MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LAB

Artificial Intelligence
Memo No. 210

December 1970

A USER'S GUIDE TO THE A. I. GROUP LISCOM LISP COMPILER: INTERIM REPORT

Jeffrey P. Golden

Work reported herein was supported by Project MAC and the Artificial Intelligence Laboratory, M.I.T. research programs sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts number N0001 4-70-A-0362-0001 and -0002.

Table of Contents

I. Introduction

The LISCOM version of the A.I. Group PDP/6(*10*) Lisp compiler is a descendant of the original Greenblatt-Nelson compiler, and is a friendly sibling to the COMPLR version maintained by Jon L. White. The compiler operates in two passes to translate LISP code into LAP code. The first pass performs a general study of the S-expression function definition which is to be compiled, producing as output a modified S-expression and various 'tables' attached to free variables. The second pass does the actual compilation (generation of assembly code), making use of the transformations performed and the information gathered by the first pass.

The LISCOM version of the compiler is being used as a vehicle for the implementation of 'fast arithmetic' in LISP. This work is being done under the auspices of the MATHLAB project of the A.I. Laboratory. The early stages of the compiler implementation were handled by W. Diffie, and the work has been continued by the present author. Corresponding changes to the LISP system were implemented by W. A. Martin and Jon L. White. The idea is to use user declarations of fixed-point and floating-point variables and functions - as to the value they return, as well as other mechanisms more convenient in certain situations (to be described at a later date), to enable the open-compilation of the LISP arithmetic functions. At the present date, the conversion of the first pass of the compiler to handle 'fast arithmetic' has been completed. The conversion of the second pass is now under way. Also, some improvement in the output code and many bugs in the compiler were removed as a joint effort of the present author and Jon L. White.

Every attempt has been made to make the LISCOM and COMPLR versions compatible in the sense that a user need not make any changes to his file to switch from one compiler to the other. There are, however, differences in some of the top level functions through which the user corresponds with the compilers, in the format of the LAP outputted, and perhaps also, in the speed of operation of the compilers, in the relative efficiencies of some of the LAP code, and in convenience to the user in terms of warning and error messages, etc. The usage conventions of the LISCOM version are described in the following section. The only descriptions of the usage of the COMPLR version available at the present date are found in A.I. Memo *116A* (PDP-6 LISP Revised), p. *5* and A.I. Memo *190* (An Interim LISP User's Guide), Appendix X, both written by J. L. White. Some discussion relating to the user's interaction with the compilers (e.g. relating to DECLARE) may be found in A.I. Memo *190*, p. *34* ff. A description of the actual operation of the compiler as of June *1969*, especially the LISCOM version, written by W. Diffie, is available from the present author.

II. Operation

One may load in the LISCOM compiler by typing at DDT 'LISCOM↑H' or ':LISCOM CR'. Since the LISCOM version uses the formatting functions of the GRIND package (see A.I. Memo *190*, Appendix E), one may use LISCOM for both compiling and formatting S-expressions, although the latter is surely inconvenient when compared to loading in the file GRIND LISP on device COM. In any event, both features of LISCOM are described below. Let us note here, that the features described below are subject to change (hopefully with notice).

Compiling

A. LISCOM's top-level functions:

These functions are all FSUBRs (unless they take no argument in which case they happen to be SUBRs).

1. COMFILE: COMFILE is the major function which enables the user to translate one or more files of LISP code (these files are called the 'source files', and are denoted here as *fn1 fn2*, *gn1 gn2*, etc.) into a single file of LAP code (this file is called the 'target file' and is denoted here as *tn1 tn2*).

A file to be compiled may contain function definitions (via DEFPROP or DEFUN) to be compiled, MACRO-definitions to be expanded, LAP code, declarations to the compiler (via DECLARE - described below), comments (via COMMENT), and any other random S-expressions. All but function definitions via DEFPROP or DEFUN, MACRO-definitions, and declarations via DECLARE are simply passed on through to the target file 'untouched' (except see the discussion of the *GRIND switch below). MACRO-definitions are expanded in Pass *1* and take priority over (i.e. can be used to redefine) system-defined functions. Since MACRO-definitions are placed on the property list of the MACRO-name, a conflict will occur (of which the user will be informed) if a user MACRO has the same name as a compiler function and if the MACRO is called from within other (or the same) MACRO-definitions. (The LISP system may be hacked later on to prevent this rare conflict from occurring by 'prefixing' a 'file name' to each function name). Also, noting the description of DECLARE below, if DECLARE is used carelessly (e.g. foolishly modifying global variables intended solely for the compiler's internal use), trouble may ensue. No other conflicts between user names (for variables and functions) and compiler names can occur. Finally, let us note here that it is intended that the compiler be able to compile (almost) any function that runs interpretively and can be compiled. If any user runs into difficulties with the compiler, he should see the author.

COMFILE takes a list of n >= 0 file-names (with device and sname specifications if desired - the usual default options are available) as argument. There are three special cases: n = 0, n = 1, and n >= 2.

(a) n = 0: Typing (COMFILE) to LISCOM is completely analogous to typing (CMP1) to COMPLR. This gives the user control over the compilation process. An example of this use, beginning here with the loading operation, is as follows: The user is talking to DDT, which has typed '*' at him.

```
'*'LISCOM↑H'↑!'          (that which the system types at the
                             user is enclosed in single
                             quotes.)
'LISCOM COMPILER (etc.)'
(UREAD fn1 fn2)          (the file to be compiled)
'(device sname)'
(UWRITE)
'(device sname)'
(COMFILE)
'COMPILER-LISTENING'
↑R ↑W ↑Q                 (I/O switches are set as desired)

   .   .   .             (the compilation)

'(func . *EXPR)'         (the EXPR 'func' has been compiled)

   .   .   .

'FINISHED'               (the compilation has been
                             completed)
(UFILE tn1 tn2)
'(device sname)'
```

Many variations on the above are possible, e.g. outputting to the line-printer, rather than to the disk; or defining functions on-line and compiling them.

The n >= 1 cases are the more usual ones. Here, the compiler handles the entire compilation process for the user, including the setting of I/O switches and filing (UWRITEing, UREADing, and UFILEing).

(b) n = 1: In this case, COMFILE takes a list of one source file-name

(fn1 fn2 dev sname)

as argument, and it outputs the compiled file as

(fn1 LAP dev sname),

('dev sname' perhaps being specified by default, of course.) An example call is

(COMFILE (FOO LISP)).

In this case, only one source file can be translated into a target file. The INITIALIZE function described below is relevant here.

(c) n >= 2: In this case, COMFILE takes a list of n >= 2 file-names as argument; the first file-name is taken to be the target file and the remaining n - 1 file-names are source files. The files are picked up in left-to-right order. In this manner, one or more source files may be translated into a single target file. A sample call is

(COMFILE (OUT COMP) (FOO LISP1) (GOO LISP2 DSK JJ)).

When n >= 2, the *INITIAL switch (global variable) described below may be relevant.

2. CF: The function CF is available as a convenience for those users who need to or wish to compile the same file sequence twice. The file specifications made in the last call to COMFILE with n >= 1 are preserved on the property list of the atom TRY, so that one need only 'eval' (CF) (for Compile File) in order to repeat that call to COMFILE.

3. COMPILE: COMPILE is used to compile or recompile individual functions, perhaps only in order to investigate the code being produced. COMPILE takes as argument a list of the functions (i.e. their names) to be compiled, i.e.

(COMPILE fun1 fun2 ... funi).

COMPILE assumes that the S-expression definitions for the functions have been loaded in already by the user. In formatting, COMPILE uses the line-length (LINEL) of the device (console) at which the user is logged-in, as opposed to COMFILE which uses the line-length of the line printer (actually 80. and not 120.) This may be changed by the user by SETQing LINEL to PAGEWIDTH. When recompiling, the function GENPREFIX discussed below is useful.

4. CMP, CN: These functions are useful only for investigating the code produced by the compiler. CMP takes as argument a LAMBDA expression to be compiled, i.e.

(CMP (LAMBDA . . .)),

which is compiled as a SUBR called TRY. I.e. CMP combines in one step a

(DEFPROP TRY (LAMBDA . . .) EXPR)

and

(COMPILE TRY).

CN (for Compile Name), which takes no argument, is used to repeat the last call to COMPILE, through the use of the global variable FUNNAME, mentioned below.

B. Handy functions, global variables, and notes:

The following is a description of functions and global variables available in the compiler to aid in the compilation process:

(a) Functions

1. DECLARE: DECLARE, an FSUBR, is an all-purpose function used in handling declarations of variables (via the functions SPECIAL and UNSPECIAL) and functions (via *FEXPR and *LEXPR) as well as in causing evaluations to take place immediately. Indeed,

DECLARE [expr.] = MAPC [(FUNCTION EVAL) (CDR expr.)].

(Thus, one need not use DECLARE upon having loaded in the compiler, before initiating the compilation process.) DECLARE will also enable the user to inform the compiler as to how he wishes 'fast arithmetic' to be done, when such is available - a full description will be given at that time. DECLARE is treated by the LISP interpreter just as is COMMENT; however, in the compiler it takes effect whether used at the top level or in a function definition. DECLARE or an alternative must be used to inform the compiler

(*1*) via the compiler function SPECIAL of variables which appear free in some function(s) in the file or of those which the user would like to refer to by name later on. The compiler and the user may access these variables via the 'SPECIAL cell' on their property list. Much will be said of the former use in later sections. The function UNSPECIAL is used to remove the SPECIAL declaration.

(2) via the compiler functions *FEXPR and *LEXPR, respectively, of FEXPRs and LEXPRs (i.e. EXPRs with atomic, non-NIL LAMBDA-lists) which are called within function definitions before they themselves appear (are defined) in the file, (the default option for such 'undefined' functions is that they are of EXPR-type, and an error message will be printed out if this is not done), if indeed they appear there at all.

DECLARE is also used in SETQing the 'user-accessible' global variables described below. Thus, a sample call is

(DECLARE (SPECIAL var*1* var2) (*FEXPR fun*1*) (SETQ *GRIND 3)
(GENPREFIX H)).

(SPECIAL, UNSPECIAL, *EXPR, *FEXPR, *LEXPR, and GENPREFIX are all FSUBRs defined by the compiler.)

2. GENPREFIX: GENPREFIX is useful when recompiling individual functions to be placed in a file alongside other LAP-code or in compiling several files at different times which are later to be assembled into one system. GENPREFIX is used to avoid conflicts between tags and between GENSYMs (generated symbols) which are used by the compiler for generated functions

(see the 'Notes' section below). (Tags are local to the LAP function in which they appear, but only one occurrence of each tag may appear in a DDT symbol table). The syntax is

(GENPREFIX atom)

and the atom, e.g. BAR, is used (1) to prefix a GENSYM, e.g. converting G0105 to BAR0105 in the case of generated function names; and (2) as the prefix to an integer (counter) TAGCNT in the case of tags. See the discussion of the global variables GENLIST and TAGCNT below. A one-character atom is usually used, and indeed the generated atom (BAR0105 here) may not have more than 6 characters if it is to be placed in a DDT symbol table. GENPREFIX also resets TAGCNT (see later) to 0. (It is hoped that some day functions associated with GENSYMming will be added to the LISP system to replace the need for GENPREFIX as used for generated functions.)

3. INITIALIZE: INITIALIZE, a SUBR, is useful when compiling more than one file with a single loading of the compiler. INITIALIZE takes as argument 1, 2, or 3 which indicates its mode: If arg = 1, INITIALIZE removes all SPECIALs from the OBLIST. This is obviously a less efficient alternative to declaring all of one's SPECIAL variables UNSPECIAL at the end of the file. If arg = 2, INITIALIZE removes all *EXPR, *FEXPR, *LEXPR, and MACRO properties from the OBLIST, except for those *EXPR flags used by the compiler. If arg = 3, INITIALIZE takes both of the actions described above under args 1 and 2. Thus INITIALIZE can be used to obtain a clean environment for compiling the next file, without the necessity of reloading the compiler. When using COMFILE with n > 2, INITIALIZ(E)ation is accomplished through the setting of the *INITIAL switch, described below.

(b) Global variables - switches:

Some of the user-accessible global variables in the compiler are switches which may be set (e.g. via DECLARE) to T or NIL, or to an integer, as the case may be. To indicate which global variables are switches, * is used as the first character of their names.

1. *INITIAL: As mentioned above, when using COMFILE with n > 2, INITIALIZE may be called between each pair of files inputted for compilation by simply setting *INITIAL appropriately. *INITIAL is the argument to INITIALIZE, and the actions caused by setting *INITIAL to 1,2, or 3 were described above. The default setting is 0, in which case INITIALIZE is not called.

2. *SYMBOLS: Setting *SYMBOLS to T causes (SYMBOLS T) followed by a tag to be inserted into the LAP-code immediately after the (LAP funname type) pseudo-instruction for each function then compiled. The tag is useful in cases where two function names begin with the same 6 characters, since these cannot be differentiated within the DDT symbol table. For an explanation of (SYMBOLS T-or-NIL) see A.I. Memo 190, p. I-5. The default setting of *SYMBOLS is NIL.

3. *GRIND: Any S-expression (including MACRO-definitions) in the source file that is not a function definition is output 'as is' (as described below) into the target file. (Even DECLAREs are output, so that if the user wishes to investigate his LAP code, he can easily check out the compiling environment as to SPECIALs, UNSPECIALs, etc.) *GRIND gives the user control over the manner in which both the random S-expressions mentioned above and the LAP-code is output, as follows:

*GRIND = 0 (the default setting): everything is formatted ('ground').
= 1: only the LAP-code is ground.
= 2: only the random S-expressions are ground.
= 3: nothing is ground.

Furthermore, if *GRIND = 0 or 1, the compiler outputs a form-feed after every 56 lines (or so - if *GRIND = 1); if *GRIND = 2 or 3, the compiler outputs a form-feed after the LAP-code for each compiled function. Whereas formatting makes for much more aesthetic and readable output, it is time-consuming and the user may prefer not to grind out e.g. his LAP-code.

4. *REDEF: When set to T, *REDEF tells the user about functions which are defined more than once in the source files. This may be useful when compiling more than one source file with one loading of the compiler, with the intention of assembling these files into one file or system. Users who intend to compile the same file more than once with one compiler loading, e.g. to take advantage of the compiler's declaring as SPECIAL undeclared free variables, should not have *REDEF set to T on the first go-round. The default setting of *REDEF is NIL.

5. *CLOSED, *ARITH, *MUZZLED: These switches have to do with fast-arithmetic compiling, and hence, are not yet of interest. It should be pointed out, though, that setting *CLOSED to T, which in fact is its current default setting, causes the compiler not to attempt to do any fast-arithmetic compiling, which is now the intention.

6. *DEBUG: As the compiler is loquacious in its error and warning messages (see section (d) 1. below), several LISP users have used the compiler for debugging purposes - seeking out typing errors, etc. For this reason, a special faster debugging mode, which is entered by setting *DEBUG to T, has been added to the compiler, in which no LAP code is generated, no output file is opened, and in fact, the compiler does not go through its second pass at all. The lists UNDFUNS and NEWSPECVARS (see below) are still maintained. The only warning message which is not issued is that for undefined GO tags (see below). One may use COMFILE in the manner described above. (For n >= 2, the target file should be NIL). The default setting of *DEBUG is NIL.

(c) Global variables - other:

1. UNDFUNS: Upon completing the compilation for each call to COMFILE, the compiler prints out the elements of the list UNDFUNS. These are the names of user functions which though called within the source file(s), were not defined there. The functions appear in UNDFUNS in the order of their first call. The user can ask for this list to be printed again (or for the first time in the case of COMPILE) by evalling UNDFUNS.

2. NEWSPECVARS: NEWSPECVARS is a list of all those variables seen since the last call to COMFILE or COMPILE which appeared free in some function but were not declared SPECIAL. These variables are made SPECIAL by the compiler. To have this list printed out, the user may eval NEWSPECVARS.

3. GENLIST: Evalling (GENPREFIX atom) as discussed above, say atom = BAR, sets GENLIST to the list (B A R). Thus, the same result can be had by setting GENLIST accordingly, except that GENPREFIX also resets TAGCNT to 0. The default setting of GENLIST is (G).

4. TAGCNT: Tags generated by the compiler are obtained by
```
(MAKNAM (APPEND GENLIST
                (EXPLODE (SETQ TAGCNT (ADD1 TAGCNT)))))
```
Upon loading in the compiler, TAGCNT is set to 0. The user may set TAGCNT to some positive integer if he wishes.

5. FUNNAME: FUNNAME is set by COMFILE and COMPILE to the name of the current function being compiled, or to the last such function if the compiler has just completed compiling a function. Thus, if an error occurs while compiling, evalling FUNNAME enables the user to determine the name of the function being compiled.

(d) Notes on compiling:

1. Errors which cause the compiler to stop, fall into three general categories: (1) breaks: which are generally caused by compiler errors, (2) data-errors: which the compiler believes to be caused by errors within the user's function definitions, and (3) errors in data or due to the compiler which the compiler does not check for, but which eventually lead to LISP errors. We will address ourselves here only to the first two categories in the hope that those of the third category will rarely if ever occur.

In the case of (1) or (2), the compiler enters a read-eval-print loop which is useful in debugging the compiler or in investigating further the existing state of affairs. To exit

from this loop, the user usually types ⌈G or ⌈Z; of course, with the intention of notifying the author in case of a compiler error. The user may also wish to type $P_ or $X_ ($ = <alt. mode>, _ = <space>) to proceed. In the case of (1) compiler errors, in which case the compiler prints out *BREAK* followed by the error message, typing either $P_ or $X_ will cause the compiler to recommence compiling with the next function in the file. (In this case only, the compiler may have output some spurious LAP-code.) In the case of (2) data-errors, the compiler prints out either *NRDATAERR* or *RDATAERR* followed by the error message. In the case of *NRDATAERR* or 'non-recoverable data error', typing $P_ or $X_ again causes the compiler to go on to the next function. In the case of *RDATAERR* or 'recoverable data error', typing $P_ causes the compiler to continue compiling the same function, perhaps after making some reasonable adjustment if necessary; while typing $X_ again causes the compiler to go on to the next function. The intention is to give the user the option of continuing with the same function with the possibility that the compiler will discover more data-errors therein; or indeed, the compiler may correct the error, as described below. The user may in any event decide to recompile the function or the file later on. At present, there are only four cases of 'recoverable data-errors':

(1) The compiler encounters an FEXPR or LEXPR definition, having previously compiled the function as an EXPR. Typing $P_ will cause the compiler to continue compiling, taking the function to have F-type or L-type, respectively, from now on.

(2) The user has a MACRO-definition with the same name as a compiler function. The user will lose only in the situation described above (i.e. the MACRO is called within other (or the same) MACRO-definitions.) If this is not the case, the user may type $P_ , and the compiler will continue, making certain that a conflict will not occur.

(3) The user calls a system function with the wrong number of arguments. The user may type $P_ , causing the compiler to continue as follows: If he called the function with too few arguments, the compiler appends NILs for each of the remaining arguments. If he called the function with too many arguments, the compiler makes no change in the case of closed-compiled functions. The more commonly used open-compiled functions pull off only as many arguments as they need. Later, the user may wish to edit the LAP-code for this function or recompile it.

(4) A RETURN is used not in the context of a PROG. Typing $P_ will cause the compiler to strip off the RETURN, i.e. convert (RETURN arg) to arg, and continue.

Other possible errors in user code may simply cause the compiler to print out a warning message and go on. This does not mean that these situations are not really errors. For example, the compiler may complain that it has encountered a free variable which has not been declared, which it then makes

SPECIAL. If this variable has been used bound in previous functions and the user intends for these occurrences to mean the same variable, then a miscompilation has occurred. Warning messages are also given in case of unused PROG or LAMBDA variables - bound variables which are not declared SPECIAL and which are not evaluated within that PROG or LAMBDA; undefined GO tags, in which case the compiler simply outputs a JRST (jump) to the end of the LAP-code for the PROG; etc. These may all signify errors.

2. There are two circumstances which cause the compiler to compile code out of context, i.e. to extract LISP code from a function definition and to compile it as a generated function:
(1) Lambda funargs: any occurrence of
(FUNCTION (LAMBDA . . .))
in code causes the LAMBDA expression to be compiled as a separate function.
(2) Any call to ERRSET of the form
(ERRSET *arg1* *arg2*)
or
(ERRSET *arg1*)
in which *arg1* is neither an atom nor a list of a single atom (i.e. a function call with no arguments) causes *arg1* to be compiled as a separate function of the form
(LAMBDA NIL *arg1*).
The latter circumstance as a cause for compilation out of context will probably be eliminated in the near future.
Clearly, any variables which appear free in these LAMBDA expressions (perhaps as a result of their being compiled out of context) must be declared SPECIAL.

3. Any variables appearing in arguments to F-type functions which are to be evaluated (e.g. the latter arguments to ARRAY) must be declared SPECIAL. Unfortunately, here the compiler does no checking for the user. Hence, the user will lose if he does not heed this warning. Also, the user must remember to declare as SPECIAL any free variables appearing in functional position, or the compiler will take them to be undefined functions and compile them as SUBRs.

4. In his LISP code to be compiled, the user may redefine a LISP system function as a MACRO but not as an EXPR or FEXPR. If the user wishes the latter, e.g. to redefine SUBST via an EXPR, he may do the following instead:

```
(DEFUN SUBST MACRO (X) (CONS 'SUBST1 (CDR X)))
(DEFUN SUBST1 EXPR  .  .  .) .
```

(The atom 'EXPR' of course is unnecessary.)

III. Formatting

There are two functions available in the compiler for formatting: FORMFILE, analogous to COMFILE, for formatting a file and FORMAT, analogous to COMPILE, for formatting function definitions. Except for their obviously different purpose, the syntax and semantics for these functions is similar to that for COMFILE and COMPILE, respectively, except that (1) at present, FORMFILE has no n = 0 mode; (2) when n = 1, the assumption is that the user when evalling (FORMFILE (fn1 fn2)) wishes to clobber the source file, i.e. wishes to give the target file the same name as the source file.

- - - - - - - - - - -

Please notify the author in case of compiler bugs or if you have other comments to make.