
;;; Read-Write-Area              456K/512K bytes,    8/8 segments
;;; Static-Area                  609K/640K bytes,    10/10 segments
;;; Dynamic-0-Area               20K/3968K bytes,    1/62 segments
;;; Dynamic-1-Area               0K/2624K bytes,     0/41 segments
NIL
```

p. 10-15, 16 The "Examples" section should change as follows:

```
> (gc-off)
T
> (room t)
;;; 6020 words [24080 bytes] of dynamic storage in use.
;;; 1337216 words [5348864 bytes] of free storage available before a GC.
;;; 1337216 words [5348864 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Disabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                         Size [used/allocated]
;;; ----                         ----
;;; Foreign-Area                 25K/64K bytes,      1/1 segment
;;; Readonly-Non-Pointer-Area    2265K/2304K bytes,  36/36 segments
;;; Readonly-Pointer-Area        257K/320K bytes,    5/5 segments
;;; Read-Write-Area              456K/512K bytes,    8/8 segments
;;; Static-Area                  609K/640K bytes,    10/10 segments
;;; Dynamic-0-Area               24K/3968K bytes,    1/62 segments
;;; Dynamic-1-Area               0K/2624K bytes,     0/41 segments
NIL
> (gc-on)
T
> (room t)
```

```
;;; 6212 words [24848 bytes] of dynamic storage in use.
;;; 665280 words [2661120 bytes] of free storage available before a GC.
;;; 1336772 words [5347088 bytes] of free storage available if GC is disabled.
;;; Semi-space Size: 2624K bytes [41 segments]
;;; Current Dynamic Area: Dynamic-0-Area
;;; GC Status: Enabled
;;; Reserved Free Space: 512K bytes [8 segments]
;;; Memory Growth Limit: 12928K bytes [202 segments], total
;;; Memory Growth Rate: 1024K bytes [16 segments]
;;; Reclamation Ratio: 25% desired free after garbage collection
;;; Area Information:
;;; Name                        Size [used/allocated]
;;; ----                        ----
;;; Foreign-Area                25K/64K bytes,      1/1 segment
;;; Readonly-Non-Pointer-Area   2265K/2304K bytes,  36/36 segments
;;; Readonly-Pointer-Area       257K/320K bytes,    5/5 segments
;;; Read-Write-Area             456K/512K bytes,    8/8 segments
;;; Static-Area                 609K/640K bytes,    10/10 segments
;;; Dynamic-0-Area              25K/2624K bytes,    1/41 segments
;;; Dynamic-1-Area              0K/2624K bytes,     0/41 segments
NIL
```

p. 11-45 The following sentence

> If a default form is specified, it is evaluated as the initial value of the form *var*.

should appear in the second paragraph of the "Remarks" section for **defflavor**, rather than in the third.

The keyword **:required-methods** should be listed as a valid option to **defflavor**, as follows:

> **:required-methods** specifies one or more message names. Any flavor that inherits *flavor-name* must provide a method for each specified message in order to be instantiated; otherwise, an error is signaled.

p. 12-60 The ninth paragraph of the "Remarks" section for **initialize-windows** should read as follows:

> If you try to initialize the Window Tool Kit but it has already been initialized, the following message appears:
>
> `Window system already initialized`

p. 12-64 In the "Remarks" section for **listen-any**, paragraph two should be omitted. No end-of-file can occur in a mouse-input-stream.

p. 12-67 The following sentence should be added to the second paragraph of the "Remarks" section for **make-active-region:**

> The **:mouse-still** keyword argument to **make-active-region** is not supported in this release.

p. 12-75 In paragraph three of the "Remarks" section for **make-mouse-cursor**, for **boole-or** read **boole-ior**.

p. 12-84 In the discussion of the keywords **:width** and **:height** for **make-window**, the last sentence should read as follows:

> If **:width** is omitted or **nil**, its default value is the width of the root viewport; if **:height** is omitted or **nil**, its default value is the height of the root viewport.

p. 12-101 The second paragraph of the "Remarks" section for **peek-any** should read as follows:

> The arguments *eof-error-p* and *eof-value* are provided for compatibility with the Common Lisp functions **read-char** and **read-char-no-hang** and are ignored.

p. 12-107 The third paragraph of the "Remarks" section for **read-any, read-any-no-hang** should read as follows:

> The arguments *eof-error-p* and *eof-value* are provided for compatibility with the Common Lisp functions **read-char** and **read-char-no-hang** and are ignored.

p. 13-17 The key binding **Ctrl-H** should be listed for the editor commands **Delete Previous Character** and **Echo Area Delete Previous Character**.

p. 13-21 The following additions should be made to the options list for **Query Replace**:

Backspace Do not perform the replacement but keep searching.

Ctrl-R Go into a recursive edit at the current position.

Sun Common Lisp Reference Manual

p. 3-13 In the "Examples" section for **subtypep**, the expression `(subtypep 'integer 'string)` should return the following results:

```
> (subtypep 'integer 'string)
NIL
T
```

p. 5-34 The following sentence should be added to paragraph three of the "Purpose" section for **let, let***:

> If no *value* is specified for a *var* argument that is a list element, a warning is signaled.

An example of this warning is as follows:

```
> (defun let-test ()
    (let ((x)) 5))
;;; Warning: The form (LET ((X)) ...) is not legal Common Lisp syntax
LET-TEST
> (let-test)
;;; Warning: The form (LET ((X)) ...) is not legal Common Lisp syntax
5
> (compile 'let-test)
;;; Compiling function LET-TEST...
;;; Warning: The form (LET ((X)) ...) is not legal Common Lisp syntax
;;; Warning: Variable X is bound but its value is ignored.
;;;    assembling...emitting...done.
LET-TEST
> (let-test)
5
```

p. 8-3 In the section "Syntax for Declaration Specifiers," the syntax for the **optimize** declaration should read as follows:

|(`optimize` {(*quality value*) | *quality*}*)

p. 8-4 The following sentence should precede the last sentence of paragraph five:

> If a class is specified and no value is assigned, 3 is used.

p. 8-6 The form **define-setf-method** should be included in the list of forms that may use the special form **declare**.

p. 12-26 In the "Examples" section, the expression `(complex 1 .99)` should return `#C(1.0 0.99)`.

p. 12-29 In the "Examples" section, the expression `(decode-float .5)` should return the following results:

```
> (decode-float .5)
0.5
0
1.0
```

The expression `(decode-float 1.0)` should return the following results:

```
> (decode-float 1.0)
0.5
1
1.0
```

p. 12-66 In the "Examples" section, the value returned by **pi** should be `3.1415927`.

p. 12-77 In the "Examples" section, the expression `(tan #C(0 1))` should return `#C(0.0 0.7615941)`.

p. 12-80 In the "Examples" section, the expression `(truncate .5)` should return the following results:

```
> (truncate .5)
0
0.5
```

The expression `(round .5)` should return the following results:

```
> (round .5)
0
0.5
```

p. 18-9 In the "Remarks" section for **make-hash-table**, the second sentence of paragraph six should read as follows:

> This value may be a positive integer less than **:size** or a floating-point value greater than 0.0 and less than or equal to 1.0.

p. 24-11, 12 In the "Remarks" section for **compile-file**, references to the extension `.2bin` should be ignored. This extension is not used by default for files that are compiled with a **:target** option of `68020`.

The discussion of the **:target** option should be replaced with the following paragraphs:

The possible values for the **:target** option are as follows:

- `68020/68881`

  If this value for the **:target** option is specified, the compiler generates binary files specifically for the MC68881 coprocessor. Such files use the machine-specific floating-point hardware and produce faster floating-point operations. A binary file produced for the MC68881 coprocessor must be loaded on a machine with the correct hardware; otherwise a continuable error is signaled.

- `68020`

  If this value for the **:target** option is specified, the compiler generates binary files specifically for the MC68020 processor. Such files may run slightly faster in some cases, but they will not run on MC68010 processors.

- `68K`

  This is the default value of the **:target** option.

p. 24-33 The "Remarks" section for **sleep** should read as follows:

The *seconds* argument can be any nonnegative noncomplex number.

Additions to the Documentation

This section lists additions to the documentation.

Sun Common Lisp Reference Manual

The additional function pages for the Sun Common Lisp Reference Manual pertain to the following chapters:

- Chapter 6. "Macros"

  destructuring-bind

- Chapter 22. "File System Interface"

  close-all-files

  ignore-binary-dependencies

  load

  load-binary-pathname-types

  load-if-source-newer*, *load-if-source-only

  load-source-pathname-types

  load-verbose

Sun Common Lisp User's Guide

The additional function pages for the Sun Common Lisp User's Guide pertain to the following chapters:

- Chapter 7. "The Foreign Function Interface"

  foreign-undefined-symbol-names

  unintern-foreign-symbol

- Chapter 8. "Running UNIX Programs from Lisp"

  cd, pwd

  command-line-argument

  enter-top-level-hook

environment-variable

lisp-image-name

run-program

shell

working-directory

- Chapter 10. "Storage Management in Common Lisp"

  gc-silence

  gc-size

  with-static-area

- Chapter 12. "The Window Tool Kit"

  delete-viewport

- Chapter 13. "The Editor"

  Edit Definition (*meta-.*)

  Arglist (Ctrl-C Ctrl-A)

New Chapters

- Chapter 9. "Compiling Lisp Programs"

  A new version of this chapter provides expanded information about compilation strategy, using declarations, and compiling fast floating-point operations. It also includes the following new constructs:

  def-compiler-macro

  defsubst

  disable-stack-lists

  enable-stack-lists

  uncompile

  unproclaim

  undef-compiler-macro

- Chapter 14. “Miscellaneous Programming Features”

  This new chapter discusses the Source File Recording Facility and the Advice Facility. It contains the following new constructs:

  advice-continue, apply-advice-continue

  defadvice

  describe-advice

  discard-source-file-info

  get-source-file

  load-instance

  record-source-file

  record-source-files

  remove-advice

  source-pathname

  ***terse-redefinitions**

destructuring-bind

Purpose: The macro **destructuring-bind** extracts components from, or destructures, a list.

Syntax: **destructuring-bind** *pattern form-to-destructure* {*form*}* [*Macro*]

Remarks: The *pattern* argument is equivalent to a lambda list of a **defmacro** form.

The symbols specified in the *pattern* argument name variables that are bound in the body of the macro to corresponding pieces of *form-to-destructure.* The body of the macro expansion function is specified by the *form* arguments. They are executed in order.

The value of the last form executed is returned as the result of executing the macro.

The destructuring mechanism provided by **destructuring-bind** is similar to the destructuring mechanism provided by **defmacro**; however, **destructuring-bind** can be used in contexts other than macro expansion. The keyword **&whole** is not permitted with **destructuring-bind**.

The macro **destructuring-bind** is an extension to Common Lisp.

Examples:

```
> (destructuring-bind (&key a b c) '(:b 2 :c 3 :a 1)
    (list a b c))
(1 2 3)
> (destructuring-bind ((a . b) c) (list (cons 'eh 'bee) 'sea)
    (list a b c))
(EH BEE SEA)
```

See Also: **defmacro**

close-all-files

Purpose: The function **close-all-files** closes all open files. This function can be used to close files for which the associated stream object is no longer accessible.

Syntax: **close-all-files** [*Function*]

Remarks: The function **close-all-files** is an extension to Common Lisp.

See Also: **close**

open

ignore-binary-dependencies

Purpose: The variable ***ignore-binary-dependencies*** provides a default value for the **:ignore-binary-dependencies** keyword argument to the function **load**. The initial value is **nil**.

Syntax: ***ignore-binary-dependencies*** [*Variable*]

Remarks: The variable ***ignore-binary-dependencies*** is an extension to Common Lisp.

See Also: **load**

load

Purpose: The function **load** reads the file specified by the *filename* argument and evaluates each form in that file. A non-**nil** value is returned if the operation is successful.

Syntax: **load** *filename* &key `:verbose :print :if-does-not-exist` [*Function*]
`:if-source-only :if-source-newer`
`:ignore-binary-dependencies`

Remarks: The *filename* argument is a pathname, stream, string, or symbol.

If the *filename* argument specifies a stream, **load** determines the type of the stream and loads either source code or binary data directly from the stream.

If the *filename* argument is a pathname that is not fully specified, default values are taken from the value of the variable __*default-pathname-defaults*__ by using the function **merge-pathnames**.

If the *filename* argument specifies a pathname that has an explicit type component, **load** searches for a file that matches the given pathname. If the specified file exists and the pathname type is a member of the list given by the variable __*load-binary-pathname-types*__, the file is loaded as a binary file. If the specified file exists and the pathname type is not a member of the list given by the variable __*load-binary-pathname-types*__, the file is loaded as a source file. If the specified file does not exist, by default **load** signals a continuable error and prompts for a new filename.

If the *filename* argument specifies a pathname that does not have an explicit type component, **load** searches for both source files and binary files. The function searches for source files by merging successive elements of __*load-source-pathname-types*__ with the given pathname until it finds a match. The function then searches for binary files by merging successive elements of __*load-binary-pathname-types*__ with the given pathname until it finds a match. The file that is loaded is determined as follows:

- If neither a source nor a binary file exist, **load** signals a continuable error and prompts for a new filename by default.
- If only a binary file exists, it is loaded.
- If only a source file exists, the value of the **:if-source-only** keyword argument determines whether the source file is loaded or compiled.
- If both a source file and a binary file exist and the binary file is newer than the source file, the binary file is loaded.

- If both a source file and a binary file exist and the source file is newer than the binary file, the value of the keyword **:if-source-newer** determines which file is loaded.

The following keyword arguments control details of loading a file:

- If the **:verbose** argument is non-**nil**, **load** prints information about its progress on the standard output. The default value of **:verbose** is the value of the variable ***load-verbose***. Its initial value is **t**.
- If the **:print** argument is non-**nil**, **load** prints the value of each expression that is loaded on the standard output. The default value of this argument is **nil**.
- The **:if-does-not-exist** argument controls what happens if the specified file does not exist. The function **load** calls the function **open** with the **:if-does-not-exist** argument bound to this value. This value can be either **:error** (signal an error) or **nil** (return **nil** from the function **load**). It defaults to **:error**.
- The **:if-source-only** argument controls what happens when only a source file exists. Its value is one of the following keywords:
  - If the value is **:load-source**, the source file is loaded.
  - If the value is **:query** and the function **compile-file** is defined, **load** asks whether to load the source file or compile the source file and load the resulting binary file. If **compile-file** is not defined, the source file is loaded.
  - If the value is **:compile**, the source file is compiled and the resulting binary file is loaded.

  The default value of **:if-source-only** is the value of the variable ***load-if-source-only***. Its initial value is **:load-source**.
- The **:if-source-newer** argument controls what happens when both a source file and a binary file exist for the specified *filename* argument and the source is newer than the binary. Its value can be one of the following:
  - If the value is **:query** and the function **compile-file** is defined, **load** asks whether to load the source file, load the binary file, or compile the source file and load the resulting binary file. If the function **compile-file** is not defined, **load** asks whether to load the source file or the binary file.
  - If the value is **:load-source**, the source file is loaded.
  - If the value is **:load-binary**, the binary file is loaded.
  - If the value is **:compile** and the function **compile-file** is defined, the source file is compiled and the resulting binary file is loaded. If **compile-file** is not defined, **load** asks whether to load the source file or the binary file.

The default value of **:if-source-newer** is the value of the variable ***load-if-source-newer***. Its initial value is **:query**.

- The **:ignore-binary-dependencies** argument controls what happens when a specified binary file that has been compiled for a particular run-time feature is loaded on a machine that does not have the correct architecture. Its value is one of the following:
  - If the value is **t**, the file is loaded.
  - If the value is **nil**, a continuable error is signaled.
  - If the value is **:warn**, a warning is issued and the file is loaded.

  The default value of **:ignore-binary-dependencies** is the value of the variable ***ignore-binary-dependencies***. Its initial value is **nil**.

The standard output is defined by the value of the variable ***standard-output***.

The keyword arguments **:if-source-only**, **:if-source-newer**, and **:ignore-binary-dependencies** are extensions to Common Lisp.

Examples:

```
;;; assuming the file /test/load-test-file.lisp contains
;;;
;;; 1
;;; (setq a 888)
;;;
;;; then...

> (load "/test/load-test-file")
#P"/test/load-test-file.lisp"
> a
888
> (load (setq p (merge-pathnames "/test/load-test-file")) :verbose t)
;;; Loading source file "/test/load-test-file.lisp"
#P"/test/load-test-file.lisp"
> (load p :print t)
1
888
#P"/test/load-test-file.lisp"

;;; Assume the current directory contains "tourist.lisp" and
;;; "tourist.lbin" and that "tourist.lisp" is newer than
;;; "tourist.lbin".

> *load-if-source-newer* ; the default value is :QUERY
:QUERY
> (load "tourist")
Source file tourist.lisp is newer than binary file tourist.lbin.
Load Source, Binary, or Compiled source (S, B or C): s
```

```
;;; Loading source file "tourist.lisp"
#P"/u/r/chkout/test/tourist.lisp"
> (load "tourist")
Source file tourist.lisp is newer than binary file tourist.lbin.
Load Source, Binary, or Compiled source (S, B or C): b
;;; Loading binary file "tourist.lbin"
#P"/u/r/chkout/test/tourist.lbin"
> (load "tourist")
Source file tourist.lisp is newer than binary file tourist.lbin.
Load Source, Binary, or Compiled source (S, B or C): c
;;; Reading input file #P"/u/r/chkout/test/tourist.lisp"
;;; Compiling toplevel form...assembling...emitting...done.
;;; Compiling function MAKE-TOURIST...assembling...emitting...done.
;;; Compiling toplevel form...assembling...emitting...done.
;;; Wrote output file #P"/u/r/chkout/test/tourist.lbin"
;;; Loading binary file "tourist.lbin"
#P"/u/r/chkout/test/tourist.lbin"
```

See Also: error

ignore-binary-dependencies

load-if-source-newer

load-if-source-only

load-binary-pathname-types

load-source-pathname-types

load-verbose

merge-pathnames

redefinition-action

load-binary-pathname-types

Purpose: The variable ***load-binary-pathname-types*** determines which pathnames are considered by **load** to denote binary files. Its value is a list of pathname types.

Syntax: ***load-binary-pathname-types*** [*Variable*]

Remarks: The value of ***load-binary-pathname-types*** must be a list in which each element is either a string or **nil**. The initial value is `("lbin")`.

The variable ***load-binary-pathname-types*** is an extension to Common Lisp.

See Also: **load**

load-source-pathname-types

pathname-type

load-if-source-newer, *load-if-source-only*

Purpose: These variables provide default values for keyword arguments to the function **load**.

The variable ***load-if-source-newer*** provides a default value for the **:if-source-newer** keyword argument. The initial value is **:query**.

The variable ***load-if-source-only*** provides a default value for the **:if-source-only** keyword argument. The initial value is **:load-source**.

Syntax: ***load-if-source-newer*** [*Variable*]

load-if-source-only [*Variable*]

Remarks: The variables ***load-if-source-newer*** and ***load-if-source-only*** are extensions to Common Lisp.

See Also: **load**

load-source-pathname-types

Purpose: The variable *load-source-pathname-types* determines which pathnames are considered by **load** to denote source files. Its value is a list of pathname types.

Syntax: *load-source-pathname-types* [*Variable*]

Remarks: The value of *load-source-pathname-types* must be a list in which each element is either a string or **nil**. The initial value is `(nil "lisp")`.

The function **load** can only consider a pathname without an explicit extension as a source file if the variable *load-source-pathname-types* contains the element **nil**.

The variable *load-source-pathname-types* is an extension to Common Lisp.

Examples:

```
;;; Assume the current directory contains the source files "dragon"
;;; and "dragon.lisp".

> *load-source-pathname-types*
(NIL "lisp")
> (load "dragon")
;;; Loading source file "dragon"
#P"/u/r/chkout/test/dragon"
> (setq *load-source-pathname-types* '("lisp" NIL))
("lisp" NIL)
> (load "dragon")
;;; Loading source file "dragon.lisp"
#P"/u/r/chkout/test/dragon.lisp"
```

See Also: **load**

load-binary-pathname-types

pathname-type

load-verbose

Purpose: The variable ***load-verbose*** provides a default value for the **:verbose** keyword argument to the function **load**. The initial value is **t**.

Syntax: ***load-verbose*** [*Variable*]

See Also: **load**

foreign-undefined-symbol-names

Purpose: The function **foreign-undefined-symbol-names** returns a list of the names of foreign symbols that are not currently associated with any foreign code or data.

Syntax: **foreign-undefined-symbol-names** [*Function*]

Remarks: The names are returned as Lisp strings.

This function is an extension to Common Lisp.

Examples:

```
> (define-fortran-function i-am-undefined())
I-AM-UNDEFINED
> (foreign-undefined-symbol-names)
("_i_am_undefined_")
```

See Also: **unintern-foreign-symbol**

unintern-foreign-symbol

Purpose: The function **unintern-foreign-symbol** makes a foreign symbol inaccessible for general use. You can use the name of the uninterned symbol in newly loaded foreign object code without changing previously loaded code.

Syntax: **unintern-foreign-symbol** *symbol-name* [*Function*]

Remarks: The *symbol-name* argument is a Lisp string that names the foreign symbol as it appears in a foreign object file as follows:

- `"_sym"` for C symbols named `sym`
- `"_sym_"` for FORTRAN symbols named `sym`

If you have loaded a file into a Lisp image, you can edit, recompile, and reload a new version of the file into the same image. You can then successfully invoke **define-c-function**, **define-fortran-function**, or **define-foreign-symbol** with names of symbols that you have uninterned with this function.

This function is an extension to Common Lisp.

Examples:

```
; This example makes a foreign function unbound and unintterns the
; respective foreign symbol.

> (define-c-function atoi ((x :string)) :result-type :integer)
ATOI
> (load-foreign-libraries nil '("-lc"))
T
> (atoi "231")
231
> (fmakunbound 'atoi)
ATOI
> (unintern-foreign-symbol "_atoi")
T
```

See Also: **foreign-undefined-symbol-names**

cd, pwd

Purpose:

These functions allow you to move around in the file system without exiting the Lisp environment.

The function **cd** sets the current working directory. If a pathname argument is provided, the current working directory is set to it. If no argument is provided, the current working directory is set to your home directory.

The function **pwd** returns a pathname for the current working directory.

Syntax:

cd `&optional` *directory-pathname* [*Function*]

pwd [*Function*]

Remarks:

The *directory-pathname* argument is either a simple string or a pathname.

If **cd** is used to set the current working directory, the variable *default-pathname-defaults* is set to the new working directory.

Note: Invoking the function **cd** to set the current working directory from the top level of Lisp is preferable to setting the value of the variable *default-pathname-defaults* explicitly; **cd** keeps the operating system informed about which working directory is current. The function **working-directory** is the preferable method for changing working directories within programs.

These functions are extensions to Common Lisp.

Examples:

```
> (pwd)
#P"/lucid6/win/"
> (cd)
#P"/u/win/"
> (cd "/lucid/bin/")
#P"/lucid/bin/"
> (pwd)
#P"/lucid/bin/"
```

See Also: **working-directory**

command-line-argument

Purpose: The function **command-line-argument** returns a string representing a command line argument that was provided when Lisp was invoked. This function returns **nil** if the command line argument does not exist.

Syntax: **command-line-argument** *argument-number* [*Function*]

Remarks: The *argument-number* argument must be a fixnum.

This function is an extension to Common Lisp.

Examples:

```
> (command-line-argument 0)
"lisp"
> (command-line-argument 1)
NIL
```

enter-top-level-hook

Purpose: The variable ***enter-top-level-hook*** controls the behavior of Lisp when an image is started.

If the variable is bound to a compiled function or to a symbol that names a compiled function, that function is called once before Lisp enters the top level.

If the variable is not bound to a function, Lisp searches your home directory for a `"lisp-init"` file to load. The file that is loaded is determined as if by the following function call:

```
> (load "lisp-init" :if-does-not-exist nil)
```

Syntax: ***enter-top-level-hook*** [*Variable*]

Remarks: The variable ***enter-top-level-hook*** must be bound to a compiled function. Binding the variable to a compiled function requires you to check the function for errors before it is bound.

By default, the variable ***enter-top-level-hook*** is unbound.

This function is an extension to Common Lisp.

Examples:

```
;;; This example assumes you are working in a Lisp image named
;;; /scratch/mylisp.  After *enter-top-level-hook* is bound, the image
;;; is saved on disk.  The next time /scratch/mylisp is started up, the
;;; the appropriate message appears.

> (defun ignore-lisp-init-file ()
   (format  t "Ignoring lisp init-file name~%"))
IGNORE-LISP-INIT-FILE

> (compile 'IGNORE-LISP-INIT-FILE)
;;; Compiling function IGNORE-LISP-INIT-FILE...tail merging...assembling
...emitting...done.
IGNORE-LISP-INIT-FILE

> (setq *enter-top-level-hook* #'IGNORE-LISP-INIT-FILE)
#<Compiled-Function IGNORE-LISP-INIT-FILE 522507>

> (disksave "/scratch/mylisp")
#P"/scratch/mylisp"

> (quit)

% /scratch/mylisp
```

```
; After executing its initialization routines, Lisp responds with this
; message.

Ignoring lisp init-file name
>
```

environment-variable

Purpose: The function **environment-variable** returns the string value of a specified environment variable. The function returns **nil** if the variable does not exist.

Syntax: **environment-variable** *variable-name* [*Function*]

Remarks: The *variable-name* argument must be a string.

You can use the **setf** macro with **environment-variable** to assign a value to a specified environment variable. If the variable already exists, its value is reassigned. If the variable does not exist, a new environment variable is created. If the value of the variable is set to **nil**, the variable is deleted.

Environment variable modifications are visible only to Lisp and to programs invoked from Lisp by the functions **shell** or **run-program**; the programs that are invoked inherit the modifications. Environment variables that have been changed or added revert to their original values when Lisp exits.

This function is an extension to Common Lisp.

Examples:

```
> (environment-variable "SHELL")
"/bin/csh"
> (setf (environment-variable "SHELL") nil)
NIL
> (environment-variable "SHELL")
NIL
> (setf (environment-variable "SHELL") "/bin/sh")
"/bin/sh"
> (environment-variable "SHELL")
"/bin/sh"
```

See Also: **shell**

run-program

lisp-image-name

Purpose: The function **lisp-image-name** returns the pathname of the object file that started the current invocation of Lisp. If the object file cannot be determined, the function returns **nil**.

Syntax: **lisp-image-name** *[Function]*

Remarks: This function is an extension to Common Lisp.

Examples:

```
> (lisp-image-name)
#P"/usr/local/bin/lisp"
```

run-program

Purpose: The function **run-program** provides the ability to run other UNIX programs from the Lisp environment.

Syntax: **run-program** *name* &key `:input :output :error-output` *[Function]*

```
:wait :arguments
:if-input-does-not-exist
:if-output-exists
:if-error-output-exists
```

Remarks: The *name* argument is a pathname or an object that can be coerced to a pathname. It represents the name of the program to be run. If the pathname is a relative pathname, each directory in the environment variable **PATH** is searched for the filename that corresponds to *name*. If the pathname is an absolute pathname, that is, if the name begins with a slash (/), then that file is used. The namestring of *name* is the **argv**[0] parameter for the program.

Four values are returned by **run-program**.

1. The first value is a stream. If either the **:input** or the **:output** keyword argument is **:stream**, that stream communicates with the running process and is the first value returned. If neither keyword is **:stream**, the first value is **nil**.
2. If the **:error-output** keyword argument is **:stream**, the second value returned is the resulting input stream from which Lisp can read the program's error output. If **:error-output** is not **:stream**, the second value returned is **nil**.
3. If the **:wait** keyword argument is **t**, the third value is the exit status of the program that was run. Otherwise the exit status is **nil**.
4. If the program is running, the fourth value is its UNIX process id. If the program has run to completion, the fourth value is **nil**.

The function **run-program** replaces the function **run-unix-program**. Though **run-unix-program** is available for backward compatibility, you should use **run-program** in most instances.

This function is an extension to Common Lisp.

Examples:

```
;; This example shows how to run a program and have Lisp wait for it to
;; complete.  The example assumes you have a program called "banner"
;; on /usr/games.

> (run-program "csh")
% /usr/games/banner test

  #####  ######   ####    #####
    #    #       #          #
    #    #####    ####      #
    #    #            #     #
    #    #       #    #     #
    #    ######   ####      #

% exit
% NIL
NIL
0
NIL
```

shell

Purpose: The function **shell** invokes a Shell. If **shell** is invoked from a Lisp that is running inside a Sunview window, it opens a Shelltool window.

Syntax: **shell** `&optional` *command-line* [*Function*]

Remarks: The optional *command-line* argument is a string to be interpreted by the Shell as a command line. If the argument is supplied, the function **shell** invokes an inferior Shell that executes the command line and returns. If the argument is not specified or is **nil**, the function **shell** invokes an inferior Shell.

The Shell to be invoked is determined by the value of the **SHELL** environment variable.

The function **shell** returns a fixnum status code.

This function is an extension to Common Lisp.

Examples:

```
> (shell "ls /usr/games")
adventure       boggle          chess           gammontool      random
arithmetic      boggledict      chesstool       hack            teachgammon
backgammon      boggletool      cribbage        hangman         wump
banner          btlgammon       factor          lib
bcd             canfield        fish            life
bj              canfieldtool    fortune         number
bogdict         cfscores        gammonscore     primes
0
> (shell)
% ls /usr/games
adventure       boggle          chess           gammontool      random
arithmetic      boggledict      chesstool       hack            teachgammon
backgammon      boggletool      cribbage        hangman         wump
banner          btlgammon       factor          lib
bcd             canfield        fish            life
bj              canfieldtool    fortune         number
bogdict         cfscores        gammonscore     primes
```

working-directory

Purpose: The function **working-directory** returns the current working directory.

Syntax: **working-directory** *[Function]*

Remarks: The variable ***default-pathname-defaults*** is set to the new working directory.

You can use the **setf** macro with **working-directory** to set the current working directory; this method of setting the working directory is preferable to explicitly setting the variable ***default-pathname-defaults*** because it keeps the operating system and the variable synchronized.

This function is an extension to Common Lisp.

Examples:

```
> (working-directory)
#P"/lucid/bin/"
> (setf (working-directory) "/u/win/")
#P"/u/win/"
> (working-directory)
#P"/u/win/"
> (setf (working-directory) "/scratch/")
#P"/scratch/"
> (working-directory)
#P"/scratch/"
> (setf (working-directory) (user-homedir-pathname))
#P"/u/win/"
> (working-directory)
#P"/u/win/"
> (pwd)
#P"/u/win/"
```

See Also: **cd**

pwd

gc-silence

Purpose: The variable ***gc-silence*** controls garbage collection messages.

Syntax: ***gc-silence*** [*Variable*]

Remarks: The variable ***gc-silence*** can have one of the following values:

- **nil**

  If the value of ***gc-silence*** is **nil**, garbage collection messages are displayed on the standard output. This value is the default.

- **t**

  If the value of ***gc-silence*** is **t**, all garbage collection messages are suppressed.

- a compiled function

  If the value of ***gc-silence*** is a compiled function, that function is called with a single argument whenever a garbage collection message might normally appear. The argument can be bound to any of the following values:

  - **:before**

    When the function is invoked immediately before a garbage collection is performed, the argument has the value **:before**.

  - **:after**

    When the function is invoked immediately after a garbage collection is performed, the argument has the value **:after**.

  - **:dynamic-expansion**

    When the function is invoked immediately after the dynamic areas are expanded, the argument has the value **:dynamic-expansion**.

  - **:reserved-expansion**

    When the function is invoked immediately before the reserved free space is expanded, the argument has the value **:reserved-expansion**.

Because ***gc-silence*** procedures are called when normal memory allocation is impossible, an executing function that is bound to ***gc-silence*** should not use more than 1000 bytes of dynamic storage. Any objects created by a function that is bound to ***gc-silence*** exist only while the function executes; it is an error to refer to these objects after the function exits.

This variable is an extension to Common Lisp.

Examples:

```
> (room)
;;; 46872 words [187488 bytes] of dynamic storage in use.
;;; 624620 words [2498480 bytes] of free storage available before a GC.
;;; 1296112 words [5184448 bytes] of free storage available if GC is disabled.
NIL
> (gc)
;;; GC: 4908 words [19632 bytes] of dynamic storage in use.
;;; 666584 words [2666336 bytes] of free storage available before a GC.
;;; 1338076 words [5352304 bytes] of free storage available if GC is disabled.
19632
2666336
5352304
> (let ((*gc-silence* t)) (gc))
20000
2665968
5351936
> (room)
;;; 5014 words [20056 bytes] of dynamic storage in use.
;;; 666478 words [2665912 bytes] of free storage available before a GC.
;;; 1337970 words [5351880 bytes] of free storage available if GC is disabled.
NIL
> (defun gc-silence-function (when)
   (case when
       (:before  (format t "Before GC ... "))
       (:after   (format t "After  GC ... "))
       (:dynamic-expansion  (format t "Expanding Dynamic ..."))
       (:reserved-expansion (format t "Expanding Reserved ..."))))
GC-SILENCE-FUNCTION
> (compile 'GC-SILENCE-FUNCTION)
;;; Compiling function GC-SILENCE-FUNCTION...tail merging...assembling...
emitting...done.
GC-SILENCE-FUNCTION
> (setf *gc-silence* #'gc-silence-function)
#<Compiled-Function GC-SILENCE-FUNCTION 6188E7>
> (gc)
Before GC ... After  GC ...
35120
1143520
2322160
```

gc-size

Purpose: The function **gc-size** returns the following multiple values:

- the number of bytes of storage already used in the current dynamic semi-space
- the number of bytes still available in the current dynamic semi-space
- the number of bytes available in the combined dynamic semi-spaces

Syntax: **gc-size** [*Function*]

Remarks: The values returned by this function are the same as those returned by the function **gc**.

This function is an extension to Common Lisp.

Examples:

```
> (gc-size)
309816
999896
2309608
> (defmacro how-much-consing-macro (&body body)
   '(let ((initial-gc-size (gc-size)))
      (unwind-protect (progn ,@body)    ; multiple-value-prog1
        (format t "~&~S words consed"   ; that doesn't cons.
               (floor (- (gc-size) initial-gc-size)
                     4)))))             ; This machine has 4 bytes
HOW-MUCH-CONSING-MACRO                  ; per word.
> (defun how-much-consing (fn &rest args)
  (how-much-consing-macro
   (apply fn args)))
HOW-MUCH-CONSING
> (compile 'how-much-consing)
;;; Compiling function HOW-MUCH-CONSING...assembling...emitting...done.
HOW-MUCH-CONSING
> (defun test (x y)
  (cons x y))
TEST
> (how-much-consing 'test 'a 'b)
92 words consed
(A . B)
> (compile 'test)
;;; Compiling function TEST...assembling...emitting...done.
TEST
> (how-much-consing 'test 'a 'b)
2 words consed
(A . B)
```

See Also: **gc**

room

with-static-area

Purpose: The macro **with-static-area** executes a sequence of forms while forcing all consing to be done in the static area. Objects that are created are permanent and are never garbage collected.

Syntax: **with-static-area** {*form*}* [*Macro*]

Remarks: The macro **with-static-area** should be used with short sequences of forms to prevent the unintentional creation of objects in static space.

Note: This macro should be used sparingly; for example, it should never be executed in interpreted code because objects created as the result of interpretation would be created in the static area.

This macro is an extension to Common Lisp.

Examples:

```
;;; This example assumes you have a large defstruct called large-object
;;; that creates large-object structures that are retained
;;; indefinitely.  Creating large-object structures in the static area will
;;; therefore speed garbage collection.  The following code provides a
;;; wrapper to the defstruct constructor for large-object so that
;;; large-object structures are created in the static area:

> (defstruct (large-object (:constructor new-large-object))
    field-1
    field-2
    field-3
    field-4
    field-n)
   LARGE-OBJECT
> (compile (defun make-large-object (&rest arguments)
             (with-static-area
              (apply #'new-large-object arguments))))
;;; Compiling function MAKE-LARGE-OBJECT...assembling...emitting...done.
MAKE-LARGE-OBJECT
> (make-large-object :field-4 35)
#S(LARGE-OBJECT FIELD-1 NIL FIELD-2 NIL FIELD-3 NIL FIELD-4 35 FIELD-N NIL)
```

delete-viewport

Purpose: The function **delete-viewport** deletes a viewport or window and removes it from the viewport hierarchy. The viewport's resources can then be garbage collected if no user-defined data structures refer to the viewport.

Syntax: **delete-viewport** [*Function*]

Remarks: This function is an extension to Common Lisp.

Editor Commands

Edit Definition *meta-.* [*Command*]

This command prompts for a Lisp symbol that names a function or a macro and attempts to find the source code that defines the symbol. If it locates the source file in which the symbol is defined, it reads the file into a buffer and searches for the start of the definition. If the start of the definition is located, the point is positioned there. If the function was not defined with **defun** or **defmacro**, or if the text of the definition is unusual, the command may have difficulty locating the appropriate starting point in the file; in this case, a warning appears in the echo area.

Arglist **Ctrl-CCtrl-A** [*Command*]

This command displays in the echo area the argument list of the ***current function***, where the current function is the function to which you are currently typing arguments. If the Editor cannot determine which function is current or if the current function is not defined, an error is signaled.

Chapter 9. Compiling Lisp Programs

Chapter 9. Compiling Lisp Programs

Introduction to the Compiler 9–3
  Compiling Functions and Files 9–3
  Compilation Strategy 9–4
Using Declarations 9–6
  Special Declarations 9–6
  Type Declarations 9–7
  Ftype Declarations 9–8
  Restrictive-Ftype Declarations 9–9
  Type-Reduce Declarations 9–9
  Using Type Declarations 9–10
  Inline and Notinline Declarations 9–15
  Ignore Declarations 9–17
  Optimization Declarations 9–17
  Arglist Declarations 9–21
  Requesting Stack Allocation of &rest Arguments 9–21
Compiling Fast Floating-Point Operations 9–24
Code Optimizations 9–26
  Function Call Optimizations 9–26
  Macro Expansion 9–27
  Constant Folding 9–27
  Case Macro Optimization 9–29
clear-undef 9–30
compile 9–31
compile-file 9–32
compiled-function-p 9–34
compiler-options 9–35
declare 9–37
def-compiler-macro 9–39
defsubst 9–40
disable-stack-lists 9–42
enable-stack-lists 9–43
eval-when 9–44
locally 9–45
proclaim 9–46
the 9–47
uncompile 9–48
undef-compiler-macro 9–49
unproclaim 9–50

Introduction to the Compiler

The ***Compiler*** reads Lisp expressions and generates native machine code that runs faster than the interpretation of the original source code. Compiled code generally produces the same values and has the same side effects as the interpretation of the original source code. Compiled code, however, may behave obscurely when errors occur and may require recompilation if macros or constants are redefined.

The more information you give the Compiler, the more efficient your compiled code will be. You can give the Compiler information by adding declarations and definitions to your source code and by using code that the Compiler can optimize more efficiently. This chapter explains how to compile functions and files, describes how to increase the efficiency of source code by adding declarations, and discusses various optimizations performed by the Compiler.

The constructs listed below are used when compiling code. See the function pages at the end of this chapter for complete syntactic descriptions of these functions and forms.

clear-undef
compile
compile-file
compiled-function-p
compiler-options
declare
def-compiler-macro
defsubst
disable-stack-lists
enable-stack-lists
eval-when
locally
proclaim
the
uncompile
undef-compiler-macro
unproclaim

Compiling Functions and Files

Compiling can be done in two ways: by compiling an individual function in the current environment or by compiling a file of source code.

- Compiling a function

  To compile an interpreted function in the current Lisp environment, use the function **compile**, which replaces the interpreted function's definition with the compiled version. Evaluating the expression `(compile 'foo)` compiles the function `foo`. Compiled code and interpreted code can be used interchangeably; that is, a compiled function can call an interpreted function, and an interpreted function can call a compiled function.

- Compiling a file

  To compile a file, use the function **compile-file**, which takes a Lisp source file and produces a binary file containing the compiled code. By convention, Lisp source files usually have the extension `.lisp` and binary files have the extension `.lbin`. Evaluating the expression `(compile-file "foo.lisp")` produces the file `"foo.lbin"`. To use the compiled functions, you must load the binary file into Lisp by using the **load** function.

Compilation Strategy

Compiled code is generally faster than its interpreted counterpart, but it may also be harder to modify and debug than interpreted code. Compiled code generally does less error checking and retains less environment information than interpreted code. As a result, where interpreted code would signal an error, compiled code may simply behave unexpectedly. Compiled code is also less sensitive to redefinitions; in particular, changes to macro definitions that occur after compilation are not reflected in the behavior of the compiled code.

In general, as you increase the amount of optimization performed by the Compiler, you increase the differences between the compiled code and its original source code and consequently decrease the amount of available debugging information. Sun Common Lisp allows you to incrementally compile and optimize your code by adding declarations and other sources of compilation information to your source code.

Generally, you should begin with interpreted code because it is easier to debug. You can then selectively compile portions of your code and gradually increase the amount of optimization that the Compiler performs. Finally, when your code has been tested, you can fully optimize your code to produce high-performance production code.

You can increase the running speed of your code in a number of ways:

- Specify the data types of the arguments and returned values of Lisp expressions (see the section "Using Type Declarations").
- Compile code to use hardware-supported floating-point operations (see the section "Compiling Fast Floating-Point Operations").
- Use fixnum arithmetic when possible by declaring both the arguments and the values of a Lisp expression to be of type fixnum (see the section "Increasing the Efficiency of Arithmetic Operations").
- Use simple arrays and simple vectors (see the section "Increasing the Efficiency of Array Access").

- Code simple functions in-line (see the section "Inline and Notinline Declarations").
- Use optimization declarations to emphasize speed (see the section "Optimization Declarations").
- Use code that the Compiler can easily optimize, such as macros and tail-recursive functions (see the section "Code Optimizations").

Using Declarations

A ***declaration*** is a statement that supplies information about a Lisp program to the Lisp environment. With the exception of the **special** declaration, declarations are optional and are ignored by the Interpreter. They can be used as advice to the Compiler, however, to produce faster and more efficient code. This applies in particular to **type** declarations.

You can make declarations with either the **declare** special form or the **proclaim** function. You can use the **declare** special form to make local declarations within other Common Lisp forms. Local declarations observe the rules of lexical scope. The syntax for a local declaration is the following:

```
(declare declaration-form ...)
```

You can use the **proclaim** function to make global declarations, which are also called ***proclamations***. A global declaration may be overridden by a local declaration. Note that the **proclaim** function evaluates its argument, while the **declare** special form does not. The syntax for a global declaration is the following:

```
(proclaim declaration-form)
```

Sun Common Lisp provides the following categories of declarations:

special
type
ftype
restrictive-ftype
type-reduce
inline
notinline
ignore
optimization
arglist
dynamic-extent

Special Declarations

A **special** declaration specifies that a given variable will be dynamically scoped rather than lexically scoped. References to the variable will thus refer to its dynamic binding. To make a local declaration, which observes the rules of lexical scope, use the **declare** special form. To make a global declaration, which affects all dynamic bindings of the variable, use the **proclaim** function. Proclaiming a variable as a certain type has no effect on the lexical bindings of that variable.

The effect of a **special** declaration is exactly the same in both the Compiler and the Interpreter. However, compiled access to special variables occurs through in-line coding of the function **symbol-value**. If the special variable is unbound, obscure errors may result. To check special variables, you can either declare the **symbol-value** function as **notinline** or compile code with the **safety** optimization level set to 3 (see the section "Optimization Declarations").

Type Declarations

Type declarations specify the data types of the values of Lisp expressions. A **type** declaration allows the Compiler to eliminate type checking. The Compiler extensively uses **type** declarations to produce faster and more efficient code; the Interpreter ignores **type** declarations.

The syntax for a **type** declaration is the following:

(declare (type *type-specifier variable-1 variable-2* ...))

If the type specifier is one of the Common Lisp atomic types, you can use a shorter form of the **type** declaration (see the chapter "Data Types" in the *Sun Common Lisp Reference Manual* for a list of atomic types):

(declare (*type-specifier variable-1 variable-2* ...))

Adding type declarations to arithmetic operations can make the operations faster by significantly reducing the type-checking and type-dispatching overhead of function calls. To obtain the most efficient form of arithmetic operators, you must specify the value types of an expression as well as the argument types of the expression. This is especially true for arithmetic that uses values of type fixnum.

The type ***fixnum*** denotes the range of Common Lisp integers that can be directly represented as machine integers on the underlying hardware. These are the integers in the range **most-negative-fixnum** to **most-positive-fixnum** (see the *Sun Common Lisp Reference Manual*). The Compiler can directly code applications of arithmetic operators for which the arguments and the values are known to be fixnums in the corresponding machine operations; fixnum arithmetic is thus extremely fast. See the section "Using Type Declarations" for a discussion of the most effective ways to use **type** declarations.

Ftype Declarations

An **ftype** declaration specifies the manner in which the value type of a declared function depends on the argument types of the function.. Whenever the arguments to a declared function are of the indicated types, the result of the function will also be of the indicated type. A function may have more than one **ftype** declaration associated with it.

The syntax for an **ftype** declaration is the following:

(declare (ftype *type function-name-1 function-name-2 ...*))

A **function** declaration is an abbreviated form of an **ftype** declaration. Its syntax is the following:

(declare (function *name arglist result-type-1 result-type-2 ...*))

An **ftype** declaration does not require the arguments to an expression to be of a particular type; it merely specifies that the result of the function will be of a certain type if the arguments of the function have been declared as a certain type. For example, the following proclamation declares that if the argument to the function square is a fixnum, the value of the function will also be a fixnum:

```
(proclaim '(ftype (function (fixnum) fixnum) square))

(defun square (x) (* x x))
```

The proclamation has no effect on the following code because the argument x is not declared to be of type fixnum:

```
(defun do-some-arithmetic (x)
       (the fixnum (+ x (square x))))
```

If, however, you add a **type** declaration for x, the Compiler can assume that the expression (square x) is a fixnum, and it will use the fixnum-specific version of the + operator.

```
(defun do-some-arithmetic (x)
  (declare (type fixnum x))
  (the fixnum (+ x (square x))))
```

Restrictive-Ftype Declarations

A **restrictive-ftype** declaration specifies both the argument types and the result types of a series of functions. While an **ftype** declaration conditionally restricts the value type of a function, a **restrictive-ftype** declaration unconditionally restricts both the argument types and the value type of a function.

The **restrictive-ftype** declaration is an extension to Common Lisp. Its syntax is the following:

```
(declare (restrictive-ftype type function-name-1 function-name-2 ...))
```

The following proclamation declares that the function baz will only accept two fixnum arguments and will always return a fixnum value:

```
(proclaim '(restrictive-ftype (function (fixnum fixnum) fixnum) baz))
```

Thus, in the following code, all of the arithmetic operators can be compiled as fixnum operators without requiring any additional declarations:

```
(defun do-some-arithmetic (i j)
  (declare (fixnum i j))
  (baz (+ (baz (* i j) (+ i j)) j) i))
```

Type-Reduce Declarations

A **type-reduce** declaration specifies that within the scope of the declaration all objects that belong to a specified supertype can be treated as belonging to a specified subtype. A program might declare, for example, that all numbers should be treated as fixnums. A **type-reduce** declaration may be simpler to use than adding many **type** declarations to your code.

The **type-reduce** declaration is an extension to Common Lisp. Its syntax is the following:

```
(declare (type-reduce type1 type2))
```

The arguments *type1* and *type2* must be type specifiers; the type *type2* must be a subtype of type *type1*.

For example, if an application uses only fixnum arithmetic, the following proclamation allows the Compiler to compile all arithmetic operators as fixnum operators without requiring any additional declarations:

```
(proclaim '(type-reduce number fixnum))
```

If an application uses both fixnum and nonfixnum arithmetic, but uses only fixnum integer values, the following declaration could be used:

```
(declare (type-reduce integer fixnum))
```

See the section "Using Type Declarations" for more information about using **type-reduce** declarations to restrict arithmetic operators.

Using Type Declarations

Type declarations, **ftype** declarations, **restrictive-ftype** declarations, and **type-reduce** declarations are most effective when used to do the following:

- Increase the efficiency of arithmetic operations
- Increase the efficiency of array and sequence operations
- Propagate type information

Increasing the Efficiency of Arithmetic Operations

To obtain the most efficient form of a Common Lisp operator, you may need to specify both the types of the arguments and the type of the value of an expression. There are four ways of specifying types:

- To declare the type of all occurrences of a variable, use the special form **declare**.

  For example, the following code defines a function that sums a list of numbers:

  ```
  (defun list-add (l)
    (let ((sum 0))
      (dolist (i l sum)
        (setq sum (+ i sum)))))
  ```

  To increase the efficiency of `list-add`, you should declare both the argument types and the result types of the expression. In the following code, the **declare** special form is used to declare the variables `i` and `sum` to be of type fixnum:

  ```
  (defun list-add (l)
    (let ((sum 0))
      (declare (fixnum sum))
      (dolist (i l sum)
        (declare (fixnum i))
        (setq sum (+ i sum)))))
  ```

Note that the value of the expression (+ i sum) is also implicitly declared to be of type fixnum because the value of the expression is assigned to sum, which has been declared to be of type fixnum. The Compiler is thus free to depend on these types and will use the fixnum-specific version of +.

- To specify the type of an individual expression, use the special form **the**.

  The following example shows an alternate form of list-add that uses the special form **the** to declare both the value type of the expression and the types of the list elements as they are retrieved:

```
(defun labels-list-add (l)
  (labels ((help-add (sum restl)  ; Define a recursive, helper function.
                     (declare (fixnum sum))
                     (if (null restl) sum
                         (help-add
                          (the fixnum
                               (+ sum (the fixnum (first restl))))
                          (rest restl)))))
    (help-add 0 l)))
```

  Because both the argument types and the value type have been declared, the Compiler can use a fixnum-specific version of +.

- To propagate or restrict types for all expressions that use specified operators, use **ftype** or **restrictive-ftype** declarations.

  You can use **ftype** declarations to restrict arithmetic operators in certain contexts. In most cases the result of applying arithmetic operators to fixnum values might not be a fixnum unless the fixnum values are small. The expression (1+ most-positive-fixnum) does not produce a fixnum value, for example. If the fixnum values are small enough, however, you can use **ftype** declarations to restrict arithmetic operators. For example, the following declarations inform the Compiler that whenever the arguments to the declared operators are fixnums the value types will also be fixnums. The Compiler can thus use the fixnum forms of the operators:

```
(declare (ftype (function (&rest fixnum) fixnum) + -)
         (ftype (function (fixnum) fixnum) 1+ 1-))
```

- To restrict the types of numerical quantities, use **type-reduce** declarations.

  Often only fixnum arithmetic is needed in a program. The following example shows another form of `list-add` that uses a single **type-reduce** declaration to restrict all numerical quantities within the function:

  ```
  (defun fixnum-list-add (l)
    (declare (type-reduce number fixnum))
    (let ((sum 0))
      (dolist (i l sum)
        (setq sum (+ i sum)))))
  ```

You can use **type** declarations to write fixnum-specific versions for generic routines. You must use caution when defining fixnum-specific operators. Expressions that apply arithmetic operators to more than two arguments may not produce optimal compiled code. For example, the following expression cannot be fully optimized:

```
(the fixnum (+ (the fixnum x) (the fixnum y) (the fixnum z))))
```

Although each of the arguments and the result of the entire expression are declared as fixnums, the Compiler cannot assume that the intermediate results will also be fixnums.

The following macro defines a fixnum-specific addition operator. Note that it automatically expresses the sum in terms of binary operators and adds the necessary declarations:

```
(def-macro fixnum-plus (&rest args)
  (case (length args)
    (0 '0)
    (1 (first args))
    (2 `(the fixnum (+ (the fixnum ,(first args))
                       (the fixnum ,(second args)))))
    (t `(the fixnum
            (+ (the fixnum ,(first args))
               (fixnum-plus ,@(rest args)))))))
```

When this code is compiled, calls to `fixnum-plus` will be coded directly into hardware add instructions.

To increase the efficiency of floating-point arithmetic, you should declare the types of all floating-point variables and compile code to use the floating-point hardware for your system. See the section “Compiling Fast Floating-Point Operations” for more information.

Increasing the Efficiency of Array Access

You can make array and sequence operations faster by adding type declarations. To use the most efficient form of array access, you must do the following three things:

- Declare the array to be a simple array or simple vector.
- Declare the element types of the array.
- Declare the number of dimensions of the array.

If you do not specify all three pieces of information, the Compiler cannot generate the most efficient array access code. Each of the following declarations contains insufficient information:

```
;; The type vector does not imply the type simple-array.
(declare (type (vector (unsigned-byte 8)) v))

;; The element-type is not known.
(declare (type (simple-array * (* *)) a))

;; The number of dimensions is not specified.
(declare (type (simple-array t) a1))
```

Each of the following declarations, however, allows the Compiler to generate the most efficient array access code:

```
;; The type simple-vector is equivalent to the type (simple-array t (*)).
(declare (type simple-vector v))

(declare (type (simple-array t (* *)) a))

(declare (type (simple-array (unsigned-byte 8) (*)) v8)))
```

If all arrays in a piece of code are of a certain type, it may be simpler to use a **type-reduce** declaration to restrict the types of all arrays. If an application uses only simple one-, two-, or three-dimensional arrays of type **t**, for example, the following declaration could be used:

```
(declare (type-reduce (array * (*)) simple-vector)
         (type-reduce (array * (* *)) (simple-array t (* *)))
         (type-reduce (array * (* * *)) (simple-array t (* * *))))
```

Propagating Type Information

Type propagation is the process of passing type information to all values assigned to a declared variable, returned by a special form, or returned by a function that contains type declarations. Type propagation is another means of passing type information to the Compiler, which may allow it to generate more efficient code.

Type information is propagated in the following three ways:

- The Compiler automatically propagates the type of special forms and standard Common Lisp functions to all components of the special form or function that return values.

  The following example defines a function that adds the absolute values of a list of fixnums and produces a fixnum total:

  ```
  (defun add-list (l)
    (let ((sum 0))
      (declare (fixnum sum))
      (dolist (i l sum)
        (declare (fixnum i))
        (setq sum
              (+ sum
                 (the fixnum
                      (if (> i 0) i (- i))))))))
  ```

  In this definition, the type of the variable sum is propagated to the expression whose value is assigned to sum. The explicitly declared type of the second term of sum is propagated to the i and (- i) components of the if expression. The Compiler can thus compile all of the operators as fixnum-specific operators.

- The Compiler automatically propagates type information that is declared in the fields of structures that are defined by using **defstruct**.

  For example, suppose you define a structure bird as follows:

  ```
  (defstruct bird
    (weight 0.0 :type single-float)
    (height 0.0 :type single-float)
    (age 0 :type fixnum))
  ```

Because the **age** field has been declared to be of type fixnum, in the following code the Compiler expects both the argument to 1+ and the returned value of the expression to be fixnums:

```
(setf (bird-age bird1)
      (1+ (bird-age bird1)))
```

The Compiler can thus generate the fixnum-specific form of 1+.

- You can explicitly propagate type information for user-defined functions by using **ftype** or **restrictive-ftype** declarations (see the section "Ftype Declarations").

Inline and Notinline Declarations

An **inline** declaration asks the Compiler to replace the procedure call to a function with the machine-language code for that function. The Compiler may choose to ignore this declaration.

Using in-line code eliminates some of the overhead in calling functions. In-line coding is especially useful for simple functions; it is not as useful for large functions where the cost of calling the function is small compared to the execution time of the function. In-line coding should be used selectively because it may increase the size of compiled code.

To declare local functions that have been defined by using the **flet** or the **labels** special form, use an **inline** declaration, as shown in the following example:

```
;;; Compute the distance between two 2-dimensional points.
(defun distance-2d (p1 p2)
  (flet ((square (x) (* x x)))
    (declare (inline square))
    (flet ((square-d (x1 x2) (square (- x2 x1))))
      (declare (inline square-d))
      (sqrt
       (+ (square-d (point-x p1) (point-x p2))
          (square-d (point-y p1) (point-y p2)))))))
```

To declare top-level functions, use an **inline** proclamation, as shown in the following example:

```
(proclaim '(inline square))

(defun square (x) (* x x))
```

You can also use the function **defsubst** to define a function and declare it **inline**. The following expression produces the same result as the proclamation and definition in the preceding example:

```
(defsubst square (x) (* x x))
```

In addition to avoiding function overhead, declaring a function **inline** allows the Compiler to propagate type information within the body of that function. For example, the following code defines a function, `use-square`, that calls `square`:

```
(defun use-square (z)
  (declare (fixnum z))
  (the fixnum (square z)))
```

Because the variable `z` and the result of `square` are both declared to be of type fixnum, the type information propagates to the call to `*` in the body of `square`. The Compiler can thus use a fixnum-specific form of the `*` operator.

Unless **notinline** declarations are used, the Compiler automatically generates in-line code for many system-provided function calls, such as **car** and **cdr**. The Compiler normally codes the following functions in-line:

Simple-array accessors where the element type and number of dimensions are known at compile-time

Structure accessors

Standard type predicates, such as **symbolp** and **consp**

Symbol accessors, such as **symbol-value** and **symbol-function**

Fixnum arithmetic

Character operations, such as **char** and **string-char**

eq

Although in-line code is usually fast, it may have the following disadvantages:

- It may prevent you from tracing calls to the in-line function during execution.
- It may prevent you from dynamically redefining the in-line function.
- It may increase the code size of the program.

A **notinline** declaration tells the Compiler not to use in-line code for a function, even for a function it would normally code in-line. If a function is declared **notinline**, a call to the function replaces the in-line code. The Compiler must obey a **notinline** declaration.

Ignore Declarations

An **ignore** declaration prevents the Compiler from issuing a warning when the variable specified in the declaration is not referred to in the body of the code. For example, the Compiler will not produce a warning about the variables `lies` and `halftruths` when the following code is compiled:

```
(defun just-the-facts (case)
  (multiple-value-bind
      (lies halftruths facts)
    (case-report case)
    (declare (ignore lies halftruths))
    facts))
```

Optimization Declarations

An **optimize** declaration controls the type and amount of optimization you want the Compiler to perform. You can adjust the following classes of optimizations:

- The speed at which compiled code runs
- The amount of safety (error checking) retained during compilation
- The amount of space the compiled code needs
- The speed at which the code is compiled

You can adjust the amount of optimization by assigning each class an integer value between 0 and 3 inclusive that represents the level of optimization. To assign a value to an optimization class, use an **optimize** declaration, as shown in the following example:

```
(optimize (speed 2) (safety 1))
```

Specifying an optimization class without an integer assigns the highest possible value to the optimization class. The following code assigns **safety** the value 3:

```
(optimize safety)
```

The levels of an optimization class are cumulative; each level includes the effects of the previous levels. The optimization classes and their values are as follows:

- **speed**

  Increasing the level of **speed** increases the speed at which your code will run and decreases the ease with which you can debug the code. This class can have the following values:

  0 This value turns off in-line coding. Function calls can be traced, and redefinitions affect compiled code.

  1 This value turns off variable elimination and other optimizations that affect the evaluation order of expressions. All user-defined variables will appear in stack frames, and function calls are evaluated in their original order.

  2 This value turns off tail merging. Frames for tail calls appear normally on the stack (see the section "Function Call Optimizations" for an explanation of tail calls and tail merging).

  3 This value turns off all restrictions that affect speed. This is the default value.

- **safety**

  Increasing the level of **safety** increases the amount of error checking in compiled code. This class can have the following values:

  0 This value indicates that no safety constraints are imposed on compiled code.

  1 This value indicates that functions with a fixed number of arguments are checked on entry for the correct number of arguments. It is the same as setting **:fast-entry** to **nil**. This is the default value.

  2 This value indicates that write access operations are checked as follows:

  - Accessor functions for data objects that are used as arguments to **setf**, **rplaca**, **rplacd**, or **set** are type checked. The accessor functions are **car**, **cdr**, **symbol-value**, **symbol-function**, **symbol-package**, **symbol-name**, **symbol-plist**, and **macro-function**.
  - Structure accessor functions that are used as arguments to **setf** are checked to verify that their arguments are structures and that the offset is within range.
  - Array reference functions that are used as arguments to **setf** are checked to verify that their arguments are of the correct type, within bounds, and within the offset range. The array reference functions are **svref**, **aref**, **char**, **schar**, **bit**, and **sbit**.

3 This value indicates that read access operations for the access operations described above are type checked. In addition, type declarations are ignored, which means that generic operators are not replaced by specific operators.

- **space**

  Increasing the level of **space** decreases the size of the compiled code. This class can have the following values:

  0 This value imposes no size constraints on the compiled code. This is the default value.

  3 This value turns off in-line coding of safe access functions, which may decrease the size of compiled code.

- **compilation-speed**

  Increasing the level of **compilation-speed** may decrease the amount of time required to compile a file. This class can have the following values:

  0 This value allows the Compiler to use optimizations that may decrease compilation speed. This value is the default.

  3 This value prevents some optimizations that affect compilation speed.

The table in Figure 9-1 summarizes the optimization classes and their default values.

Class	Value	Compiler Action
speed	0	Does no in-line coding.
	1	Does no optimizations that affect evaluation order.
	2	Does no tail merging.
	3	All restrictions are removed; this is the default.
safety	0	Imposes no safety constraints.
	1	Checks arguments on entry to a function with a fixed number of arguments; this is the default.
	2	Checks write access.
	3	Checks read access.
space	0	Imposes no space constraints; this is the default.
	3	Turns off in-line coding of safe accessors.
compilation-speed	0	Imposes no constraint on compilation speed; this is the default.
	3	Prevents optimizations that affect compilation speed.

Figure 9–1. Table of Optimization Classes

To adjust the level of optimization, you can also specify the keyword options **:fast-entry**, **:tail-merge**, and **:notinline** to **compile-file** and **compiler-options**. These keyword options, which are extensions to Common Lisp, may be activated or suppressed when the four optimization classes are used. Thus, using an **optimize** declaration to set the value of safety to 0 implies that the value of the keyword **:fast-entry** is **t**.

The conditions under which the keywords are activated are the following:

:fast-entry	**safety = 0**
:tail-merge	**speed = 3**
:notinline	**speed = 0**

Arglist Declarations

An **arglist** declaration explicitly specifies the lambda list that is returned by a call to the function **arglist**. An **arglist** declaration allows you to specify a descriptive lambda list that would be helpful to a user. Its syntax is the following:

(declare (arglist (*declared-arguments*)))

The declared lambda list should be an unquoted list. If a function or macro definition does not contain an **arglist** declaration, the function **arglist** returns the lambda list used in the original source code, as shown in the following example:

```
> (defun multiply (x y)
    (* x y))
MULTIPLY
> (arglist 'multiply)
(X Y)
> (defun new-multiply (x y)
    (declare (arglist (factor1 factor2)))
    (* x y))
NEW-MULTIPLY
> (arglist 'new-multiply)
(FACTOR1 FACTOR2)
```

Requesting Stack Allocation of &rest Arguments

You can request stack allocation of **&rest** arguments by using **dynamic-extent** declarations. Allocating **&rest** list structures on a stack rather than in dynamic storage reduces the amount of dynamic memory used by a program and thus reduces the number of garbage collections.

The list of argument values bound to an **&rest** argument is normally constructed in dynamic storage. When the function that created the list exits, this dynamic storage is not deallocated until the next garbage collection because the list may be stored in a data structure or bound to a variable that survives the function call.

Often, however, the **&rest** argument has ***dynamic-extent***, which means that the list is not accessible after the function call that created it exits. The storage allocated to it is therefore not needed after the function call exits. In the following code, for example, the **&rest** argument is only needed to access the components of the list during the execution of the function:

```
(defun fixnum-+ (&rest numbers)
    (let ((sum 0))
    (dolist (number numbers)
      (incf sum number))
    sum))
```

A **dynamic-extent** declaration informs the Compiler that an **&rest** argument has dynamic extent and that the storage allocated to it can be released when the function call returns. The Compiler can then generate code that allocates and deallocates the storage for the list on a stack. Lists that are allocated and deallocated on a stack are called ***stack lists***.

To allocate lists on a stack, you must do the following:

- Add **dynamic-extent** declarations for the **&rest** arguments to be allocated on a stack.

  You can declare a stack list **&rest** argument by using the following declaration:

  ```
  (declare (dynamic-extent variable))
  ```

- Make sure stack list allocation is enabled at run-time. Stack list allocation is enabled by default. You can explicitly disable it by calling the function **disable-stack-lists**; you can explicitly enable it by calling the function **enable-stack-lists**.

When stack list allocation is enabled, the following code produces a function that sums its fixnum arguments without using dynamic allocation:

```
(defun fixnum-+ (&rest numbers)
  (declare (dynamic-extent numbers))
  (let ((sum 0))
    (dolist (number numbers)
      (incf sum number))
    sum))
```

Using **dynamic-extent** declarations should only affect the amount of dynamic memory used by your program. If code that contains **dynamic-extent** declarations is behaving incorrectly, the declarations may be incorrect. You can check the use of **dynamic-extent** declarations by disabling stack list allocation with a call to the **disable-stack-lists** function.

For example, the following two functions incorrectly declare their **&rest** arguments to be stack lists; when these functions exit, the **&rest** argument list structure is still accessible through the global variables `*f*` and `*g*`.

```
(compile (defun f (&rest arguments)
           (declare (dynamic-extent arguments)
                    (special *f*))
           (setq *f* arguments)))

(compile (defun g (&rest arguments)
           (declare (dynamic-extent arguments)
                    (special *g*))
           (setq *g* arguments)))
```

When stack list allocation is enabled, the program behaves incorrectly; a call to the function **g** has side effects on the global variable *f*, even though *f* is not explicitly manipulated by function **g**:

```
> (enable-stack-lists)
T
> (f 1 2 3)
(1 2 3)
> *f*
(1 2 3)
> (g 4 5 6 7)
(4 5 6 7)
> *f*
(5 6 7)
> *g*
(4 5 6 7)
```

When stack list allocation is disabled, the program behaves correctly, which indicates that the program probably contains incorrect **dynamic-extent** declarations:

```
> (disable-stack-lists)
T
> (f 1 2 3)
(1 2 3)
> *f*
(1 2 3)
> (g 4 5 6 7)
(4 5 6 7)
> *f*
(1 2 3)
> *g*
(4 5 6 7)
```

Compiling Fast Floating-Point Operations

You can increase the speed of floating-point operations by compiling code so that it explicitly uses the floating-point hardware on the Sun. Calls to hardware-supported floating-point operations are compiled in-line and are thus faster because the function-calling overhead is eliminated. In addition, local variables that are declared as floating-point types are allocated either in registers or in a special block on the stack. These variables are not allocated in dynamic memory and thus never need to be garbage collected.

To get the fastest results from your floating-point hardware, you must do the following:

- Compile code so that it uses the floating-point hardware for the Sun.

  To use the floating-point hardware, call the **compile-file** function with the **:target** keyword set to **68020/68881**. The following expression compiles a file and generates code that uses the the MC68881 coprocessor for floating-point operations:

  ```
  (compile-file "lots-of-floating-point" :target '68020/68881)
  ```

  To use the normal hardware, call the **compile-file** function with the **:target** keyword set to the value **68K**.

  You can set a default value for **:target** by calling the function **compiler-options**. The following expression sets the MC68881 coprocessor as the default target:

  ```
  (compiler-options :target '68020/68881)
  ```

- Declare the explicit types of all floating-point variables.

  To declare floating-point variables, use **type** declarations or the special form **the** to declare the explicit types of all floating-point quantities. This allows the Compiler to allocate variables in registers or on the stack.

  When compiled with the **:target** option to **compile-file** set to the appropriate target, the following example produces extremely efficient code that requires no dynamic storage allocation:

  ```
  (defun 2d-matrix-add (a b c)
    (declare (type (simple-array single-float (* *)) a b c))
    (let ((m (array-dimension a 1))
          (n (array-dimension a 2)))
      (declare (fixnum m n))
      (dotimes (i m)
        (declare (fixnum i))
  ```

```
      (dotimes (j n)
        (declare (fixnum j))
        (setf (aref c i j) (+ (aref a i j) (aref b i j)))))))
```

- Use functions coded in-line for the computationally intensive portions of your code whenever possible. This avoids the allocation of arguments and return values in dynamic storage and allows floating-point local variables to remain in registers.
- Represent aggregates of floating-point data as simple floating-point arrays whenever possible. Compiled code that moves data between simple floating-point arrays and floating-point local variables does not require dynamic allocation.

  For example, the following code defines a data structure **ship**:

  ```
  (defstruct ship
    name
    (x-position 0.0 :type single-float)
    (y-position 0.0 :type single-float))
  ```

  The following definitions provide the same functionality as the above definition and increase the efficiency of **ship** by using floating-point arrays, which allow the Compiler to generate more efficient access code:

  ```
  (defstruct ship
    name (position
          (make-array 2
                      :element-type 'single-float
                      :initial-contents '(0.0 0.0))
          :type (simple-array single-float 2)))
  (defsubst ship-x (s) (aref (ship-position s) 0))
  (defsubst ship-y (s) (aref (ship-position s) 1))
  ```

Code that includes in-line floating-point operations will only run on machines equipped with the appropriate floating-point hardware. The binary files produced by **compile-file** are tagged to indicate the type of floating-point hardware required. Any binary file produced for a specific target is tagged regardless of whether any floating-point code actually appears in the file.

When a binary file is loaded, Lisp determines whether the appropriate floating-point hardware is available on the host machine. If the hardware is not present, the value of the **:ignore-binary-dependencies** keyword option to the **load** function determines whether Lisp will load the file with no warnings, signal a continuable error, or load the file with a warning (see the chapter "File System Interface" in the *Sun Common Lisp Reference Manual*).

Code Optimizations

The Compiler performs a large number of transformations on the source code to improve its performance. While these transformations preserve Common Lisp semantics, they may also cause the execution of the compiled program to differ from that of the interpreted program. This section describes some of the optimizations the Compiler performs so that you can take advantage of them and can understand the resulting differences between the compiled code and the interpreted code.

Function Call Optimizations

Unless directed otherwise, the Compiler optimizes self-recursive function calls, tail calls, and self-tail calls. In particular, self-tail calls are automatically compiled as loops. While these function calls are efficient, they may be difficult to trace because they do not appear on the stack.

- A function call is a ***self-recursive call*** if the function calls itself. For example, the following code contains a recursive function call to `fact`:

  ```
  (defun fact (n)
    (if (< n 1) 1 (* n (fact (1- n)))))
  ```

  Unless compiled as **notinline**, self-recursive function calls jump directly to the start of the code rather than going through the **symbol-function** slot. A trace of a self-recursive function shows only the initial call, not the resulting recursive calls. For example, if you trace `fact`, a call to `(fact 2)` will only show the initial call to `fact`, not the recursive calls to `(fact 1)` and `(fact 0)`.

- A function call is a ***tail call*** if it is the last operation performed by the calling function. In the following code, all of the function calls in the case statement are tail calls:

  ```
  (defun analyze-cereal (cereal)
    (case (cereal-kind cereal)
      (oat (analyze-oat-cereal cereal))
      (wheat (analyze-wheat-cereal cereal))
      (sugar (analyze-sugar-cereal cereal))
      (t (analyze-other-cereal cereal))))
  ```

- A function call is a ***self-tail call*** if it calls itself as the last operation. In the following example, the call to `my-last` is a self-tail call:

  ```
  (defun my-last (x)
   (if (cdr x) (my-last (cdr x)) x))
  ```

> When the Compiler compiles either a tail call or a self-tail call, it reuses the calling function's stack frame rather than creating a new stack frame. This optimization is called ***tail merging***; it is indicated by the message `"...tail merging..."` or `"...self tail merging..."`.

A function defined using self-tail calls is a ***tail-recursive*** function. Because of tail merging, tail-recursive functions are usually as efficient as functions defined using loop constructs.

As a consequence of tail merging, the caller of a tail-called function will not appear in a back trace because the stack frame for the caller has been used by the called procedure. For example, suppose you define a function `analyze-food` that calls the function `analyze-cereal`. A back trace from `analyze-oat-cereal` will show only the calls to `analyze-oat-cereal` and `analyze-food`; the frame for `analyze-cereal` will have been replaced by the frame for `analyze-oat-cereal`.

If the call to `analyze-cereal` is also a tail-recursive call, the trace will not show `analyze-food` either, as that frame has been replaced by the call to `analyze-cereal`.

To trace tail calls, you can turn off tail merging by setting the **:tail-merge** keyword argument to **compile-file** or **compiler-options** to **nil** when you compile tail-recursive functions.

Macro Expansion

When an application of a macro, a compiler macro, or a function defined by using **defsubst** is evaluated at compile time, the form is evaluated and the resulting expression is substituted for the original form. This process eliminates the overhead of a function call.

While these forms may make the code faster, they have the following disadvantages:

- Because a macro is not invoked during execution, the macro call does not appear on the stack and cannot be traced.
- When a file is compiled, the macro definition must precede the first use of the macro in text; otherwise the Compiler will treat the macro call as an ordinary function call, which causes a run-time error.
- If a macro is redefined, any code using the macro definition must be recompiled.

Constant Folding

The Compiler may perform an optimization called ***constant folding*** on a form that has no side effects and that can be evaluated repeatedly to produce values that are eql. During compilation the form is evaluated and replaced by its value. Forms that may be constant folded include symbols defined by using **defconstant**, atoms other than symbols, lists whose car is **quote**, and constant numerical expressions.

The Compiler may use constant folding on the following expressions:

```
pi
(+ 1 2)
(* 2.0 pi)
'(this is a constant list)
#(a constant vector)
(aref #(a b c) 1)
(car '(a b))
"Hi, I'm a string"
'(a b)
```

The Compiler does not use constant folding on the following expressions:

```
random-symbol
(list 'values 'of 'this 'expression 'are 'not 'eql)
(cons 'a 'b)
(make-array 10)
(* 0 (random-function-that-may-have-side-effects)
```

Nonnumerical expressions that allocate new storage are usually not folded because the values are not **eql**.

Note: You should never update structures created from constant forms because constant objects that appear in compiled code may be allocated by the loader in a read-only area in memory.

For example, the following function is incorrect:

```
(defun this-will-lose ()
  (let ((x #(0 0 0)))
    (setf (aref x 0) 1)  ; Cannot update constants.
    x))
```

The following function is correct:

```
(defun this-will-win ()
  (let ((x (copy-seq #(0 0 0))))
    (setf (aref x 0) 1)
    x))
```

Case Macro Optimization

When a call to the **case** macro contains only fixnum key values within a certain range, the Compiler improves run-time performance by expanding the **case** macro into code that jumps through a dispatch table.

To use a dispatch table, the application of the **case** macro must satisfy the following qualifications:

- All keys must be of type fixnum except the last key, which can also be either the symbol **t** or the symbol **otherwise**.
- The range of the keys must satisfy the following algorithm:

  (*highest-key* - *lowest-key*) < (10 * *number-of-keys*)
- The lowest key must be less than 68, and the highest key must be greater than -60.

For example, the following code can be compiled into code that uses a dispatch table:

```
(case x (1 'one)
        (2 'two)
        (4 'four)
        (t 'other-num))
```

clear-undef

Purpose: At the end of compilation, the Compiler prints out a list of all the currently undefined functions. The function **clear-undef** resets this list to **nil**.

Syntax: **clear-undef** [*Function*]

Remarks: This function is an extension to Common Lisp.

compile

Purpose: The function **compile** compiles an interpreted function in the current Lisp environment. It replaces the interpreted function's definition with the compiled version. It produces a compiled function object from a lambda expression. The lambda expression is the argument *definition* if it is present; otherwise the function definition of the symbol *name* is the relevant lambda expression.

If the *name* argument is **nil**, **compile** returns the compiled function object; otherwise the function definition of the symbol *name* is set to the compiled function object, and **compile** returns that symbol.

Syntax: **compile** *name* `&optional` *definition* [*Function*]

Examples:

```
> (defun foo (x) (+ x x))
FOO
> (compile 'foo)
;;; Compiling function FOO...assembling...emitting...done.
FOO
> (foo 2)
4
> (funcall (compile nil '(lambda (x) (+ x x))) 3)
;;; Compiling function...assembling...emitting...done.
6
```

compile-file

Purpose: The function **compile-file** produces binary files from Lisp source files. The compiled functions contained in the binary files become available for use when the binary files are loaded into Lisp.

Syntax: **compile-file** *input-pathname* &key

```
:output-file
:messages
:warnings
:fast-entry
:tail-merge
:notinline
:target
```

[*Function*]

Remarks: The function converts a file specified by the *input-pathname* argument into compiled code.

If given, the **:output-file** option specifies where the compiled code is sent; its argument should be a pathname or a string describing a valid filename. The binary file that is produced from the source file is given that name, and any existing file with that name is overwritten.

If this option is not specified or if its argument is bound to **nil**, the file extension for the binary file is determined as follows:

- Source file has the extension `.lisp`.

  In this case, the extension `.lisp` is replaced by `.lbin`.

- Source file does not have the extension `.lisp`.

  If the extension is different from `.lisp` or if there is no extension, the extension `.lbin` is attached to the end of the source filename.

The default value of this option is **nil**.

The **:messages** option controls the fate of the progress messages issued by the Compiler. A value of **nil** means issue no progress messages; otherwise the value should specify a stream to which messages can be sent. The default value is **t**, which sends the messages to the standard terminal device.

The **:warnings** option controls the warnings issued by the Compiler. A value of **nil** means issue no warnings; otherwise the value must specify a stream to which warnings can be sent. The default value is **t**, which sends the warnings to ***error-output***.

If the **:fast-entry** option has a non-**nil** value, the Compiler does not insert code to check the number of arguments on entry to a function with a fixed number of arguments. Thus, calls to functions compiled in this manner are slightly faster. The default value is **nil**.

If the **:tail-merge** option has a non-**nil** value, the Compiler converts tail-recursive calls to iterative constructions and performs other types of tail call optimizations, such as tail merging, that eliminate the overhead of some function calls. The default value is **t**.

If the **:notinline** option has a non-**nil** value, the Compiler behaves as if all functions have been declared notinline; see the section on inline and notinline declarations for more details. The default value is **nil**.

The possible values for the **:target** option are as follows:

- 68020/68881

  If you specify this value for the **:target** option, the Compiler generates binary files specifically for the MC68881 coprocessor. Such files use the machine-specific floating-point hardware and produce faster floating-point operations. A binary file produced for the MC68881 coprocessor must be loaded on a machine with the correct hardware; otherwise a continuable error is signaled.

- 68020

  If you specify this value for the **:target** option, the Compiler generates binary files specifically for the MC68020 processor. Such files may run slightly faster in some cases, but they will not run on MC68010 processors.

- 68K

  This is the default value of the **:target** option.

The keywords **:messages**, **:warnings**, **:fast-entry**, **:tail-merge**, **:notinline**, and **:target** are extensions to Common Lisp.

See Also: **compiler-options**

compiled-function-p

Purpose: The predicate **compiled-function-p** is true if its argument is a compiled code object; otherwise it is false.

Syntax: **compiled-function-p** *object* [*Function*]

Examples:

```
> (compiled-function-p (symbol-function 'append))
T
> (compiled-function-p #'(lambda (x) x))
NIL
```

compiler-options

Purpose: The function **compiler-options** resets the default values of the keyword options **:messages**, **:warnings**, **:fast-entry**, **:tail-merge**, **:notinline**, and **:target** of the function **compile-file**.

Syntax: **compiler-options** &key **:messages** [*Function*]

```
                           :warnings
                           :fast-entry
                           :tail-merge
                           :notinline
                           :target
```

Remarks: The keyword **:messages** controls the fate of the progress messages issued by the Compiler. A value of **nil** means issue no progress messages; otherwise the keyword should specify a stream to which messages can be sent. The default value is **t**, which sends the messages to the standard terminal device.

The keyword **:warnings** controls the warnings issued by the Compiler. A value of **nil** means issue no warnings; otherwise the keyword value must specify a stream to which warnings can be sent. The default value is **t**, which sends the warnings to ***error-output***.

If the **:fast-entry** keyword has a non-**nil** value, the Compiler does not check the number of arguments on entry to a function with a fixed number of arguments. Thus, calls to functions compiled in this manner are slightly faster. The default value of **:fast-entry** is **nil**.

If the **:tail-merge** option has a non-**nil** value, the Compiler converts tail-recursive calls to iterative constructions and performs other types of tail call optimizations, such as tail merging, that eliminate the overhead of some function calls. The default value of **:tail-merge** is **t**.

If the **:notinline** option has a non-**nil** value, the Compiler behaves as if all functions have been declared notinline; see the section on inline and notinline declarations for more details. The default value of **:notinline** is **nil**.

The possible values for the **:target** option are as follows:

- 68020/68881

  If you specify this value for the **:target** option, the Compiler generates binary files specifically for the MC68881 coprocessor. Such files use the machine-specific floating-point hardware and produce faster floating-point operations. A binary file produced for the MC68881 coprocessor must be loaded on a machine with the correct hardware; otherwise a continuable error is signaled.

- 68020

  If you specify this value for the **:target** option, the Compiler generates binary files specifically for the MC68020 coprocessor. Such files may run slightly faster in some cases, but they will not run on MC68010 processors.

- 68K

  This is the default value of the **:target** option.

This function is an extension to Common Lisp.

See Also: **compile-file**

declare

Purpose: The **declare** special form may be used to make declarations within certain forms. Declarations may occur in lambda expressions and in the following forms:

define-setf-method
defmacro
defsetf
deftype
defun
do
do*
do-all-symbols
do-external-symbols
do-symbols
dolist
dotimes
flet
labels
let
let*
locally
macrolet
multiple-value-bind
prog
prog*
with-open-stream
with-open-file
with-output-to-string
with-input-from-string

Syntax: **declare** {*decl-spec*}* [*Special Form*]

```
decl-spec::= (special {var}*) |
             (type type-specifier {var}*) |
             (type-specifier {var}*) |
             (ftype type-specifier {function-name}*) |
             (function function-name ({type-specifier}*)  {type-specifier}*) |
             (restrictive-ftype type-specifier {function-name}*) |
             (type-reduce type1 type2) |
             (inline {function-name}*) |
             (notinline {function-name}*) |
             (ignore {var}*) |
             (optimize {(quality value) | quality}*) |
             (declaration {declaration-name}*) |
             (arglist (declared-arguments)) |
             (dynamic-extent variable)

quality::= speed | space | safety | compilation-speed

value::= 0 | 1 | 2 | 3
```

Remarks: Declarations may only occur where specified by the syntax of these forms.

Macros may expand into declarations as long as this syntax is observed.

The declaration specifier argument is not evaluated.

declare

The declarations **restrictive-ftype**, **type-reduce**, **arglist**, and **dynamic-extent** are extensions to Common Lisp.

For more information, see the *Sun Common Lisp Reference Manual.*

Examples:

```
> (defun foo (y)                                    ; This y is regarded
      (declare (special y))                         ; as special.
      (let ((y t))                                  ; This y is regarded
         (list y                                    ; as lexical.
               (locally (declare (special y)) y)))) ; This y refers to the
                                                    ; special binding of y.
FOO
> (foo nil)
(T NIL)
```

def-compiler-macro

Purpose: The macro **def-compiler-macro** defines a Compiler macro. A Compiler macro is a macro that affects only compiled code, not interpreted code.

Syntax: **def-compiler-macro** *name lambda-list* {*form*}* [*Macro*]

Remarks: The *name* argument is a symbol; it is not evaluated. A global Compiler macro expansion function is associated with the symbol *name.* It is possible to have a Compiler macro defined with **def-compiler-macro** associated with the same name as a function defined with **defun** or a macro defined with **defmacro**.

The *name* is returned as the result of **def-compiler-macro**.

The *lambda-list* argument specifies the arguments to the macro being defined; this argument is equivalent to a lambda list of a **defmacro** form.

The body of the macro expansion function is specified by the *form* arguments. They are executed in order.

This macro is an extension to Common Lisp.

Examples:

```
> (defmacro xx1 (x) `(format nil "~A interpreted" ,x))
XX1
> (def-compiler-macro xx1 (x) `(format nil "~A compiled" ,x))
XX1
> (defun xx2 () (xx1 "test is"))
XX2
> (xx2)
"test is interpreted"
> (compile 'xx2)
;;; Compiling function XX2...tail merging...assembling...emitting...done.
XX2
> (xx2)
"test is compiled"
```

See Also: **defmacro**

defsubst

Purpose: The function **defsubst** defines a function and makes its definition available for in-line expansion by the Compiler.

Syntax: **defsubst** *name lambda-list* {*form*}* [*Function*]

Remarks: The *name* argument must be a symbol; it is not evaluated. A global function definition is attached to the symbol *name* as the contents of the symbol's function cell. The *name* of the new function is returned as the value of the **defsubst** form.

The *lambda-list* argument specifies the arguments to the function being defined.

The body of the function consists of the forms specified by the *form* arguments; they are executed in order when the function is called.

The expression (defsubst *new-function* ...) is equivalent to the expression (progn (proclaim '(inline *new-function*)) (defun *new-function* ...)). When a call to a function defined with **defsubst** has been compiled, it cannot be traced or redefined.

This function is an extension to Common Lisp.

Examples:

```
> (defsubst test1 () "This is a test")
TEST1
> (defun test2 () (test1))
TEST2
> (trace test1)
(TEST1)
> (test2)
1 Enter TEST1
1 Exit TEST1 "This is a test"
"This is a test"
> (compile 'test2)
;;; Compiling function TEST2...assembling...emitting...done.
TEST2
> (test2)
"This is a test"
> (untrace test1)
(TEST1)
> (defun test1 () "This is a new test")
TEST1
> (test2)
"This is a test"
```

See Also: **defun**

proclaim

disable-stack-lists

Purpose: The function **disable-stack-lists** disables stack list allocation. The Compiler generates code that allocates **&rest** arguments in dynamic storage.

Syntax: **disable-stack-lists** [*Function*]

Remarks: This function is an extension to Common Lisp.

See Also: **enable-stack-lists**

enable-stack-lists

Purpose: The function **enable-stack-lists** turns on stack list allocation. Stack list allocation allows the Compiler to generate code that allocates **&rest** arguments that are declared to have dynamic extent on a stack instead of in dynamic memory. This reduces the number of garbage collections.

Syntax: **enable-stack-lists** `&key :max-depth :max-length` *[Function]*

Remarks: The keyword argument **:max-depth** specifies the maximum number of stack lists that may be allocated at any time. If specified, the value of **:max-depth** must be an integer between 10 and 10000 inclusive. The value defaults to 100.

The keyword argument **:max-length** specifies the maximum number of cons cells that may be allocated in stack lists at any time. If specified, the value of **:max-length** must be an integer between 100 and 100000. The value defaults to 1000.

If stack list allocation is repeatedly enabled in a Lisp process, the maximum depth and maximum length are never reduced, even if the **:max-depth** and **:max-length** arguments are given smaller values than were previously used.

If the creation of a stack list would exceed either the maximum depth or the maximum length, the list is created in dynamic memory instead.

Stack list allocation is enabled by default. You need only call the function **enable-stack-lists** to increase the available stack list space or to reenable stack list allocation after a call to **disable-stack-lists**.

See Also: **disable-stack-lists**

eval-when

Purpose: The special form **eval-when** specifies when a particular body of code is to be executed.

This time is defined by the *situation* arguments. The value of each argument must be **compile**, **load**, or **eval**.

If **eval** is specified, the evaluator evaluates the *form* arguments at execution time. If **compile** is specified, the Compiler evaluates the *form* arguments at compilation time. If **load** is specified and the file containing the **eval-when** is compiled, then the forms are compiled; they are executed when the output file produced by the Compiler is loaded.

The value of the last *form* evaluated is returned as the result of **eval-when**. If no forms are executed, **eval-when** returns **nil**.

Syntax: **eval-when** ({*situation*}*) {*form*}* [*Special Form*]

Remarks: The *form* arguments are executed in order.

Examples:

```
> (setq foo 3)
3
> (eval-when (compile) (setq foo 2))
NIL
> foo
3
> (eval-when (eval) (setq foo 2))
2
> foo
2
```

locally

Purpose: The **locally** macro makes local declarations that affect only its *form* arguments.

Syntax: **locally** {*declaration*}* {*form*}* [*Macro*]

Examples:

```
> (defun foo (y)                                  ; This y is regarded
       (declare (special y))                      ; as special.
       (let ((y t))                               ; This y is regarded
          (list y                                 ; as lexical.
                (locally (declare (special y)) y)))) ; This y refers to the
                                                  ; special binding of y.
FOO
> (foo nil)
(T NIL)
```

proclaim

Purpose: The **proclaim** function makes a global declaration, or proclamation.

A proclamation whose declaration specifier declares a variable to be **special** makes all occurrences of that variable name special references.

Syntax: **proclaim** *decl-spec* [*Function*]

Remarks: Although the effect of a proclamation is global, it may be overridden by a local declaration.

The argument of **proclaim** is evaluated. It may therefore be a computed declaration specifier.

Examples:

```
> (proclaim '(special prosp))
T
> (setq prosp 1 reg 1)
1
> (let ((prosp 2)(reg 2))
    (set 'prosp 3)(set 'reg 3)
    (list prosp reg))
(3 2)
> (list prosp reg)
(1 3)
```

See Also: **defvar**

defparameter

(In the *Sun Common Lisp Reference Manual*)

the

Purpose: The special form **the** specifies that the value produced by a form will be of a certain type.

The *value-type* argument is a type specifier; it is not evaluated. The *form* argument is evaluated.

The **the** special form returns the value or values that result from *form*.

Syntax: **the** *value-type form* [*Special Form*]

Remarks: You can use the macro **setf** with **the** type declarations. In this case the declaration is transferred to the form that specified the new value. The resulting **setf** form is then analyzed.

Examples:

```
> (the list '(a b))
(A B)
> (the (values integer list) (values 5 '(a b)))
5
(A B)
> (let ((i 100))
    (declare (fixnum i))
    (the fixnum (1+ i)))
101
```

uncompile

Purpose: The function **uncompile** replaces the compiled version of a function or macro with its original interpreted definition. The function or macro must have been compiled with a call to the function **compile**; otherwise an error is signaled.

Syntax: **uncompile** *name* [*Function*]

Remarks: This function is an extension to Common Lisp.

See Also: **compile**

undef-compiler-macro

Purpose: The macro **undef-compiler-macro** removes a Compiler macro definition.

Syntax: **undef-compiler-macro** *name* [*Macro*]

Remarks: The *name* argument is a symbol.

This macro is an extension to Common Lisp.

See Also: **def-compiler-macro**

unproclaim

Purpose: The function **unproclaim** reverses a proclamation.

Syntax: **unproclaim** *decl-spec* [*Function*]

Remarks: The *decl-spec* argument must be a valid declaration specifier for **declare**; otherwise an error is signaled. With the exception of the **optimize** declaration, the declaration specifier must exactly match the most recently proclaimed declaration specifier. If it does not, the call to **unproclaim** has no affect.

If **unproclaim** is called with a declaration specifier of the form `(optimize` *quality*`)`, *quality* is set to its default value. If **unproclaim** is called with a declaration specifier of the form `(optimize (`*quality value*`))`, *quality* is reset to its default value only if the current value of *quality* is *value*; otherwise, the call has no affect.

This function is an extension to Common Lisp.

See Also: **proclaim**

Chapter 14. Miscellaneous Programming Features

Chapter 14. Miscellaneous Programming Features

Introduction 14–3
The Source File Recording Facility 14–4
  About Source File Definitions 14–4
  Source File Recording Examples 14–5
The Advice Facility 14–6
  About Advice 14–6
  Advice Examples 14–7
advice-continue, apply-advice-continue 14–9
defadvice 14–10
describe-advice 14–11
discard-source-file-info 14–12
get-source-file 14–13
load-instance 14–14
record-source-file 14–15
record-source-files 14–16
redefinition-action 14–17
remove-advice 14–19
source-pathname 14–20
terse-redefinitions 14–21

Introduction

This chapter describes two miscellaneous programming features. The Source File Recording Facility records the name of the source file in which a Lisp object is defined. The Advice Facility allows you to wrap a function with additional advice. All of the constructs used by these facilities are extensions to Common Lisp.

The function pages for both facilities are at the end of the chapter in alphabetical order.

The Source File Recording Facility

The ***Source File Recording Facility*** records the name of the source file in which a Lisp object is defined. When this facility is enabled, Lisp warns you when an object that has been previously defined in a file is redefined in another file; the facility is enabled by default. Lisp distinguishes between redefinitions of an object, multiple definitions of the same object in one file, and redefinitions caused by reloading a file.

The Source File Recording Facility consists of the following functions and variables:

discard-source-file-info
get-source-file
load-instance
record-source-file
record-source-files
redefinition-action
source-pathname
terse-redefinitions

About Source File Definitions

Defining an object is a way to name and describe an object; for example, you can define a named function and specify its behavior by using **defun**. The Source File Recording Facility records definitions of functions, macros, and structures. It can be extended to record definitions of objects of other types.

Redefining an object assigns a new definition to a previously defined object. Giving a named object the same definition that you originally specified is also considered a redefinition.

When source file recording is enabled, defining and redefining objects has the following effects:

- When you redefine an object, the redefinition warning gives you the name of the source file containing the original definition.
- When you load a file, redefinition warnings distinguish between multiple definitions, redefinitions, and reloading a file as follows:
  - If an object has been defined in a previous instance of loading the same file, warnings are suppressed.
  - If an object has been defined earlier in the same file, you are warned that the object has multiple definitions.
  - If an object has been previously defined in another file, you are warned that the object is being redefined and are given the name of the source file containing the previous definition.

- You can get the pathname of the source file that contains the definition of an object by using the function **get-source-file**.
- In the Editor, you can locate the source code for a function or a macro and place it in an Editor buffer by using the **Edit Definition** command.

The Source File Recording Facility is enabled by default. To stop source file recording, set the variable *record-source-files* to **nil**. Stopping source file recording does not delete old information; to remove source file information, use the function **discard-source-file-info**. You may want to remove source file information before you save an image to reduce the image size.

To restart source file recording, set the variable *record-source-files* to **t**.

Source File Recording Examples

The following examples illustrate the effects of source file recording on redefinition warnings.

```
;; Suppose the files "foo.lisp" and "bar.lisp" both contain definitions
;; of the function DOIT.
;;
;; Loading "foo" defines DOIT for the first time.
> (load "foo")
;;; Loading source file "foo.lisp"
#P"/u/kdo/foo.lisp"

;; Now, loading "bar", which also defines DOIT, produces a warning.
> (load "bar")
;;; Loading source file "bar.lisp"
;;; Warning: Redefining function DOIT which used to be defined in
file /u/kdo/foo.lisp
#P"/u/kdo/bar.lisp"

;; Reloading the same file produces no warning.
> (load "bar")
;;; Load source file "bar.lisp".
#P"/u/kdo/bar.lisp"

;; Get the pathname of the source file.
> (get-source-file 'doit)
#P"/u/kdo/bar.lisp"
```

The Advice Facility

The ***Advice Facility*** allows you to modify the behavior of an existing function by attaching named pieces of advice. You can attach multiple pieces of advice to the same function in a specified order.

The Advice Facility consists of the following functions and macros:

advice-continue
apply-advice-continue
defadvice
describe-advice
remove-advice

About Advice

A symbol normally has a function definition. When some advice is applied, the symbol is changed to have a new definition, called a ***piece of advice***. From within the advice, the original definition is called the ***continuation***.

When a symbol is called as a function, the advice is called instead of the original definition. The advice accepts the arguments to the call and returns the value for the call. The advice may or may not call its continuation, and it may call its continuation several times or with different arguments. The advice may pass on its return values or it may modify or ignore them. It may do exactly what the continuation does, or it may do something totally unrelated. The advice takes the place of the original definition for all interfaces to the external world.

You can supply multiple, nested pieces of advice. The ***outermost*** piece of advice is the one that is invoked first when the function is called. The continuation for the outermost advice is the next inner piece of advice, and so on; the continuation for the innermost piece of advice is the original definition of the function.

To create a piece of advice, use the **defadvice** macro. You can specify the order of the pieces of advice by using the *place* argument to **defadvice**. To call the advised function from within the body of **defadvice**, use the macros **advice-continue** and **apply-advice-continue**.

To remove a piece of advice, use the **remove-advice** function. If the advice is removed, the continuation is restored to its position as the definition of the symbol.

Advice Examples

The following examples illustrate the use of **defadvice** and related advising constructs.

```
;; Define a function that adds 1 to its argument.
> (defun foo (x) (+ x 1))
FOO
> (foo 3)               ; 3+1 = 4
4
;; Put some advice on it that gives a default value for the argument.
> (defadvice (foo default-to-zero) (&optional (x 0))
      (advice-continue x))
#<Interpreted-Function (:ADVICE FOO DEFAULT-TO-ZERO) BOOD37>
> (foo)                 ; No argument supplied, so 0+1 = 1
1
;; Put some more advice on to double the result.
> (defadvice (foo double-result) (&rest args)
             (* 2 (apply-advice-continue args)))
#<Interpreted-Function (:ADVICE FOO DOUBLE-RESULT) B261BF>
> (foo)                 ; (default 0+1)*2 = 2
2
> (foo 5)               ; (5+1)*2 = 12
12
;; See what is there now.
> (describe-advice 'foo)
Advice DOUBLE-RESULT:
 #<Interpreted-Function (:ADVICE FOO DOUBLE-RESULT) B362D7>
  Advice DEFAULT-TO-ZERO:
   #<Interpreted-Function (:ADVICE FOO DEFAULT-TO-ZERO) B3619F>
    Original definition: #<Interpreted-Function (NAMED-LAMBDA FOO (X)
(BLOCK FOO (+ X 1))) AF1A57>
NIL
;; Get rid of one of the pieces of advice.
> (remove-advice 'foo 'default-to-zero)
Removing #<Interpreted-Function (:ADVICE FOO DEFAULT-TO-ZERO) B3619F>
#<Interpreted-Function (:ADVICE FOO DOUBLE-RESULT) B60747>
;; Now install more advice to add 1 to the result.  Put it outside
;; the advice that does the doubling.
> (defadvice (foo add-one-to-result (:outside double-result)) (x)
             (+ 1 (advice-continue x)))
#<Interpreted-Function (:ADVICE FOO ADD-ONE-TO-RESULT) B7732F>
> (foo 2)               ; ((2+1)*2)+1 = 7
7
```

```
;; Replace that advice with the same piece of advice but put it inside
;; the doubling advice instead.
> (defadvice (foo add-one-to-result (:inside double-result)) (x)
            (+ 1 (advice-continue x)))
#<Interpreted-Function (:ADVICE FOO DOUBLE-RESULT) B7ED9F>
> (foo 2)               ; ((2+1)+1)*2 = 8
8
;; Redefine the basic function inside the advice to return its argument
;; unchanged.
> (defun foo (x) x)
;;; Warning: Redefining FOO, keeping advice DOUBLE-RESULT, ADD-ONE-TO-
RESULT
FOO
> (foo 2)               ; (2+1)*2 = 6
6
;; Remove all the advice from the function.
> (remove-advice 'foo)
Removing 2 pieces of advice from FOO
NIL
> (foo 1)
1
```

advice-continue, apply-advice-continue

Purpose: The macros **advice-continue** and **apply-advice-continue** call an advised function from within the body of **defadvice**.

The macro **advice-continue** applies the continuation to the specified arguments.

The macro **apply-advice-continue** applies the continuation to a specified list of arguments.

Syntax: **advice-continue** **&rest** *args* [*Macro*]

apply-advice-continue *args* **&rest** *more-args* [*Macro*]

Remarks: The last argument specified for **apply-advice-continue** must be a list. It is appended to a list of all the other arguments.

These macros are extensions to Common Lisp.

See Also: **apply**

funcall

(in the *Sun Common Lisp Reference Manual*)

defadvice

Purpose: The macro **defadvice** defines a piece of advice for a specified function.

Syntax: **defadvice** (*function-to-advise name-of-advice* `&optional` *place*) (*lambda-list*) {*form*}* [*Macro*]

Remarks: The *function-to-advise* argument is the name of the function being advised. Only functions that are stored as definitions of symbols can be advised.

The *name-of-advice* argument specifies a name for the advice. Any existing advice with this name is replaced; the names are compared using the function **string-equal**.

The optional *place* keyword argument specifies how the advice that is being defined relates to other pieces of advice on the same function. This argument should be a list of keywords and values. The following options are valid:

- (**:outside** *name*), (**:before** *name*)

  The **:outside** keyword puts the piece of advice that is being defined outside the advice with the specified *name*; the keyword **:before** is synonymous.

- (**:inside** *name*), (**:after** *name*)

  The **:inside** keyword puts the piece of advice that is being defined inside the advice with the specified *name*; the keyword **:after** is synonymous.

If the *place* argument is omitted, the new piece of advice goes outside any advice defined earlier and inside any advice defined later. If the order of the pieces of advice is important, you should explicitly specify the position of the new advice in relation to the existing pieces.

The *lambda-list* argument specifies the arguments to the advised function.

The forms specified by the *form* arguments make up the body of **defadvice**, which is any normal function body.

If **defadvice** appears in a compiled file, the advice function is compiled.

This macro is an extension to Common Lisp.

describe-advice

Purpose: The function **describe-advice** describes all levels of advice for a specified function.

Syntax: **describe-advice** *function* [*Function*]

Remarks: The *function* argument names the function whose advice is being described.

This function is an extension to Common Lisp.

See Also: **defadvice**

discard-source-file-info

Purpose: The function **discard-source-file-info** discards all recorded source file information. This function allows you to reduce the size of an image; it is particularly useful before an image is saved.

Syntax: **discard-source-file-info** [*Function*]

Remarks: Using **discard-source-file-info** does not affect the setting of ***record-source-files***.

This function is an extension to Common Lisp.

See Also: **record-source-file**

record-source-files

get-source-file

Purpose: The function **get-source-file** returns information about the file or files in which an object is defined.

Syntax: **get-source-file** *object* `&optional` *type want-list* [*Function*]

Remarks: The optional *type* argument is a symbol that identifies the type of definition sought. The value of *type* can be either **function**, **macro**, **structure**, or a definition type that you have created by using the function **record-source-file**.

If the *type* argument is specified, the function searches for only a specific type of definition. If the definition exists, the pathname of the source file that contains the definition is returned. If the definition does not exist, an error is signaled.

If the *type* argument is **nil** and the *want-list* argument is specified, the function searches for all definitions of the specified object and returns a list of the form (*type* . *pathname*) for each definition of the object.

If neither the *type* nor the *want-list* argument is specified, the function searches for all definitions of the specified object. If there is a single definition of the object, the function returns two values: the pathname and the type. If there are multiple definitions of the same object, a continuable error is signaled.

This function is an extension to Common Lisp.

See Also: **record-source-file**

load-instance

Purpose: The variable *load-instance* identifies an instance of loading a file. When a file is loaded, the variable *load-instance* is bound to a unique object that distinguishes this instance of loading the file from previous loads of the same file. This variable is used to detect multiple definitions of an object in the same file.

Syntax: *load-instance* [*Variable*]

Remarks: This variable is an extension to Common Lisp.

See Also: **record-source-file**

load (in the *Sun Common Lisp Reference Manual*)

record-source-file

Purpose: The function **record-source-file** records information about the source file that defines a particular object.

Syntax: **record-source-file** *object type* `&optional` *pathname load-instance* [*Function*]

Remarks: The *type* argument is a symbol that identifies the type of definition. The value of *type* can be either **function**, **macro**, **structure**, or a definition type you create by supplying a symbol.

The optional argument *pathname* gives the pathname of the file that is defining the object. The default is the value of the variable ***source-pathname***.

The value of the optional argument *load-instance* is an object that distinguishes the current instance of loading a file from other instances of loading the same file. The value of *load-instance* is used to detect multiple definitions of the same object in the same file. The default is the value of the variable ***load-instance***.

This function is an extension to Common Lisp.

See Also: ***redefinition-action***

source-pathname

record-source-files

Purpose: The variable ***record-source-files*** determines whether to record the name of the file in which an object is defined.

Syntax: ***record-source-files*** [*Variable*]

Remarks: If the variable is non-**nil**, source file names are recorded. Redefinition warnings are not produced when functions are redefined by loading the same file again; redefinition warnings are produced when an object is redefined in the same file. Redefinition messages include the name of the original source file that defines the object. The default value of ***record-source-files*** is **t**.

If the variable is **nil**, source file names are not recorded. Redefinition warnings are produced when a file is reloaded and when an object is redefined in the same file. Redefinition warnings do not include source file information.

If the variable is **nil**, no new information is recorded; however, old information is not lost.

This variable is an extension to Common Lisp.

See Also: **record-source-file**

redefinition-action

redefinition-action

Purpose: The global variable ***redefinition-action*** specifies what action is taken when a redefinition occurs.

Syntax: ***redefinition-action*** [*Variable*]

Remarks: The variable ***redefinition-action*** can have one of the following values:

- **:warn**

  If the value is **:warn**, you are warned when a function or macro is redefined as a result of loading a different file from the source file that contains the original definition. If the source file that contains the original definition is unknown, the warning is always given. The default value is **:warn**.

- **:query**

  If the value is **:query**, you are asked whether you wish to proceed with the redefinition.

- **nil**

  If the value is **nil**, no action is taken.

This variable is an extension to Common Lisp.

Examples:

```
; Suppose that *record-source-files* is set, which is the default.  If the
; file temp1.lisp contains
;                         (defun test ())
; and the file temp2.lisp also contains
;                         (defun test ())
; the following session could occur:
;

> *redefinition-action*                 ; The default action is :warn.
:WARN
> (load "~/temp1.lisp")                 ; Define "test" the first time.
;;; Loading source file "/u/kdo/temp1.lisp"
#P"/u/kdo/temp1.lisp"
> (load "~/temp1.lisp")                 ; Redefine from same file--no message.
;;; Loading source file "/u/kdo/temp1.lisp"
#P"/u/kdo/temp1.lisp"
> (load "~/temp2.lisp")                 ; Redefine it by loading new file.
;;; Loading source file "/u/kdo/temp2.lisp"
;;; Warning: Redefining function TEST which used to be defined in file
/u/kdo/temp1.lisp
#P"/u/kdo/temp2.lisp"
```

```
> (defun test ())                       ; Redefine it from top level.
;;; Warning: Redefining function TEST which used to be defined in file
/u/kdo/temp2.lisp
TEST
> (setq *redefinition-action* :query)   ; Reset to ask for confirmation.
:QUERY
> (load "~/temp1.lisp")
;;; Loading source file "/u/kdo/temp1.lisp"
Redefining function TEST which used to be defined at top level  OK?
(Y or N) y
#P"/u/kdo/temp1.lisp"
> (setq *redefinition-action* nil)      ; Don't print warnings.
NIL
> (load "~/temp2.lisp")
;;; Loading source file "/u/kdo/temp2.lisp"
#P"/u/kdo/temp2.lisp"
```

See Also: **define-function**

define-macro

load

(in the *Sun Common Lisp Reference Manual*)

remove-advice

Purpose: The function **remove-advice** removes some or all of the advice from a specified function.

Syntax: **remove-advice** ***function*** `&optional` ***advice-name*** [*Function*]

Remarks: The ***function*** argument specifies the name of the function from which the advice should be removed.

The ***advice-name*** argument specifies the name of the advice to be removed. If the ***advice-name*** argument is specified, only advice that matches the given name is removed. The names are compared using the function **string-equal**. If the ***advice-name*** argument is not specified, all advice is removed from the function.

This function is an extension to Common Lisp.

See Also: **defadvice**

source-pathname

Purpose: The variable ***source-pathname*** contains the truename of the source file of the file being loaded. If a file with the extension `.lisp` is being loaded, the value of this variable is the truename of the file being loaded. If a compiled file is being loaded, the value of this variable is the truename of the source file from which the compiled file was created. You can bind this variable to another value to pretend that a different file is being loaded.

Syntax: ***source-pathname*** [*Variable*]

Remarks: This variable is an extension to Common Lisp.

See Also: **record-source-file**

terse-redefinitions

Purpose: The variable ***terse-redefinitions*** controls the level of detail printed in redefinition warnings. If the variable is non-**nil**, the warnings are terse and may be harder to understand. The default value is **nil**.

Syntax: ***terse-redefinitions*** [*Variable*]

Remarks: This variable is an extension to Common Lisp.

See Also: ***redefinition-action***