17. The Compiler

17.1 The Basic Operations of the Compiler

The purpose of the Lisp compiler is to convert Lisp functions into programs in the Lisp Machine's instruction set, so that they run more quickly and take up less storage. Compiled functions are represented in Lisp by FEFs (Function Entry Frames), which contain machine code as well as various other information. The printed representation of a FEF is

#<DTP-FEF-POINTER *address name*>

If you want to understand the output of the compiler, refer to chapter 31, page 752.

There are three ways to invoke the compiler from the Lisp Machine. First, you may have an interpreted function in the Lisp environment that you would like to compile. The function **compile** is used to do this. Second, you may have code in an editor buffer that you would like to compile. The Zmacs editor has commands to read code into Lisp and compile it. Third, you may have a program (a group of function definitions and other forms) written in a file on the file system. The function **compile-file** can translate this file into a *QFASL* file that describes the compiled functions and associated data. The QFASL file format is capable of representing an arbitrary collection of Lisp objects, including shared structure. The name derives from 'Q', a prefix once used to mean "for the Lisp Machine, not for Maclisp", and 'FASL', an abbreviation for "fast loading".

17.2 How to Invoke the Compiler

compile *function-spec* &optional *definition*

Compiles an individual interpreted function definition. If *definition* is supplied, it is the definition to be compiled. Otherwise, the current definition of *function-spec* is used. If *function-spec* is non-nil, the compiled function is stored as the definition of *function-spec*, and *function-spec* is returned. Otherwise, the compiled function object itself is returned. (However, it is preferable to use **compile-lambda** if your wish is to create a compiled function object without storing it anywhere.)

The compiled function object created by **compile** records the interpreted definition it was made from on its debugging info alist (see page 242). This is useful in two ways: the function **uncompile** can be used to reinstall the interpreted definition, and **compile** invoked again on the same *function-spec* can find the interpreted definition used before and compile it again. The latter is useful if you have changed some macros or subst functions which the definition refers to.

uncompile *function-spec*

If *function-spec* is defined as a compiled function that records the original definition that was compiled, then *function-spec* is redefined with that original definition. This undoes the effect of calling **compile** on *function-spec*.

compile-lambda *lambda-exp function-spec*

Returns a compiled function object produced by compiling *lambda-exp*. The function name recorded by the compiled function object is *function-spec*, but that function spec is not defined by compile-lambda. This function is preferable to compile with first argument nil in that it allows you to specify the name for the function to record internally.

compile-encapsulations *function-spec*

Compiles all encapsulations that function-spec currently has. Encapsulations (see section 11.9, page 244) include tracing, breakons and advice. Compiling tracing or breakons makes it possible (or at least more possible) to trace or breakon certain functions that are used in the evaluator. Compiling advice makes it less costly to advise functions that are used frequently.

Any encapsulation that is changed will cease to be compiled; thus, if you add or remove advice, you must do compile-encapsulations again if you wish the advice to be compiled again.

compile-encapsulations-flag *Variable*

If this is non-nil, all encapsulations that are created are compiled automatically.

compile-file *input-file* &key *output-file set-default-pathname package*

Compiles the file specified by *input-file*, a pathname or namestring. The format for files input to the compiler is described on section 17.3, page 303.

If *output-file* is specified, it is a pathname used for the compiled file. Otherwise, the ouptut file name is computed from the input file name by specifying :qfasl as the type component.

package, if non-nil specifies the package in which compilation should be performed. Normally the system knows, or asks interactively, and you need not supply this argument.

set-default-pathname, if non-nil, means that the defaults should be set to the input file's name. *set-default-pathname* defaults to **t**.

qc-file *filename* &optional *output-file load-flag in-core-flag package file-local-declarations dont-set-default-p read-then-process-flag*

An older, obsolete way of invoking the compiler on a file.

file-local-declarations is for compiling multiple files as if they were one. *dont-set-default-p* suppresses the changing of the default file name to *filename* that normally occurs. The *load-flag* and *in-core-flag* arguments were not fully implemented and should not be used. *read-then-process-flag* causes the entire file to be read and then the entire file to be compiled; this is no longer advantageous now that there is enough memory to avoid thrashing when forms are read and compiled one by one, and it prevents compile-time reader-macros defined in the file from working properly.

qc-file-load *filename* &optional *output-file load-flag in-core-flag package functions-defined file-local-declarations dont-set-default-p read-then-process-flag*

Compiles a file and then loads in the resulting QFASL file.

compiler:compiler-verbose *Variable*

If this variable is non-nil, the compiler prints the name of each function that it is about to compile.

compiler:peep-enable *Variable*

The peephole optimizer is used if this variable is non-nil. The only reason to set it to nil is if there is a suspicion of a bug in the optimizer.

See also the disassemble function (page 792), which lists the instructions of a compiled function in symbolic form.

17.3 Input to the Compiler

The purpose of compile-file is to take a file and produce a translated version which does the same thing as the original except that the functions are compiled. compile-file reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the QFASL file so that when the QFASL file is loaded the effect of that source form will be reproduced. The differences between source files and QFASL files are that QFASL files are in a compressed binary form, which reads much faster but cannot be edited, and that function definitions in QFASL files have been translated from Lisp forms to FEFs.

So, if the source contains a (defun ...) form at top level, then when the QFASL file is loaded the function will be defined as a compiled function. If the source file contains a form that is not of a type known specially to the compiler, then that form (encoded in QFASL format) is output "directly" into the QFASL file, so that when the QFASL file is loaded that form will be evaluated. Thus, if the source file contains (princ "Hello") at top level, then the compiler puts in the QFASL file instructions to create the list (princ "Hello") and then evaluate it.

The Lisp Machine editor Zmacs assumes that source files are formatted so that an open parenthesis at the left margin (that is, in column zero) indicates the beginning of a function definition or other top level list (with a few standard exceptions). The compiler assumes that you follow this indentation convention, enabling it to tell when a close-parenthesis is missing from one function as soon as the beginning of the next function is reached.

If the compiler finds an open parenthesis in column zero in the middle of a list, it invents enough close parentheses to close off the list that is in progress. A compiler warning is produced instead of an error. After that list has been processed, the open parenthesis is read again. The compilation of the list that was forcefully closed off is probably useless, but the compilation of the rest of the file is usually correct. You can read the source file into the editor to fix and recompile the function that was unbalanced.

A similar thing happens on end of file in the middle of a list, so that you get to see any warnings for the function that was unbalanced.

Certain special forms including eval-when, progn, local-declare, declare-flavor-instance-variables, and comment are customarily used around lists that start in column zero. These symbols have a non-nil si:may-surround-defun property that makes the compiler permit this. You can add such properties to other symbols if you want.

compiler:qc-file-check-indentation *Variable*

If nil, inhibits the compiler from checking for open-parentheses in column zero.

When a macro definition (macro and defmacro forms) is encountered at top level in the file being compiled, the macro definition is recorded for the rest of the compilation so that the macro thus defined can be used in the same file following its definition. This is in addition to writing the compiled macro definition into the QFASL file.

Flavor definitions (defflavor forms, see page 414) and global special declarations (made with proclaim, page 54, or with defvar, page 33) are likewise recorded for the rest of the compilation, as well as written into the QFASL file so that they will be recorded permanently when the file is loaded.

sys:file-local-declarations *Variable*

During file-to-file compilation, the value of this variable is a list of all declarations that are in effect for the rest of the file. Macro definitions, defdecl's, proclaim's and special declarations that come from defvars are all recorded on this list.

Package-defining and altering functions such as defpackage, in-package, export and use-package are executed by the compiler in the ordinary, permanent fashion. They are also written in the QFASL file so that the form is executed just the same when the file is loaded. If you load the file later in the same session, the package altering form is executed twice. This is normally harmless. require receives the same treatment.

You can control explicitly whether a form is evaluated by the compiler, and whether it is written into the QFASL file to be executed when the file is loaded, using the eval-when construct. You might want a form to be:

Put into the QFASL file (compiled, of course), or not.

Evaluated within the compiler, or not.

Evaluated if the source file loaded, or not.

An eval-when form looks like

```
(eval-when times-list
  form1 form2 ...)
```

The *times-list* may contain one or more of the symbols load, compile, or eval. If load is present, the *forms* are written into the QFASL file to be evaluated when the QFASL file is loaded (except that defun forms put the compiled definition into the QFASL file instead). If compile is present, the *forms* are evaluated in the compiler. If eval is present, the *forms* are evaluated when read into Lisp; this is because eval-when is defined as a special form in Lisp. (The compiler ignores eval in the *times-list*.) For example,

```
(eval-when (compile eval) (macro foo (x) (cadr x)))
```

would define foo as a macro in the compiler and when the file is read in interpreted, but not when the QFASL file is fasloaded.

eval-when (*time...*) *body...* *Special form*

When seen by the interpreter, if one of the *times* is the symbol eval then the *body* forms are evaluated; otherwise eval-when does nothing.

But when seen by the compiler, this special form does the special things described above.

Nested use of eval-when is permitted but its meaning is tricky. If an inner eval-when form appears in an ordinary context where a general form would be written into the QFASL file but not executed at compile time, then it behaves in the usual fashion: the *body* forms are written into the QFASL file if load is one of the *times*, and they are evaluated at compile time if compile is one of the *times*.

If the inner eval-when form appears in a context which says to evaluate at compile time only, then the *body* forms are evaluated if eval is one of the *times*.

If the inner eval-when appears in a context which says to write into the QFASL file and evaluate at compile time, the the *body* forms are written into the QFASL file if load is one of the *times*, and they are evaluated at compile time if either compile or eval is one of the *times*.

For the rest of this section, we will use lists such as are given to eval-when, e.g. (load eval), (load compile), etc., to describe when forms are evaluated.

If a form is not enclosed in an eval-when, then the times at which it is evaluated depend on the form. The following table summarizes at what times evaluation takes place for any given form seen at top level by the compiler.

(eval-when *times-list form ...*)
times-list specifies when the *form...* should be performed.

(declare (special ...)) or (declare (unspecial ...))
The special or unspecial is performed at (load compile) time.

(declare *anything-else*)
anything-else is performed only at (compile) time.

(proclaim ...) is performed at (load compile eval) time.

(special ...) or (unspecial ...)
(load compile eval)

(macro ...) or (defmacro ...) or (defsubst ...)
or (defflavor ...) or (defstruct ...)
(load eval). However, during file to file compilation, the definition is recorded temporarily and used for expanding calls to the macro, or macros defined by the defstruct for the rest of the file.

(comment ...) Ignored at all times.

(compiler-let ((*var val*) ...) *body...*)
Processes the *body* in its normal fashion, but with the indicated variable bindings in effect. These variables will typically affect the operation of the compiler or of macros. See section 18.5.6, page 339.

(local-declare (*decl decl*...) *body*...)
: Processes the *body* in its normal fashion, with the indicated declarations added to the front of the list which is the value of local-declarations.

(defun ...) or (defmethod ...) or (defselect ...)
: (load eval), but at load time what is processed is not this form itself, but the result of compiling it.

(require ...) or (in-package ...)
or various other package functions
: (load compile eval)

anything-else (load eval)

Sometimes a macro wants to return more than one form for the compiler top level to see (and to be evaluated). The following facility is provided for such macros. If a form

```
(progn form1 form2 ...)
```

is seen at the compiler top level, all of the *forms* are processed as if they had been at compiler top level. (Of course, in the interpreter they are all evaluated.)

To prevent an expression from being optimized by the compiler, surround it with a call to dont-optimize.

dont-optimize *form* *Special form*

In execution, this is equivalent to simply *form*. However, any source-level optimizations that the compiler would normally perform on the top level of *form* are not done.
Examples:

```
(dont-optimize (apply 'foo (list 'a 'b)))
```

actually makes a list and calls apply, rather than doing

```
(foo 'a 'b)
```

```
(dont-optimize (si:flavor-method-table flav))
```

actually calls si:flavor-method-table as a function, rather than substituting the definition of that defsubst.

dont-optimize can even be used around a defsubst inside of setf or locf, to prevent open-coding of the defsubst. In this case, a function will be created at load time to do the setting or return the location.

```
(setf (dont-optimize (zwei:buffer-package buffer))
      (pkg-find-package "foo"))
```

Subforms of *form*, such as arguments, are still optimized or open coded, unless additional dont-optimize's appear around them.

17.4 Compile-Time Properties of Symbols

When symbol properties are referred to during macro expansion, it is desirable for properties defined in a file to be "in effect" for the the rest of the file if the file is compiled. This does not happen if get and defprop are used, because the defprop will not be executed until the QFASL file is loaded. Instead, you can use getdecl and defdecl. These are normally the same as get and defprop, but during file-to-file compilation they also refer to and create declarations.

getdecl *symbol property*

This is a version of get that allows the properties of the *symbol* to be overridden by declarations.

If a declaration of the form (*property symbol value*) is in effect, getdecl returns *value*. Otherwise, getdecl returns the result of (get *symbol property*).

If you intend to create such declarations with proclaim or local use of declare, you must make sure that a declaration declaration is in effect for *property*. You can do this with (proclaim '(declaration *property*)).

getdecl is typically used in macro definitions. For example, the setf macro uses getdecl to get the properties which say how to store in the specified place. See page 340 for an example of a macro that uses getdecl.

putdecl *symbol property value*

Causes (getdecl *symbol property*) to return *value*.

putdecl usually simply does a putprop. But if executed at compile time during file-to-file compilation, it instead makes an entry on file-local-declarations of the form (*property symbol value*).

In either case, this stores *value* where getdecl can find it; but if putdecl is done during compilation, it affects only the rest of that compilation.

defdecl *symbol property value* *Special form*

When executed, this is like putdecl except that the arguments are not evaluated. It is usually the same as defprop except for the order of the arguments.

Unlike defprop, when defdecl is encountered during file-to-file compilation, a declaration is recorded which remains in effect for the rest of the compilation. (The defdecl form also goes into the QFASL file to be executed when the file is loaded). defprop would have no effect whatever at compile time.

defdecl is often useful as a part of the expansion of a macro. It is also useful as a top-level expression in a source file.

Example:

```
(defdecl foo locf foo-location)
```

in a source file would allow (locf (foo *args*...)) to be used in the rest of that source file; and, once the file was loaded, by anyone.

Simple use defsetf expands into a defdecl.

17.5 Using Compiler Warnings

When the compiler prints warnings, it also records them in a data base, organized by file and by function within file. Old warnings for previous compilations of the same function are thrown away, so the data base contains only warnings that are still applicable. This data base can be used to visit, in the editor, the functions that got warnings. You can also save the data base and restore it later.

There are three editor commands that you can use to begin visiting the sites of the recorded warnings. They differ only in how they decide which files to look through:

Meta-X Edit Warnings
: For each file that has any warnings, asks whether to edit the warnings for that file.

Meta-X Edit File Warnings
: Reads the name of a file and then edits the warnings for that file.

Meta-X Edit System Warnings
: Reads the name of a system and then edits the warnings for all files in that system (see defsystem, page 660).

While the warnings are being edited, the warnings themselves appear in a small window at the top of the editor frame, and the code appears in a large window which occupies the rest of the editor frame.

As soon as you have finished specifying the file(s) or system to process, the editor proceeds to visit the code for the first warning. From then on, to move to the next warning, use the command Control-Shift-W. To move to the previous warning, use Meta-Shift-W. You can also switch to the warnings window with Control-X O or with the mouse, and move around in that buffer. When you use Control-Shift-W and there are no more warnings after the cursor, you return to single-window mode.

You can also insert the text of the warnings into any editor buffer:

Meta-X Insert File Warnings
: Reads the name of a file and inserts into the buffer after point the text for that file's warnings. The mark is left after the warnings, but the region is not turned on.

Meta-X Insert Warnings
: Inserts into the buffer after point the text for the warnings of all files that have warnings. The mark is left after the warnings, but the region is not turned on.

You can also dump the warnings data base into a file and reload it later. Then you can do Meta-X Edit Warnings again in the later session. You dump the warnings with si:dump-warnings and load the file again with load. In addition, make-system with the :batch option writes all the warnings into a file in this way.

si:dump-warnings *output-file-pathname* &rest *warnings-file-pathnames*

Writes the warnings for the files named in *warnings-file-pathnames* (a list of pathnames or strings) into a file named *output-file-pathname*.

compiler:warn-on-errors *Variable*

If this variable is non-nil, errors in reading code to be compiled, and errors in macro expansion within the compiler, produce only warnings; they do not enter the debugger. The variable is normally **t**.

The default setting is useful when you do not anticipate errors during compilation, because it allows the compilation to proceed past such errors. If you have walked away from the machine, you do not come back to find that your compilation stopped in the first file and did not finish.

If you find an inexplicable error in reading or macroexpansion, and wish to use the debugger to localize it, set compiler:warn-on-errors to nil and recompile.

17.5.1 Controlling Compiler Warnings

By controlling the compile-time values of the variables run-in-maclisp-switch, obsolete-function-warning-switch, and inhibit-style-warning-switch (explained above), you can enable or disable some of the warning messages of the compiler. The following special form is also useful:

inhibit-style-warnings *form* *Macro*

Prevents the compiler from performing style-checking on the top level of *form*. Style-checking is still done on the arguments of *form*. Both obsolete function warnings and won't-run-in-Maclisp warnings are done by means of the style-checking mechanism, so, for example,

```
(setq bar (inhibit-style-warnings (value-cell-location foo)))
```

does not warn that value-cell-location will not work in Maclisp, but

```
(inhibit-style-warnings (setq bar (value-cell-location foo)))
```

does warn, since inhibit-style-warnings applies only to the top level of the form inside it (in this case, to the **setq**).

Sometimes functions take arguments that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, there are three things you can do.

The first thing is to name the variables **ignore** or **ignored**. The compiler does not complain if a variable by one of these names is not used. Furthermore, by special dispensation, it is all right to have more than one variable in a lambda-list that has one of these names.

Another thing you can do is write an **ignore** declaration. Example:

```
(defun the-function (list fraz-name fraz-size)
  (declare (ignore fraz-size)))
```

This has the advantage that arglist (see page 242) will return a more meaningful argument list for the function, rather than returning something with **ignore**'s in it.

Finally, you can simply use the variable for effect (ignoring its value) at the front of the function. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size        ; This argument is not used.
  ...)
```

The following function is useful for requesting compiler warnings in certain esoteric cases. Normally, the compiler notices whenever any function *x* uses (calls) any other function *y*; it makes notes of all these uses, and then warns you at the end of the compilation if the function *y* got called but no definition of it has been seen. This usually does what you want, but sometimes there is no way the compiler can tell that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and funcall is done on it. There is no way that the compiler can see that this is going to happen, and so it can't notice the function usage, and so it can't create a warning message. In order to make such warnings happen, you can explicitly call the following function at compile-time.

compiler:function-referenced *what by*

what is a symbol that is being used as a function. *by* may be any function spec. compiler:function-referenced must be called at compile-time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

You can also tell the compiler about any function it should consider "defined":

compiler:compilation-define *function-spec*

function-spec is marked as "defined" for the sake of the compiler; future calls to this function will not produce warnings.

compiler:make-obsolete *function reason* *Macro*

This special form declares a function to be obsolete; code that calls it will get a compiler warning, under the control of obsolete-function-warning-switch. This is used by the compiler to mark as obsolete some Maclisp functions which exist in Zetalisp but should not be used in new programs. It can also be useful when maintaining a large system, as a reminder that a function has become obsolete and usage of it should be phased out. An example of an obsolete-function declaration is:

```
(compiler:make-obsolete create-mumblefrotz
        "use MUMBLIFY with the :FROTZ option instead")
```

17.5.2 Recording Warnings

The warnings data base is not just for compilation. It can record operations for any number of different operations on files or parts of files. Compilation is merely the only operation in the system that uses it.

Each operation about which warnings can be recorded should have a name, preferably in the keyword package. This symbol should have four properties that tell the system how to print out the operation name as various parts of speech. For compilation, the operation name is :compile and the properties are defined as follows:

```
(defprop :compile "compilation" si:name-as-action)
(defprop :compile "compiling" si:name-as-present-participle)
(defprop :compile "compiled" si:name-as-past-participle)
(defprop :compile "compiler" si:name-as-agent)
```

The warnings system considers that these operations are normally performed on files that are composed of named objects. Each warning is associated with a filename and then with an object within the file. It is also possible to record warnings about objects that are not within any file.

To tell the warnings system that you are starting to process all or part of a file, use the macro si:file-operation-with-warnings.

sys:file-operation-with-warnings *Macro*
(*generic-pathname operation-name whole-file-p*) *body...*

body is executed within a context set up so that warnings can be recorded for operation *operation-name* about the file specified by *generic-pathname* (see page 563).

In the case of compilation, this is done at the level of compile-file (actually, it is done in compiler:compile-stream).

whole-file-p should be non-nil if the entire contents of the file are to be processed inside the *body* if it finishes; this implies that any warnings left over from previous iterations of this operation on this file should be thrown away on exit. This is only relevant to objects that are not found in the file this time; the assumption is that the objects must have been deleted from the file and their warnings are no longer appropriate.

All three of the special arguments are specified as expressions that are evaluated.

Within the processing of a file, you must also announce when you are beginning to process an object:

sys:object-operation-with-warnings (*object-name location-function*) *body...Macro*
Executes *body* in a context set up so that warnings are recorded for the object named *object-name*, which can be a symbol or a list. Object names are compared with equal.

In the case of compilation, this macro goes around the processing of a single function.

location-function is either **nil** or a function that the editor uses to find the text of the object. Refer to the file SYS: ZWEI; POSS LISP for more details on this.

object-name and *location-function* are specified with expressions that are evaluated.

You can enter this macro recursively. If the inner invocation is for the same object as the outer one, it has no effect. Otherwise, warnings recorded in the inner invocation apply to the object specified therein.

Finally, when you detect exceptions, you must make the actual warnings:

sys:record-warning *type severity location-info format-string* &rest *args*
Records one warning for the object and file currently being processed. The text of the warning is specified by *format-string* and *args*, which are suitable arguments for **format**, but the warning is *not* printed when you call this function. Those arguments will be used to reprint the warning later.

sys:record-and-print-warning *type severity location-info format-string* &rest *args*
Records a warning and also prints it.

type is a symbol that identifies the specific cause of the warning. Types have meaning only as defined by a particular operation, and at present nothing makes much use of them. The system defines one type: **si:premature-warnings-marker**.

severity measures how important a warning this is, and the general causal classification. It should be a symbol in the keyword package. Several severities are defined, and should be used when appropriate, but nothing looks at them:

:implausible This warning is about something that is not intrinsically wrong but is probably due to a mistake of some sort.

:impossible This warning is about something that cannot have a meaning even if circumstances outside the text being processed are changed.

:probable-error
This is used to indicate something that is certainly an error but can be made correct by a change somewhere else; for example, calling a function with the wrong number of arguments.

:missing-declaration
This is used for warnings about free variables not declared special, and such. It means that the text was not actually incorrect, but something else that is supposed to accompany it was missing.

:obsolete This warning is about something that you shouldn't use any more, but which still does work.

:very-obsolete
This is about something that doesn't even work any more.

:maclisp This is for something that doesn't work in Maclisp.

:fatal This indicates a problem so severe that no sense can be made of the object at all. It indicates that the presence or absence of other warnings is

not significant.

:error There was a Lisp error in processing the object.

location-info is intended to be used to inform the editor of the precise location in the text of the cause of this warning. It is not defined as yet, and you should use nil.

If a warning is encountered while processing data that doesn't really have a name (such as forms in a source file that are not function definitions), you can record a warning even though you are not inside an invocation of sys:object-operation-with-warnings. This warning is known as a *premature warning* and it will be recorded with the next object that is processed; a message will be added so that the user can tell which warnings were premature.

Refer to the file SYS: SYS; QNEW LISP for more information on the warnings data base.

17.6 Compiler Source-Level Optimizers

The compiler stores optimizers for source code on property lists so as to make it easy for the user to add them. An optimizer can be used to transform code into an equivalent but more efficient form (for example, (eq *obj* nil) is transformed into (null *obj*), which can be compiled better). An optimizer can also be used to tell the compiler how to compile a special form. For example, in the interpreter do is a special form, implemented by a function which takes quoted arguments and calls eval. In the compiler, do is expanded in a macro-like way by an optimizer into equivalent Lisp code using prog, cond, and go, which the compiler understands.

The compiler finds the optimizers to apply to a form by looking for the compiler:optimizers property of the symbol that is the car of the form. The value of this property should be a list of optimizers, each of which must be a function of one argument. The compiler tries each optimizer in turn, passing the form to be optimized as the argument. An optimizer that returns the original form unchanged (eq to the argument) has "done nothing", and the next optimizer is tried. If the optimizer returns anything else, it has "done something", and the whole process starts over again.

Optimizers should not be used to define new language features, because they only take effect in the compiler; the interpreter (that is, the evaluator) doesn't know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory or something. That is why they are called optimizers. In principle, the code ought to compile just as correctly if the optimizer is eliminated.

compiler:add-optimizer *function optimizer optimized-into...* *Macro*

Puts *optimizer* on *function*'s optimizers list if it isn't there already. *optimizer* is the name of an optimization function, and *function* is the name of the function calls which are to be processed. Neither is evaluated.

(compiler:add-optimizer *function optimizer optimize-into-1 optimize-into-2...*) also remembers *optimize-into-1*, etc., as names of functions which may be called in place of *function* as a result of the optimization. Then who-calls of *function* will also mention callers of *optimize-into-1*, etc.

compiler:defoptimizer *Macro*

function optimizer-name (optimizes-into...) lambda-list body...

Defines an optimizer and installs it. Equivalent to

```
(progn
  (defun optimizer-name lambda-list
    body...)
  (compiler:add-optimizer function optimizer-name
                          optimizes-into...))
```

compiler:defcompiler-synonym *function for-function* *Macro*

Makes *function* a synonym for *for-function* in code being compiled. Example:

```
(compiler:defcompiler-synonym plus +)
```

is how the compiler is told how to compile **plus**.

17.7 Maclisp Compatibility

Certain programs are intended to be run both in Maclisp and in Zetalisp. Their source files need some special conventions. For example, all **special** declarations must be enclosed in **declare**'s, so that the Maclisp compiler will see them. The main issue is that many functions and special forms of Zetalisp do not exist in Maclisp. It is suggested that you turn on **run-in-maclisp-switch** in such files, which will warn you about a lot of problems that your program may have if you try to run it in Maclisp.

The macro-character combination **# + lispm** causes the object that follows it to be visible only when compiling for Zetalisp. The combination **# + maclisp** causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros **if-for-lispm** and **if-for-maclisp** around them. The **if-for-lispm** macro turns off **run-in-maclisp-switch** within its object, preventing spurious warnings from the compiler. The **# + lispm** reader construct does not dare do this, since it can be used to conditionalize any object, not just a expression that will be evaluated.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

if-for-lispm *form* *Macro*

If (**if-for-lispm** *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a QFASL file intended for Zetalisp. If the Zetalisp interpreter sees this it evaluates *form* (the macro expands into *form*).

if-for-maclisp *form* *Macro*

If (**if-for-maclisp** *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp (e.g. if the compiler is COMPLR). If the Zetalisp interpreter ignores this form entirely (the macro expands into **nil**).

if-for-maclisp-else-lispm *maclisp-form lispm-form* *Macro*

If (if-for-maclisp-else-lispm *form1 form2*) is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

if-in-lispm *form* *Macro*

In Zetalisp, (if-in-lispm *form*) causes *form* to be evaluated; in Maclisp, *form* is ignored.

if-in-maclisp *form* *Macro*

In Maclisp, (if-in-maclisp *form*) causes *form* to be evaluated; in Zetalisp, *form* is ignored.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make the file start with a prelude, which should look like:

```
(eval-when (compile)
  (cond ((not (status feature lispm))
         (load '|PS:<L.SYS2>CONDIT.LISP|))))
          ;; Or other suitable filename
```

This does nothing when you compile the program on the Lisp Machine. If you compile it with the Maclisp compiler, it loads in definitions of the above macros, so that they will be available to your program. The form (status feature lispm) is generally useful in other ways; it evaluates to t when evaluated on the Lisp Machine and to nil when evaluated in Maclisp.

There are some advertised variables whose compile-time values affect the operation of the compiler. Mostly these are for Maclisp compatibility features. You can set these variables by including in his file forms such as

```
(eval-when (compile) (setq open-code-map-switch t))
```

However, these variables seem not to be needed very often.

run-in-maclisp-switch *Variable*

If this variable is non-nil, the compiler tries to warn the user about any constructs that will not work in Maclisp. By no means all Lisp Machine system functions not built in to Maclisp cause warnings; only those that could not be written by the user in Maclisp (for example, make-array, value-cell-location, etc.). Also, lambda-list keywords such as &optional and initialized prog variables are be mentioned. This switch also inhibits the warnings for obsolete Maclisp functions. The default value of this variable is nil.

obsolete-function-warning-switch *Variable*

If this variable is non-nil, the compiler tries to warn the user whenever an obsolete Maclisp-compatibility function such as maknam or samepnamep is used. The default value is t.

allow-variables-in-function-position-switch *Variable*

If this variable is non-nil, the compiler allows the use of the name of a variable in function position to mean that the variable's value should be funcall'ed. This is for compatibility with old Maclisp programs. The default value of this variable is nil.

open-code-map-switch *Variable*

If this variable is non-nil, the compiler attempts to produce inline code for the mapping functions (mapc, mapcar, etc., but not mapatoms) if the function being mapped is an anonymous lambda-expression. The generated code is faster but larger. The default value is t.

If you want to turn off open coding of these functions, it is preferable to use (declare (notinline mapc mapcar ...)).

inhibit-style-warnings-switch *Variable*

If this variable is non-nil, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings, won't-run-in-Maclisp warnings, and other sorts of warnings. The default value is nil. See also the inhibit-style-warnings macro, which acts on one level only of an expression.

compiler-let ((*variable value*)...) *body*... *Macro*

Allows local rebinding of global switches that affect either compilation or the behavior of user-written macros. Its syntax is like that of let, and in the interpreter it is identical to let. When encountered in compiled code, the variables are bound around the compilation of *body* rather than around the execution at a later time of the compiled code for *body*. For example,

Example:

```
(compiler-let ((open-code-map-switch nil))
  (mapc (function (lambda (x) ...)) foo))
```

prevents the compiler from open-coding the mapc.

The same results can be obtained more cleanly using declare. User-written macros can examine the declarations using getdecl.

The next three functions are primarily for Maclisp compatibility. In Maclisp, they are declarations, used within a declare at top level in the file.

***expr** *symbol*... *Special form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

***lexpr** *symbol*... *Special form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

***fexpr** *symbol*... *Special form*

Declares each *symbol* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

17.8 Putting Data in QFASL Files

It is possible to make a QFASL file containing data, rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in printed representations. Also, certain data structures such as arrays do not have a convenient printed representation as text, but can be saved in QFASL files. For example, the system stores fonts this way. Each font is in a QFASL file (on the SYS: FONTS; directory) that contains the data structures for that font. When the file is loaded, the symbol that is the name of the font gets set to the array that represents the font. Putting data into a QFASL file is often referred to as "*fasdumping* the data".

In compiled programs, the constants are saved in the QFASL file in this way. The compiler optimizes by making constants that are **equal** become **eq** when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a QFASL file is loaded, objects that were **eq** when the file was written are still **eq**; this does not normally happen with text files.

The following types of objects can be represented in QFASL files: Symbols (uninterned or uninterned), numbers of all kinds, lists, strings, arrays of all kinds, named structures, instances, and FEFs.

:fasd-form *Operation on instances*

When an instance is fasdumped (put into a QFASL file), it is sent a :fasd-form message, which must return a Lisp form that, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply fasdumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain **eq**; the :fasd-form message is only sent the first time a particular instance is encountered during writing of a QFASL file. If the instance does not accept the :fasd-form message, it cannot be fasdumped.

Loading a QFASL file in which a named structure has been fasdumped creates a new named structure with components identical to those of the one that was dumped. Then the :fasd-fixup operation is invoked, which gives the new structure the opportunity to correct its contents if they are not supposed to be just the same as what was dumped.

The meaning of a QFASL file is greatly affected by the package used for loading it. Therefore, the file itself says which package to use.

In **dump-forms-to-file**, you can specify the package to use by including a :package attribute in the *attribute-list* argument. For example, if that argument is the list (:package "SI") then the file is dumped and loaded in the si package. If the package is not specified in this way, **user** is used. The other fasdumping functions always use **user**.

dump-forms-to-file *filename forms-list* &optional *attribute-list*

Writes a QFASL file named *filename* which contains, in effect, the forms in *forms-list*. That is to say, when the file is loaded, its effect will be the same as evaluating those forms.

Example:

```
(dump-forms-to-file "foo" '((setq x 1) (setq y 2)))
(load "foo")
x => 1
y => 2
```

attribute-list is the file attribute list to store in the QFASL file. It is a list of alternating keywords and values, and corresponds to the -*- line of a source file. The most useful keyword in this context is :package, whose value in the attribute list specifies the package to be used both in dumping the forms and in loading the file. If no :package keyword is present, the file will be loaded in whatever package is current at the time.

compiler:fasd-symbol-value *filename symbol*

Writes a QFASL file named *filename* which contains the value of *symbol*. When the file is loaded, *symbol* will be setq'ed to the same value. *filename* is parsed and defaulted with the default pathname defaults. The file type defaults to :qfasl.

compiler:fasd-font *name*

Writes the font named *name* into a QFASL file with the appropriate name (on the SYS: FONTS; directory).

compiler:fasd-file-symbols-properties *filename symbols properties dump-values-p dump-functions-p new-symbol-function*

This is a way to dump a complex data structure into a QFASL file. The values, the function definitions, and some of the properties of certain symbols are put into the QFASL file in such a way that when the file is loaded the symbols will be setqed, fdefined, and putpropped appropriately. The user can control what happens to symbols discovered in the data structures being fasdumped.

filename is the name of the file to be written. It is defaulted with the default pathname defaults. The file type defaults to "QFASL".

symbols is a list of symbols to be processed. *properties* is a list of properties which are to be fasdumped if they are found on the symbols. *dump-values-p* and *dump-functions-p* control whether the values and function definitions are also dumped.

new-symbol-function is called whenever a new symbol is found in the structure being dumped. It can do nothing, or it can add the symbol to the list to be processed by calling compiler:fasd-symbol-push. The value returned by *new-symbol-function* is ignored.

17.9 Analyzing QFASL Files

QFASL files are composed of 16-bit nibbles. The first two nibbles in the file contain fixed values, which are there so the system can tell a proper QFASL file. The next nibble is the beginning of the first *group*. A group starts with a nibble that specifies an operation. It may be followed by other nibbles that are arguments.

Most of the groups in a QFASL file are there to construct objects when the file is loaded. These objects are recorded in the *fasl-table*. Each time an object is constructed, it is assigned the next sequential index in the fasl-table. The indices are used by other groups later in the file, to refer back to objects already constructed.

To prevent the fasl-table from becoming too large, the QFASL file can be divided into *whacks*. The fasl-table is cleared out at the beginning of each whack.

The other groups in the QFASL file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

If you are having trouble with a QFASL file and want to find out exactly what it does when it is loaded, you can use UNFASL to find out.

si:unfasl-print *input-file-name*
Prints on *standard-output* a description of the contents of the QFASL file *input-file-name*.

si:unfasl-file *input-file-name* &optional *output-file-name*
Writes a description of the contents of the QFASL file *input-file-name* into the output file. The output file type defaults to :unfasl and the rest of the pathname defaults from *input-file-name*.