18. COMPILER

The compiler is contained in the standard Medley system. It may be used to compile functions defined in Medley, 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 (Chapter 17), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in Medley, by directly calling the compiler using functions such as COMPILE. No matter how the compiler is called, the function COMPSET is called which asks you 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 compiler does *not* respect the value of DFNFLG (Chapter 10) when it redefines functions to their compiled definitions. Therefore, if you set DFNFLG to PROP to completely avoid inadvertantly redefining something in your running system, you *must* not answer YES to this question.

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.

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

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

F Compile to **F**ile, without redefining functions.

ST **ST**ore new definitions, saving EXPR definitions.

STF **ST**ore new definitions; **F**orget 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.

Compiler Printout

In Medley, 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*.

If the compilation of *FN* causes the generation of one or more auxilary functions, 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 in the last section of this chapter.

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.

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, 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 a global variable is equivalent to calling GETTOPVAL on the 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 Medley 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, you might write

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

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 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: The variables that a given function accesses as global variables can be determined by using the function CALLS.

Local Variables and Special Variables

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. 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.

Note: The variables that a given function binds as local variables or accesses as special variables can be determined by using the function CALLS.

Note: LOCALVARS and SPECVARS declarations affect the compilation of local variables within a function, but the arguments to functions are always accessible as specvars. This can be changed by redefining the following function:

(**DASSEM.SAVELOCALVARS** *FN*) [Function]

This function is called by the compiler to determine whether argument information for *FN* should be written on the compiled file for *FN*. If it returns NIL, the argument information is *not* saved, and the function is stored with arguments U, V, W, etc instead of the originals.

Initially, DASSEM.SAVELOCALVARS is defined to return T. (MOVD 'NILL 'DASSEM.SAVELOCALVARS) causes the compiler to retain no local variable or argument names. Alternatively, DASSEM.SAVELOCALVARS could be redefined as a more complex predicate, to allow finer discrimination.

Constants

Interlisp allows the expression of constructions which are intended to be description of their constant values. The following functions are used to define constant values. The function SELECTC provides a mechanism for comparing a value to a number of constants.

(**CONSTANT** *X*) [Function]

This function enables you to define that the expression *X* should be treated as a "constant" value. When CONSTANT is interpreted, *X* 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 *X* 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 definition, as if (QUOTE *VALUE-OF-EXPRESSION*) had appeared instead of (CONSTANT *EXPRESSION*).

If the value of *X* does not have a readable print name, then the expression *X* itself will be saved with the function, and it will be evaluated when the function is first loaded. The value will then be stored in the function's literals, and will be retrieved on future references.

If a program needed a list of 30 NILs, you 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 differ 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.

Whereas the function CONSTANT attempts to evaluate the expression as soon as possible (compile-time, load-time, or first-run-time), other options are available, using the folowing two function:

(**LOADTIMECONSTANT** *X*) [Function]

Similar to CONSTANT, except that the evaluation of *X* is deferred until the compiled code for the containing function is loaded in. For example, (LOADTIMECONSTANT (DATE)) will return the date the code was loaded. If LOADTIMECONSTANT is interpreted, it merely returns the value of *X*.

(**DEFERREDCONSTANT** *X*) [Function]

Similar to CONSTANT, except that the evaluation of *X* is always deferred until the compiled function is first run. 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. If DEFERREDCONSTANT is interpreted, it merely returns the value of *X*.

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

Defines 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*).

The compiler prints a warning if user code attempts to bind a variable previously declared as a constant.

Constants can be saved using the CONSTANTS file package command.

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 first look for a definition among the functions in the file that is being compiled. The function can be defined anywhere in any of the files given as arguments to BCOMPL, TCOMPL, BRECOMPILE or RECOMPILE. If the function is not contained in the file, the compiler will look for other information in the variables NLAMA, NLAML, and LAMS, which can be set by you:

NLAMA [Variable]

(For NLAMbda Atoms) A list of functions to be treated as nlambda nospread functions by

the compiler.

NLAML [Variable]

(For NLAMbda Llist) 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.

COMPILEUSERFN is only called when the compiler encounters a *list* CAR of which is not the name of a defined function. You can instruct the compiler about how to compile other data types via COMPILETYPELST.

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

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.

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. 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.

FUNCTION and Functional Arguments

Compiling the function FUNCTION 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 to the end of the name of the function in which they appear, e.g., FOOA0003. For example, suppose FOO

is defined as (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled. When FOO is run, FOO1 will be called with two arguments, X, and FOOA000*N* and FOO1 will call FOOA000*N* 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.). A considerable savings in time could be achieved by making FOO1 compile open via a computed macro, e.g.

```
(PUTPROP 'FOO1 'MACRO
      '(Z (LIST (SUBST (CADADR Z)
                       (QUOTE FN)
                       DEF)
                (CAR Z)))
```

DEF is the definition of FOO1 as a function of just its first argument, and FN is the name used for its functional argument in its definition. 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 you must decide which is more important.

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 significant 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. You can make other functions compile open via MACRO definitions. You can also affect the compiled code via COMPILEUSERFN and COMPILETYPELST.

COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (symbols). You can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. macros, declarations, COMPILEUSERFN, etc. COMPILETYPELST allows you 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 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*).

Compiling CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be DWIMIFYed. You 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 file 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 definition has a local CLISP declaration, including a null declaration, i.e., just (CLISP:), the definition will be automatically DWIMIFYed before compiling.

Note: COMPILEUSERFN is defined 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 dwimified 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 unless DWIMESSGAG = T. In this case, the expression is just compiled as is, i.e. as though CLISP had not been enabled. You can exit the break in one of these ways:

1. Type OK to the break, which will cause the compiler to try again, e.g. you could define some missing records while in the break, and then continue

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 first place

3. Return an expression to be compiled in its place by using the RETURN break command.

Note: TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to NOFIXFNSLST and NOFIXVARSLST, respectively. Thus, 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 is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

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. 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 (the compiled file extension is stored as the value of the variable COMPILE.EXT). 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 for the following:

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 (the actual string printed is the value of COMPILEHEADER).
2. Every DEFINEQ in the symbolic file is replaced by the corresponding compiled definitions in the compiled file.
3. Expressions following a DONTCOPY tag inside of a DECLARE**:** 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**:**. This "compiled" file can be loaded into any Interlisp system with LOAD.

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 by an expr definition, 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*.

If compiled definitions are being written to a file, the file is closed unless *FLG* = T.

(**COMPILE1** *FN DEF*) [Function]

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

If DWIMIFYCOMPFLG is T, or *DEF* contains a CLISP declaration, *DEF* is dwimified before compiling.

(**TCOMPL** *FILES*) [Function]

TCOMPL is used to "compile files"; given a symbolic LOAD file (e.g., one created by MAKEFILE), it produces a "compiled file". *FILES* is a list of symbolic files 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 file is written on a file of the same name suffixed with DCOM, e.g., (TCOMPL '(SYM1 SYM2)) produces two files, SYM1.DCOM and SYM2.DCOM.

TCOMPL processes the files one at a time, reading in the entire file. For each FILECREATED expression, the list of functions that were marked as changed by the file package is noted, and the FILECREATED expression is written onto the output file. For each DEFINEQ expression, TCOMPL adds any nlambda functions defined in the DEFINEQ to NLAMA or NLAML, and adds lambda functions to LAMS, so that calls to these functions will be compiled correctly. NLAMA, NLAML, and LAMS are rebound to their top level values (using RESETVAR) by all of the compiling functions, so that any additions to these lists while inside of these functions will not propagate outside. Expressions beginning with DECLARE: are processed specially. All other expressions are collected to be subsequently written onto the output file.

After processing the file in this fashion, TCOMPL compiles each function, except for those functions which appear on the list DONTCOMPILEFNS (initially NIL), and writes the compiled definition onto the output file. TCOMPL then writes onto the output file the other expressions found in the symbolic file. DONTCOMPILEFNS might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that DONTCOMPILEFNS can be set via block declarations.

Note: If the rootname of a file 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.

TCOMPL returns a list of the names of the output files. All files are properly terminated

and closed. If the compilation of any file is aborted via an error or Control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

(**RECOMPILE** *PFILE CFILE FNS*) [Function]

The purpose of RECOMPILE is to allow you to update a compiled file without recompiling every function in the file. RECOMPILE does this by using the results of 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 **P**retty file (source file) to be compiled; *CFILE* is the name of the **C**ompiled 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 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.

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
2. *FNS* = T or EXPRS and the function is defined by an expr definition
3. *FNS* = CHANGES and the function is marked as having been changed in the FILECREATED expression in *PFILE*
4. *FNS* = ALL

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*.

Note: 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).

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 CHANGES. Thus you can perform his edits, dump the file, and then simply (RECOMPILE '*FILE*) to update the compiled file.

The value of RECOMPILE is the file name of 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.

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

Note: Since PRETTYDEF automatically outputs a suitable DECLARE: expression to indicate which functions in the file (if any) are defined 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.

Block Compiling

In Interlisp-10, 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. 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. A block can be entered via several different functions, called entries. These must be specified when the block is compiled.

In Medley, block compiling is handled somewhat differently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Medley works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, 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.

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, etc. To help with this, there is the BLOCKS file package command, which has the form:

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

 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.

A block declaration is a list of the form:

(*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 $VALUE_i$, 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)).

For example, consider the block declaration below. The block name is EDITBLOCK, it includes a number of functions (EDITL0, EDITL1, ... EDITH), and it sets the variables ENTRIES, SPECVARS, RETFNS, and GLOBALVARS.

```
(EDITBLOCK
         EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA
         EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM EDITH
         (ENTRIES EDITL0 ## UNDOEDITL)
         (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS)
         (RETFNS EDITL0)
         (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS))
```

Whenever BCOMPL or BRECOMPILE encounter a block declaration, they rebind RETFNS, SPECVARS, GLOBALVARS, BLKLIBRARY, 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.

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 finishing all blocks, BCOMPL and BRECOMPILE treat any functions in the file that did not appear in a block declaration in the same way as do TCOMPL and RECOMPILE. If you wish a function compiled separately as well as in a block, or if you wish to compile some functions (not blockcompile), with some compiler variables changed, you 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 first setting *VAR*$_1$ ... *VAR*$_N$ as described above.

The following variables control other aspects of compiling a block:

RETFNS [Variable]

Value is a list of internal block functions whose names must appear on the stack, e.g., if the function is to be returned from RETFROM, RETTO, RETEVAL, etc. Usually, internal calls between functions in a block are not put on the stack.

BLKAPPLYFNS [Variable]

Value is a list of internal block functions called by other functions in the same block using BLKAPPLY or BLKAPPLY* for efficiency reasons.

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.

BLKAPPLYFNS [Variable]

Value is a list of functions that are considered to be in the "block library" of functions that should automatically be included in the block if they are called within the block.

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 you 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.

Block Compiling Functions

There are three user level functions for block compiling, BLOCKCOMPILE, BCOMPL, and BRECOMPILE, corresponding to COMPILE, TCOMPL, and RECOMPILE. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, you can intermix in a single file functions compiled normally 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 specified, 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 specified, 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 file, the file 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 definition for *BLKNAME*, plus definitions 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. 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.

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.

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 you 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, 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 FN_1 ...), 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 CHANGES.

Compiler Error Messages

 Messages describing errors in the function being compiled are also printed on the terminal. 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 definition 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 definition of *FN* from *CFILE*, and cannot find 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 specified 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*)

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 undefined.

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

If you intended to treat the value of *FORM* as a function, you should use APPLY* (Chapter 10). *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 you intended to treat the *value* of *FN* as a function, APPLY* (Chapter 10) should be used. This message is printed when *FN* is not defined, and is also a local variable of the function being compiled.

(*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 defined more than once in a single PROG. The second definition is ignored.

(*TG* – UNDEFINED TAG)

TG is a PROG label that is referenced but not defined 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, 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 defined 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 defined. LAP treats *TG* as though it were a COREVAL, and continues the compilation.

(*TG* – MULTIPLY DEFINED TAG, ASSEMBLE)

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

(*TG* – UNDEFINED TAG, ASSEMBLE)

TG is a label that is referenced but not defined in an assemble form.

(*OP* – OPCODE? – ASSEMBLE)

OP appears as CAR of an assemble statement, and is illegal.

(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.