CHAPTER 12

THE COMPILER

The compiler is contained in the standard Interlisp system. It may be used to compile functions defined in the user's Interlisp system, or to compile definitions stored in a file. The resulting compiled code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to use one of the file package functions, such as MAKEFILE (page 11.6), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in the user's Interlisp system, by directly calling the compiler using functions such as COMPILE (page 12.10). No matter how the compiler is called, the function COMPSET is called which asks the user certain questions concerning the compilation. (COMPSET sets the free variables LAPFLG, STRF, SVFLG, LCFIL and LSTFIL which determine various modes of operation.) Those that can be answered ''yes''or ''no'' can be answered with YES, Y, or T for ''yes''; and NO, N, or NIL for ''no''. The questions are:

LISTING? This asks whether to generate a listing of the compiled code. The LAP and machine code are usually not of interest but can be helpful in debugging macros. Possible answers are:

- 1 Prints output of pass 1, the LAP macro code.
- 2 Prints output of pass 2, the machine code.
- YES Prints output of both passes.
- NO Prints no listings.

The variable LAPFLG is set to the answer.

FILE: This question (which only appears if the answer to LISTING? is affirmative) ask where the compiled code listing(s) should be written. Answering T will print the listings at the terminal. The variable LSTFIL is set to the answer.

REDEFINE? This question asks whether the functions compiled should be redefined to their compiled definitions. If this is answered YES, the compiled code is stored and the function definition changed, otherwise the function definition remains unchanged.

The variable STRF is set to T (if this is answered YES) or NIL.

SAVE EXPRS? This question asks whether the original defining EXPRs of functions should be saved. If answered YES, then before redefining a function to its compiled definition, the EXPR definition is saved on the property list of the function name. Otherwise they are discarded.

It is very useful to save the EXPR definitions, just in case the compiled function needs to be changed. The editing functions will retrieve this saved definition if it

exists, rather than reading from a source file.

The variable SVFLG is set to T (if this is answered YES) or NIL.

OUTPUT FILE? This question asks whether (and where) the compiled definitions should be written into a file for later loading. If you answer with the name of a file, that file will be used. If you answer Y or YES, you will be asked the name of the file. If the file named is already open, it will continue to be used. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done.

The variable LCFIL is set to the name of the file.

In order to make answering these questions easier, there are four other possible answers to the LISTING? question, which specify common compiling modes:

S Same as last setting. Uses the same answers to compiler questions as given for the last compilation.

F Compile to File, without redefining functions.

ST STore new definitions, saving EXPR definitions.

STF STore new definitions; Forget EXPR definitions.

Implicit in these answers are the answers to the questions on disposition of compiled code and EXPR definitions, so the questions REDEFINE? and SAVE EXPRS? would not be asked if these answers were given. OUTPUT FILE? would still be asked, however. For example:

```
_COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.DCOM
(FACT COMPILING)
.
.
(FACT REDEFINED)
.
.
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
_
```

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on the file FACT.DCOM for subsequent loading.

12.1 COMPILER PRINTOUT

In Interlisp-D, for each function FN compiled, whether by TCOMPL, RECOMPILE, or COMPILE, the compiler prints:

(FN (ARG$_1$ ... ARG$_N$) (uses: VAR$_1$ ... VAR$_N$) (calls: FN$_1$ ... FN$_N$))

The message is printed at the beginning of the second pass of the compilation of FN. (ARG$_1$ ... ARG$_N$) is the list of arguments to FN; following "uses:" are the free variables referenced or set in FN (not including global variables); following "calls:" are the undefined functions called within FN.

In Interlisp-10, for every function compiled, the compiler prints (FN (ARG$_1$ ... ARG$_N$) (FREE$_1$ ... FREE$_N$)), where FREE$_1$ ... FREE$_N$ are the free variables referenced or set in FN.

If the compilation of FN causes the generation of one or more auxilary functions (see page 12.8), a compiler message will be printed for these functions before the message for FN, e.g.,

```
(FOOA0027 (X) (uses: XX))
(FOO (A B))
```

When compiling a block, the compiler first prints (BLKNAME ... BLKFN$_1$ BLKFN$_2$ ...). Then the normal message is printed for the entire block. The names of the arguments to the block are generated by suffixing "#" and a number to the block name, e.g., (FOOBLOCK (FOOBLOCK#0 FOOBLOCK#1) FREE-VARIABLES). Then a message is printed for each *entry* to the block.

In addition to the above output, both RECOMPILE and BRECOMPILE print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

The compiler prints out error messages when it encounters problems compiling a function. For example:

```
----- In BAZ:
***** (BAZ - illegal RETURN)
-----
```

The above error message indicates that an "illegal RETURN" compiler error occurred while trying to compile the function BAZ. Some compiler errors cause the compilation to terminate, producing nothing; however, there are other compiler errors which do not stop compilation. The compiler error messages are described on page 12.20.

Compiler printout and error messages go to the file COUTFILE, initially T. COUTFILE can also be set to the name of a file opened for output, in which case all compiler printout will go to COUTFILE, i.e. the compiler will compile "silently." However, any error messages will be printed to both COUTFILE as well as T.

12.2 GLOBAL VARIABLES

Variables that appear on the list GLOBALVARS, or have the property GLOBALVAR with value T, or are declared with the GLOBALVARS file package command (page 11.25), are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of this variable is equivalent to (GETTOPVAL (QUOTE VARIABLE)), regardless of whether or not it is bound in the current access chain. Similarly, (SETQ VARIABLE VALUE) will compile as (SETTOPVAL (QUOTE VARIABLE) VALUE).

All system parameters, unless otherwise specified, are declared as global variables. Thus, *rebinding* these variables in a deep bound system (like Interlisp-D) will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, the user might write

```
(SETQ GLOBALVARIABLE NEWVALUE)
FORM
(SETQ GLOBALVARIABLE OLDVALUE)
```

Note that in this case, if an error occurred during the evaluation of `FORM`, or a control-D was typed, the global variable would not be restored to its original value. The function `RESETVAR` (page 9.20) provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or control-D is typed.

Note: Interlisp-10 employs a shallow binding scheme as described on page 7.1. There is no distinction between global variables and other types of variables: all variable references are to the variable's value cell. Thus, the cost of *accessing* a variable is small and independent of the depth of computation, whereas in a deep bound system, it can be expensive to search the stack for the most recent binding of a variable, hence the need for a mechanism like global variables. Note however that in a shallow bound system, the cost of rebinding a variable is somewhat higher than in a deep bound system (except when the variable is a `LOCALVAR`). For the purposes of compilation, global variables are treated the same as `SPECVARS`, i.e. their names are always visible on the stack when they are rebound.

12.3 LOCALVARS AND SPECVARS

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable's name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

`LOCALVARS` and `SPECVARS` are variables that affect compilation. During regular compilation, `SPECVARS` is normally `T`, and `LOCALVARS` is `NIL` or a list. This configuration causes all variables bound in the functions being compiled to be treated as special *except* those that appear on `LOCALVARS`. During block compilation, `LOCALVARS` is normally `T` and `SPECVARS` is `NIL` or a list. All variables are then treated as local *except* those that appear on `SPECVARS`.

Declarations to set `LOCALVARS` and `SPECVARS` to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope.

(1) The declarations may be included in the filecoms of a file, by using the `LOCALVARS` and `SPECVARS` file package commands (page 11.25). The scope of the declaration is then the entire file:

```
(LOCALVARS . T) (SPECVARS X Y)
```

(2) The declarations may be included in block declarations; the scope is then the block, e.g.,

```
(BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS X)))
```

(3) The declarations may also appear in individual functions, or in PROG's or LAMBDA's within a function, using the DECLARE function. In this case, the scope of the declaration is the function or the PROG or LAMBDA in which it appears. LOCALVARS and SPECVARS declarations must appear immediately after the variable list in the function, PROG, or LAMBDA, but intervening comments are permitted. For example:

```
(DEFINEQ ((FOO
          (LAMBDA (X Y)
              (DECLARE (LOCALVARS Y))
              (PROG (X Y Z)
                    (DECLARE (LOCALVARS X))
                    ... ]
```

If the above function is compiled (non-block), the outer X will be special, the X bound in the PROG will be local, and both bindings of Y will be local.

Declarations for LOCALVARS and SPECVARS can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
          (DECLARE (SPECVARS . T))
          (PROG (Z) ... ]
```

will cause X, Y, and Z to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
          (DECLARE (SPECVARS X))
          ... ]
```

will make X a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be T, and therefore *both* X and Y will be specvars by default.

Although LOCALVARS and SPECVARS declarations have the same form as other components of block declarations such as (LINKFNS . T), their operation is somewhat different because the two variables are not independent. (SPECVARS . T) will cause SPECVARS to be set to T, and LOCALVARS to be set to NIL. (SPECVARS V1 V2 ...) will have *no* effect if the value of SPECVARS is T, but if it is a list (or NIL), SPECVARS will be set to the union of its prior value and (V1 V2 ...). The operation of LOCALVARS is analogous. Thus, to affect both modes of compilation one of the two (LOCALVARS or SPECVARS) must be declared T before specifying a list for the other.

12.4 CONSTANTS

The function CONSTANT enables the user to define certain expressions as descriptions of their ''constant'' values. For example, if a user program needed a scratch list of length 30, the user could specify

(CONSTANT (to 30 collect NIL)) instead of (QUOTE (NIL NIL)). The former is more concise and displays the important parameter much more directly than the latter. CONSTANT can also be used to denote values that cannot be quoted directly, such as (CONSTANT (PACK NIL)), (CONSTANT (ARRAY 10)). It is also useful to parameterize quantities that are constant at run time but may di er at compile time, e.g. (CONSTANT BITSPERWORD) in a program is exactly equivalent to 36, if the variable BITSPERWORD is bound to 36 when the CONSTANT expression is evaluated at compile time.

When interpreted, the expression occuring as the argument to CONSTANT is evaluted each time it is encountered. If the CONSTANT form is compiled, however, the expression will be evaluated only once:

If the value of the expression has a readable print-name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's de nition, as if (QUOTE VALUE-OF-EXPRESSION) had appeared instead of (CONSTANT EXPRESSION).

If the value does not have a readable printname (e.g. the PACK and ARRAY examples above), then the expression itself will be saved with the function, and it will be evaluated when the function is rst executed. The value will then be stored in the function's literals, and will be retrieved on future references.

Whereas the function CONSTANT attempts to evaluate the expression as soon as possible (compile-time, load-time, or rst-run-time), the function DEFERREDCONSTANT will always defer the evaluation until rst running. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked.

Note: The function SELECTC (page 4.3) provides a mechanism for conparing a value to a number of constants.

(CONSTANTS VAR_1 VAR_2 ... VAR_N) [NLambda NoSpread Function]

De nes VAR_1, ... VAR_N (unevaluated) to be compile-time constants. Whenever the compiler encounters a (free) reference to one of these constants, it will compile the form (CONSTANT VAR_i) instead.

If VAR_i is a list of the form (VAR FORM), a free reference to the variable will compile as (CONSTANT FORM).

Constants can be saved using the CONSTANTS le package command (page 11.27).

12.5 COMPILING FUNCTION CALLS

When compiling the call to a function, the compiler must know the type of the function, to determine how the arguments should be prepared (evaluated/unevaluated, spread/nospread). There are three seperate cases: lambda, nlambda spread, and nlambda nospread functions.

To determine which of these three cases is appropriate, the compiler will rst look for a de nition among the functions in the le that is being compiled. The function can be de ned anywhere in any of the les given as arguments to BCOMPL, TCOMPL, BRECOMPILE or RECOMPILE. If the function is not contained in the le, the compiler will look for other information in the variables NLAMA, NLAML, and LAMS, which can be set by the user:

NLAMA [Variable]

(for NLAMbda Atoms) A list of functions to be treated as nlambda nospread functions by the compiler.

NLAML [Variable]

(for NLAMbda List) A list of functions to be treated as nlambda spread functions by the compiler.

LAMS [Variable]

A list of functions to be treated as lambda functions by the compiler. Note that including functions on LAMS is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

If the function is not contained in a file, or on the lists NLAMA, NLAML, or LAMS, the compiler will look for a current definition in the Interlisp system, and use its type. If there is no current definition, next COMPILEUSERFN is called:

COMPILEUSERFN [Variable]

When compiling a function call, if the function type cannot be found by looking in files, the variables NLAMA, NLAML, or LAMS, or at a current definition, then if the value of COMPILEUSERFN is not NIL, the compiler calls (the value of) COMPILEUSERFN giving it as arguments CDR of the form and the form itself, i.e., the compiler does (APPLY* COMPILEUSERFN (CDR FORM) FORM). If a non-NIL value is returned, it is compiled instead of FORM. If NIL is returned, the compiler compiles the original expression as a call to a lambda spread that is not yet defined.

Note that COMPILEUSERFN is only called when the compiler encounters a *list* CAR of which is not the name of a defined function. The user can instruct the compiler about how to compile other data types via COMPILETYPELST, page 12.9.

CLISP uses COMPILEUSERFN to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs (See page 12.9).

If the compiler cannot determine the function type by any of the means above, it assumes that the function is a lambda function, and its arguments are to be evaluated. The function is also added to the value of ALAMS:

ALAMS [Variable]

(for Assumed LAMbdaS) A list of functions to that the compiler has assumed to be lambda functions. ALAMS is not used by the compiler; it is maintained for the user's benefit so that the user can check to see whether any incorrect assumptions were made.

If there are nlambda functions called from the functions being compiled, and they are only defined in a separate file, they must be included on NLAMA or NLAML, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not ''know'' about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the

compiler will automatically handle calls to that function correctly.

12.6 FUNCTION AND FUNCTIONAL ARGUMENTS

Compiling the function FUNCTION (page 5.15) may involve creating and compiling a seperate ''auxiliary function'', which will be called at run time. An auxiliary function is named by attaching a GENSYM (page 2.11) to the end of the name of the function in which they appear, e.g., FOOA0003. For example, suppose FOO is de ned as (LAMBDA (X) (FOO1 X (FUNCTION))) and compiled. When FOO is run, FOO1 will be called with two arguments, X, and FOOA000N and FOO1 will call FOOA000N each time it uses its functional argument.

Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.). Note that a considerable savings in time could be achieved by making FOO1 compile open via a computed macro (page 5.17), e.g.

```
(Z (LIST (SUBST (CADADR Z)
                (QUOTE FN)
                DEF )
         (CAR Z)))
```

DEF is the de nition of FOO1 as a function of just its rst argument, and FN is the name used for its functional argument in its de nition. In this case, (FOO1 X (FUNCTION)) would compile as an expression, containing the argument to FUNCTION as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and the user must decide which is more important.

12.7 OPEN FUNCTIONS

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. If the amount of time spent *inside* the function is small, this function calling time will be a signi cant percentage of the total time required to use the function. Therefore, many ''small'' functions, e.g., CAR, CDR, EQ, NOT, CONS are always compiled ''open'', i.e., they do not result in a function call. Other larger functions such as PROG, SELECTQ, MAPC, etc. are compiled open because they are frequently used. The user can make other functions compile open via MACRO de nitions (see page 5.17). The user can also a ect the compiled code via COMPILEUSERFN (page 12.7) and COMPILETYPELST (page 12.9).

12.8 COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (literal atoms). The user can a ect the compiler's behaviour with respect to lists and literal atoms in a number of ways,

e.g. macros, declarations, COMPILEUSERFN (page 12.7), etc. COMPILETYPELST allows the user to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to DEFEVAL (page 5.11) for the interpreter.

COMPILETYPELST [Variable]

A list of elements of the form (TYPENAME . FUNCTION). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type name of the datum is looked up on COMPILETYPELST. If an entry appears CAR of which is equal to the type name, CDR of that entry is applied to the datum. If the value returned by this application is *not* EQ to the datum, then that value is compiled instead. If the value *is* EQ to the datum, or if there is no entry on COMPILETYPELST for this type name, the compiler simply compiles the datum as (QUOTE DATUM).

12.9 COMPILING CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the de nitions must rst be DWIMIFYed (page 16.14). The user can automate this process in several ways:

(1) If the variable DWIMIFYCOMPFLG is T, the compiler will always DWIMIFY expressions before compiling them. DWIMIFYCOMPFLG is initially NIL.

(2) If a le has the property FILETYPE with value CLISP on its property list, TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE will operate as though DWIMIFYCOMPFLG is T and DWIMIFY all expressions before compiling.

(3) If the function de nition has a local CLISP declaration (see page 16.10), including a null declaration, i.e., just (CLISP:), the de nition will be automatically DWIMIFYed before compiling.

Note: COMPILEUSERFN (page 12.7) is de ned to call DWIMIFY on iterative statements, IF-THEN statements, and fetch, replace, and match expressions, i.e., any CLISP construct which can be recognized by its CAR of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, IF statements, etc., the function does not have to be dwimi ed before compiling.

If DWIMIFY is ever unsuccessful in processing a CLISP expression, it will print the error message UNABLE TO DWIMIFY followed by the expression, and go into a break.[8] The user can exit the break in several di erent ways: (1) type OK to the break, which will cause the compiler to try again, e.g. the user could de ne some missing records while in the break, and then continue; or (2) type ^, which will cause the compiler to simply compile the expression as is, i.e. as though CLISP had not been enabled in the rst place; or (3) return an expression to be compiled in its place by using the RETURN break command (page 9.3).

Note: TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE all scan the entire le before doing any compiling, and take note of the names of all functions that are de ned in the le as well as the names of all variables that are set by adding them to NOFIXFNSLST and NOFIXVARSLST, respectively. Thus,

[8]unless DWIMESSGAG = T. In this case, the expression is just compiled as is, i.e. as though clisp had not been enabled.

if a function is not currently defined, but *is* defined in the file being compiled, when DWIMIFY is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as CAR of a form. DWIMIFY also takes into account variables that have been declared to be LOCALVARS, or SPECVARS, either via block declarations or DECLARE expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration USEDFREE may also be used to declare variables simply used freely in a function. These variables will also be left alone by DWIMIFY. Finally, NOSPELLFLG (page 15.12) is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

12.10 COMPILER FUNCTIONS

Normally, the compiler is envoked through file package commands that keep track of the state of functions, and manage a set of files, such as MAKEFILE (page 11.6). However, it is also possible to explicitly call the compiler using one of a number of functions. Functions may be compiled from in-core definitions (via COMPILE), or from definitions in files (TCOMPL), or from a combination of in-core and file definitions (RECOMPILE).

TCOMPL and RECOMPILE produce ''compiled'' files. Compiled files usually have the same name as the symbolic file they were made from, suffixed with DCOM (Interlisp-D) or COM (Interlisp-10).[9] The file name is constructed from the name field only, e.g., (TCOMPL '<BOBROW>FOO.TEM;3) produces FOO.DCOM on the connected directory. The version number will be the standard default.

A ''compiled file'' contains the same expressions as the original symbolic file, except that (1) a special FILECREATED expression appears at the front of the file which contains information used by the file package, and which causes the message COMPILED ON DATE to be printed when the file is loaded; [10] (2) every DEFINEQ in the symbolic file is replaced by the corresponding compiled definitions in the compiled file; and (3) expressions following a DONTCOPY tag inside of a DECLARE: (page 11.26) that appears in the symbolic file are not copied to the compiled file. The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, *regardless of where they appear in the symbolic file.* The only exceptions are expressions that follow a FIRST tag inside of a DECLARE: (page 11.26). This ''compiled'' file can be loaded into any Interlisp system with LOAD (page 11.4).

Note: When a function is compiled from its in-core definition (as opposed to being compiled from a definition in a file), and the function has been modified by BREAK, TRACE, BREAKIN, or ADVISE, it is first restored to its original state, and a message is printed out, e.g., FOO UNBROKEN. If the function is not defined as an EXPR, the value of the function's EXPR property is used for the compilation, if there is one. If there is no EXPR property, and the compilation is being performed by RECOMPILE, the definition of the function is obtained from the file (using LOADFNS). Otherwise, the compiler prints (FN NOT COMPILEABLE), and goes on to the next function.

(COMPILE X FLG) [Function]

X is a list of functions (if atomic, (LIST X) is used). COMPILE first asks the standard compiler questions, and then compiles each function on X, using its in-core definition. Returns X.

---

[9] The compiled file suffix is stored as the value of the variable COMPILE.EXT.

[10] The actual string printed is the value of COMPILEHEADER, initially "compiled on".

If compiled de nitions are being written to a le, the le is closed unless FLG = T.

(COMPILE1 FN DEF _) [Function]

Compiles DEF , rede ning FN if STRF= T (STRF is one of the variables set by COMPSET, page 12.1). COMPILE1 is used by COMPILE, TCOMPL, and RECOMPILE.

If DWIMIFYCOMPFLG is T, or DEF contains a CLISP declaration, DEF is dwimi ed before compiling. See page 12.9.

(TCOMPL FILES) [Function]

TCOMPL is used to ''compile les''; given a symbolic LOAD le (e.g., one created by MAKEFILE), it produces a ''compiled le''. FILES is a list of symbolic les to be compiled (if atomic, (LIST FILES) is used). TCOMPL asks the standard compiler questions, except for ''OUTPUT FILE:''. The output from the compilation of each symbolic le is written on a le of the same name su xed with DCOM, e.g., (TCOMPL '(SYM1 SYM2)) produces two les, SYM1.DCOM and SYM2.DCOM.

TCOMPL processes the les one at a time, reading in the entire le. For each FILECREATED expression, the list of functions that were marked as changed by the le package is noted, and the FILECREATED expression is written onto the output le. For each DEFINEQ expression, TCOMPL adds any nlambda functions de ned in the DEFINEQ to NLAMA or NLAML, and adds lambda functions to LAMS, so that calls to these functions will be compiled correctly (see page 12.7).[11] Expressions beginning with DECLARE: are processed specially (see page 11.26). All other expressions are collected to be subsequently written onto the output le.

After processing the le in this fashion, TCOMPL compiles each function, except for those functions which appear on the list DONTCOMPILEFNS,[12] and writes the compiled de nition onto the output le. TCOMPL then writes onto the output le the other expressions found in the symbolic le.

Note: If the rootname of a le has the property FILETYPE with value CLISP, or value a list containing CLISP, TCOMPL rebinds DWIMIFYCOMPFLG to T while compiling the functions on FILE, so the compiler will DWIMIFY all expressions before compiling them. See page 12.9.

TCOMPL returns a list of the names of the output les. All les are properly terminated and closed. If the compilation of any le is aborted via an error or control-D, all les are properly closed, and the (partially complete) compiled le is deleted.

(RECOMPILE PFILE CFILE FNS) [Function]

The purpose of RECOMPILE is to allow the user to update a compiled le without recompiling every function in the le. RECOMPILE does this by using the results of

[11] NLAMA, NLAML, and LAMS are rebound to their top level values (using RESETVAR) by TCOMPL, RECOMPILE, BCOMPL, BRECOMPILE, COMPILE, and BLOCKCOMPILE, so that any additions to these lists while inside of these functions will not propagate outside.

[12] Initially NIL. DONTCOMPILEFNS might be used for functions that compile open, since their de nitions would be super uous when operating with the compiled le. Note that DONTCOMPILEFNS can be set via block declarations (see page 12.14).

a previous compilation. It produces a compiled file similar to one that would have been produced by TCOMPL, but at a considerable savings in time by only compiling selected functions, and copying the compiled definitions for the remainder of the functions in the file from an earlier TCOMPL or RECOMPILE file.

PFILE is the name of the Pretty file (source file) to be compiled; CFILE is the name of the Compiled file containing compiled definitions that may be copied. FNS indicates which functions in PFILE are to be recompiled, e.g., have been changed or defined for the first time since CFILE was made. Note that PFILE, not FNS, drives RECOMPILE.

RECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with TCOMPL, the output automatically goes to PFILE.DCOM. RECOMPILE processes PFILE the same as does TCOMPL except that DEFINEQ expressions are not actually read into core. Instead, RECOMPILE uses the filemap (see page 11.38) to obtain a list of the functions contained in PFILE. The filemap enables RECOMPILE to skip over the DEFINEQs in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-DEFINEQs and the processing of the DECLARE: expressions as with TCOMPL). A map is built if the symbolic file does not already contain one, for example if it was written in an earlier system, or with BUILDMAPFLG = NIL (page 11.39).

After this initial scan of PFILE, RECOMPILE then processes the functions defined in the file. For each function in PFILE, RECOMPILE determines whether or not the function is to be (re)compiled. Functions that are members of DONTCOMPILEFNS are simply ignored. Otherwise, a function is recompiled if (1) FNS is a list and the function is a member of that list; or (2) FNS = T or EXPRS and the function is an EXPR; or (3) FNS = CHANGES and the function is marked as having been changed in the FILECREATED expression in PFILE; or (4) FNS = ALL.[13] If a function is not to be recompiled, RECOMPILE obtains its compiled definition from CFILE, and copies it (and all generated subfunctions) to the output file, PFILE.DCOM. If the function does not appear on CFILE, RECOMPILE simply recompiles it. Finally, after processing all functions, RECOMPILE writes out all other expressions that were collected in the prescan of PFILE.

If CFILE = NIL, PFILE.DCOM (the old version of the output file) is used for copying *from*. If both FNS and CFILE are NIL, FNS is set to the value of RECOMPILEDEFAULT, which is initially EXPRS. This is the most common usage. Typically, the functions the user has changed will have been UNSAVEDEFed by the editor, and therefore will be EXPRs. Thus the user can perform his edits, dump the file, and then simply (RECOMPILE 'FILE) to update the compiled file.

The value of RECOMPILE is the new compiled file, PFILE.DCOM. If RECOMPILE is aborted due to an error or control-D, the new (partially complete) compiled file will be closed and deleted.

[13] If FNS = ALL, CFILE is superfluous, and does not have to be specified. This option may be used to compile a symbolic file that has never been compiled before, but which has already been loaded (since using TCOMPL would require reading the file in a second time).

RECOMPILE is designed to allow the user to conveniently and *e ciently* update a compiled le, even when the corresponding symbolic le has not been (completely) loaded. For example, the user can perform a LOADFROM (page 11.6) to ''notice'' a symbolic le, edit the functions he wants to change (the editor will automatically load those functions not already loaded), call MAKEFILE (page 11.6) to update the symbolic le (MAKEFILE will copy the unchanged functions from the old symbolic le), and then perform (RECOMPILE PFILE).

Note: Since PRETTYDEF automatically outputs a suitable DECLARE: expression to indicate which functions in the le (if any) are de ned as NLAMBDAs, calls to these functions will be handled correctly, even though the NLAMBDA functions themselves may never be loaded, or even looked at, by RECOMPILE.

12.11 BLOCK COMPILING

Block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times, e.g., EQUAL, COPY, and COUNT are block compiled in Interlisp- 10.

The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls, except that they are usually linked (see page 12.18). A block can be entered via several di erent functions, called entries.[14] These must be speci ed when the block is compiled. For example, the error block has three entries, ERRORX, INTERRUPT, and FAULT1. Similarly, the compiler block has nine entries.

Note: In Interlisp- D, block compiling is handled somewhat di erently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Interlisp- D works by automatically renaming the block functions with special names, and calling these functions with the normal function- calling mechanisms. Speci cally, a function FN is renamed to \BLOCK- NAME /FN. For example, function FOO in block BAR is renamed to ''\BAR/FOO''. Note that it is possible with this scheme to break functions internal to a block.

12.11.1 RETFNS

Another savings in block compilation arises from omitting most of the information on the stack about internal calls between functions in the block. However, if a function's name must be visible on the stack, e.g., if the function is to be returned from RETFROM, RETTO, RETEVAL, etc., it must be included on the list RETFNS.

[14] Actually the block is entered the same as every other function, i.e., at the top. However, the entry functions call the main block with their name as one of its arguments, and the block dispatches on the name, and jumps to the portion of the block corresponding to that entry point. The e ect is thus the same as though there were several di erent entry points.

12.11.2 BLKAPPLYFNS

Normally, a call to APPLY from inside a block would be the same as a call to any other function outside of the block. If the first argument to APPLY turned out to be one of the entries to the block, the block would have to be reentered. BLKAPPLYFNS enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list BLKAPPLYFNS those functions which will be called in this fashion, and by using BLKAPPLY in place of APPLY, and BLKAPPLY* in place of APPLY*. If BLKAPPLY or BLKAPPLY* is given a function not on BLKAPPLYFNS, the effect is the same as a call to APPLY or APPLY* and no error is generated. Note however, that BLKAPPLYFNS must be set at *compile* time, not run time, and furthermore, that all functions on BLKAPPLYFNS must be in the block, or an error is generated (at compile time), NOT ON BLKFNS.

12.11.3 BLKLIBRARY

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an EXPR definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list BLKLIBRARY, and their EXPR definitions on the property list of the functions under the property BLKLIBRARYDEF. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on BLKLIBRARY, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block. The functions ASSOC, EQUAL, GETPROP, LAST, LENGTH, LISPXWATCH, MEMB, MEMBER, NCONC1, NLEFT, NTH, /RPLNODE, and TAILP already have BLKLIBRARYDEF properties.

12.11.4 Block Declarations

Block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, linking, etc. To help with this, there is the BLOCKS file package command (page 11.25), which has the form:

(BLOCKS $BLOCK_1$ $BLOCK_2$ ... $BLOCK_N$)

where each $BLOCK_i$ is a block declaration. The BLOCKS command outputs a DECLARE: expression, which is noticed by BCOMPL and BRECOMPILE. BCOMPL and BRECOMPILE are sensitive to these declarations and take the appropriate action.

Note: Masterscope includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in ENTRIES not in the block, variables that may not need to be SPECVARS because they are not used freely

below the places they are bound, etc. See page 13.1

The form of a block declaration is:

(BLKNAME BLKFN$_1$ ... BLKFN$_M$ (VAR$_1$. VALUE$_1$) ... (VAR$_N$. VALUE$_N$))

BLKNAME is the name of a block. BLKFN$_1$ ... BLKFN$_M$ are the functions in the block and correspond to BLKFNS in the call to BLOCKCOMPILE. The (VAR$_i$. VALUE$_i$) expressions indicate the settings for variables affecting the compilation of that block. If VALUE$_i$ is atomic, then VAR$_i$ is set to VALUE$_i$ (e.g. (LINKFNS . T)), otherwise VAR$_i$ is set to the UNION of VALUE$_i$ and the current value of the variable VAR$_i$. Also, expressions of the form (VAR * FORM) will cause FORM to be evaluated and the resulting list used as described above (e.g. (GLOBALVARS * MYGLOBALVARS)).

As an example, one of the block definitions for the editor is shown below. The block name is EDITBLOCK, it includes a number of functions (EDITL0, EDITL1, ... EDITH), and it sets the variables ENTRIES, SPECVARS, RETFNS, GLOBALVARS, BLKAPPLYFNS, BLKLIBRARY, and NOLINKFNS:

```
(EDITBLOCK
   EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA EDITCOML
   EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM UNDOEDITCOM1
   EDITSMASH EDITNCONC EDIT1F EDIT2F EDITNTH BPNT BPNT0
   BPNT1 RI RO LI LO BI BO EDITDEFAULT ## EDUP EDIT* EDOR
   EDRPT EDLOC EDLOCL EDIT: EDITMBD EDITXTR EDITELT
   EDITCONT EDITSW EDITMV EDITTO EDITBELOW EDITRAN TAILP
   EDITSAVE EDITH
   (ENTRIES EDITL0 ## UNDOEDITL)
   (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS **COMMENT**FLG
             PRETTYFLG UNDOLST UNDOLST1)
   (RETFNS EDITL0)
   (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS HISTORYCOMS
             EDITRACEFN)
   (BLKAPPLYFNS RI RO LI LO BI BO EDIT: EDITMBD EDITMV
             EDITXTR)
   (BLKLIBRARY LENGTH NTH LAST)
   (NOLINKFNS EDITRACEFN))
```

Whenever BCOMPL or BRECOMPILE encounter a block declaration, they rebind RETFNS, SPECVARS, GLOBALVARS, BLKLIBRARY, NOLINKFNS, LINKFNS, and DONTCOMPILEFNS to their top level values, bind BLKAPPLYFNS and ENTRIES to NIL, and bind BLKNAME to the first element of the declaration. They then scan the rest of the declaration, setting these variables as described above. When the declaration is exhausted, the block compiler is called and given BLKNAME, the list of block functions, and ENTRIES.

If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed NOT ON FILE, COMPILING IN CORE DEFINITION. Otherwise, the message NOT COMPILEABLE, is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

Note that since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After nishing all blocks, BCOMPL and BRECOMPILE treat any functions in the le that did not appear in a block declaration in the same way as do TCOMPL and RECOMPILE. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not blockcompile), with some compiler variables changed, he can use a special pseudo- block declaration of the form

(NIL BLKFN $_1$ ... BLKFN $_M$ (VAR $_1$. VALUE $_1$) ... (VAR $_N$. VALUE $_N$))

which means that BLKFN $_1$ ... BLKFN $_M$ should be compiled after rst setting VAR $_1$ ... VAR $_N$ as described above. For example,

```
(NIL CGETD FNTYP ARGLIST NARGS NCONC1 GENSYM (LINKFNS . T))
```

appearing as a ''block declaration'' will cause the six indicated functions to be compiled while LINKFNS= T so that all of their calls will be linked (except for those functions on NOLINKFNS).

12.11.5 Block Compiling Functions

There are three user level functions for block compiling, BLOCKCOMPILE, BCOMPL, and BRECOMPILE, corresponding to COMPILE, TCOMPL, and RECOMPILE. All of them ultimately call the same low level functions in the compiler, i.e., there is no ''block compiler'' per se. Instead, when block compiling, a ag is set to enable special treatment for SPECVARS, RETFNS, BLKAPPLYFNS, and for determining whether or not to link a function call. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, the user can intermix in a single le functions compiled normally, functions compiled normally with linked calls, and block compiled functions.

(BLOCKCOMPILE BLKNAME BLKFNS ENTRIES FLG) [Function]

BLKNAME is the name of a block, BLKFNS is a list of the functions comprising the block, and ENTRIES a list of entries to the block.

Each of the entries must also be on BLKFNS or an error is generated, NOT ON BLKFNS. If only one entry is speci ed, the block name can also be one of the BLKFNS , e.g., (BLOCKCOMPILE 'FOO '(FOO FIE FUM) '(FOO)). However, if more than one entry is speci ed, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

If ENTRIES is NIL, (LIST BLKNAME) is used, e.g., (BLOCKCOMPILE 'COUNT '(COUNT COUNT1))

If BLKFNS is NIL, (LIST BLKNAME) is used, e.g., (BLOCKCOMPILE 'EQUAL)

BLOCKCOMPILE asks the standard compiler questions and then begins compiling. As with COMPILE, if the compiled code is being written to a le, the le is closed unless FLG= T. The value of BLOCKCOMPILE is a list of the entries, or if ENTRIES = NIL, the value is BLKNAME .

The output of a call to BLOCKCOMPILE is one function de nition for BLKNAME , plus de nitions for each of the functions on ENTRIES if any. These entry functions

are very short functions which immediately call BLKNAME .

(BCOMPL FILES CFILE _ _) [Function]

FILES is a list of symbolic files (if atomic, (LIST FILES) is used). BCOMPL differs from TCOMPL in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files. (If you have several files to be BCOMPLed *separately*, you must make several calls to BCOMPL.) Output is to CFILE if given, otherwise to a file whose name is (CAR FILES) suffixed with DCOM. For example, (BCOMPL '(EDIT WEDIT)) produces one file, EDIT.DCOM.

BCOMPL asks the standard compiler questions, except for "OUTPUT FILE:", then processes each file exactly the same as TCOMPL (page 12.11). BCOMPL next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

If *any* of the files have property FILETYPE with value CLISP, or a list containing CLISP, then DWIMIFYCOMPFLG is rebound to T for *all* of the files. See page 12.9.

The value of BCOMPL is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

Note that it is permissible to TCOMPL files set up for BCOMPL; the block declarations will simply have no effect. Similarly, you can BCOMPL a file that does not contain any block declarations and the result will be the same as having TCOMPLed it.

(BRECOMPILE FILES CFILE FNS _) [Function]

BRECOMPILE plays the same role for BCOMPL that RECOMPILE plays for TCOMPL. Its purpose is to allow the user to update a compiled file without requiring an entire BCOMPL.

FILES is a list of symbolic files (if atomic, (LIST FILES) is used). CFILE is the compiled file produced by BCOMPL or a previous BRECOMPILE that contains compiled definitions that may be copied. The interpretation of FNS is the same as with RECOMPILE.

BRECOMPILE asks the standard compiler questions except for "OUTPUT FILE:". As with BCOMPL, output automatically goes to FILE.DCOM, where FILE is the first file in FILES.

BRECOMPILE processes each file the same as RECOMPILE (page 12.11), then processes each block declaration. If *any* of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from CFILE as with RECOMPILE. For pseudo-block declarations of the form (NIL FN1), all variable assignments are made, but only those functions indicated by FNS are recompiled.

After completing the block declarations, BRECOMPILE processes all functions that do not appear in a block declaration, recompiling those dictated by FNS, and copying the compiled definitions of the remaining from CFILE.

Finally, BRECOMPILE writes onto the output file the ''other expressions'' collected in the initial scan of FILES.

The value of BRECOMPILE is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

If CFILE = NIL, the old version of FILE.DCOM is used, as with RECOMPILE. In addition, if FNS and CFILE are both NIL, FNS is set to the value of RECOMPILEDEFAULT, initially EXPRS.

12.12 LINKED FUNCTION CALLS

Note: Linked function calls are not implemented in Interlisp-D.

Conventional (non-linked) function calls from a compiled function go through the function definition cell, i.e., the definition of the called function is obtained from its function definition cell at call time. Thus, when the user breaks, advises, or otherwise modifies the definition of the function FOO, every function that subsequently calls it instead calls the modified function. For calls from the system functions, this is clearly *not* a desirable feature. For example, suppose that the user wishes to break on basic functions such as PRINT, EVAL, RPLACA, etc., which are used by the break package. We would like to guarantee that the system packages will survive through user modification (or destruction) of basic functions (unless the user specifically requests that the system packages also be modified). This protection is achieved by linked function calls.

For linked function calls, the definition of the called function is obtained at *link* time, i.e., when the calling function is defined, and stored in the literal table of the calling function. At *call* time, this definition is retrieved from where it was stored in the literal table, *not* from the function definition cell of the called function as it is for non-linked calls.

Note that while function calls from block compiled functions are *usually* linked (i.e. the default for blocks is to link),[15] and those from standardly compiled functions are *usually* non-linked, linking function calls and blockcompiling are independent features of the Interlisp compiler, i.e., linked function calls are possible, and frequently employed, from standardly compiled functions.

Note that normal function calls require only the called function's name in the literals of the compiled code, whereas a *linked* function call uses two literals and hence produces slightly larger compiled functions.

The compiler's decision as to whether to link a particular function call is determined by the variables LINKFNS and NOLINKFNS as follows:

(1) If the function appears on NOLINKFNS, the call is not linked;

[15]In Interlisp-10, linked function calls are actually a little slower and take more space than non-linked calls, so that the user might want to include (NOLINKFNS . T) in block declarations to override the default.

(2) If block compiling and the function is one of the block functions, the call is internal as described earlier;

(3) If the function appears on LINKFNS, the call is linked;

(4) If NOLINKFNS= T, the call is not linked;

(5) If block compiling, the call is linked;

(6) If LINKFNS= T, the call is linked;

(7) Otherwise the call is not linked.

Note that (1) takes precedence over (2), i.e., if a function appears on NOLINKFNS, the call to it is *not* linked, even if it is one of the functions in the block, i.e., the call will go outside of the block.

NOLINKFNS is initialized to various system functions such as ERRORSET, BREAK1, etc. LINKFNS is initialized to NIL. Thus if the user does not specify otherwise, all calls from a block compiled function (except for those to functions on NOLINKFNS) will be linked; all calls from standardly compiled functions will not be linked. However, when compiling system functions such as HELP, ERROR, ARGLIST, FNTYP, BREAK1, et al, LINKFNS is set to T so that even though these functions are not block compiled, all of their calls will be linked.

If a function is not de ned at link time, i.e., when an attempt is made to link to it, it is linked instead to the function NOLINKDEF. When the function is later de ned, the link can be completed by relinking the calling function using RELINK described below. Otherwise, if a function is run which attempts a linked call that was not completed, NOLINKDEF is called. If the function is now de ned, i.e., it was de ned at some point after the attempt was made to link to it, NOLINKDEF will quietly perform the link and continue the call. Otherwise, it will call FAULTAPPLY and proceed as described in page 15.6.

CALLS, BREAK on FN1-IN-FN2 and ADVISE FN1-IN-FN2 all work correctly for linked function calls, e.g., (BREAK '(FOO IN FIE)), where FOO is called from FIE via a linked function call. Note that control-H will *not* interrupt at linked function calls.

12.12.1 Relinking

The function RELINK is available for relinking a compiled function, i.e., updating all of its linked calls so that they use the de nition extant at the time of the relink operation.

(RELINK FN) [Function]

FN is either the name of a function, a list of functions, an atom whose value is a list of functions, or the atom WORLD. RELINK performs the corresponding relinking operations. RELINK returns FN.

(RELINK 'WORLD) is possible because the compiled code reader maintains on LINKEDFNS a list of all user functions containing any linked calls. SYSLINKEDFNS is a list of all *system* functions that have any linked calls. (RELINK 'WORLD) performs both (RELINK LINKEDFNS) and (RELINK SYSLINKEDFNS).

Note: To relink a function in a block, one should relink the block, not the function.

It is important to stress that linking takes place when a function is *de ned* . Thus, if FOO calls FIE via a linked call, and a bug is found in FIE, changing FIE is not su cient; FOO must be relinked. Similarly, if FOO1, FOO2, and FOO3 are de ned (in that order) in a le, and each call the others via linked calls, when a new version of the le is loaded, FOO1 will be linked to the *old* FOO2 and FOO3, since those de nitions will be extant at the time it is read and de ned. Similarly, FOO2 will link to the new FOO1 and *old* FOO3. Only FOO3 will link to the new FOO1 and FOO2. The user would have to perform (RELINK '(FOO1 FOO2 FOO3)) following the LOAD.

12.13 COMPILER ERROR MESSAGES

Messages describing errors in the function being compiled are also printed on the teletype. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(FN NOT ON FILE, COMPILING IN CORE DEFINITION)
From calls to BCOMPL and BRECOMPILE.

(FN NOT COMPILEABLE)
An EXPR de nition for FN could not be found. In this case, no code is produced for FN , and the compiler proceeds to the next function to be compiled, if any.

(FN NOT FOUND) Occurs when RECOMPILE or BRECOMPILE try to copy the compiled de nition of FN from CFILE , and cannot nd it. In this case, no code is copied and the compiler proceeds to the next function to be compiled, if any.

(FN NOT ON BLKFNS)
FN was speci ed as an entry to a block, or else was on BLKAPPLYFNS, but did not appear on the BLKFNS . In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(FN CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)
In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(BLKNAME - USED BLKAPPLY WHEN NOT APPLICABLE)
BLKAPPLY is used in the block BLKNAME , but there are no BLKAPPLYFNS or ENTRIES declared for the block.

(VAR SHOULD BE A SPECVAR - USED FREELY BY FN)
In Interlisp- 10, while compiling a block, the compiler has already generated code to bind VAR as a LOCALVAR, but now discovers that FN uses VAR freely. VAR should be declared a SPECVAR and the block recompiled.

((* --) COMMENT USED FOR VALUE)
A comment appears in a context where its value is being used, e.g. (LIST X (* --) Y). The compiled function will run, but the value at the point where the comment was used is ''unde ned.''

((FORM) - NON-ATOMIC CAR OF FORM)

If user intended to treat the value of FORM as a function, he should use APPLY* (page 5.12). FORM is compiled as if APPLY* had been used.

((SETQ VAR EXPR --) BAD SETQ)

SETQ of more than two arguments.

(FN - USED AS ARG TO NUMBER FN?)

The value of a predicate, such as GREATERP or EQ, is used as an argument to a function that expects numbers, such as IPLUS.

(FN - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

The compiler has assumed FN is the name of a function. If the user intended to treat the *value* of FN as a function, he must use APPLY* (page 5.12).

This message is printed when FN is not de ned, and is also a local variable of the function being compiled. Note that earlier versions of the Interlisp-10 compiler did treat FN as a functional argument, and compiled code to evaluate it.

(FN - ILLEGAL RETURN)

RETURN encountered when not in PROG.

(TG - ILLEGAL GO)

GO encountered when not in a PROG.

(TG - MULTIPLY DEFINED TAG)

TG is a PROG label that is de ned more than once in a single PROG. The second de nition is ignored.

(TG - UNDEFINED TAG)

TG is a PROG label that is referenced but not de ned in a PROG.

(VAR - NOT A BINDABLE VARIABLE)

VAR is NIL, T, or else not a literal atom.

(VAR VAL -- BAD PROG BINDING)

Occurs when there is a prog binding of the form (VAR VAL_1 VAL_N).

(TG - MULTIPLY DEFINED TAG, ASSEMBLE)

TG is a label that is de ned more than once in an assemble form.

(TG - UNDEFINED TAG, ASSEMBLE)

TG is a label that is referenced but not de ned in an ASSEMBLE form.

(TG - MULTIPLY DEFINED TAG, LAP)

TG is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply de ned tag during pass one, the tag is in a LAP macro.

(TG - UNDEFINED TAG, LAP)

TG is a label that is referenced during the second pass of compilation and is not de ned. LAP treats TG as though it were a COREVAL, and continues the compilation.

(OP - OPCODE? - ASSEMBLE)

OP appears as CAR of an assemble statement, and is illegal. See page 22.12 for legal assemble statements.

(NO BINARY CODE GENERATED OR LOADED FOR FN)

A previous error condition was sufficiently serious that binary code for FN cannot be loaded without causing an error.