SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 541–605 (1981)

A Portable LISP Compiler*

MARTIN L. GRISS AND ANTHONY C. HEARN†
Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, U.S.A.

SUMMARY

The programming language LISP is usually implemented via an interpreter, and a compiler is added later as a LISP program. However, all such production compilers known to the authors produce explicit instructions for the given computer being used. This paper describes the development of a portable LISP compiler in the sense that only Standard LISP functions are used in its definition and the output is a sequence of abstract machine codes, easily mapped to instruction sequences on current computers. The resulting code is quite efficient, demonstrating once again the maxim that most compiler optimization is largely machine independent.

KEY WORDS LISP Compilers Machine independent compilers Optimization

1. INTRODUCTION

The programming language LISP has been in existence for over twenty years[1] and still remains as a unique programming language from several points of view. For one thing it is usually implemented via an interpreter, with a compiler added at a later stage. Most modern LISP compilers are written in LISP itself and their structure can be traced from the first such compilers written by Maling and by Hart and Levin for the IBM 704. However, it is very hard to find descriptions of these compilers in the literature and most of what is available is sketchy or limited to basic principles or special features. The most complete description may be found in a recent book by Allen[2] which describes the basic compilation process in terms of a simple abstract machine. Fairly detailed descriptions of working LISP compilers we have found are by Saunders on his Q-32 LISP system compiler,[3] London's paper on verifying compilers for a subset of LISP,[4] a paper by Blair on a compiler for a LISP system for the IBM System/360[5] and some papers describing parts of the Maclisp and related compilers.[6, 7] There are also descriptions of some experimental machine independent compilers in the literature,[8, 9] but our emphasis on high quality code in a production environment as well as portability gives our work a different emphasis.

In addition, in spite of the fact that most LISP compilers are written in LISP, a higher level language, the code generation of past production compilers has been effected directly in terms of the machine instructions for the particular machine for which the compiler was written. There is now enough general experience in compiler construction, however, to suggest that the majority of the code generation techniques

* Work supported in part by the National Science Foundation under Grant No. MCS76–15035.

† *Current address*: Rand Corporation, Santa Monica, California 90406, U.S.A.

0038–0644/81/060541–65$06.50

Received 2 July 1979
Revised 20 August 1980

are not machine dependent, and, more precisely, a large number of the optimization techniques are independent of the particular machine for which the compiler is written.[10] This point has been well emphasised in the case of FORTRAN, for example.[11]

With these thoughts in mind we decided to look once again at LISP compilation and determine to what degree it was possible to produce a compiler for Standard LISP[12] which was as machine independent as possible, but which produced good quality code and was more reliable than currently available compilers. The aim was to make it easy to install this compiler on top of any LISP system that supports Standard LISP, replacing the existing compiler (if any). As a result, the maintenance would be eased, and Standard LISP compatibility enhanced. In support of these goals we decided therefore that the output from this compiler would be a set of instructions for an abstract Standard LISP machine rather than assembly language instructions for a particular computer. These instructions would then (most likely) be macro expanded into the code for the particular target computer. For convenience in this paper, we will refer to these instructions as c-macros.

In order to ease our task in describing this compiler, we shall assume that the reader is familiar with LISP at the level of Allen's book,[2] and, in particular, is familar with his chapter on compiling. In addition, we base our definitions on Standard LISP and therefore assume also that the reader has read the Standard LISP Report.[12] To present examples of the compiler source code, we use the REDUCE Algol-like implementation language RLISP.[13] A subset of this language is also used in the Standard LISP Report. When it is necessary to illustrate particular implementation details, we shall talk in terms of specific LISP interpreters on two computers with which we are most familiar, namely the PDP–10 family (which includes the DEC system 10 and 20) and the IBM System/360 family (whose members and relatives amount by now to a considerable group). For conciseness, we shall refer to these classes of machines as the PDP–10 and IBM 360, respectively. There are of course other LISP interpreters for these machines which use different protocols from those we assume here. Use of these interpreters would therefore require modification of our machine specific code to support the differing conventions.

2. CHOICE OF PORTABLE LISP COMPILER MACROS

In designing the architecture of our target abstract machine, we were naturally influenced by the machines currently at our disposal, which included IBM Systems 360 and 370, PDP–10s and DEC systems 20, a Burroughs B1800 and B6700, Univac 1108 and CDC 6600/7600. Moreover, the abstract LISP machine had to be consistent with the LISP interpreter implementation with which the compiler was to be used. After careful consideration of the facilities offered by such machines, and the structure of the corresponding LISP interpreters, we decided on an abstract machine with these general characteristics:

(a) A number of general purpose registers;
(b) a frame allocation mechanism, which provides for a stack of frames for storing temporary results and local variable values during a function evaluation; and

(c) a register saving protocol which leaves to each function the responsibility of saving registers that contain values required later in this function (usually the argument values); i.e it is assumed that most function calls destroy the contents of registers.

The model assumes that arguments of a function are passed in this set of general purpose registers rather than directly on a stack. The value of a function is assumed to be returned in the first such register. The number of such registers is a parameter in the model, although Standard LISP states that a maximum of 15 is expected. This would seem to impose a limit on the number of arguments permitted in function definitions, but in fact this is not the case. A number of possible solutions exist; the simplest is to have the last register contain a list of the remaining arguments. Moreover, the registers in the abstract machine need not be real registers in the target machine, as memory locations set aside for this purpose can also be used. Finally, it is rare that the full 15 real registers are actually used; only two or three 'real' registers are required for acceptable efficiency. For example, in our IBM 360 implementation, only the first two registers in the compiler implementation are actual registers. The remainder are in a contiguous block of memory. Since such registers are fixed place holders for expression references their use often avoids the cost of allocation and deallocation of frame locations. Even when registers are actually simulated by memory locations and thus suffer the same access times as frame references, the access may in fact be slightly faster, since no indexing is required.

The most critical design decision which we made was clearly the choice of an architecture with general purpose registers rather than a pure stack. Our experience suggests that a pure stack design is much better suited to list processing in general and leads to a much simpler compiler design. However, machines with high speed general purpose registers appear to be with us for a long time to come and this biased our design in favour of a register based model. A stack based compiler is also under investigation as part of another project.[14]

Finally, the register saving protocol chosen permits many of the frequently used small functions to do all their computation in registers, without having to use stack locations; this would not be possible in a scheme that automatically saves all registers on function entry, or even a scheme that saves only those registers that are changed by a function. It is important to note that most of the c-macros do not change any registers, with the instructions to load an explicit register, and to link to an external function being the major exception.

The actual c-macros chosen are fairly straightforward and can usually be implemented on a given machine in one or two instructions, or via a subroutine call. The basic c-macros are 16 in number as follows:

i. *function entry and exit*	
ENTRY name type nargs	indicates start of this function and declares its name, type and expected number of arguments.
EXIT	exits to caller of this function.
ii. *frame handling*	
ALLOC n	allocate a frame of 'n' words, where 'n' is a non-negative integer.
DEALLOC n	deallocate this frame.

iii. *loading and storing*

LOAD reg exp	load expression 'exp' into register number 'reg', where 'reg' is an integer greater than 0. The allowed form of 'exp' will be described later in Section 4.2 (ii).
STORE reg floc	store the contents of register number 'reg' in 'floc', an integer less than or equal to 0 representing a frame location, or the form (FLUID ⟨id⟩) or (GLOBAL ⟨id⟩) representing a non-local cell. Because of its common occurrence, the form (STORE NIL floc) is also allowed to store a NIL in floc.

iv. *control c-macros*

JUMP adr	an unconditional jump to the identifier 'adr'.
JUMPNIL adr	jump to 'adr' if register 1 contains NIL.
JUMPT adr	jump to 'adr' if register 1 does not contain NIL.
JUMPE adr exp	jump to 'adr' if contents of register 1 is **eq** to 'exp' (**eq** being the LISP function of that name).
JUMPN adr exp	jump to 'adr' if contents of register 1 is not **eq** to 'exp'.
LBL adr	used to define an address for jumps.

v. *subroutine linkage*

LINK name type nargs	link to function 'name' of type 'type' with 'nargs' arguments. The arguments are in registers 1 through 'nargs'.

vi. *non-local variable handling*

LAMBIND regs alst	bind the values in the list of 'regs' to the **fluid** variables named in 'alst'.
PROGBIND alst	bind NIL to the **fluid** variables named in 'alst'.
FREERSTR alst	restore the **fluid** variables named in 'alst' to their previous bindings.

This set of c-macros is complete enough to compile efficient code from most LISP definitions. However, some additional c-macros can be used on many machines for even more efficient code production, and these have been included as options in the current compiler. They are:

JUMPC adr reg type	jump to 'adr' if contents of register 'reg' is of type 'type' (i.e. the condition is satisfied).
JUMPNC adr reg type	jump to address 'adr' if contents of register 'reg' is not of type 'type' (i.e. the condition is not satisfied).
LINKE name type nargs n	link to function 'name' of type 'type' with 'nargs' arguments, deallocate frame of size 'n' and exit to caller directly from 'name' (i.e. compile tail recursion and function chains more efficiently).

A more complete description of these c-macros is given in Appendix A. The absence of CAR, CDR and so on, as c-macros does not mean that our LISP 'machine' does not

consider these of primary importance; rather, we have implemented a general mechanism for efficiently coding certain expressions inline (the 'exp' in iii and iv above), which include such functions (see Section 4.2 (ii)).

With this brief description of our abstract machine in mind, we can now take a look at the general structure of the compiler itself.

3. GENERAL DESCRIPTION OF THE COMPILER

3.1. Preliminary

In general terms, the compiler is a Standard LISP program which occupies approximately 2000 lines of LISP code in a prettyprinted form or 1350 lines of RLISP code (including comments). A listing is given in Appendix B, and a magnetic tape of this compiler may be obtained by writing to the authors. The output from the compiler after assembly is expected to work in a Standard LISP environment, augmented by seven functions which are required to support the compiled code. A description of these functions is given in Appendix C. The compiler module contains two entry functions **compile** and **compd**. **Compile** takes a list of uncompiled function names as argument, converts them into their compiled equivalents and returns its argument. **Compd** is analogous to the Standard LISP function **putd**. It has three arguments, the function name, its type (for example, **expr**, **fexpr** or LISP **macro**) and its body.

It compiles the body into a sequence of c-macros which are then assembled and the relevant code pointer stored with the function name. It returns the name of the function. The actual code for a given function is generated by a function **comproc** which takes two arguments, a lambda expression and a function name, and returns a list of c-macros. **Comproc** generates this code in three passes, with each pass a LISP function with the names **pass1**, **pass2** and **pass3**, respectively. **Pass1** performs a global analysis of the function body producing as output another LISP expression representing the function body with a more precise syntax (reflecting such things as variable and function type) more easily translated into code. **Pass2** converts this expression into c-macros, using a recursive function **comval** to do the basic code generation. Finally **pass3** uses a series of optimizing functions to improve the generated code. The detailed structure of each of these functions is given below.

3.2. The first pass of the compiler

The very first LISP compilers written in LISP included a first pass which was essentially a preprocessing of the body of the function in order to make the actual code generation a more straightforward process. Most LISP compilers written since then have followed this practice. In the present compiler, the first pass performs the following transformations on the function body:

i. LISP **macro** expansion;
ii. Non-local variable and constant identification;
iii. Non-local variable binding;
iv. Functional argument identification and transformation; and
v. Pseudo-function Analysis.

The actual details of these transformations are as follows.

i. *LISP* **macro** *expansion*

Most LISP systems include a function type called LISP **macro** (not to be confused with c-macro), which is used to customize code: During LISP evaluation, a LISP **macro** is first evaluated once to produce a new form, and this form then evaluated for the value; during compilation, the new form should instead be compiled to generate as efficient code as possible. However, in order to make the code generation itself as efficient as possible, this is done completely before any code generation begins so that LISP **macro** expansion, and other preprocessing of the new forms need not be done during **pass2**. Since **pass2** is top-down in nature, and if the LISP **macro** expansion was deferred, it would have to be performed whenever an argument involving the invocation of a LISP **macro** was examined, leading to unnecessary expansion of the same expression many times over.

In addition to regular LISP **macros**, the first pass also considers a separate set of macro-like open codings of functions defined by means of a **cmacro** indicator. The corresponding property is a **lambda** expression which is folded into the relevant form by **pass1**. In particular, this mechanism is used to expand chains of **cars** and **cdrs** inline. For example, **caar** has the property

```
(LAMBDA (U) (CAR (CAR U)))
```

under the **cmacro** indicator.

ii. *Non-local variable and constant identification*

Standard LISP recognizes three different binding mechanisms for variables namely, **global**, **fluid** and **local** each of which must be handled by the compiler. Standard LISP conceals any differences between shallow and deep binding implementations of the **fluid** mechanism, and our compiler likewise conceals these differences in the definitions of the c-macros described below. In point of fact, these distinctions only become important when expressions are being compiled since in interpreted expressions, all variables are considered **fluid**, although they are not explicitly declared as such; only in compiled functions do **local** variables become relevant. In fact, in order to generate as efficient code as possible, most LISP compilers consider as many variables as possible as **local**, and consequently any non-local variables must be recognized so that their names may be explicitly retained during the compilation process. Consistent with this model, the most logical explicit declarations would be **global** and **local**, with **fluid** as the default; both **global** and **local** would then be optimizing declarations; **global** to indicate that no binding occurs for this variable (and hence only one fixed cell need be used to reference its value in **alist** models, as opposed to the more expensive **alist** search for a **fluid** in deep binding models), and **local** to indicate that the name of the variable need not be preserved and hence a single cell known only to the compiler is sufficient to reference it. However, since **local** is the most common variable type in compiled functions, we have followed the usual LISP philosophy of considering variables **fluid** by default in interpreted functions and **local** in compiled functions. This represents a major semantic difference between interpreted and compiled functions, and one which has caused many problems in LISP implementations. These differences are discussed further in Section 6.

As a result, each variable in the function body is inspected to ensure that it is local within the body or has previously been declared as non-local. If an undeclared non-local variable is detected, a diagnostic message '⟨variable⟩ declared fluid' is printed and the variable so declared. Constants such as numbers and strings and global variables with constant values such as T and NIL are also quoted explicitly during this pass to make their later analysis as straightforward as possible.

iii. *Non-local variable binding*

If a **fluid** variable occurs in a **lambda** or **prog** list, then the old binding of the variable must be saved, and a new binding established for the current environment. The compiler handles this by assigning a frame location for each such variable on entry to the block (this frame location may not actually be used in some implementations). A c-macro (LAMBIND or PROGBIND) whose argument is a list of pairs of variable name and frame locations is added at the beginning of the block, and a c-macro FREERSTR at the end. The intent of the binding c-macros then is to save the current **fluid** value and to replace it with the appropriate new value (which is either in a register or NIL). The system will then use the appropriate **fluid** value mechanism for handling the variables in the new block. On exit from the block, FREERSTR simply restores the old **fluid** values from where they were saved. In the case of **progs**, a NIL will be stored in the **fluid** cell by PROGBIND after saving the original value whereas in LAMBIND the relevant register value is stored (thus LAMBIND also receives a list of registers as one of its arguments).

It is important to realize that the old value of the fluid variable does not actually have to be stored in the location left in the indicated frame location; rather, it is up to the implementor of LAMBIND, PROGBIND and FREERSTR to decide whether to use these locations. This is one reason that these three additional c-macros are provided when they could be expanded inline in terms of simpler c-macros as described in Appendix A; namely to provide maximum flexibility for implementors who wish to exploit additional features in their interpreters. For example, in our interpreter for the PDP–10, an additional push down stack is employed for **fluid** bindings, so that the interpreter can access the value at any time, and to provide support for an efficient **errorset** and **funarg**. Alternatively, in a deep binding **alist** model, the LAMBIND or PROGBIND can consist of a push on the **alist** and the FREERSTR a pop off the **alist**, again with no actual saving of values in the frame.

iv. *Functional argument identification and transformation*

An area that has led to a variety of LISP implementations (and some controversy) is in the handling of functional arguments. It is not our intent here to join that controversy, but simply to provide a mechanism for handling such constructs in existing interpreters. If the functional argument is the name of a function, in **pass1** the form (**function** ⟨name⟩) is replaced by the value of (**mkfunc** (**quote** ⟨name⟩)). On the other hand, a lambda expression is given a name generated by the compiler, the expression compiled as a function of that name, and the generated name then treated as above. In many systems, **mkfunc** may simply be defined as:

```
expr procedure mkfunc u; list('QUOTE,u);
```

provided a constant environment is assumed. On the other hand, it can be a call to **function** or **funarg** if the underlying LISP supports a complete functional argument

handling mechanism. Baker's article[15] on the combination of deep and shallow binding schemes into a single consistent model is an illuminating discussion of these and related issues.

Finally, a parameter variable in a **lambda** or **prog** form used as a function in the body of the definition is easily recognized by our variable analysis. Calls of such functions are transformed by this pass into a call to **apply**, so that their application becomes the responsibility of the interpreter.

v. *Pseudo-function analysis*

The first pass also checks for any **fexprs** apart from a standard set **cond**, **prog**, **and**, **or** and so on, for which open coding is provided in the second pass. The arguments of such functions are listed together and then passed quoted to the second pass, to be compiled as the single argument of the **fexpr**; the LINK c-macro emitted in **pass2** will still indicate that an **fexpr** is being called. For example, if F1 were an **fexpr**, then

(F1 U V)

would be transformed into

(F1 (QUOTE (U V))).

There remains the problem of deciding how to process the identifiers U and V in this expression. Most often, they are intended to be parameters, passed by name, and their values found during the execution of F1. In this case, it is necessary to declare them **fluid** so that their names are not lost in the compiled code. On the other hand, the intent might be simply to pass U and V as symbols, with no evaluation intended in F1. In this case, a **fluid** declaration would be inappropriate, although correct. For this reason, we leave it up to the programmer to add whatever declarations are necessary, even though an oversight in this regard may lead to errors which are difficult to find.

In many **alist** model interpreters, **fexprs** are assumed to be invoked with the current **alist** as a second argument. This has two effects: an **fexpr** definition must have a second argument appended, if needed; and, the value of **alist** should be loaded into register 2 in the appropriate cases. We will assume that the relevant **fexpr** function is defined with this second argument, or it is added by the Standard LISP functions **df** and **putd** if needed. We also leave to the implementor of the c-macros to emit the relevant (LOAD 2 (GLOBAL **alist**)) prior to expanding a LINK to an **fexpr**.

Thus at the end of **pass1**, only 'ordinary' functions remain. In other words, in generating a call to a currently undefined function, the compiler assumes that it is an 'ordinary' function in the sense that it expects exactly the number of arguments given and that they are all evaluated. Thus any attempt later to define such a function as other than of this type will lead to run time errors. Users must therefore be sure that all **fexprs**, LISP **macros** and so on, are defined before their call in any function being compiled to avoid such errors.

Rather than handling each special case function explicitly in **pass1**, we associate a special pass one evaluation function with the function name, using the indicator **pass1fn**. Special cases in the second pass are similarly handled by an indicator **compfn**. This table driven structure for the compiler makes it more straightforward to add specific open coding for any given function, although to design the code for such a case requires a fairly complete knowledge of the various tables which the compiler uses to keep track of such things as labels and jumps. Also, a LISP **macro** is expanded in

pass1 only if the function does not have an indicator **compfn** which is used later in **pass2** for open coding. Thus one can override a LISP **macro** definition of a function by such indicators.

Some earlier compilers included in **pass1** a test for top level tail recursion which was then converted into an iterative definition. In the present compiler, we found it more straightforward to do such recursion removal in the second pass as we shall discuss later. If however a more general recursion removal scheme were implemented to include many of the cases discussed by Risch[16] for example, then such transformations would probably best be done in the first pass.

3.3. The second pass of the compiler

The basic code generation is done in the second pass of the compiler by the function **pass2**. Its definition is as follows:

expr procedure pass2(exp); comval(exp,0);

Comval is a recursive function which takes as arguments a LISP S-expression and an integer (whose role we shall describe later in Section 4.2 (iii)) and generates the c-macros corresponding to the input expression. These c-macros are not actually returned by **comval** but instead stored in a global variable **codelist**, since we have found it easier to manipulate the generated code in the later optimization steps in this manner.

Like **pass1**, the design of **comval** is also that of a straightforward LISP evaluator, except that there are fewer special cases to worry about after **pass1** has processed the expression. It first checks for an atomic or tagged constant argument (constants are returned by **pass1** as (QUOTE 'expression')), in which case it simply loads register 1 with the value of the argument. Next it handles functions marked for open coding by a **compfn** indicator, such as **cond**, **and**, **or**, **list**, **prog2**, **progn**, **prog**, **go**, **return** and **setq** for which the appropriate function to perform the specific open coding is used. Functions defined via a lambda construction are next considered followed by an attempt to convert recursive forms to iteration using the function **comrec**.

For the remaining functions, the code is generated by a function **call**. In this case, the arguments of the function must be separately compiled (using a function **comlis** to perform repeated calls on **comval**), and the values loaded into the appropriate argument registers before a LINK to the function can be emitted. This means in general that each argument expression must be compiled and its value saved in a frame location, to be restored to the appropriate register after all compilations are complete. However, if the expression can be compiled using only the single register into which its value must ultimately be loaded and there are no side effects during its evaluation, then its loading may be deferred until all other argument compilations are complete. In particular, if the value of such an expresssion already resides in a register, the value may be taken directly from that register without recomputation. Such expressions will be referred to as having an **anyreg** property, and will be considered in more detail in Section 4.2 (ii).

The special function cases handled by **comval** through compfn indicators vary in complexity. Some are quite simple, such as **prog2** and **progn**, in which case each argument is evaluated sequentially by **comval** so that the value of the last remains in register 1. For **setq**, the second argument is evaluated and a STORE c-macro is

emitted to store register 1 in the location of **setq**'s first argument, which is either a variable whose location is known or for which an access function can be inserted by the loader (**global** or **fluid**). Other cases, such as **cond**, are much more complicated. However, a fairly complete account of the general principals involved in compiling conditional expressions is given in Allen's book,[2] so we shall not repeat that discussion here.

One of the most interesting examples of special case function handling occurs in compiling any expression where a boolean value is needed.[17] In many cases, the truth or falsity of such expressions indicates the need for branching in the program flow, (for example in a conditional expression), rather than a need for the actual value. As a result, such expressions (involving **and**, **or**, **not**, etc.) are compiled by general functions **combool** and **comtst**, reminiscent of the optimized boolean code generator in the original 704 compiler. These use two **fluid** variables; **fn** to keep track of whether we are in an 'and' or 'or' situation, and **switch** to determine under what conditions a branch is necessary.

Comtst takes two arguments; the boolean expression, and a label. **Comtst** generates a jump to this label if required, otherwise the program flow is assumed to continue. If the variable **switch** is true, **comtst** generates this jump when the value of the expression would be true. If **switch** is false, then transfer occurs on the falsity of the expression. This enables the compilation of top level tests involving **null** (or **not**) to be trivially performed; one simply negates **switch** and considers the argument of **null** instead.

In the general case, therefore, **comtst** uses the values of the variables **fn** and **switch** to determine whether a JUMPT or JUMPNIL c-macro should be added at the appropriate points in the generated code. However, like **comval**, it also considers special cases, such as **eq**, for which it generates a JUMPN or JUMPE c-macro. A test is also made for an indicator **comtst** on functions seen by **comtst**. This allows for the open coding of other functions in boolean expressions at the implementor's desire. For example, to test for expressions tagged in a simple way (such as atoms and numbers in most systems) and to set up the necessary transfer instructions can often be done in a way which leaves the results in the compiler registers unchanged. For this reason the optional c-macros JUMPC and JUMPNC mentioned ealier are available as compiler c-macros. If the expression analysis indicates that the value of the boolean expression is required in some register, or must be saved in the function's frame, then its value must be loaded after the expression is compiled. This could be done by having **combool** first call **comtst** and then append a subsequent load of T or NIL into the required location at the appropriate points in the code. However, our studies have shown that more compact code usually results if such expressions are compiled by direct calls to the relevant boolean functions (e.g. **eq**, **null**, etc.) than by open coding, since the direct call of a function usually generates less instructions than the test c-macro followed by the necessary loads of T and NIL.

The compilation of **prog** is messy but straightforward. A **fluid** variable **golist** is used to keep track of labels, and interrogated during the compilation of a **go** form to ensure that the label exists in the current **prog**. A return label is also generated for the compilation of **return** forms (which are compiled as a load of their argument and a jump to this label). Frame locations must also be allocated for local variables in the **prog** and these variables initialized to NIL by the generation of the appropriate STORE c-macro. Other constructs are then compiled by sequential calls to **comval**.

Finally, a **lambda** expression is compiled essentially as an 'open coded' function with the arguments loaded into registers like a normal function call. The function body is then compiled using **comval**. In the current compiler we have restricted the number of arguments in a **lambda** expression to **maxnargs**, the maximum number of arguments allowed in a function. It should be noted however, that as with arguments to a function, we do not really have to impose this restriction; we can once again use the last register for a list of the remaining arguments, or these remaining arguments can be placed directly in new frame locations beyond those already allocated.

The c-macros for the allocation and deallocation of a function's frame are added by **comproc** after the code for the body of the function is generated and the optimizations discussed in the next section have been completed. An ALLOC is attached at the beginning of the code sequence even for a frame of size 0, since some implementations must always mark their own push down stack at this point. A DEALLOC is attached at the end of the code sequence as long as not all exits terminate in a LINKE. Finally, after this is done, an ENTRY c-macro is added at the beginning and an EXIT at the end (again only when all exits do not terminate in a LINKE) to complete the code generation.

4. CODE OPTIMIZATIONS

With this simple definition of **comval**, it is possible to generate our c-macros from any given LISP input. However, a quick perusal of the output reveals many inefficiencies in the generated code sequence. For example, consider the following function compiled in this manner:

```
symbolic procedure 1stchr(u,v);
  if null(cdr(u)) then car(u) . (NIL . v)
    else car(u) . (list(1stchr(cdr(u),v)) . NIL);
```

Using the open coding of the basic LISP functions we have described in the previous section, the c-macros generated by **comval** for this function are as follows:

```
(ENTRY LSTCHR EXPR 2)
(ALLOC 3)
(STORE 1 0)
(STORE 2 -1)
(LINK CDR EXPR 1)
(JUMPNIL L1)
(JUMP L2)
(LBL L1)
(LOAD 1 0)
(LINK CAR EXPR 1)
(STORE 1 -2)
(LOAD 1 (QUOTE NIL))
(LOAD 2 -1)
(LINK CONS EXPR 2)
(LOAD 2 1)
(LOAD 1 -2)
(LINK CONS EXPR 2)
(JUMP L3)
(LBL L2)
```

```
(LOAD 1 0)
(LINK CAR EXPR 1)
(STORE 1 -2)
(LOAD 1 0)
(LINK CDR EXPR 1)
(LOAD 2 -1)
(LINK LSTCHR EXPR 2)
(LOAD 2 (QUOTE NIL))
(LINK CONS EXPR 2)
(LOAD 2 (QUOTE NIL))
(LINK CONS EXPR 2)
(LOAD 2 1)
(LOAD 1 -2)
(LINK CONS EXPR 2)
(JUMP L3)
(LBL L3)
(DEALLOC 3)
(EXIT)
```

for a total of 37 c-macros. As explained earlier, positive integers in the arguments of a LOAD or STORE denote compiler registers and 0 or negative integers denote frame locations.

An immediate problem with the code generation so far is evident from a study of this c-macro sequence, namely that the code is generated in a very local manner, by processing the function body expression by expression. The need for optimizations in the code generation process is therefore obvious.

The optimizations we consider in this compiler may be divided into three general categories:

1. Optimizations arising by specific open coding of various functions;
2. Optimizations which result from a knowledge of global information collected during the compilation process; and
3. Optimizations arising by post code generation modification of the c-macro sequence.

Before describing these optimizations, it is important to understand that any given optimization can save either in execution time or in the number of instructions generated, or both. When a conflict arose we preferred to save instructions at the expense of increased execution time rather than vice versa, because space for compiled code is usually the more critical resource in the large scale LISP applications of particular interest to us.

Since very little is written in the literature about the mechanics of such optimizations for LISP compilation, although their use in other language compilers is well understood, we shall discuss each of these in some detail. A review of the more formal aspects of such optimizations can be found in References 18 and 19.

4.1. Optimizations arising from open coding of functions

i. *Preliminary*

It is clear that the overall cost of linking to functions in LISP is very high since most LISP programs are constructed from a large number of relatively small functions.

Therefore, one obvious optimization is the use of open coding of such functions whenever possible. Even the very earliest LISP compilers open coded the LISP constructs such as **cond**, **prog**, **go**, **return** and the boolean functions, because they are **fexprs** and would lead to very inefficient code if they were not open coded. We have already considered some of these in Section 3.3. In these cases not only does the open coding avoid explicit calls to these functions but it also generates less code. Most of the functions open coded do not use all registers, often only one or two, so that the code needed to save registers for a general function call is eliminated; also, information on the register contents is available after the open coded call, permitting further savings in later code segments.

The current compiler provides specific functions for open coding 15 such functions, namely **and**, **cond**, **go**, **list**, **map**, **mapc**, **mapcan**, **mapcar**, **mapcon**, **maplist**, **prog**, **prog2**, **progn**, **return** and **setq**. In addition to this method of open coding which uses the **compfn** indicator, there are three other mechanisms in use. The first of these, involving boolean compilation, was also discussed in Section 3.3, namely the use of an indicator **comtst** to tag certain functions which can be open coded in boolean tests. An example of such open coding is **eq**, although in the IBM 360 compiler **atom** and **numberp** are also handled in this manner. The next mechanism recognizes that many simple functions such as **car** and **cdr**, have no side effects and can, in fact, be loaded by direct instructions into any register. Such functions are given an indicator **anyreg** whose corresponding property is a pattern for the replacement of the function. For example, **car** in our PDP–10 implementation has the pattern (HLRZ x y) and in the IBM 360 implementation (L x 0 (R0 y)). (Here 'x' represents the result register for the (CAR exp), and 'y' refers to the register, constant or frame location of the **anyreg** 'exp'). This mechanism is discussed in more detail in Section 4.2 (ii).

The third mechanism is used when the function can be handled inline, but fixed registers are needed. The normal **comlis** argument preparation is executed, but instead of simply LINKing to the function, a property under the indicator **open** is used to 'replace' the (LINK fn type n) by the appropriate inline code.

ii. *Optimized* CONS *compilation*

Since the basic functions **car**, **cdr**, **atom** and **eq** can usually be implemented in one or two instructions using the techniques we have described so far, it is worth asking whether the fifth basic function, **cons**, can also be implemented in an optimal manner. Unfortunately, **cons** takes several instructions to remove cells from and update the free storage list, apart from updates of a **cons** counter or calls on the garbage collector. It is therefore usually impractical to have open code for this function. However, it is fairly easy to add to the interpreter efficient implementations of the functions

symbolic procedure ncons u; u . NIL;

and

symbolic procedure xcons(u,v); v . u;

In terms of these, chains of **cons** calculations (such as in one form of **list** evaluation) can be compiled with a minimum of register loading activity.
For example, the form (**list** U V W), if expanded into

(**cons** U (**cons** V (**cons** W NIL))),

compiles to:

```
(LOAD 1 -2)
(LINK NCONS EXPR 1)
(LOAD 2 -1)
(LINK XCONS EXPR 2)
(LOAD 2 0)
(LINK XCONS EXPR 2)
```

where we assume U in frame position 0, V in −1 and W in −2. Because of the frequent use of **cons** in LISP, this particular special case optimization is of some importance.

We do not use this method of **list** compilation in the current compiler, however. Instead, a function **comlist** converts a call on **list** into a call on one of the functions **ncons**, **list2**, and so on, themselves compiled using the optimizing **cons**. This produces more compact code in general than any other open coding for **list** that we know. In addition, this technique could be extended even further by defining (**xlist2** U V) to be (**list2** V U) etc, to mimic the **cons** case. However, this latter option has not been explored in the current compiler. For more than five arguments to **list** which occurs very rarely in most LISP programming, the compiler uses the **cons** expansion for the additional arguments.

4.2. Optimizations which depend on global information

The next general class of optimizations we consider are those which depend on the gathering of global information during the code generation in **pass2**. Those considered in the current compiler are as follows:

i. *Maintaining register status*

A very important optimization involves keeping track, as much as possible, of the expressions stored in each register at each stage of the compilation. This requires a detailed analysis whenever a branch occurs so that the differences in the registers after the branch can be recognized. The utility of this optimization is clearly enhanced by open coding since a link to an arbitrary function destroys all knowledge of the previous contents of the registers. However, since most of the functions supported by Standard LISP, even those which are not open coded, tend to be written in assembly language in a manner which uses very few of the compilers's registers, it is possible to use flags **onereg** and **tworeg** to indicate that such functions use only the first one or two registers respectively and therefore keep a little more register information around when they are used. Such information could also be generated during the compilation of a function and stored with it for later use. We do not however collect such 'history' information in the current compiler, although it is a possibility for a future version.

ii. *Deferred register loading*

Another optimization of great importance is already encountered in the implemented form of **comval**. Since our function linking mechanism requires that each argument be loaded into its appropriate register, it is useful to determine whether the value of an expression can be loaded directly into that register without the intermediate need of any other compiler registers in the process. A variable whose value may be found in another register or in the current frame is obviously of this type, as are **fluid** variables and quoted expressions. However, as discussed in the previous section, most LISP

implementations allow for a class of functions (usually simple ones such as **car** and **cdr**) which also have this property. The general test for this property is therefore handled by a function **anyreg**, which looks for the indicator **anyreg** on the property list of such functions. **Comval** therefore checks first for such a property before going on to the other cases. If the expression has this property, its loading is deferred for direct loading later, since it will not destroy any other registers already loaded.

As a result, the allowed expressions ⟨exp⟩ which may occur in a LOAD, JUMPE or JUMPN c-macro can be one of the following:

1. a number <= 0 giving relative frame position of a local variable;
2. a number >0 which denotes the register where the argument can already be found;
3. (QUOTE ⟨S-expression⟩);
4. (FLUID ⟨literal atom⟩) or (GLOBAL ⟨literal atom⟩); or
5. (⟨function⟩ ⟨exp⟩) where ⟨function⟩ has an **anyreg** property, and ⟨exp⟩ is another allowed expression.

A complication arises when one attempts to defer register loading during the evaluation of the list of arguments of an ordinary function. The majority of LISP interpreters employ a left-to-right argument evaluation in this case, and it is possible that the value of a variable in one argument may change in the evaluation of a subsequent argument. If we defer the loading of the value of the former variable until the latter argument has been evaluated, the incorrect value will be loaded, as in F(X, X: = Y). In addition, if other side effects such as printing are required during the argument evaluation, then these too must occur at the right time. As a result, we also require that functions tagged **anyreg** have no side effects so that their deferred loading is possible. To ensure that the argument evaluation occurs in the correct order where necessary, **comlis** calls **anyreg** with a second argument consisting of a list of all remaining arguments to the function. In other words, the enforcement of a LISP left-to-right argument evaluation protocol requires that an argument evaluation only be deferred if it is a quoted expression; and it is **anyreg** and the remaining arguments are all **anyreg** or cannot change variable values.

Since such deferred loading can really save a lot of instructions, and it is difficult in general to know whether a given function changes the value of a variable or not (it might do an **rplaca** on it, for example), another possibility is to ignore the left-to-right evaluation protocol. In fact, some compilers do this automatically, leading at times to unexpected results. As a compromise, we resolve this problem by means of a global variable ***ord**. If ***ord** is true, then left-to-right evaluation is enforced usually leading to extra code being generated. The default setting of ***ord** is false, so it is up to the user to set this variable to true if left-to-right argument evaluation is needed in any function evaluation.

iii. *Avoiding unnecessary loads*

The use of the knowledge of the current contents of the registers which we maintain in **regs** permits us to avoid many redundant LOADs during the code generation. However, this is a rather local optimization and does not permit the elimination of unnecessary LOADs in loops and other large structures. Some such LOADs can be removed by post code generation optimization as discussed in Section 4.3 but further attempts at such code elimination would require the use of a much more graphical

analysis of the function body than we have attempted in the current compiler. However, another class of redundant LOADs arises from the fact that LISP requires every function to return a value, and a compiler must follow this protocol unless it is known that the value is not really needed. To determine if this is in fact true, and thus avoid the generation of an unnecessary LOAD, requires keeping status information to determine whether the value of a particular calculation is actually required. This controls the loading of NILs inside nested **progs**, the loading of NILs for unterminated **conds**, and so on. For this purpose, we introduce a **status** variable as the second argument to **comval**. This variable can take the following five values:

0—top level in function body
1—lower level but value required (in register 1)
2—lower level but value not required
3—top level in **prog**
4—top level in a **prog** which is top level in a **prog**.

This thorough handling of expression status allows us to avoid loading values if they are not really required (i.e. when **status** >1). In addition we can recognize a non-terminating **cond** expression (i.e. one with a missing **else** clause) and return NIL as the default termination if **status** <2. Finally a **prog** within a **prog** need not load a NIL (or any **anyreg**, or no-side-effect expression) if one 'drops through' (or in any **return** at such a level).

As a result of this analysis, several restrictions in some previous compilers have been removed in this version. In particular, a **go** or **return** statement may appear anywhere within a **prog** where a value is not required. For example, a **go** at a second or lower level in a **cond**, as in

(**cond** ((**null** u) (**cond** ((**null** v) (**go** A))))),

is permitted in a **prog** , but (FN (**go** A)) is not.

Finally, knowledge of the fact that we are at the top-level of a function is used in **comrec** to aid in the detection of tail recursion.

iv. *Avoiding unnecessary stores*

It is also possible to eliminate many unnecessary STOREs generated during the compilation process by keeping track of the subsequent use of the stored values in later computations. This requires maintaining global information across segments of code, since STOREs will have to be inserted in the code sequence some distance before the subsequent LOAD that needs stored value. Two global lists are therefore maintained for this purpose; **stlst** and **slst**. The former maintains a list of all STOREs emitted during the code generation process, mainly to enable certain post code generation optimizations to be performed. **slst** is more relevant here; it is a list of all stored identifiers which have not yet been used. Whenever a variable is loaded it is removed from **slst**. On the other hand, when a STORE is emitted, **slst** is checked to see if a previous unused STORE of the identifier has been made. If so, the previous store is replaced by a NOOP c-macro which can be later removed in the third pass. (We do not consider the NOOP c-macro in our 'official' set, since it is purely internal to the compiler. Another c-macro in this category used in the compiler listing is CODE, which can be used for crutch coding on specific machines when needed). Of course, when we lose control of our variable layout, such as when a label is encountered, **slst**

has to be set to NIL. Even in this case, though, a preanalysis of the code to check label usage could prevent the loss of this information, although we do not attempt this in the current compiler.

Another optimization of some importance, especially in small functions, involves keeping track of whether a function's formal parameters can be directly loaded from the registers as initialized on entry to a function. Such variables are maintained in a separate register list, **iregs** and only stored in the frame when we encounter a situation where the registers are possibly used. In this manner, we can defer the storing of these values as long as possible, and, in some cases, never need to store them at all.

v. *Optimizing control structures*

Because of the local nature of the code generation, it is quite easy to generate a sequence like

(JUMPNIL L1) (JUMP L2) (LBL L1).

Such sequences can in fact be optimized by the general peephole mechanism to be described in Section 4.3 (i). A more difficult case involves sequences such as

(JUMP L1)...(LBL L1) (JUMP L2)...

To remove such redundant jumps in an efficient manner and to perform several other similar optimizations requires keeping track of all labels and jumps in the generated code, in preference to the less efficient process of searching for them in the whole code body. Consequently, whenever a LBL c-macro is first generated a table entry for that label is made in the list **lblist**. Whenever any kind of jump is made to that label, the unique reference to the jump c-macro list is added to the table associated with that label by a function **addjmp**.

If one now omits the jump to L2 in the above sequence, **addjmp** can determine that a label precedes it and rename all references to label L1 to refer to L2. The preceding label L1 can then be eliminated. In addition, if the previous c-macro is not a JUMP, a jump to L2 is inserted. Similar considerations can be made by the function **addlbl** which adds a label to the code; if the preceding entry in the code is a label, then all references to it can be renamed to the new label which is then inserted in its place.

4.3. Optimization of the generated code sequence

There are a large number of optimizations which can be performed on the code sequence generated by **pass2**. These optimizations are all lumped together in the present compiler in a function **pass3**. This is the most unsatisfactory part of the current compiler because most of these optimizations are **ad hoc** in nature. However, many of these could probably be handled by a general pattern-matching program which would enable the optimizations to be expressed in a more natural manner. This approach is a topic for continuing research, and we have made some progress in the context of pattern-match program transformations, a general 'optimizing' compiler-compiler, and efficient 'hashed' pattern matchers.[20, 21] The use of pattern-like target code optimizations in the 'PO' optimizer of Frazer[22] is an example of this.

We have investigated a number of such optimizations as described below. However, the potential list is much larger than this and will no doubt grow as more and more patterns are discovered in the generated code.

i. *Peephole optimization*

A wide class of optimizations arise from replacing local sequences of instructions by fewer instructions. This process is often referred to as 'peephole optimization'.[23] These optimizations differ from the others in this paper which are more structural in nature and usually depend on global information collected during the second pass for their efficient implementation. There are, in fact, three possible classes of peephole optimizations we could consider in our compiler model namely:

a. Replacing sequences of c-macros by other c-macros;
b. replacing sequences of c-macros by explicit instructions for the given machine; and
c. replacing sequences of explicit instructions by other instructions.

If one is interested in preserving the machine independence of the compiler output, then only the first class of optimization can be carried out in the c-macro generation phase. However, since all these classes of optimization start from a list of c-macros or instructions and result in a list of c-macros or instructions, the same mechanism can be used for all three since we assume that the loader program can handle a mixed stream of instructions and c-macros. Our implemented compiler therefore considers these all in one pass although it would be possible to divide them into separate passes if c-macro portability were a requirement. Alternatively, to output truly portable c-macros one can simply omit optimizations in class (b) and (c) above from the appropriate table.

The peephole optimizations which we consider are performed by a function **peepholeopt**. This function looks at each entry on the generated code list (actually considered in the reverse order since this is how it is passed from **pass2**). It checks for an indicator **optfn** on each c-macro considered. If this is found, the corresponding property is applied to the code sequence at that point, and if a valid pattern is found, the code is replaced by the improved code sequence.

As an example of an optimization in the first class, consider the sequence of c-macros

(JUMPNIL L1) (JUMP L2) (LBL L1)

which is often generated during a conditional expression compilation. The recognition of this sequence and its replacement by the shorter sequence

(JUMPT L2) (LBL L1)

is performed by a function **lblopt**, whose name is found under the indicator **optfn** on the property list of LBL. This optimization is applied to any of the testing c-macros, each of which has a conjugate c-macro for the opposite test.

ii. *Optimizing the* LINK *c-macro*

As we mentioned earlier, most of the c-macros we use may be implemented in a few instructions on most machines. However, in most cases the LINK c-macro must be implemented as a subroutine in the interpreter. Since such a subroutine can provide several entry points for the compiler, we investigated reducing the number of c-macros generated by providing several varieties of LINK c-macro. The two most common sequences encountered for which such code reduction is possible are

(LINK fn type nargs) (JUMP L1)...(LBL L1) (DEALLOC n) (EXIT)

and

(LOAD 1 n) (LINK fn type nargs).

By replacing the first case by the c-macro (LINKE fn type nargs n), we can eliminate the jump, and in some cases the label and DEALLOC c-macro if all exits from the function are via LINKEs. The second case could be replaced by the c-macro (LINKL fn type nargs n), which eliminates the LOAD. However, this second option has not been included in the current compiler. The check for the LINKE c-macro is made by the function **fixlinks**.

In the implementation of these linking c-macros, it is usually necessary to place information about fn, nargs, etc. in a data word. If the additional space needed to store the additional arguments in LINKE (and LINKL) takes as much space as the c-macro eliminated, there is clearly no advantage in using them. As a consequence, the use of LINKE is optional under control of the boolean variable ***nolinke**. LINKE is however used for example in our implementation on the PDP–10 which uses a stack for return addresses, and so permits a direct jump to the function, letting it return to the caller directly.

iii. *Eliminating common chains of code*

Another useful optimization is possible if one keeps careful track of JUMPs during the code generation. For this purpose, all jumps are added with their preceding code to a list **jmplist**. A post-code generation function **fixchains** is now used as follows: Each such code chain which terminates in an explicit JUMP is checked against all other code chains for any which terminate in the same jump and have common c-macros preceding the jump. If such chains are found with at least two common c-macros, then the initial chain is truncated to the point where the chains differ, and a jump inserted in the first chain to a label added at the point of divergence in the second. Although this makes the code marginally slower (because of the inclusion of the jump), it is clearly shorter than previously.

In searching for such common chains of code, we only look for the first matching chain to the one under study. In principal, it would be better to search all matching jumps for the largest such chain, but a study of our test code showed that little was gained by such a search and so it was not included.

The importance of the technique of common code chain elimination was stressed to us by Bill Wulf, and is discussed in detail in the description of the BLISS compiler.[24]

iv. *Optimizations dependent on having only one reference to a label*

Another advantage of keeping close track of all labels, and jumps to them, as described in Section 4.2 (v), is that we can easily determine in the final fixup stage if a label is actually referenced at all (if not, there will be no jump entries in the table) or only once, in which case some additional optimization is possible. For this scheme to work, it is of course necessary to remove jumps from the table, if optimization removes them from the code list. This is accomplished by a function **remjmp**.

One such optimization recognizes that the sequence

(JUMPx L1) ⟨M1⟩...⟨Mn⟩...(LBL L1) ⟨M1⟩...⟨Mn⟩...

may be shortened when the JUMPx (a conditional jump c-macro) is the only reference to the label and ⟨M1⟩ through ⟨Mn⟩ are c-macros which do not use register 1. In this case, the sequence can be replaced by

⟨M1⟩ ... ⟨Mn⟩ (JUMPx L1) ... (LBL L1) ...

Another optimization performed in the case of only one reference to a label is in fact an elementary form of loop invariant recognition and removal. A common sequence of code in this case is

(LBL L1) (LOAD 1 n) ... (STORE 1 n) (JUMP L1).

Since the LOAD is unnecessary after the JUMP, it may be taken outside the loop. If the c-macro preceding the label is also (LOAD 1 n), we can eliminate a LOAD. It is worth noting, however, that this code sequence suggests the use of a **while** or **repeat** loop in the source, which can be better compiled direct (as mentioned by Wirth regarding Pascal compilation).

v. *Frame contraction*

It is possible for the compiler to store a value in the frame and then never use it again. Since all such unused stores are present in **slst** as explained in Section 4.2 (iv), one action of **pass3** is to remove any such stores from the code sequence. However, the removal of these and other redundant stores during the code generation may also make the effective frame smaller by creating unused positions in it. To remove these holes in a more efficient manner than searching the whole code body requires keeping all references to frame positions in a list, and updating their positions in the frame if any holes are found. Such frame contraction is performed by the function **fixfrm**.

vi. *Use of registers in place of a frame*

In functions which do not need a LINK c-macro, or which link only to functions which use no registers apart from their argument registers (which we identify by flags as mentioned in Section 4.2 (i)) or which terminate in a LINK as the last c-macro before returning (i.e. only LINKEs), it is possible to use registers, if available, in place of frame locations for the storage of temporary results. If no registers are available, then fixed locations in memory could be used instead, although this is not done in the current compiler. Besides providing faster access (if registers are used), this alleviates the need for frame allocation and deallocation in the case when all frame locations are replaced in this manner. This check is also made in the function **fixfrm**.

It is instructive to see what happens to our test function **lstchr** when these optimizations are applied. The result is the code:

```
(ENTRY LSTCHR EXPR 2)
(ALLOC 1)
(STORE 1 0)
(LOAD 1 (CDR 1))
(JUMPT L1)
(LINK CONS EXPR 2)
(JUMP L2)
(LBL L1)
(LINK LSTCHR EXPR 2)
(LINK NCONS EXPR 1)
(LINK NCONS EXPR 1)
(LBL L2)
(LOAD 2 (CAR 0))
(LINKE XCONS EXPR 2 1)
```

a total of 14 c-macros now instead of the original 37 and with a frame size of 1 rather than 3! We have however done this at the expense of introducing more complex arguments in the c-macros, which consequently increases the complexity of the code generation process on the target computer.

We leave it to the reader to see how these optimizations came about using the techniques we have described here.

5. FURTHER OPTIMIZATIONS NOT IMPLEMENTED IN CURRENT COMPILER

The problem with compiler projects is that there are always further improvements which can be made to any current version, so that in some sense, such a project is never finished. However, in the interests of potential users of this compiler, we decided to 'freeze' it at the point described in this paper, and to leave any further improvements for later versions. We have already mentioned in this paper some possibilities for improvement, such as a pattern driven post code generation optimization. There are undoubtedly a large number of specific post code generation fixups for given computers which could be added fairly easily using such a pattern driven mechanism. However, there are in addition a number of general enhancements possible which we are currently studying and, in the interests of completeness, we present below those we consider of most importance.

5.1. Optimization of arithmetic expressions

LISP is notoriously slow at arithmetic, mainly because of the cost involved in converting a LISP number to its machine representation on entry to each LISP numerical routine. In addition, all type checking is done at run time. Such calculations can be significantly speeded up by recognizing that a given block contains only arithmetic operations and doing the number conversion only on entry and exit from the block. Such improvements require the addition of arithmetic c-macros to our set, and additional frame or garbage collection operations. An implementation of such optimizations is in fact available in the Maclisp compiler.[7] We have also tested such optimizations, by means of a preprocessing type checking phase, [25, 26] rather than in the compiler itself. This makes the addition of the arithmetic code improvements fairly modular using the open coding techniques available in the current compiler. We expect to make this extension available in the near future.

5.2. Block compilation

Another significant optimization that has been partially explored is the process of compiling a set of related functions at one time. It is an observed fact that such 'modules' provide only a few of the functions as entry points to the external environment; the rest of the functions serve only to provide support for the module. By analysing the entire set as a single entity, many more opportunities for shared code and open coding arise. Certain functions are used only once, and can be moved inline, others can be more efficiently compiled knowing the environment in which they will be called. This work will be discussed further in a later paper.

5.3. Generalizations of current compiler c-macros

There are several situations where the c-macro sequences would become significantly shorter if the restriction to testing on register 1 were removed from the various testing c-macros. The most obvious generalization is to replace testing on this register by a test on the general expression allowed as an argument in the JUMPE and JUMPN c-macros. With this generalization, the relevant c-macros would take the following forms:

```
(JUMPNIL adr exp)
(JUMPT adr exp)
(JUMPE adr exp1 exp2)
(JUMPN adr exp1 exp2)
(JUMPC adr exp type)
(JUMPNC adr exp type)
```

To implement this generalization would require the use of an auxiliary register (or registers) in general to compute the expression being tested. If a particular machine implementation required one of the compiler registers to be used for this purpose, then this would have to be saved and restored as part of the computation. Since some of the machines we are interested in have few real registers, we did not implement these generalizations in the current compiler. However, in some functions, especially small ones, the results could be dramatic . For example, consider the function defined by:

```
symbolic procedure mkex u;
  if mode eq 'symbolic then u else mkex1 u;
```

Assuming mode to be **fluid** this function currently compiles to:

```
(ENTRY MKEX EXPR 1)
(LOAD 3 1)
(LOAD 1 (FLUID MODE))
(JUMPN L1 (QUOTE SYMBOLIC))
(LOAD 1 3)
(JUMP L2)
(LBL L1)
(LOAD 1 3)
(LINK MKEX1 EXPR 1)
(LBL L2)
(EXIT)
```

However, with the c-macro generalizations above, this would become:

```
(ENTRY MKEX EXPR 1)
(JUMPN L1 (FLUID MODE) (QUOTE SYMBOLIC))
(LINK MKEX1 EXPR 1)
(LBL L1)
(EXIT)
```

Again, in this vein, one could replace the LOAD and STORE c-macros by a single c-macro MOVE having the form:

```
(MOVE exp1 exp2)
```

Thus (LOAD reg exp) and (STORE reg floc) would then become special cases of this single c-macro.

To generalize this process still further, one could provide for **anyreg** functions which take more than the one argument provided for in our current implementation. However, there is no practical reason why two or more argument functions couldn't be used. In particular, some of the binary arithmetic functions could benefit from such an extension, especially in an open coded arithmetic model. We would also have to address the problem of register allocation more carefully, to take full advantage of these two-register **anyreg**s.

5.4. Graphical analysis of the function structure

A final modification of the compiler, which would be the most radical change of those discussed so far, would be to do a complete graphical analysis of the function structure before any code was generated. By such means, register and frame layout could be determined on a global rather than a local basis, and a more thorough analysis of labels, loops and chains of common code would allow for even greater optimization. Applications of such analyses to other languages have been already described in the literature[10, 11, 18, 27] and would apply equally well here.

6. SEMANTIC DIFFERENCES BETWEEN INTERPRETED AND COMPILED CODE

A clear goal of this compiler is to produce optimal code, and, in order to achieve this, some semantic differences between interpreted and compiled code are required. Depending on the specific implementation of the c-macros and underlying LISP, these include:

1. local variables are not referenced by name, so we must declare non-local variables explicitly, otherwise an error can result;
2. LISP **macros** and **fexpr** functions must be defined before compiling a function in which they are used;
3. in interpreted expressions, the arguments of a function are usually evaluated from left to right; the order in compiled expressions is essentially arbitrary. (Unless of course ***ord** has been set to true during the relevant compilation);
4. the compatibility of the number of arguments supplied to a function and the number expected is not checked in open compiled expressions and this too can lead to errors;
5. open coded **anyreg** functions such as **car**, **cdr** and so on, do not check for the correctness of the data supplied; and
6. the functions that are open compiled may not be traced or redefined with respect to a compiled function, which must therefore be completely recompiled to effect such changes.

Of course, most of these differences could be eliminated in the compiled code at the cost of additional instructions, and less efficient evaluation, but in the interest of efficiency we forego compatibility on the above points.

7. THE LOADER

In any given machine implementation of this compiler, there still remains the problem of designing the compiler c-macro expander, assembler and loader for the code we have

generated. LISP assemblers (called LAP), are usually written in LISP, and the c-macro expansion can be easily accomplished at the same time. The LAP assembly and loading task remains quite machine dependent, of course, because there are many details, such as instruction lengths, determination of word boundaries and so on, which depend on the machine. Nevertheless, an essentially portable assembler and loader has been described in Standard LISP by Frick,[28] and we refer the reader to this paper for a general discussion of the principals involved.

8. HISTORY OF THE IMPLEMENTATIONS

A few words on our method of implementing this compiler. We started with a working compiler in LISP using Stanford LISP/360 on an IBM System/360. This was translated by machine into RLISP syntax[29] and made more structured by the insertion of **while** statements where possible. The resulting program was much more transparent as a result. All code generation was then replaced by 'machine independent' c-macros and the resulting code produced from compiling a wide range of programs studied. Several inefficiencies were apparent which were then corrected by improvements in the compiler itself. We then determined how to implement the c-macros themselves in fewer instructions by appropriate changes to the interpreter. The results of this was a compiler which produced approximately 20 per cent fewer instructions in compiling a large program (REDUCE[30]).

Our next step was to move this compiler to the PDP–10 using LISP 1.6 as the embedding LISP system. A comparison of the code sequences produced with the existing compiler revealed many further inefficiencies in the portable version. To remove these required a major restructuring of the code generation phase of the compiler. In addition, the c-macros were modified to be simpler in implementation and to provide a more natural LISP machine. Eventually we were able to produce code a little more efficient than the LISP 1.6 compiler we had used previously. Reimplementing the revised compiler on the IBM 360 gave a further 15 per cent code improvement for a total of 35 per cent since the project began. Further improvements since that time have taken this number to over 40 per cent. The code produced on the PDP–10 in compiling a large program such as REDUCE is now at least 10 per cent shorter than that produced by any other LISP compiler (such as that for Maclisp) on this machine. Implementations of REDUCE under both Standard LISP and Maclisp show also that the running time of the former is no worse than the latter, although interpreter design differences make it hard to compare actual compiled code running times in the two systems.

After these initial implementations, the compiler was then installed by Cedric Griss on a Univac 1108 using a LISP system from the Univeristy of Wisconsin. A further implementation was also made on a CDC 7600 using a LISP interpreter from the Univeristy of Texas. The feedback provided by these experiences told us where to improve the description of the implementation and also of remaining trouble spots in the compiler (such as fluid variable binding, for example) which were then corrected. In addition, the code produced in these cases was again significantly shorter than that produced by existing LISP compilers on those machines.

It should be stressed that each improvement was tested against the previous version which was known to work. This way we could isolate bugs in the improvements themselves and correct them quickly. Software refinement really works! It also goes

without saying that working interactively on a PDP–10 rather than in batch mode on an IBM 360 was also a decided advantage.

The most recent implementation was completed in the summer of 1978 by David Dahm on a Burroughs B6700. This particular implementation is rather interesting, in that a completely new LISP interpreter was written following the Standard LISP Report very closely; most functions are compiled from the LISP definitions given in the Report, and the target language is ALGOL, rather than machine code, as is the case for all of the other LISP systems for which our compiler has been implemented. Some work has also been done by one of us (MLG) on a FORTRAN based system, with the compiler emitting FORTRAN. This language is far less suitable than ALGOL, since recursion must be simulated in some way, usually violating the FORTRAN standard.

9. SAMPLE STATISTICS

We present in this Section some sample statistics relating to the performance of this compiler on the PDP–10 and the IBM 360 using the Utah Standard LISP interpreters on these machines.

9.1. Resource requirements for loading compiler

The whole compiler module is in two parts, exclusive of the loader; namely the part which generates the machine independent c-macros, and that part which defines the c-macros as explicit machine instructions. In Table I we therefore show numbers for both these parts in order to give a realistic indication of the resource use of the compiler. We show the number of eight bit bytes needed for both binary program and heap to load the compiler on both machines. On the PDP 10, we assume 4.5 bytes per 36 bit word, and on the IBM 360, 4 bytes per 32 bit word. However, it should be noted that a heap cell is 4.5 bytes on the PDP–10 and 8 bytes on the IBM 360.

Table I

	Lines of Rlisp		Heap space		Binary code space	
Module	PDP–10	IBM 360	PDP–10	IBM 360	PDP–10	IBN 360
Portable compiler	1,350	1,350	8,200	9,200	20,140	24,020
C-macro definition	335	240	6,545	7,040	5,110	4,072
Total compiler	1,685	1,590	14,745	16,240	25,250	28,092

9.2. *Compiler c-macro measurements*

The static and dynamic compiler c-macro distributions were investigated on the PDP–10 for two cases, the compiler self-compilation and a standard test set of algebraic calculations using REDUCE. The relevant percentage distributions are shown in Table II, with the c-macros ordered roughly by their relative rankings. It should be noted that the c-macros JUMPC and JUMPNC do not occur as they are not used in the PDP–10 implementation. In addition, ALLOC 0 is not counted in these statistics.

We notice that the c-macro usage is completely dominated by LOAD, with all LINKs in second place. Of course, some c-macros such as LINK and LAMBIND for

example require more code space and time than others such as JUMP and simple LOADs. On the other hand, the LOAD usage includes calls to **car**, **cdr** and so on, which are handled by the **anyreg** mechanism and so can be quite expensive on occasion. Therefore, on the whole, the above statistics are a reasonable measure of both space and time usage.

We also notice that there is little difference between the static and dynamic profiles, nor between the compiler self-compilation and the algebraic calculations. Our statistics are therefore similar to those found with other compilers and other test cases.[31]

Table II. Dynamic c-macro usage for test examples

	% static distribution		% dynamic distribution	
C-macro	Compiler	Algebra	Compiler	Algebra
LOAD	37·22	35·61	37·21	31·16
LINK	18·55	18·02	14·52	15·54
STORE	11·23	11·28	10·52	12·14
JUMPNIL	5·22	5·76	9·31	10·85
LBL	9·68	10·36	3·87	3·20
ENTRY	2·82	2·33	5·24	4·35
EXIT	1·85	1·65	3·70	2·67
ALLOC	1·79	1·55	2·22	2·89
DEALLOC	2·66	3·34	3·27	4·13
JUMP	3·95	3·78	3·35	3·05
JUMPT	1·68	2·75	1·73	6·60
LINKE	1·40	2·06	1·54	1·70
JUMPN	1·21	0·82	2·08	0·80
JUMPE	0·18	0·57	0·49	0·90
FREERSTR	0·28	0·06	0·47	0·01
LAMBIND	0·14	0·02	0·45	0·01
PROGBIND	0·14	0·04	0·03	0·00
TOTAL	100·00	100·00	100·00	100·00

10. CONCLUSIONS

The benefits of writing our compiler in a portable manner should be obvious; the code is carefully written to isolate the machine dependent and machine independent optimizations, making implementation and maintenance much simpler. In particular, a given optimization either applies to all versions, or just to some of them. By having only a single compiler for all systems, we have been able to avoid the wasted effort of reimplementing essentially the same code; rather we can concentrate on generalizing any optimizations found for one system so that they improve all compilers in a relatively machine independent way. What has been even more surprising to us has been the fact that the resulting compiler is smaller and more efficient than most existing production compilers; rethinking the compilation problem in terms of an abstract machine has not been an idle exercise.

ACKNOWLEDGEMENTS

We have benefitted in this work from discussions with a large number of our colleagues. In particular, the comments of David Dahm, John Fitch, Inge Frick, Cedric Griss,

Kevin Kay, Augustin Lux, Arthur Norman, Herbert Stoyan and Bill Wulf have been partcularly helpful.

APPENDIX A: The Portable Compiler C-macros

In this Appendix, we provide a detailed description of the compiler c-macros, and illustrate their implementation by a discussion of existing implementations on the IBM 360 and the PDP–10. These machines are sufficiently different that their description should be an adequate guide for anyone interested in implementing this compiler on a new machine. For completeness, a listing of the relevant c-macro expansion files for these two machines is given in Appendix B. In presenting these details, we assume that the reader has sufficient familiarity with these two machines to understand the instructions used on each; it is not our intent to teach assembly language programming for the two machines! Complete details on these instructions can be found in References 32, and 33 and the manufacturers' manuals.

A.1. General principles

We assume the existence of a set of abstract registers on the target computer up to a maximum set by the particular implementation. However, to be consistent with Standard LISP, 15 registers should be available. We also assume the existence of a contiguous region in memory which can be indexed in some manner from a base point. This region is normally called a push down list or a push down stack in LISP and is used to store the compiled function frames. This region is assumed to be scanned by the garbage collector.

On the IBM 360, the first two abstract registers are real registers, addressed symbolically as A and Q, and the remainder a fixed block of memory addressed via an index register. This notation of course reflects the influence of the original IBM 704 design on the particular interpreter being used. The push down stack is a contiguous block of memory used by both the interpreter and compiled functions. A register PDL points to the first word in a given function's frame, and a second register, PDS, points to the next free word in the push down stack. NIL is contained permanently in a register NILR, used also as the base register for atom addressability.

On the PDP–10, the first five registers are real registers and correspond to registers 1 to 5 in the machine. The remainder are in a contiguous block of memory. The push down stack is also a contiguous block of memory used by both interpreter and compiled functions. A register P points to the last word used on the push down stack. Only one stack register need be used on this machine, since unlike the IBM 360, one is permitted to index in the negative direction. Register 0 contains the value of NIL, and 0 is also the 'address' of NIL.

With the exception of LOAD, LINK and LINKE, the c-macros are assumed not to change any of the compiler registers. LOAD must only change the register being loaded. In general, LINK is assumed to change all registers unless the function linked to is flagged **onereg**, in which case only register 1 is assumed changed, or **tworeg** in which only registers 1 and 2 are assumed changed. Since LINKE includes an exit, it is irrelevant what it does to the registers, except, of course, that it, like LINK, must return its value in register 1.

The c-macros themselves are normally implemented as functions in LISP so that they may be called recursively if needed, and of course compiled. They also have a flag

MC on their property lists. The loader is assumed to know about this flag, and to apply the definition of such functions to the arguments provided. The value of each c-macro is a list of LAP instructions equivalent to the original c-macro, produced in the correct sequence for loading. The general format of the LAP instruction is left unspecified, but will probably be an atom or list (see Frick's discussion of the portable LISP loader for a comprehensive format [28]).

In describing the c-macros, the following argument conventions are used:

LABL:	⟨id⟩ representing a label for jumps
NREG:	{⟨integer⟩ >0 representing a register} ¦ NIL
REG:	⟨integer⟩ >0 representing a register
REGS:	list of REG
NAME:	⟨id⟩ representing a function name
TYPE:	EXPR ¦ FEXPR
NARGS:	⟨integer⟩ > = 0 representing the number of arguments in a function
N:	⟨integer⟩ >0 representing the number of frame locations
LOC:	⟨integer⟩ < = 0 representing a frame location
FLOC:	LOC ¦ (FLUID ⟨id⟩) ¦ (GLOBAL ⟨id⟩)
ALST:	list of pairs of form (⟨id⟩LOC) representing a fluid variable name and a frame location for storing its previous value
EXP:	REG ¦ FLOC ¦ (QUOTE ⟨S-expression⟩) ¦ (⟨afn⟩ EXP) where ⟨afn⟩ is the name of a function with an **anyreg** property.

A.2 The c-macros

ENTRY(NAME:id,TYPE:id,NARGS:integer):list

This c-macro sets up an entry point for the function named NAME. The actual details of this are usually deferred to the loader.

On the IBM 360, the loader associates a code pointer with the function named NAME under the indicator TYPE. This association is usually deferred until the code assembly is complete in case a function referenced by the loader is being redefined. In addition, a subroutine call is issued to set up register R3 as the base register of the function, and set PDL to PDS. Moreover, given that a subroutine call is needed anyway, the common cases of ALLOC 0 through ALLOC 3 are included by means of special case subroutines.

On the PDP–10, the loader associates a code pointer with the function named NAME under the indicator TYPE. As above, this association is deferred until the code assembly is complete.

EXIT():list

This c-macro causes a transfer back to the calling function.

On the IBM 360, a link register R2 must be restored before the branch can take place. As a result, this c-macro is implemented as a subroutine call of the form ((CNOP) (BC 15 48 (R0 R12))). The instruction CNOP tells the loader that the next instruction must be assembled at a full word boundary in order to make the code end with a full word. The offset 48 references the internal exit subroutine in the interpreter. On the PDP–10, the single instruction (POPJ P) causes the return address to be picked up from the location addressed by the P register and the relevant transfer made.

ALLOC(N:integer):list

N is assumed non-negative. If $0 \leqslant N < 4$ then NIL is returned since these cases are handled by ENTRY. Otherwise the value is a list of instructions which allocate a frame of length N on the push down stack and check to see that the bound on the stack is not exceeded.

For $N > 3$, then the following instructions are generated:

```
(LA PDS [4*N-4] (R0 R12))
(BXH PDS K4 0 (R0 R12))
```

where the value of the expression in the square brackets is used. To make this operation fast, register K4 has been dedicated to store the constant 4, and R12 used for the address of the relevant trap handler.

On the PDP–10, the sequence ALLOC(1), STORE(1, 0) can be done by the single instruction (PUSH P 1), so this can be used for N = 1 with or without the STORE in the c-macro sequence. The hardware stack bound check deals with the overflow case on this machine. The sequence ALLOC(2), STORE(1,0), STORE(2, −1) can be done by the two instructions (PUSH P 2) (PUSH P 1). Otherwise the P register is advanced N words, using (ADD P (C 0 0 N N)), and an instruction issued to make sure that the stack bound is not exceeded.

DEALLOC(N:integer):list

N is assumed non-negative. If N = 0 then NIL is returned. Otherwise, the value is a list of instructions which remove a frame of N words from the push down stack.

For $N > 0$, all that is required on the IBM 360 is the instruction (LR PDS PDL). However, since EXIT must be effected by a subroutine call, the sequence DEALLOC(N), EXIT( ) is also done by a subroutine call which includes the above instruction in addition to the EXIT sequence.

On the PDP–10 the P register is reset to a position N words less than its current position using the instruction (SUB P (C 0 0 N N)). (This is faster than N calls of (POP P 1), even for N = 1).

JUMP(LABL:id):list

This c-macro generates an unconditional jump to the location specified by the c-macro LBL(LABL).

On the IBM 360, this is the single instruction (BC 15 LABL), which limits addressability to 4K bytes. On the PDP–10, the instruction (JRST 0 LABL) is used.

JUMPNIL(LABL:id):list

This c-macro generates a jump to the location specified by the c-macro LBL(LABL) if the value of the contents of register 1 is NIL.

On the IBM 360, the instructions generated are ((CR A NILR) (BC 8 LABL)). On the PDP–10, ((JUMPE 1 LABL)), using the fact that a pointer to NIL is stored in register 0.

JUMPT(LABL:id):list

This c-macro generates a jump to the location specified by the c-macro LBL(LABL) if the value in register 1 is not NIL.

On the IBM 360, the instructions generated are ((CR A NILR) (BC 7 LABL)). On the PDP–10, ((JUMPN 1 LABL)).

JUMPE(LABL:id,EXP:any):list

This c-macro generates a jump to the location specified by the c-macro LBL(LABL) if the value of register 1 is to **eq** to EXP.

On the IBM 360, instructions are first generated to set up a condition code, and a final instruction (BC 8 LABL) issued to complete the transfer. The initial code is generated as follows: If EXP is register 2, then (CR A Q) is issued. If EXP is any other number (we assume register 1 is not possible!) then an instruction of the form (C A ⟨location of EXP⟩) is issued. Otherwise the load c-macro is used to load EXP into an auxiliary register R1, followed by the instruction (CR A R1) to set up the condition code.

On the PDP–10, code is generated to compare register 1 with EXP, and a skip if not **eq** is emitted to skip over a following (JRST 0 LABL), using (CAMN 1 exp) or (CAIN 1 (QUOTE ⟨S-expression⟩)). If the **anyreg** expression requires a register for its computation, an auxiliary register (D) is used, by first loading the exp into D via LOAD(D, exp) and then emitting ((CAMN 1 D) (JRST 0 LABL)).

JUMPN(LABL:id,EXP:any):list

This c-macro generates a jump to the location specified by the c-macro LBL(LABL) if the value of register 1 is not **eq** to EXP.

On the IBM 360, the same instructions are generated initially as for the JUMPE c-macro, and the instruction (BC 7 LABL) finally issued to complete the transfer.

On the PDP–10 the process is exactly the same as for JUMPE, but (CAME 1 exp) or (CAIE 1 (QUOTE ⟨S-expression⟩)) is used to skip around the (JRST 0 LABL) if register 1 **eq** EXP.

JUMPC(LABL:id,REG:atom,NAME:id):list

NAME in this case is the boolean function being tested for (e.g., **atom** or **numberp**). If the contents of REG satisfy the NAME test, then this c-macro generates a jump to the location specified by the c-macro LBL(LABL).

On the IBM 360, If REG is 1 or 2, then the c-macro expands into the sequence (TM ⟨reg⟩ ⟨mask⟩) (BC 14 ⟨labl⟩), where the ⟨mask⟩ is appropriate to the boolean function NAME and ⟨reg⟩ = REG. If REG is any other register, then a load of REG into the auxiliary register R1 is first generated, followed by the above sequence with ⟨reg⟩ now R1. This c-macro is not currently used on the PDP–10, although it will be possible when a new version of the loader is installed.[28]

JUMPNC(LABL:id,REG:atom,NAME:id):list

As for JUMPC, NAME is the boolean function being tested for. If the contents of REG do not satisfy the NAME test, then this c-macro generates a jump to the location specified by the c-macro LBL(LABL).

On the IBM 360, if REG is 1 or 2, then the c-macro expands into the sequence (TM ⟨reg⟩ ⟨mask⟩) (BC 1 ⟨labl⟩), where the ⟨mask⟩ is appropriate to the boolean function NAME and ⟨reg⟩ = REG. If REG is any other register, then a load of REG into the auxiliary register R1 is first generated, followed by the above sequence with ⟨reg⟩ now R1. This c-macro is not currently used on the PDP–10.

LBL(LABL:id):list

This c-macro generates a label in the instruction sequence in a format recognized by the loader.

On both the IBM 360 and the PDP–10, the code sequence (LABL) is issued to put the atom LABL in the instruction list.

LOAD(REG:atom,EXP:any):list

This c-macro loads the register REG with the value of the expression EXP. If EXP is one of ⟨reg⟩, ⟨loc⟩, (QUOTE ⟨S-expression⟩), (FLUID ⟨var⟩) or (GLOBAL ⟨var⟩), the loading is usually direct. Otherwise the instructions to load the form (⟨afn⟩⟨exp⟩) must be issued first before the loading can be completed. On the IBM 360, if the loading is to registers 1 and 2, it can be direct. Otherwise, the expression is loaded into an auxiliary register R1, and then stored in the location reserved for the register REG, unless EXP is 1 or 2 as noted below. The decoding of EXP for the various cases is as follows:

reg:	If EXP is 1 or 2, then if REG is 1 or 2, the instruction (LR REG EXP) is issued, otherwise (ST EXP ⟨location of REG⟩) is produced. If EXP is greater than 2, then the instruction sequence begins with (L x ⟨location of EXP⟩), where x is A, Q, or R1, depending on whether an auxiliary register had to be used.
loc:	the instruction sequence begins with (L x ⟨location of EXP⟩) as above.
(QUOTE ⟨S-expression⟩):	If EXP is (QUOTE NIL), then the instruction sequence terminates with (LR x NILR), otherwise ((L x EXP) (AR x NILR)). The AR instruction is necessary since all quoted expressions use NILR as their base register.
(FLUID ⟨var⟩) ¦ (GLOBAL ⟨var⟩):	The instruction sequence begins with ((L x EXP) (L x 0 (NILR x))). The latter instruction is necessary to pick up the **car** of the fluid/global cell where the actual value is stored.
(⟨afn⟩⟨exp⟩):	The **anyreg** property for ⟨afn⟩ is assumed to be (LAMBDA (X Y) ⟨list of instructions⟩), where the parameters X and Y refer to the register and frame location respectively. Hence if the ⟨exp⟩ is a ⟨loc⟩, a substitution of the appropriate register and frame location is made in the lambda body. On the other hand, if ⟨exp⟩ is not a ⟨loc⟩, an **open** property is sought for ⟨fn⟩, with the form (LAMBDA (A) ⟨list of instructions⟩ . In this case, a LOAD of ⟨exp⟩ is made to the appropriate register, after which the list of instructions in the open property with the parameter A replaced by the appropriate register is added to the instruction sequence.

On the PDP–10, the process is even simpler than above: EXP can always be directly loaded to registers 1–5 (using REG itself as an intermediate register for **Anyreg** expressions); for REG > 5, auxiliary register D is used for a LOAD(D,EXP), followed by (MOVEM D REG). The relevant sub-cases are:

(QUOTE ⟨S-expression⟩)	(MOVEI reg (QUOTE ⟨S-expression⟩));
(FLUID var) or (GLOBAL var)	(MOVE reg (FLUID var));
loc	(MOVE reg loc P);
reg′	(MOVE reg reg′).

where reg is the register name corresponding to REG.

STORE(NREG:atom,FLOC:any):list

If NREG is NIL, then NIL is stored in FLOC, otherwise the contents of register NREG are stored in FLOC. If FLOC is of the form (FLUID ⟨var⟩) or (GLOBAL ⟨var⟩) then a store into the appropriate cell must be passed to the loader. It is assumed that the loader knows how to handle such forms. If FLOC is a non-positive integer, then a store into the appropriate frame location is generated.

On the IBM 360, NIL is permanently stored in a register named NILR, and need not be handled as a special case. Therefore, if NREG is NIL, 1 or 2, the store can be direct from the appropriate register. If FLOC is a frame location, this sequence has the form ((ST nreg[−4*FLOC] (R0 PDL))), where nreg is the symbolic name for NREG and the value of the expression in the square brackets is used. If NREG is one of the other forms, the code sequence is ((L M (FLUID var)) (ST nreg 0 (NILR M))) where M is another auxiliary register. On the other hand, if NREG is 3, 4 and so on, then the contents of the memory location reserved for the relevant register are first loaded into the auxiliary register R1, and this register then stored as above.

On the PDP–10, NIL again is permanently stored in a register, namely 0, and is therefore also not a special case. So if NREG is NIL, or 1 through 5, the store is again direct from the register. If FLOC is a frame location, the instruction sequence generated has the form ((MOVEM nreg FLOC P)), if it is one of the other forms, the sequence is ((MOVEM nreg FLOC)). If NREG > 5, then an auxiliary register is again used to load the contents of the memory location reserved for NREG, and this then stored as above.

LINK(NAME:id,TYPE:id,NARGS:integer):list

If NAME has an open indicator, then an open coded sequence of instructions for the link is generated. The general format of the open property is the list of instructions which replace the link. Otherwise a general function calling sequence is produced.

On the IBM 360, the general linking mechanism requires that several registers be saved on entry to a function and restored on exit. Therefore, the linking is done via a subroutine rather than a direct set of instructions. In addition, except for a class of atoms whose location is known, a data word is also generated along with the link cell to provide the name of the function to the linking subroutine.

On the PDP–10, the general linking can be done by using a single word Unimplemented User Operation (UUO), named 'CALL': (CALL noargs (E name)). This form therefore limits 'nargs' to 0 through 15. However, since register 15 is used to indicate **fexprs**, a maximum of 14 arguments can be permitted in a function.

LINKE(NAME:id,TYPE:idNARGS:integer,N:integer):list

This c-macro is equivalent to the sequence of c-macros:

```
LINK(NAME,TYPE,NARGS)
DEALLOC(N)
EXIT( ).
```

However, since DEALLOC does not affect any registers, its position may be exchanged with the LINK c-macro. If LINK followed by EXIT can then be done in one instruction or one subroutine call, the resulting code is shorter then the above sequence.

On the IBM 360, this c-macro is not used (being inhibited by setting ***nolinke** to true), since the EXIT instruction is used to determine the beginning of the function data block which immediately follows this point.

On the PDP–10, the LINK and EXIT sequence can be performed by another single word UUO: (JCALL noargs (E name)).

LAMBIND(REGS:list,ALST:list):list

This c-macro, and its associated c-macros PROGBIND and FREESTR, involve a large number of instructions if open coded. It is therefore recommended that these be implemented as subroutines, particularly if the underlying LISP uses a fairly complex **fluid** mechanism. The action for a simple shallow binding model can be described in terms of the other c-macros using an auxiliary register R as follows:

```
while REGS do
  {LOAD(R, FLUID(caar ALST));
  STORE(R, cdar ALST);
  STORE(car REGS, FLUID(caar ALST));
  ALST := cdr ALST;
  REGS := cdr REGS};
```

Note that additional operations might be required to satisfy the error handling, or **fluid** mechanism.

In other words, the elements of REGS are paired with those of ALST to give an effective triple (⟨reg⟩,⟨var⟩,⟨loc⟩). For each triple, the contents of the **fluid** variable ⟨var⟩ are stored in ⟨loc⟩ (to be restored later by FREERSTR), and the contents of ⟨reg⟩then stored in the **fluid** cell.

On both the IBM 360 and the PDP–10, this c-macro is implemented as a subroutine call. On the PDP–10, an additional special pushdown stack, indexed by a register SP is used, and the frame location is not needed at all. A ((CALL 0 (E LAMBIND*)) is followed by an appropriate number of (0 reg var) words; a pair (⟨var⟩ ⟨value-of-var⟩) is pushed onto the special stack by LAMBIND, and STORE(reg,(FLUID var)) performed . After all of ALST has been processed, a mark is pushed onto the special stack to indicate the size of this binding set.

PROGBIND(ALST:list):list

Like LAMBIND, this c-macro is best implemented as a subroutine call. Its simple shallow binding action is as follows:

```
while ALST do
  {LOAD(R, FLUID(caar ALST));
  STORE(R, cadar ALST);
  STORE(NIL, FLUID(caar ALST));
  ALST := cdr ALST};
```

In other words, for each pair (⟨var⟩,⟨loc⟩) in ALST, the value of the fluid variable ⟨var⟩ is stored in ⟨loc⟩ and a NIL then stored in the fluid cell of ⟨var⟩.

On both the IBM 360 and the PDP–10, this c-macro is implemented as a subroutine call. On the PDP–10, a set of words containing (0 0 var) for each var follow a (CALL 0 (E LAMBIND*)); NIL is represented by register 0 in the first argument position indicating that a STORE(NIL,(FLUID var)) is to be performed on each variable.

FREESTR(ALST:list):list

This c-macro is the easiest of the three c-macros handling free variable binding to describe. Its action for simple shallow binding is as follows:

```
while ALST do
  {LOAD(R, cadar ALST);
  STORE(R, FLUID(caar ALST));
  ALST := cdr ALST};
```

On both the IBM 360 and the PDP–10, this c-macro is implemented as a subroutine call. On the PDP–10, the mark pushed onto the special stack by LAMBIND or PROGBIND is used to indicate how many (var value) pairs to pop off for unbinding. Thus only a single internal function call is emitted.

C-MACROS	PDP–10	IBM 360
(ENTRY LSTCHR EXPR 2)		(BAL R3 ENTERCC)
(ALLOC 1)	(PUSH P 1)	(BXH PDS K4 0 (R0 R12))
(STORE 1 0)		(ST A 0 (R0 PDL))
(LOAD 1 (CDR 1))	(HRRZ 1 0 1)	(L A 4 (R0 A))
(JUMPT L1)	(JUMPN 1 L1)	(CR A NILR)
		(BC 7 L1)
(LINK CONS EXPR 2)	(CALL 2 (E CONS))	(BALR R2 R12)
(JUMP L2)	(JRST 0 L2)	(BC 15 L2)
(LBL L1)	L1	L1
(LINK LSTCHR EXPR 2)	(CALL 2 (E LSTCHR))	(BAL R2 174 (R0 R12))
(LINK NCONS EXPR 1)	(CALL 1 (E NCONS))	(BAL R2 116 (R0 R12))
(LINK NCONS EXPR 1)	(CALL 1 (E NCONS))	(BAL R2 116 (R0 R12))
(LBL L2)	L2	L2
(LOAD 2 (CAR 0))	(HLRZ@ 2 0 P)	(L Q 0 (R0 PDL))
		(L Q 0 (R0 Q))
(LINKE XCONS EXPR 2)	(SUB P (C 0 0 1 1))	(BAL R2 120 (R0 R12))
	(JCALL 2 (E XCONS))	(BC 15 60 (R0 R12))
Total Bytes:	50	56

To conclude this Appendix, we show the actual LAP instructions generated for our test example (**1stchr**). We notice in this case that the total code sizes are fairly similar in spite of the differences in the machine architectures. In fact, over a large number of samples we find that the number of bytes of code used in the IBM and DEC implementations are within 15 per cent of each other.

The various BALs in the IBM 360 code reference interpreter subroutine calls to a set of common functions, such as **cons**, or a routine for referencing a recursive entry to the function being defined.

APPENDIX B: Listing of the Compiler

In this Appendix, we present a complete listing of the portable compiler and its supporting compiler c-macro definition files for the IBM 360 and the PDP–10. The listing is accordingly divided into three parts; the portable compiler itself, the c-macros for the IBM 360 and the c-macros for the PDP–10. The only difference between this listing and the distributed form of these programs is that in the latter case, the character & is prefixed to most internal function names and an asterisk to all compiler c-macro names to prevent possible name clashes during execution. These characters are omitted in the listings below and throughout this paper.

B.1 The Portable Compiler Code

```
COMMENT global flags used in this compiler:

!*MODULE        indicates block compilation (a future extension of
                this compiler)
!*MSG           indicates whether certain messages should be printed
!*NOLINKE       if ON inhibits use of LINKE c-macro
!*ORD           if ON forces left-to-right argument evaluation
!*PLAP          if ON causes LAP output to be printed
!*R2I           if ON causes recursion removal where possible;

GLOBAL '(!*MODULE !*MSG !*NOLINKE !*ORD !*PLAP !*R2I);

COMMENT global variables used:

DFPRINT!*       name of special definition process (or NIL)
ERFG!*          used by REDUCE to control error recovery
MAXNARGS        maximum number of arguments permitted;

GLOBAL '(DFPRINT!* ERFG!* MAXNARGS);

MAXNARGS := 15;         %Standard LISP limit;

COMMENT fluid variables used:

ALSTS           alist of fluid parameters
CODELIST        code being built
CONDTAIL        simulated stack of position in the tail of a COND
EXIT            label for EXIT jump
FLAGG           used in COMTST, and in FIXREST
FREELST         list of free variables with bindings
GOLIST          storage map for jump labels
IREGS           initial register contents
IREGS1          temporary placeholder for IREGS during branch compilation
JMPLIST         list of locations in CODELIST of transfers
LBLIST          list of label words
LLNGTH          cell whose CAR is length of frame
```

```
NAME            name of function being currently compiled
NARG            number of arguments in function
REGS            known current contents of registers as an alist with
                 elements  of form (<reg> . <contents>)
REGS1           temporary placeholder for REGS during branch compilation
SLST            association list for stores which have not yet been used
STLST           list of active stores in function
STOMAP          storage map for variables
SWITCH          boolean expression value flag - keeps track of NULLs;

FLUID '(ALSTS CODELIST CONDTAIL EXIT FLAGG FREELST GOLIST IREGS IREGS1
        JMPLIST LBLIST LLNGTH NAME NARG REGS REGS1 SLST STLST STOMAP
        SWITCH);

COMMENT c-macros used in this compiler;

COMMENT The following c-macros must NOT change regs 1-MAXNARGS:

ALLOC n                 allocate new stack frame of n words
DEALLOC n               deallocate above frame
ENTRY name type nargs   entry point to function name of type type
                           with nargs args
EXIT                    exit to previously saved return address
STORE reg floc          store contents of reg (or NIL) in floc
JUMP adr                unconditional jump
JUMPC  adr exp type     jump to adr if exp is of type type
JUMPNC adr exp type     jump to adr if exp is not of type type
JUMPNIL adr             jump on register 1 eq to NIL
JUMPT adr               jump on register 1 not eq to NIL
JUMPE adr exp           jump on register 1 eq to exp
JUMPN adr exp           jump on register 1 not eq to exp
LBL adr                 define label
LAMBIND regs alst       bind free lambda vars in alst currently in regs
PROGBIND alst           bind free prog vars in alst
FREERSTR alst           unbind free variables in alst

COMMENT the following c-macro must only change specific register
        being loaded:

LOAD reg exp            load exp into reg;

COMMENT the following c-macros do not protect regs 1-MAXNARGS:

LINK fn type nargs      link to fn of type type with nargs args
LINKE fn type nargs n   link to fn of type type with nargs args
                           and exit removing frame of n words
CODE list               this macro allows for the inclusion of a list
                        of c-macro expressions (or even explicit
                        assembly language) in a function definition;

FLAG('(ALLOC DEALLOC ENTRY EXIT STORE JUMP JUMPC JUMPNC
       JUMPNIL JUMPT JUMPE JUMPN LBL LAMBIND PROGBIND
       FREERSTR LOAD LINK LINKE CODE),
'MC);

COMMENT General functions used in this compiler;

SYMBOLIC PROCEDURE ATSOC(U,V);
   IF NULL V THEN NIL
    ELSE IF U EQ CAAR V THEN CAR V
    ELSE ATSOC(U,CDR V);

SYMBOLIC PROCEDURE EQCAR(U,V); NOT ATOM U AND CAR U EQ V;

SYMBOLIC PROCEDURE LPRI U;
   IF ATOM U THEN LPRI LIST U
    ELSE FOR EACH X IN U DO <<PRIN2 X; PRIN2 " ">>;
```

```
SYMBOLIC PROCEDURE LPRIE U;
   <<LPRI ("******" . IF ATOM U THEN LIST U ELSE U);
     ERFG!* := T;
     TERPRI()>>;

SYMBOLIC PROCEDURE LPRIM U;
   IF !*MSG
     THEN <<TERPRI();
            LPRI ("***" . IF ATOM U THEN LIST U ELSE U);
            TERPRI()>>;

SYMBOLIC PROCEDURE MKQUOTE U; LIST('QUOTE,U);

SYMBOLIC PROCEDURE REVERSIP U;
   BEGIN SCALAR X,Y;
      WHILE U DO <<X := CDR U; Y := RPLACD(U,Y); U := X>>;
      RETURN Y
   END;

SYMBOLIC PROCEDURE RPLACW(A,B); RPLACA(RPLACD(A,CDR B),CAR B);

COMMENT the following two functions are used by the CONS open
        coding. They should be defined in the interpreter if
        possible. They should only be compiled without a COMPFN
        for CONS;

SYMBOLIC PROCEDURE NCONS U; U . NIL;

SYMBOLIC PROCEDURE XCONS(U,V); V . U;

COMMENT Top level compiling functions;

SYMBOLIC PROCEDURE COMPILE X;
   BEGIN SCALAR EXP;
      FOR EACH Y IN X DO
         <<EXP := GETD Y;
           IF NULL EXP THEN LPRIM LIST(Y,'UNDEFINED)
            ELSE COMPD(Y,CAR EXP,CDR EXP)>>
   END;

SYMBOLIC PROCEDURE COMPD(NAME,TYPE,EXP);
   BEGIN
      IF NOT FLAGP(TYPE,'COMPILE)
        THEN <<LPRIM LIST("UNCOMPILABLE FUNCTION",NAME,"OF TYPE",
                          TYPE);
               RETURN NIL>>;
      IF NOT ATOM EXP
        THEN IF !*MODULE THEN MODCMP(NAME,TYPE,EXP)
             ELSE IF DFPRINT!*
              THEN APPLY(DFPRINT!*,
                         LIST IF TYPE EQ 'EXPR
                                THEN 'DE . (NAME . CDR EXP)
                              ELSE IF TYPE EQ 'FEXPR
                               THEN 'DF . (NAME . CDR EXP)
                              ELSE IF TYPE EQ 'MACRO
                               THEN 'DM . (NAME . CDR EXP)
                              ELSE LIST('PUTD,MKQUOTE NAME,
                                        MKQUOTE TYPE,
                                        MKQUOTE EXP))
             ELSE BEGIN SCALAR X;
                    IF FLAGP(TYPE,'COMPILE)
                      THEN PUT(NAME,'CFNTYPE,LIST TYPE);
                    X :=
                     LIST('ENTRY,NAME,TYPE,LENGTH CADR EXP)
                       . COMPROC(EXP,
                                 IF FLAGP(TYPE,'COMPILE)
                                   THEN NAME);
                    IF !*PLAP THEN FOR EACH Y IN X DO PRINT Y;
                    LAP X;
```

```
                    %this is the entry point to the assembler.  LAP
                    %must remove any preexisting function definition;
                    IF (X := GET(NAME,'CFNTYPE))
                         AND EQCAR(GETD NAME,CAR X)
                      THEN REMPROP(NAME,'CFNTYPE)
                  END;
      RETURN NAME
   END;

FLAG('(EXPR FEXPR MACRO),'COMPILE);


SYMBOLIC PROCEDURE COMPROC(EXP,NAME);
   %compiles a function body, returning the generated LAP;
   BEGIN SCALAR CODELIST,FLAGG,IREGS,IREGS1,JMPLIST,LBLIST,
                LLNGTH,REGS,REGS1,ALSTS,EXIT,SLST,STLST,STOMAP,
                CONDTAIL,FREELST,
                SWITCH; INTEGER NARG;
      LLNGTH := LIST 1;
      NARG := 0;
      EXIT := GENLBL();
      STOMAP := '((NIL 1));
      CODELIST := LIST ('ALLOC . LLNGTH);
      EXP := PASS1 EXP;
      IF LENGTH CADR EXP>MAXNARGS
        THEN LPRIE LIST("TOO MANY ARGS FOR COMPILER IN",NAME);
      FOR EACH Z IN CADR EXP DO <<FRAME Z;
                                  NARG := NARG + 1;
                                  IF NOT NONLOCAL Z
                                    THEN IREGS :=
                                          NCONC(IREGS,
                                                LIST LIST(NARG,Z));
                                  REGS :=
                                   NCONC(REGS,LIST LIST(NARG,Z))>>;
      IF NULL REGS THEN REGS := LIST (1 . NIL);
      ALSTS := FREEBIND(CADR EXP,T);
      PASS2 CADDR EXP;
      FREERST(ALSTS,0);
      PASS3();
      RPLACA(LLNGTH,1 - CAR LLNGTH);
      RETURN CODELIST
   END;

SYMBOLIC PROCEDURE NONLOCAL X;
   IF FLUIDP X THEN 'FLUID ELSE IF GLOBALP X THEN 'GLOBAL ELSE NIL;

COMMENT Pass 1 of the compiler;

SYMBOLIC PROCEDURE PASS1 EXP; PA1(EXP,NIL);

SYMBOLIC PROCEDURE PA1(U,VBLS);
   BEGIN SCALAR X;
      RETURN IF ATOM U
               THEN IF CONSTANTP U OR U MEMQ '(NIL T) THEN MKQUOTE U
                     ELSE IF U MEMQ VBLS THEN U
                     ELSE IF NONLOCAL U THEN U
                     ELSE <<MKNONLOCAL U; U>>
              ELSE IF NOT ATOM CAR U
               THEN PA1(CAR U,VBLS) . PALIS(CDR U,VBLS)
              ELSE IF X := GET(CAR U,'PA1FN) THEN APPLY(X,LIST(U,VBLS))
              ELSE IF (X := GETD CAR U)
                        AND CAR X EQ 'MACRO
                        AND NOT GET(CAR U,'COMPFN)
               THEN PA1(APPLY(CDR X,LIST U),VBLS)
              ELSE IF X := GET(CAR U,'CMACRO)
               THEN PA1(SUBLIS(PAIR(CADR X,CDR U),CADDR X),VBLS)
              ELSE IF CFNTYPE CAR U EQ 'FEXPR
                        AND NOT GET(CAR U,'COMPFN)
               THEN LIST(CAR U,MKQUOTE CDR U)
```

```
            ELSE IF CAR U MEMQ VBLS OR FLUIDP CAR U
             THEN LIST('APPLY,CAR U,PALIST(CDR U,VBLS))
            ELSE CAR U . PALIS(CDR U,VBLS)

   END;

SYMBOLIC PROCEDURE PAIDEN(U,VBLS); U;

PUT('GO,'PA1FN,'PAIDEN);

PUT('QUOTE,'PA1FN,'PAIDEN);

PUT('CODE,'PA1FN,'PAIDEN);

SYMBOLIC PROCEDURE PACOND(U,VBLS);
   'COND . FOR EACH Z IN CDR U
            COLLECT LIST(PA1(CAR Z,VBLS),
                         PA1(MKPROGN CDR Z,VBLS));

PUT('COND,'PA1FN,'PACOND);

SYMBOLIC PROCEDURE PAFUNC(U,VBLS);
   IF ATOM CADR U THEN MKFUNC CADR U
    ELSE MKFUNC COMPD(MKNAM NAME,'EXPR,CADR U);

PUT('FUNCTION,'PA1FN,'PAFUNC);

SYMBOLIC PROCEDURE PALAMB(U,VBLS);
   'LAMBDA . LIST(CADR U,PA1(MKPROGN CDDR U,APPEND(CADR U,VBLS)));

PUT('LAMBDA,'PA1FN,'PALAMB);

SYMBOLIC PROCEDURE PALIST(U,VBLS); 'LIST . PALIS(U,VBLS);

SYMBOLIC PROCEDURE PAPROG(U,VBLS);
   'PROG . (CADR U . PAPROG1(CDDR U,APPEND(CADR U,VBLS)));

SYMBOLIC PROCEDURE PAPROG1(U,VBLS);
   FOR EACH X IN U COLLECT IF ATOM X THEN X ELSE PA1(X,VBLS);

PUT('PROG,'PA1FN,'PAPROG);

SYMBOLIC PROCEDURE PALIS(U,VBLS);
   FOR EACH X IN U COLLECT PA1(X,VBLS);

SYMBOLIC PROCEDURE MKNONLOCAL U;
   <<LPRIM LIST(U,"declared fluid"); FLUID LIST U; LIST('FLUID,U)>>;

SYMBOLIC PROCEDURE MKNAM U;
   %generates unique name for auxiliary function in U;
   INTERN COMPRESS APPEND(EXPLODE U,EXPLODE GENSYM());

SYMBOLIC PROCEDURE MKPROGN U;
   IF NULL U OR CDR U THEN 'PROGN . U ELSE CAR U;

COMMENT CMACRO definitions for some functions;

COMMENT We do not expand CAAAAR and similar functions, since fewer
        instructions are generated without open coding;

DEFLIST('((CAAR (LAMBDA (U) (CAR (CAR U))))
          (CADR (LAMBDA (U) (CAR (CDR U))))
          (CDAR (LAMBDA (U) (CDR (CAR U))))
          (CDDR (LAMBDA (U) (CDR (CDR U))))
          (CAAAR (LAMBDA (U) (CAR (CAR (CAR U)))))
          (CAADR (LAMBDA (U) (CAR (CAR (CDR U)))))
          (CADAR (LAMBDA (U) (CAR (CDR (CAR U)))))
          (CADDR (LAMBDA (U) (CAR (CDR (CDR U)))))
          (CDAAR (LAMBDA (U) (CDR (CAR (CAR U))))))
```

```
          (CDADR (LAMBDA (U) (CDR (CAR (CDR U)))))
          (CDDAR (LAMBDA (U) (CDR (CDR (CAR U)))))
          (CDDDR (LAMBDA (U) (CDR (CDR (CDR U)))))
          (NOT (LAMBDA (U) (NULL U)))),'CMACRO);

COMMENT Pass 2 of the compiler;

SYMBOLIC PROCEDURE PASS2 EXP; COMVAL(EXP,0);

SYMBOLIC PROCEDURE COMVAL(EXP,STATUS);
   %computes code for value of EXP;
   IF ANYREG(EXP,NIL)
     THEN IF STATUS>1 THEN NIL ELSE LREG1(EXP,STATUS)
    ELSE COMVAL1(EXP,STOMAP,STATUS);

SYMBOLIC PROCEDURE COMVAL1(EXP,STOMAP,STATUS);
   BEGIN SCALAR X;
      IF ATOM EXP THEN IF STATUS<2 THEN LREG1(EXP,STATUS) ELSE NIL
       ELSE IF NOT ATOM CAR EXP
        THEN IF CAAR EXP EQ 'LAMBDA
               THEN COMPLY(CAR EXP,CDR EXP,STATUS)
              ELSE LPRIE LIST("INVALID FUNCTION",CAR EXP)
       ELSE IF X := GET(CAR EXP,'COMPFN) THEN APPLY(X,LIST(EXP,STATUS))
       ELSE IF !*R2I AND CAR EXP EQ NAME AND STATUS=0 AND NULL FREELST
        THEN COMREC(EXP,STATUS)
       ELSE IF CAR EXP EQ 'LAMBDA
        THEN LPRIE LIST("INVALID USE OF LAMBDA IN FUNCTION",NAME)
       ELSE IF CAR EXP EQ 'CODE THEN ATTACH EXP
       ELSE CALL(CAR EXP,CDR EXP,STATUS);
      RETURN NIL
   END;

SYMBOLIC PROCEDURE ANYREG(U,V);
   %determines if U can be loaded in any register;
   %!*ORD = T means force correct order, unless safe;
   IF EQCAR(U,'QUOTE) THEN T
    ELSE (ATOM U OR GET(CAR U,'ANYREG) AND ANYREG(CADR U,NIL))
          AND (NULL !*ORD OR ANYREGL V);

SYMBOLIC PROCEDURE ANYREGL U;
   NULL U OR ANYREG(CAR U,NIL) AND ANYREGL CDR U;

SYMBOLIC PROCEDURE CALL(FN,ARGS,STATUS);
   CALL1(FN,COMLIS ARGS,STATUS);

SYMBOLIC PROCEDURE CALL1(FN,ARGS,STATUS);
   %ARGS is reversed list of compiled arguments of FN;
   BEGIN INTEGER ARGNO;
      ARGNO := LENGTH ARGS;
      LOADARGS(ARGS,STATUS);
      ATTACH LIST('LINK,FN,CFNTYPE FN,ARGNO);
      IF FLAGP(FN,'ONEREG) THEN REGS := (1 . NIL) . CDR REGS
       ELSE IF FLAGP(FN,'TWOREG)
        THEN REGS := (1 . NIL) . DELASC(2,CDR REGS)
       ELSE REGS := LIST (1 . NIL)
   END;

SYMBOLIC PROCEDURE DELASC(U,V);
   IF NULL V THEN NIL
    ELSE IF U=CAAR V THEN CDR V
    ELSE CAR V . DELASC(U,CDR V);

SYMBOLIC PROCEDURE COMLIS EXP;
   %returns reversed list of compiled arguments;
   BEGIN SCALAR ACUSED,Y;
      WHILE EXP DO
         <<IF ANYREG(CAR EXP,CDR EXP) THEN Y := CAR EXP . Y
            ELSE <<IF ACUSED THEN STORE1();
                   COMVAL1(CAR EXP,STOMAP,1);
```

```
                ACUSED := GENSYM();
                REGS := (1 . (ACUSED . CDAR REGS)) . CDR REGS;
                Y := ACUSED . Y>>;
          EXP := CDR EXP>>;
      RETURN Y
   END;

SYMBOLIC PROCEDURE STORE1; %Marks contents of register 1 for storage;
   BEGIN SCALAR X;
      X := CADAR REGS;
      IF NULL X OR EQCAR(X,'QUOTE) THEN RETURN NIL
       ELSE IF NOT ATSOC(X,STOMAP) THEN FRAME X;
      STORE0(X,1)
   END;

SYMBOLIC PROCEDURE COMPLY(FN,ARGS,STATUS);
   BEGIN SCALAR ALSTS,VARS; INTEGER I;
      VARS := CADR FN;
      LOADARGS(COMLIS ARGS,1);
      ARGS := REMVARL VARS; % The stores that were protected;
      I := 1;
      FOR EACH V IN VARS DO <<FRAME V;
                              REGS := REPASC(I,V,REGS);
                              I := I + 1>>;
      ALSTS := FREEBIND(VARS,T); %Old fluid values saved;
      I := 1;
      FOR EACH V IN VARS DO <<IF NOT NONLOCAL V THEN STORE0(V,I);
                              I := I + 1>>;
      COMVAL(CADDR FN,STATUS);
      FREERST(ALSTS,STATUS);
      RSTVARL(VARS,ARGS)
   END;

SYMBOLIC PROCEDURE COMREC(EXP,STATUS);
   BEGIN SCALAR X,Z;
      LOADARGS(COMLIS CDR EXP,STATUS);
      Z := CODELIST;
      IF NULL CDR Z
        THEN LPRIE LIST("CIRCULAR DEFINITION FOR",CAR EXP);
      WHILE CDDR Z DO Z := CDR Z;
      IF CAAR Z EQ 'LBL THEN X := CDAR Z
       ELSE <<X := GENLBL(); RPLACD(Z,LIST('LBL . X,CADR Z))>>;
      ATTJMP X
   END;

SYMBOLIC PROCEDURE LOADARGS(ARGS,STATUS);
   BEGIN INTEGER N;
      N := LENGTH ARGS;
      IF N>MAXNARGS THEN LPRIE LIST("TOO MANY ARGUMENTS IN",NAME);
      IF STATUS>0 THEN CLRREGS();
      WHILE ARGS DO
         <<LREG(N,CAR ARGS,CDR ARGS,STATUS);
           N := N - 1;
           ARGS := CDR ARGS>>
   END;

SYMBOLIC PROCEDURE LOCATE X;
   BEGIN SCALAR Y,VTYPE;
      IF EQCAR(X,'QUOTE) THEN RETURN LIST X
       ELSE IF Y := RASSOC(X,REGS) THEN RETURN LIST CAR Y
       ELSE IF NOT ATOM X THEN RETURN LIST (CAR X . LOCATE CADR X)
       ELSE IF VTYPE := NONLOCAL X THEN RETURN LIST LIST(VTYPE,X);
      WHILE Y := ATSOC(X,SLST) DO SLST := DELETE(Y,SLST);
      RETURN IF Y := ATSOC(X,STOMAP) THEN CDR Y ELSE LIST MKNONLOCAL X
   END;

SYMBOLIC PROCEDURE LREG(REG,U,V,STATUS);
   BEGIN SCALAR X,Y;
```

```
      IF (X := ASSOC(REG,REGS)) AND U MEMBER CDR X THEN RETURN NIL
       ELSE IF (Y := ASSOC(REG,IREGS))
                 AND (STATUS>0 OR MEMLIS(CADR Y,V))
        THEN <<STOREO(CADR Y,REG); IREGS := DELETE(Y,IREGS)>>;
      ATTACH ('LOAD . (REG . LOCATE U));
      REGS := REPASC(REG,U,REGS)
   END;

SYMBOLIC PROCEDURE LREG1(X,STATUS); LREG(1,X,NIL,STATUS);

COMMENT Functions for handling non-local variables;

SYMBOLIC PROCEDURE FREEBIND(VARS,LAMBP);
   %bind FLUID variables in lambda or prog lists;
   %LAMBP is true for LAMBDA, false for PROG;
   BEGIN SCALAR FALST,FREGS,X,Y; INTEGER I;
      I := 1;
      FOR EACH X IN VARS DO <<IF FLUIDP X
                               THEN <<FALST :=
                                      (X . GETFRM X) . FALST;
                                     FREGS := I . FREGS>>
                              ELSE IF GLOBALP X
                               THEN LPRIE LIST("CANNOT BIND GLOBAL ",
                                               X);
                             I := I + 1>>;
      IF NULL FALST THEN RETURN NIL;
      IF LAMBP THEN ATTACH LIST('LAMBIND,FREGS,FALST)
       ELSE ATTACH LIST('PROGBIND,FALST);
      RETURN FALST
   END;

SYMBOLIC PROCEDURE FREERST(ALSTS,STATUS); %restores FLUID variables;
   IF ALSTS THEN ATTACH LIST('FREERSTR,ALSTS);

SYMBOLIC PROCEDURE ATTACH U; CODELIST := U . CODELIST;

SYMBOLIC PROCEDURE STOREO(U,REG);
   %marks expression U in register REG for storage;
   BEGIN SCALAR X;
      X := 'STORE . (REG . GETFRM U);
      STLST := X . STLST;
      ATTACH X;
      IF ATOM U
        THEN <<CLRSTR U; SLST := (U . CODELIST) . SLST>>
   END;

SYMBOLIC PROCEDURE CLRSTR VAR; %removes unneeded stores;
   BEGIN SCALAR X;
      IF CONDTAIL THEN RETURN NIL;
      X := ATSOC(VAR,SLST);
      IF NULL X THEN RETURN NIL;
      STLST := DELEQ(CADR X,STLST);
      SLST := DELEQ(X,SLST);
      RPLACA(CADR X,'NOOP)
   END;

COMMENT Functions for general tests;

SYMBOLIC PROCEDURE COMTST(EXP,LABL);
   %compiles boolean expression EXP.
   %If EXP has the same value as SWITCH then branch to LABL,
   %otherwise fall through;
   %REGS/IREGS are active registers for fall through,
   %REGS1/IREGS1 for branch;
   BEGIN SCALAR X;
      WHILE EQCAR(EXP,'NULL) DO
         <<SWITCH := NOT SWITCH; EXP := CADR EXP>>;
      IF NOT ATOM EXP AND ATOM CAR EXP AND (X := GET(CAR EXP,'COMTST))
        THEN APPLY(X,LIST(EXP,LABL))
```

```
      ELSE <<IF EXP='(QUOTE T)
               THEN IF SWITCH THEN ATTJMP LABL ELSE FLAGG := T
              ELSE <<COMVAL(EXP,1);
                     ATTACH LIST(IF SWITCH THEN 'JUMPT
                                   ELSE 'JUMPNIL,CAR LABL);
                     ADDJMP CODELIST>>;
             REGS1 := REGS;
             IREGS1 := IREGS>>;
      IF EQCAR(CAR CODELIST,'JUMPT)
        THEN REGS := (1 . ('(QUOTE NIL) . CDAR REGS)) . CDR REGS
       ELSE IF EQCAR(CAR CODELIST,'JUMPNIL)
        THEN REGS1 := (1 . ('(QUOTE NIL) . CDAR REGS1)) . CDR REGS1
   END;

COMMENT Specific function open coding;

SYMBOLIC PROCEDURE COMANDOR(EXP,STATUS);
   BEGIN SCALAR FN,LABL,IREGSL,REGSL;
      FN := CAR EXP EQ 'AND;
      LABL := GENLBL();
      IF STATUS>1
        THEN BEGIN SCALAR REGS1;
                TSTANDOR(EXP,LABL);
                REGS := RMERGE2(REGS,REGS1)
             END
       ELSE BEGIN
              IF STATUS>0 THEN CLRREGS();
              EXP := CDR EXP;
              WHILE EXP DO
                 <<COMVAL(CAR EXP,IF CDR EXP THEN 1 ELSE STATUS);
                      %to allow for recursion on last entry;
                   IREGSL := IREGS . IREGSL;
                   REGSL := REGS . REGSL;
                   IF CDR EXP
                     THEN <<ATTACH LIST(IF FN THEN 'JUMPNIL
                                           ELSE 'JUMPT,CAR LABL);
                            ADDJMP CODELIST>>;
                   EXP := CDR EXP>>;
              IREGS := RMERGE IREGSL;
              REGS := RMERGE REGSL
           END;
      ATTLBL LABL
   END;

SYMBOLIC PROCEDURE TSTANDOR(EXP,LABL);
   BEGIN SCALAR FLG,FLG1,FN,LAB2,REGSL,REGS1L,TAILP;
      %FLG is initial switch condition;
      %FN is appropriate AND/OR case;
      %FLG1 determines appropriate switching state;
      FLG := SWITCH;
      SWITCH := NIL;
      FN := CAR EXP EQ 'AND;
      FLG1 := FLG EQ FN;
      EXP := CDR EXP;
      LAB2 := GENLBL();
      CLRREGS();
      WHILE EXP DO
         <<SWITCH := NIL;
           IF NULL CDR EXP AND FLG1
             THEN <<IF FN THEN SWITCH := T;
                    COMTST(CAR EXP,LABL);
                    REGSL := REGS . REGSL;
                    REGS1L := REGS1 . REGS1L>>
            ELSE <<IF NOT FN THEN SWITCH := T;
                   IF FLG1
                     THEN <<COMTST(CAR EXP,LAB2);
                            REGSL := REGS1 . REGSL;
                            REGS1L := REGS . REGS1L>>
```

```
                    ELSE <<COMTST(CAR EXP,LABL);
                           REGSL := REGS . REGSL;
                           REGS1L := REGS1 . REGS1L>>>>;
          IF NULL TAILP
            THEN <<CONDTAIL := NIL . CONDTAIL; TAILP := T>>;
          EXP := CDR EXP>>;
      ATTLBL LAB2;
      REGS := IF NOT FLG1 THEN CAR REGSL ELSE RMERGE REGSL;
      REGS1 := IF FLG1 THEN CAR REGS1L ELSE RMERGE REGS1L;
      IF TAILP THEN CONDTAIL := CDR CONDTAIL;
      SWITCH := FLG
   END;


PUT('AND,'COMPFN,'COMANDOR);

PUT('OR,'COMPFN,'COMANDOR);

PUT('AND,'COMTST,'TSTANDOR);

PUT('OR,'COMTST,'TSTANDOR);


SYMBOLIC PROCEDURE COMCOND(EXP,STATUS);
   %compiles conditional expressions;
   %registers REGS and IREGS are set for dropping through,
   %REGS1 and IREGS1 are set for a branch;
   BEGIN SCALAR IREGS1,REGS1,FLAGG,SWITCH,LAB1,LAB2,REGSL,IREGSL,TAILP;
      EXP := CDR EXP;
      LAB1 := GENLBL();
      IF STATUS>0 THEN CLRREGS();
      FOR EACH X IN EXP DO <<LAB2 := GENLBL();
                             SWITCH := NIL;
                             IF CDR X THEN COMTST(CAR X,LAB2)
                                        %update CONDTAIL;
                              ELSE <<COMVAL(CAR X,1);
                                     ATTACH LIST('JUMPNIL,CAR LAB2);
                                     ADDJMP CODELIST;
                                     IREGS1 := IREGS;
                                     REGS1 := (1 . '(QUOTE NIL) .
                                                 CDAR REGS) . CDR REGS>>;
                             IF NULL TAILP
                               THEN <<CONDTAIL := NIL . CONDTAIL;
                                      TAILP := T>>;
                             COMVAL(CADR X,STATUS);
                                % Branch code;
                                %test if need jump to LAB1;
                             IF NOT TRANSFERP CAR CODELIST
                               THEN <<ATTJMP LAB1;
                                      IREGSL := IREGS . IREGSL;
                                      REGSL := REGS . REGSL>>;
                             REGS := REGS1;
          %restore register status for next iteration;
        IREGS := IREGS1;
        IREGS1 := NIL;
          %we do not need to set REGS1 to NIL since all COMTSTs
          %are required to set it;
        ATTLBL LAB2>>;
      IF NULL FLAGG AND STATUS<2
        THEN <<LREG1('(QUOTE NIL),STATUS);
               IREGS := RMERGE1(IREGS,IREGSL);
               REGS := RMERGE1(REGS,REGSL)>>
       ELSE IF REGSL
        THEN <<IREGS := RMERGE1(IREGS,IREGSL);
               REGS := RMERGE1(REGS,REGSL)>>;
      ATTLBL LAB1;
      IF TAILP THEN CONDTAIL := CDR CONDTAIL
   END;
```

```
SYMBOLIC PROCEDURE RMERGE U;
   IF NULL U THEN NIL ELSE RMERGE1(CAR U,CDR U);

SYMBOLIC PROCEDURE RMERGE1(U,V);
   IF NULL V THEN U ELSE RMERGE1(RMERGE2(U,CAR V),CDR V);

SYMBOLIC PROCEDURE RMERGE2(U,V);
   IF NULL U OR NULL V THEN NIL
    ELSE (LAMBDA X;
            IF X
              THEN (CAAR U . XN(CDAR U,CDR X))
                     . RMERGE2(CDR U,DELETE(X,V))
             ELSE RMERGE2(CDR U,V))
       ASSOC(CAAR U,V);

FLAG('(JUMP LINKE ERROR),'TRANSFER);

PUT('COND,'COMPFN,'COMCOND);

SYMBOLIC PROCEDURE COMCONS(EXP,STATUS);
   IF NULL (EXP := CDR EXP) OR NULL CDR EXP OR CDDR EXP
     THEN LPRIE "MISMATCH OF ARGUMENTS"
    ELSE IF CADR EXP='(QUOTE NIL)
     THEN CALL('NCONS,LIST CAR EXP,STATUS)
    ELSE IF EQCAR(RASSOC(CADR EXP,REGS),1)
        AND ANYREG(CAR EXP,NIL)
     THEN CALL1('XCONS,COMLIS REVERSE EXP,STATUS)
    ELSE IF ANYREG(CADR EXP,NIL) THEN CALL('CONS,EXP,STATUS)
    ELSE CALL1('XCONS,REVERSIP COMLIS EXP,STATUS);

PUT('CONS,'COMPFN,'COMCONS);

SYMBOLIC PROCEDURE COMGO(EXP,STATUS);
   <<CLRREGS();
     IF STATUS>2 THEN <<ATTJMP GETLBL CADR EXP; SLST := NIL>>
      ELSE LPRIE LIST(EXP,"INVALID")>>;

PUT('GO,'COMPFN,'COMGO);

SYMBOLIC PROCEDURE COMLIST(EXP,STATUS);
   %we only support explicit functions up to 5 arguments here;
   BEGIN SCALAR M,N,FN;
      EXP := CDR EXP;
      M := MIN(MAXNARGS,5);
      N := LENGTH EXP;
      IF N=0 THEN LREG1('(QUOTE NIL),STATUS)
       ELSE IF N>M THEN COMVAL(COMLIST1 EXP,STATUS)
       ELSE CALL(IF N=1 THEN 'NCONS
                    ELSE IF N=2 THEN 'LIST2
                    ELSE IF N=3 THEN 'LIST3
                    ELSE IF N=4 THEN 'LIST4
                    ELSE 'LIST5,EXP,STATUS)
   END;

SYMBOLIC PROCEDURE LIST2(U,V); U . (V . NIL);

SYMBOLIC PROCEDURE LIST3(U,V,W); U . (V . (W . NIL));

SYMBOLIC PROCEDURE LIST4(U,V,W,X); U . (V . (W . (X . NIL)));

SYMBOLIC PROCEDURE LIST5(U,V,W,X,Y); U . (V . (W . (X . (Y . NIL))));

SYMBOLIC PROCEDURE COMLIST1 EXP;
   IF NULL EXP THEN '(QUOTE NIL)
    ELSE LIST('CONS,CAR EXP,'LIST . CDR EXP);

PUT('LIST,'COMPFN,'COMLIST);
```

```
SYMBOLIC PROCEDURE PAMAP(U,VARS);
   IF EQCAR(CADDR U,'FUNCTION)
     THEN (LAMBDA X;
             LIST(CAR U,
                  PA1(CADR U,VARS),
                  MKQUOTE (IF ATOM X THEN X ELSE PA1(X,VARS))))
       CADR CADDR U
    ELSE CAR U . PALIS(CDR U,VARS);

PUT('MAP,'PA1FN,'PAMAP);

PUT('MAPC,'PA1FN,'PAMAP);

PUT('MAPCAN,'PA1FN,'PAMAP);

PUT('MAPCAR,'PA1FN,'PAMAP);

PUT('MAPCON,'PA1FN,'PAMAP);

PUT('MAPLIST,'PA1FN,'PAMAP);

SYMBOLIC PROCEDURE COMMAP(EXP,STATUS);
   BEGIN SCALAR BODY,FN,LAB1,LAB2,LAB3,TMP,MTYPE,RESULT,SLST1,VAR,X;
      BODY := CADR EXP;
      FN := CADDR EXP;
      LAB1 := GENLBL();
      LAB2 := GENLBL();
      MTYPE :=
       IF CAR EXP MEMQ '(MAPCAR MAPLIST) THEN 'CONS
        ELSE IF CAR EXP MEMQ '(MAPCAN MAPCON)
               THEN <<LAB3 := GENLBL(); 'NCONC>>
        ELSE NIL;
      CLRREGS();
      IF MTYPE THEN <<FRAME (RESULT := GENSYM());
                      IF NULL LAB3 THEN STORE0(RESULT,NIL)>>;
      FRAME (VAR := GENSYM());
      COMVAL(BODY,1);
      REGS := LIST LIST(1,VAR);
      IF LAB3 THEN <<STORE0(VAR,1); FRAME (TMP := GENSYM());
                     COMVAL('(NCONS 'NIL),1);
                     STORE0(RESULT,1); STORE0(TMP,1);
                     LREG1(VAR,1)>>;
      ATTJMP LAB2;
      ATTLBL LAB1;
      STORE0(VAR,1);
      X := IF CAR EXP MEMQ '(MAP MAPCON MAPLIST) THEN VAR
           ELSE LIST('CAR,VAR);
     IF EQCAR(FN,'QUOTE) THEN FN := CADR FN;
     SLST1 := SLST; %to allow for store in function body;
     COMVAL(LIST(FN,X),IF MTYPE THEN 1 ELSE 3);
     IF MTYPE
       THEN <<IF LAB3 THEN <<ATTACH LIST('JUMPNIL,CAR LAB3);
                             ADDJMP CODELIST;
                             ATTACH '(LOAD 2 1);
                             LREG1(TMP,1);
                             STORE0(TMP,2);
                             ATTACH '(LINK NCONC EXPR 2);
                             ATTLBL LAB3>>
                ELSE <<LREG(2,RESULT,NIL,1);
                       ATTACH '(LINK CONS EXPR 2);
                       STORE0(RESULT,1)>>;
              REGS := LIST (1 . NIL)>>;
     SLST := XN(SLST,SLST1);
     COMVAL(LIST('CDR,VAR),1);
     ATTLBL LAB2;
     ATTACH LIST('JUMPT,CAR LAB1);
     ADDJMP CODELIST;
```

```
      IF MTYPE
        THEN COMVAL(LIST(IF LAB3 THEN 'CDR ELSE 'REVERSIP,RESULT),1)
       ELSE REGS := LIST LIST(1,MKQUOTE NIL)
   END;

SYMBOLIC PROCEDURE XN(U,V);
   IF NULL U THEN NIL
    ELSE IF CAR U MEMBER V THEN CAR U . XN(CDR U,DELETE(CAR U,V))
    ELSE XN(CDR U,V);

PUT('MAP,'COMPFN,'COMMAP);

PUT('MAPC,'COMPFN,'COMMAP);

PUT('MAPCAN,'COMPFN,'COMMAP);

PUT('MAPCAR,'COMPFN,'COMMAP);

PUT('MAPCON,'COMPFN,'COMMAP);

PUT('MAPLIST,'COMPFN,'COMMAP);

SYMBOLIC PROCEDURE COMPROG(EXP,STATUS); %compiles program blocks;
   BEGIN SCALAR ALSTS,GOLIST,PG,PROGLIS,EXIT; INTEGER I;
      PROGLIS := CADR EXP;
      EXP := CDDR EXP;
      EXIT := GENLBL();
      PG := REMVARL PROGLIS; %protect prog variables;
      FOR EACH X IN PROGLIS DO FRAME X;
      ALSTS := FREEBIND(PROGLIS,NIL);
      FOR EACH X IN PROGLIS DO IF NOT NONLOCAL X THEN STOREO(X,NIL);
      FOR EACH X IN EXP DO IF ATOM X
                           THEN GOLIST := (X . GENLBL()) . GOLIST;
      WHILE EXP DO
         <<IF ATOM CAR EXP
             THEN <<CLRREGS();
                    ATTLBL GETLBL CAR EXP;
                    REGS := LIST (1 . NIL)>>
            ELSE COMVAL(CAR EXP,IF STATUS>2 THEN 4 ELSE 3);
           IF NULL CDR EXP
               AND STATUS<2
               AND (ATOM CAR EXP OR NOT CAAR EXP MEMQ '(GO RETURN))
            THEN EXP := LIST '(RETURN (QUOTE NIL))
            ELSE EXP := CDR EXP>>;
      ATTLBL EXIT;
      IF CDR FINDLBL EXIT THEN REGS := LIST (1 . NIL);
      FREERST(ALSTS,STATUS);
      RSTVARL(PROGLIS,PG)
   END;

PUT('PROG,'COMPFN,'COMPROG);

SYMBOLIC PROCEDURE REMVARL VARS;
   FOR EACH X IN VARS COLLECT REMVAR X;

SYMBOLIC PROCEDURE REMVAR X;
   %removes references to variable X from IREGS and REGS
   %and protects SLST;
   <<REMSTORES X; PROTECT X>>;

SYMBOLIC PROCEDURE REMSTORES X;
   BEGIN
      FOR EACH Y IN IREGS DO IF X EQ CADR Y
                             THEN <<STOREO(CADR Y,CAR Y);
                                    IREGS := DELETE(Y,IREGS)>>;
      FOR EACH Y IN REGS DO WHILE X MEMBER CDR Y DO
                             RPLACD(Y,DELEQ(X,CDR Y))
   END;
```

```
SYMBOLIC PROCEDURE PROTECT U;
   BEGIN SCALAR X;
      IF X := ATSOC(U,SLST) THEN SLST := DELEQ(X,SLST);
      RETURN X
   END;

SYMBOLIC PROCEDURE RSTVARL(VARS,LST);
   FOR EACH X IN VARS DO
      <<REMSTORES X; CLRSTR X; UNPROTECT CAR LST; LST := CDR LST>>;

SYMBOLIC PROCEDURE UNPROTECT VAL; %restores VAL to SLST;
   IF VAL THEN SLST := VAL . SLST;

SYMBOLIC PROCEDURE COMPROGN(EXP,STATUS);
   BEGIN
      EXP := CDR EXP;
      IF NULL EXP THEN RETURN NIL;
      WHILE CDR EXP DO
         <<COMVAL(CAR EXP,IF STATUS<2 THEN 2 ELSE STATUS);
           EXP := CDR EXP>>;
      COMVAL(CAR EXP,STATUS)
   END;

PUT('PROG2,'COMPFN,'COMPROGN);

PUT('PROGN,'COMPFN,'COMPROGN);

SYMBOLIC PROCEDURE COMRETURN(EXP,STATUS);
   <<IF STATUS<4 OR NOT ANYREG(CADR EXP,NIL)
       THEN LREG1(CAR COMLIS LIST CADR EXP,STATUS);
     ATTJMP EXIT>>;

PUT('RETURN,'COMPFN,'COMRETURN);

SYMBOLIC PROCEDURE COMSETQ(EXP,STATUS);
   BEGIN SCALAR X;
      EXP := CDR EXP;
      IF STATUS>1 AND (NULL CADR EXP OR CADR EXP='(QUOTE NIL))
        THEN STORE2(CAR EXP,NIL)
       ELSE <<COMVAL(CADR EXP,1);
              STORE2(CAR EXP,1);
              IF X := RASSOC(CAR EXP,IREGS)
                THEN IREGS := DELETE(X,IREGS);
              REGS := (1 . (CAR EXP . CDAR REGS)) . CDR REGS>>
   END;

SYMBOLIC PROCEDURE REMSETVAR(U,V);
   %removes references to SETQ variable U from regs list V;
   IF NULL V THEN NIL
    ELSE (CAAR V . REMS1(U,CDAR V)) . REMSETVAR(U,CDR V);

SYMBOLIC PROCEDURE REMS1(U,V);
   %removes references to SETQ variable U from list V;
   IF NULL V THEN NIL
    ELSE IF SMEMQ(U,CAR V) THEN REMS1(U,CDR V)
    ELSE CAR V . REMS1(U,CDR V);

SYMBOLIC PROCEDURE SMEMQ(U,V);
   %true if atom U is a member of V at any level (excluding
   %quoted expressions);
   IF ATOM V THEN U EQ V
    ELSE IF CAR V EQ 'QUOTE THEN NIL
    ELSE SMEMQ(U,CAR V) OR SMEMQ(U,CDR V);

SYMBOLIC PROCEDURE STORE2(U,V);
   BEGIN SCALAR VTYPE;
     REGS := REMSETVAR(U,REGS);
     IF VTYPE := NONLOCAL U
       THEN ATTACH LIST('STORE,V,LIST(VTYPE,U))
```

```
      ELSE IF NOT ATSOC(U,STOMAP)
       THEN ATTACH LIST('STORE,V,MKNONLOCAL U)
       ELSE STORE0(U,V)
   END;

PUT('SETQ,'COMPFN,'COMSETQ);

COMMENT Specific test open coding;

SYMBOLIC PROCEDURE COMEQ(EXP,LABL);
   BEGIN SCALAR U,V,W;
      U := CADR EXP;
      V := CADDR EXP;
      IF U MEMBER CDAR REGS THEN W := COMEQ1(V,U)
       ELSE IF V MEMBER CDAR REGS THEN W := COMEQ1(U,V)
       ELSE IF ANYREG(V,NIL) THEN <<COMVAL(U,1); W := LOCATE V>>
       ELSE IF ANYREG(U,LIST V)
        THEN <<COMVAL(V,1); W := LOCATE U>>
       ELSE <<U := COMLIS CDR EXP; W := LOCATE CADR U>>;
      ATTACH ((IF SWITCH THEN 'JUMPE ELSE 'JUMPN)
                  . (CAR LABL . W));
      IREGS1 := IREGS;
      REGS1 := REGS;
      ADDJMP CODELIST
   END;

SYMBOLIC PROCEDURE COMEQ1(U,V);
   IF ANYREG(U,LIST V) THEN LOCATE U
    ELSE <<COMVAL(U,1); LOCATE V>>;

PUT('EQ,'COMTST,'COMEQ);

SYMBOLIC PROCEDURE TESTFN(EXP,LABL);
   %generates c-macros JUMPC and JUMPNC;
   BEGIN SCALAR X;
      IF NOT (X := RASSOC(CADR EXP,REGS)) THEN COMVAL(CADR EXP,1);
      CLRREGS();
      ATTACH LIST(IF SWITCH THEN 'JUMPC ELSE 'JUMPNC,
                    CAR LABL,
                    IF X THEN CAR X ELSE 1,CAR EXP);
      REGS1 := REGS;
      ADDJMP CODELIST
   END;

COMMENT Support functions;

SYMBOLIC PROCEDURE MEMLIS(U,V);
   V AND (MEMB(U,CAR V) OR MEMLIS(U,CDR V));

SYMBOLIC PROCEDURE MEMB(U,V);
   IF ATOM V THEN U EQ V ELSE MEMB(U,CADR V);

SYMBOLIC PROCEDURE RASSOC(U,V);
   IF NULL V THEN NIL
    ELSE IF U MEMBER CDAR V THEN CAR V
    ELSE RASSOC(U,CDR V);

SYMBOLIC PROCEDURE REPASC(REG,U,V);
   IF NULL V THEN LIST LIST(REG,U)
    ELSE IF REG=CAAR V THEN LIST(REG,U) . CDR V
    ELSE CAR V . REPASC(REG,U,CDR V);

SYMBOLIC PROCEDURE CLRREGS; %store deferred values in IREGS;
   WHILE IREGS DO
      <<STORE0(CADAR IREGS,CAAR IREGS); IREGS := CDR IREGS>>;

SYMBOLIC PROCEDURE CFNTYPE FN;
   BEGIN SCALAR X;
```

```
      RETURN IF X := GET(FN,'CFNTYPE) THEN CAR X
             ELSE IF X := GETD FN THEN CAR X
             ELSE 'EXPR
   END;

SYMBOLIC PROCEDURE GENLBL;
   BEGIN SCALAR L;
      L := GENSYM();
      LBLIST := LIST L . LBLIST;
      RETURN LIST L
   END;

SYMBOLIC PROCEDURE GETLBL LABL;
   BEGIN SCALAR X;
      X := ATSOC(LABL,GOLIST);
      IF NULL X THEN LPRIE LIST(LABL," - MISSING LABEL -");
      RETURN CDR X
   END;

SYMBOLIC PROCEDURE FINDLBL LBLST; ASSOC(CAR LBLST,LBLIST);

SYMBOLIC PROCEDURE RECHAIN(OLBL,NLBL);
   % Fix OLBL to now point at NLBL;
   BEGIN SCALAR X,Y,USES;
      X := FINDLBL OLBL;
      Y := FINDLBL NLBL;
      RPLACA(OLBL,CAR NLBL); % FIX L VAR;
      USES := CDR X; % OLD USES;
      RPLACD(X,NIL);
      RPLACD(Y,APPEND(USES,CDR Y));
      FOR EACH X IN USES DO RPLACA(CDR X,CAR NLBL)
   END;

SYMBOLIC PROCEDURE MOVEUP U;
   IF CAADR U EQ 'JUMP
     THEN <<JMPLIST := DELEQ(CDR U,JMPLIST);
            RPLACW(U,CDR U);
            JMPLIST := U . JMPLIST>>
    ELSE RPLACW(U,CDR U);

SYMBOLIC PROCEDURE ATTLBL LBL;
 · IF CAAR CODELIST EQ 'LBL THEN RECHAIN(LBL,CDAR CODELIST)
    ELSE ATTACH ('LBL . LBL);

SYMBOLIC PROCEDURE ATTJMP LBL;
   BEGIN
      IF CAAR CODELIST EQ 'LBL
        THEN <<RECHAIN(CDAR CODELIST,LBL);
               CODELIST := CDR CODELIST>>;
      IF TRANSFERP CAR CODELIST THEN RETURN NIL;
      ATTACH ('JUMP . LBL);
      ADDJMP CODELIST
   END;

SYMBOLIC PROCEDURE TRANSFERP X;
   FLAGP(IF CAR X EQ 'LINK THEN CADR X ELSE CAR X,'TRANSFER);

SYMBOLIC PROCEDURE ADDJMP CLIST;
   BEGIN SCALAR X;
      X := FINDLBL CDAR CLIST;
      RPLACD(X,CAR CLIST . CDR X);
      JMPLIST := CLIST . JMPLIST
   END;

SYMBOLIC PROCEDURE REMJMP CLIST;
   BEGIN SCALAR X;
      X := FINDLBL CDAR CLIST;
      RPLACD(X,DELEQ(CAR CLIST,CDR X));
      JMPLIST := DELEQ(CLIST,JMPLIST);
      MOVEUP CLIST
   END;
```

```
SYMBOLIC PROCEDURE DELEQ(U,V);
   IF NULL V THEN NIL
    ELSE IF U EQ CAR V THEN CDR V
    ELSE CAR V . DELEQ(U,CDR V);

SYMBOLIC PROCEDURE FRAME U; %allocates space for U in frame;
   BEGIN SCALAR Z;
      STOMAP := LIST(U,Z := CADAR STOMAP - 1) . STOMAP;
      IF Z<CAR LLNGTH THEN RPLACA(LLNGTH,Z)
   END;

SYMBOLIC PROCEDURE GETFRM U;
   (LAMBDA X;
      IF X THEN CDR X ELSE LPRIE LIST("COMPILER ERROR: LOST VAR",U))
    ATSOC(U,STOMAP);

SYMBOLIC PROCEDURE GETFFRM U;
   BEGIN SCALAR X; X := GETFRM U; FREELST := X . FREELST; RETURN X
   END;

COMMENT Pass 3 of the compiler (post code generation fixups);

SYMBOLIC PROCEDURE PASS3;
   BEGIN SCALAR FLAGG; %remove spurious stores;
      FOR EACH J IN SLST DO <<STLST := DELEQ(CADR J,STLST);
                              RPLACA(CADR J,'NOOP)>>;
      FIXCHAINS();
      FIXLINKS();
      FIXFRM();
      ATTLBL EXIT;
      IF FLAGG
        THEN <<IF NOT !*NOLINKE
                    AND CAAR CODELIST EQ 'LBL
                    AND CAADR CODELIST EQ 'LINKE
                  THEN RPLACA(CDR CODELIST,
                              LIST('LINK,CADADR CODELIST,
                                   CADR CDADR CODELIST,
                                   CADDR CDADR CODELIST));
                   %removes unnecessary LINKE;
               ATTACH ('DEALLOC . LLNGTH);
               ATTACH LIST 'EXIT>>;
      PEEPHOLEOPT();
      FIXREST()
   END;

SYMBOLIC PROCEDURE FIXCHAINS;
   BEGIN SCALAR EJMPS,EJMPS1,P,Q; %find any common chains of code;
      IF NOT CAR CODELIST='LBL . EXIT THEN ATTLBL EXIT;
      CODELIST := CDR CODELIST;
      IF NOT CAR CODELIST='JUMP . EXIT THEN ATTJMP EXIT;
      EJMPS := REVERSE JMPLIST;
      WHILE EJMPS DO
         BEGIN
            P := CAR EJMPS;
            EJMPS := CDR EJMPS;
            IF CAAR P EQ 'JUMP
              THEN <<EJMPS1 := EJMPS;
                     WHILE EJMPS1 DO
                        IF CAR P=CAAR EJMPS1 AND CADR P=CADAR EJMPS1
                          THEN <<REMJMP P;
                                 FIXCHN(P,CDAR EJMPS1);
                                 EJMPS1 := NIL>>
                         ELSE EJMPS1 := CDR EJMPS1>>
         END
   END;


SYMBOLIC PROCEDURE FIXLINKS;
   %replace LINK by LINKE where appropriate;
```

```
   BEGIN SCALAR EJMPS,P,Q;
      EJMPS := JMPLIST;
      IF NOT !*NOLINKE
        THEN WHILE EJMPS DO
                BEGIN
                   P := CAR EJMPS;
                   Q := CDR P;
                   EJMPS := CDR EJMPS;
                   IF NOT CADAR P EQ CAR EXIT THEN RETURN NIL
                    ELSE IF NOT CAAR P EQ 'JUMP
                              OR NOT CAAR Q EQ 'LINK
                      THEN RETURN FLAGG := T;
                   RPLACW(CAR Q,
                          'LINKE
                            . (CADAR Q
                                 . (CADDAR Q
                                      . (CADR CDDAR Q . LLNGTH))));
                   REMJMP P
                END
       ELSE FLAGG := T
   END;

SYMBOLIC PROCEDURE FINDBLK(U,LBL);
   IF NULL CDR U THEN NIL
    ELSE IF CAADR U EQ 'LBL AND TRANSFERP CADDR U THEN U
    ELSE IF GET(CAADR U,'NEGJMP) AND CADADR U EQ LBL THEN U
    ELSE FINDBLK(CDR U,LBL);

PUT('NOOP,'OPTFN,'MOVEUP);

PUT('LBL,'OPTFN,'LBLOPT);

SYMBOLIC PROCEDURE LBLOPT U;
   BEGIN SCALAR Z;
      IF CADAR U EQ CADADR U THEN RETURN REMJMP CDR U
       ELSE IF CAADR U EQ 'JUMP
                 AND (Z := GET(CAADDR U,'NEGJMP))
                 AND CADAR U EQ CADR CADDR U
        THEN RETURN <<Z := Z . (CADADR U . CDDR CADDR U);
                      REMJMP CDR U;
                      REMJMP CDR U;
                      RPLACD(U,Z . (CADR U . CDDR U));
                      ADDJMP CDR U;
                      T>>
       ELSE RETURN NIL
   END;


SYMBOLIC PROCEDURE PEEPHOLEOPT;
   %'peep-hole' optimization for various cases;
   BEGIN SCALAR X,Z;
      Z := CODELIST;
      WHILE Z DO
         IF NOT (X := GET(CAAR Z,'OPTFN)) OR NOT APPLY(X,LIST Z)
           THEN Z := CDR Z
   END;


SYMBOLIC PROCEDURE FIXREST;
   %checks for various cases involving unique (and unused) labels
   %and sequences like (JUMPx lab) M1 ... Mn ... (LAB lab) M1 ... Mn
   %where Mi do not affect reg 1;
   BEGIN SCALAR LABS,TLABS,X,Y,Z;
      WHILE CODELIST DO
         <<IF CAAR CODELIST EQ 'LBL
             THEN <<LBLOPT CODELIST;
                    IF CDR (Z := FINDLBL CDAR CODELIST)
                      THEN <<Y := CAR CODELIST . Y;
```

```
                    IF NULL CDDR Z
                         AND TRANSFERP CADR Z
                         AND CAADR Y EQ 'LOAD
                         AND NOLOADP(CDADR Y,
                                     CDR ATSOC(CADR Z,
                                               JMPLIST))
                      THEN <<IF
                                 NOT NOLOADP(CDADR Y,
                                             CDR CODELIST)
                                THEN RPLACW(CDR CODELIST,
                                            CADR Y
                                            . CADR CODELIST
                                              . CDDR CODELIST);
                               RPLACW(CDR Y,CDDR Y)>>
                     ELSE <<IF NULL CDDR Z
                                AND CAADR CODELIST EQ 'JUMP
                                AND GET(CAADR Z,'NEGJMP)
                               THEN LABS :=
                                      (CADR Z . Y) . LABS;
                              IF TRANSFERP CADR CODELIST
                                THEN TLABS :=
                                       (CADAR Y . Y)
                                         . TLABS>>>>>>
      ELSE IF GET(CAAR CODELIST,'NEGJMP)
                AND (Z := ATSOC(CAR CODELIST,LABS))
       THEN <<X := CAR CODELIST;
              CODELIST := CDR CODELIST;
              Z := CDDR Z;
              WHILE CAR Y=CAR Z
                      AND (CAAR Y EQ 'STORE
                             OR CAAR Y EQ 'LOAD
                                AND NOT CADAR Y=1) DO
                 <<CODELIST := CAR Y . CODELIST;
                   RPLACW(Z,CADR Z . CDDR Z);
                   Y := CDR Y>>;
             CODELIST := X . CODELIST;
             Y := X . Y>>
     ELSE IF CAAR CODELIST EQ 'JUMP
               AND (Z := ATSOC(CADAR CODELIST,TLABS))
               AND (X :=
                     FINDBLK(CDR CODELIST,
                               IF CAAR Y EQ 'LBL THEN CADAR Y
                                ELSE NIL))
      THEN BEGIN SCALAR W;
              IF NOT CAADR X EQ 'LBL
                THEN <<IF NOT CAAR X EQ 'LBL
                          THEN X :=
                                 CDR RPLACD(X,
                                            ('LBL . GENLBL())
                                              . CDR X);
                       W :=
                        GET(CAADR X,'NEGJMP)
                          . (CADAR X . CDDADR X);
                       REMJMP CDR X;
                       RPLACD(X,W . (CADR X . CDDR X));
                       ADDJMP CDR X>>
               ELSE X := CDR X;
              W := NIL;
              REPEAT <<W := CAR Y . W; Y := CDR Y>>
                 UNTIL Y EQ CDR Z;
              RPLACD(X,NCONC(W,CDR X));
              REMJMP CODELIST;
              TLABS := NIL; %since code chains have changed;
              CODELIST := NIL . (CAR Y . CODELIST);
              Y := CDR Y
           END
      ELSE Y := CAR CODELIST . Y;
     CODELIST := CDR CODELIST>>;
  CODELIST := Y
END;
```

```
SYMBOLIC PROCEDURE NOLOADP(ARGS,INSTRS);
   %determines if a LOAD is not necessary in instruction stream;
   ATOM CADR ARGS
     AND (CAAR INSTRS EQ 'LOAD AND CDAR INSTRS=ARGS
           OR CAAR INSTRS EQ 'STORE
                AND (CDAR INSTRS=ARGS
                       OR NOT CADDAR INSTRS=CADR ARGS
                            AND NOLOADP(ARGS,CDR INSTRS)));

SYMBOLIC PROCEDURE FIXCHN(U,V);
   BEGIN SCALAR X;
      WHILE CAR U=CAR V DO <<MOVEUP U; V := CDR V>>;
      X := GENLBL();
      IF CAAR V EQ 'LBL THEN RECHAIN(X,CDAR V)
       ELSE RPLACW(V,('LBL . X) . (CAR V . CDR V));
      IF CAAR U EQ 'LBL THEN <<RECHAIN(CDAR U,X); MOVEUP U>>;
      IF CAAR U EQ 'JUMP THEN RETURN NIL;
      RPLACW(U,('JUMP . X) . (CAR U . CDR U));
      ADDJMP U
   END;

SYMBOLIC PROCEDURE FIXFRM;
   BEGIN SCALAR HOLES,LST,X,Y,Z; INTEGER N;
      IF NULL STLST AND NULL FREELST THEN RETURN RPLACA(LLNGTH,1);
      N := 0;
      WHILE NOT N<CAR LLNGTH DO
         <<Y := NIL;
           FOR EACH LST IN STLST DO IF N=CADDR LST
                                      THEN Y := CDDR LST . Y;
           FOR EACH LST IN FREELST DO IF N=CAR LST THEN Y := LST . Y;
           IF NULL Y THEN HOLES := N . HOLES ELSE Z := (N . Y) . Z;
           N := N - 1>>;
      Y := Z;
      IF CAAR Z>CAR LLNGTH THEN RPLACA(LLNGTH,CAAR Z);
      WHILE HOLES DO
         <<WHILE HOLES AND CAR HOLES<CAR LLNGTH DO HOLES := CDR HOLES;
           IF HOLES
             THEN <<HOLES := REVERSIP HOLES;
                    FOR EACH X IN CDAR Z DO RPLACA(X,CAR HOLES);
                    RPLACA(LLNGTH,
                           IF NULL CDR Z OR CAR HOLES<CAADR Z
                             THEN CAR HOLES
                            ELSE CAADR Z);
                    HOLES := REVERSIP CDR HOLES;
                    Z := CDR Z>>>>;
      %now see if we can map frame to registers;
      N := IF NARG<3 THEN 3 ELSE NARG + 1;
      IF FREELST OR NULL REGP CODELIST OR CAR LLNGTH<N - MAXNARGS
        THEN RETURN NIL;
      FOR EACH X IN STLST DO RPLACW(X,
                                    LIST('LOAD,
                                         N - CADDR X,
                                         IF NULL CADR X
                                           THEN '(QUOTE NIL)
                                          ELSE CADR X));
      WHILE Y DO
         <<FOR EACH X IN CDAR Y DO NOT CAR X>0
                                     AND RPLACA(X,N - CAR X);
             %first test makes sure replacement only occurs once;
           Y := CDR Y>>;
      RPLACA(LLNGTH,1)
   END;

SYMBOLIC PROCEDURE REGP U;
   %there is no test for LAMBIND/PROGBIND
   %since FREELST tested explicitly in FIXFRM;
   IF NULL CDR U THEN T
    ELSE IF CAAR U MEMQ '(LOAD STORE)
```

```
        AND NUMBERP CADAR U AND CADAR U>2
    THEN NIL
    ELSE IF FLAGP(CAADR U,'UNKNOWNUSE)
            AND
                 NOT (IDP CADADR U
                        AND (FLAGP(CADADR U,'ONEREG)
                               OR FLAGP(CADADR U,'TWOREG))
                        OR CAR U='JUMP . EXIT)
    THEN NIL
    ELSE REGP CDR U;

FLAG('(CODE LINK LINKE),'UNKNOWNUSE);

SYMBOLIC PROCEDURE CODE U;  EVAL U;

PUT('JUMPN,'NEGJMP,'JUMPE);

PUT('JUMPE,'NEGJMP,'JUMPN);

PUT('JUMPNIL,'NEGJMP,'JUMPT);

PUT('JUMPT,'NEGJMP,'JUMPNIL);

PUT('JUMPC,'NEGJMP,'JUMPNC);

PUT('JUMPNC,'NEGJMP,'JUMPC);

COMMENT Some arithmetic optimizations to reduce the amount of code
        generated;

SYMBOLIC PROCEDURE PAPLUS2(U,VARS);
   IF CADDR U=1 THEN LIST('ADD1,PA1(CADR U,VARS))
    ELSE IF CADR U=1 THEN LIST('ADD1,!&PA1(CADDR U,VARS))
    ELSE 'PLUS2 . PALIS(CDR U,VARS);

PUT('PLUS2,'PA1FN,'PAPLUS2);

SYMBOLIC PROCEDURE PADIFF(U,VARS);
   IF CADDR U=1 THEN LIST('SUB1,PA1(CADR U,VARS))
    ELSE 'DIFFERENCE . PALIS(CDR U,VARS);

PUT('DIFFERENCE,'PA1FN,'PADIFF);

SYMBOLIC PROCEDURE PALESSP(U,VARS);
   IF CADDR U=0 THEN LIST('MINUSP,PA1(CADR U,VARS))
    ELSE 'LESSP . PALIS(CDR U,VARS);

PUT('LESSP,'PA1FN,'PALESSP);

COMMENT removing unnecessary calls to MINUS;

SYMBOLIC PROCEDURE PAMINUS(U,VARS);
   IF EQCAR(U := PA1(CADR U,VARS),'QUOTE) AND NUMBERP CADR U
     THEN MKQUOTE ( - CADR U)
    ELSE LIST('MINUS,U);

PUT('MINUS,'PA1FN,'PAMINUS);
```

B.2 Compiler c-macro definitions for the IBM 360

```
COMMENT Global variable and flag values for IBM 360 version;

!*NOLINKE := T;

!*ORD := NIL;

!*PLAP := NIL;
```

```
!*R2I := T;

MAXNARGS := 22;

COMMENT The following fluid variables are used by the loader;

FLUID '(FRAMESIZE LNAME !*HOLDDEF);

COMMENT General functions;

SYMBOLIC PROCEDURE MKFUNC FN;
   <<IF !*HOLDDEF AND GET(FN,'!*SUBR)
       THEN <<PUT(FN,'SUBR,GET(FN,'!*SUBR)); REMPROP(FN,'!*SUBR)>>;
     MKQUOTE FN>>;

COMMENT LOGX, LOGZ and *ARG are used for address calculations;

COMMENT c-macro definitions for IBM 360;

SYMBOLIC PROCEDURE ALLOC N;
   IF N<4 THEN NIL
    ELSE LIST('LA,'PDS,4*N - 4,'(R0 PDS)) . '((BXH PDS K4 92 (R12)));

SYMBOLIC PROCEDURE DEALLOC N;
   IF N=0 THEN NIL ELSE LIST '(LR PDS PDL);

COMMENT ENTRY is handled by the loader;

SYMBOLIC PROCEDURE EXIT; '((CNOP) (BC 15 48 (R0 R12)));

SYMBOLIC PROCEDURE STORE(REG,FLOC);
   IF NULL REG THEN STORE('NILR,FLOC)
    ELSE IF NUMBERP REG
     THEN IF REG=1 THEN STORE('A,FLOC)
           ELSE IF REG=2 THEN STORE('Q,FLOC)
           ELSE APPEND(LIST LIST('L,'R1,!*ARG REG,'(R0 R13)),
                       STORE('R1,FLOC))
    ELSE IF ATOM FLOC THEN LIST LIST('ST,REG, - 4*FLOC,'(R0 PDL))
    ELSE LIST(LIST('L,'M,FLOC),LIST('ST,REG,0,'(NILR M)));

SYMBOLIC PROCEDURE JUMP ADR; LIST LIST('BC,15,ADR);

SYMBOLIC PROCEDURE JUMPC(ADR,REG,TYPE); JUMP2(ADR,REG,TYPE,1);

SYMBOLIC PROCEDURE JUMPNC(ADR,REG,TYPE); JUMP2(ADR,REG,TYPE,14);

SYMBOLIC PROCEDURE JUMPNIL ADR; LIST('(CR A NILR),LIST('BC,8,ADR));

SYMBOLIC PROCEDURE JUMPT ADR; LIST('(CR A NILR),LIST('BC,7,ADR));

SYMBOLIC PROCEDURE JUMPE(ADR,EXP);
   APPEND(JUMP1 EXP,LIST LIST('BC,8,ADR));

SYMBOLIC PROCEDURE JUMPN(ADR,EXP);
   APPEND(JUMP1 EXP,LIST LIST('BC,7,ADR));

SYMBOLIC PROCEDURE LBL ADR; LIST ADR;

SYMBOLIC PROCEDURE LAMBIND(REGS,ALST);
   BEGIN SCALAR X,Y;
      X := PAIR(REGS,ALST);
      WHILE X DO
         <<Y := APPEND(Y,STORE(CAAR X,CADDAR X)); X := CDR X>>;
      RETURN APPEND(Y,
                    '(CNOP)
                      . ('(BAL R2 20 (R0 R12)) . !*FLUIDVEC ALST))
   END;

SYMBOLIC PROCEDURE PROGBIND ALST;
   '(CNOP) . ('(BAL R2 40 (R0 R12)) . !*FLUIDVEC ALST);
```

```
SYMBOLIC PROCEDURE FREERSTR ALST;
   '(CNOP) . ('(BAL R2 24 (R0 R12)) . !*FLUIDVEC ALST);

SYMBOLIC PROCEDURE LOAD(REG,EXP);
   IF NUMBERP REG
     THEN IF REG=1 THEN LOAD('A,EXP)
          ELSE IF REG=2 THEN LOAD('Q,EXP)
          ELSE IF EXP=1 THEN LIST LIST('ST,'A,!*ARG REG,'(R0 R13))
          ELSE IF EXP=2 THEN LIST LIST('ST,'Q,!*ARG REG,'(R0 R13))
          ELSE IF EXP='(QUOTE NIL)
           THEN LIST LIST('ST,'NILR,!*ARG REG,'(R0 R13))
          ELSE APPEND(LOAD('R1,EXP),
                      LIST LIST('ST,'R1,!*ARG REG,'(R0 R13)))
    ELSE IF NUMBERP EXP
     THEN IF NOT EXP>0 THEN LIST LIST('L,REG, - 4*EXP,'(R0 PDL))
          ELSE IF REG=EXP THEN NIL
          ELSE IF NUMBERP EXP
           THEN IF EXP=1
                  THEN IF REG EQ 'A THEN NIL
                       ELSE LIST LIST('LR,REG,'A)
                 ELSE IF EXP=2
                  THEN IF REG EQ 'Q THEN NIL
                       ELSE LIST LIST('LR,REG,'Q)
                 ELSE LIST LIST('L,REG,!*ARG EXP,'(R0 R13))
          ELSE LIST LIST('LR,REG,EXP)
    ELSE IF CAR EXP EQ 'QUOTE
     THEN IF EXP='(QUOTE NIL) THEN LIST LIST('LR,REG,'NILR)
          ELSE IF ORDERP(CADR EXP,'FLOAT)
           THEN LIST LIST('LA,REG,
                          LOGX CADR EXP - LOGX NIL,
                          '(R0 NILR))
          ELSE LIST(LIST('L,REG,EXP),LIST('AR,REG,'NILR))
    ELSE IF CAR EXP MEMQ '(FLUID GLOBAL)
     THEN LIST(LIST('L,REG,EXP),LIST('L,REG,0,LIST('NILR,REG)))
    ELSE IF ATOM CADR EXP AND NOT CADR EXP>0
      THEN SUBPLIS(GET(CAR EXP,'ANYREG),LIST(REG, - 4*CADR EXP))
    ELSE (LAMBDA X;
            IF NULL X
              THEN LPRIE LIST("INCOMPLETE MACRO DEFINITION FOR",
                              CAR EXP)
             ELSE IF CADR EXP=1 THEN SUBPLIS(X,LIST(REG,'A))
             ELSE IF CADR EXP=2 THEN SUBPLIS(X,LIST(REG,'Q))
             ELSE APPEND(LOAD(REG,CADR EXP),
                         SUBPLIS(X,LIST(REG,REG))))
       GET(CAR EXP,'OPEN);

SYMBOLIC PROCEDURE LINK(FN,TYPE,NARGS);
   BEGIN SCALAR Y;
      RETURN IF FN EQ LNAME
               THEN LIST LIST('BAL,'R2,4*FRAMESIZE+170,'(R0 R12))
             ELSE IF Y := GET(FN,'OPEN) THEN !*OPEN Y
             ELSE LIST('(CNOP),
                       '(BAL R2 56 (R0 R12)),
                       LIST('AC,LOGZ(FN,NARGS)))
   END;

COMMENT LINKE is not used in IBM 360 version;

COMMENT Auxiliary functions used by the c-macros;

SYMBOLIC PROCEDURE !*OPEN U;
   IF CAR U EQ 'LAMBDA THEN SUBPLIS(U,'(A A)) ELSE U;

SYMBOLIC PROCEDURE SUBPLIS(U,V); SUBLIS(PAIR(CADR U,V),CADDR U);

SYMBOLIC PROCEDURE JUMP1 EXP;
   IF ATOM EXP
     THEN IF NOT EXP>0 THEN LIST LIST('C,'A, - 4*EXP,'(R0 PDL))
          ELSE IF EXP=2 THEN '((CR A Q))
          ELSE LIST LIST('C,'A,!*ARG EXP,'(R0 R13))
    ELSE APPEND(LOAD('R1,EXP),'((CR A R1)));
```

```
SYMBOLIC PROCEDURE JUMP2(ADR,REG,TYPE,BCSWITCH);
   BEGIN SCALAR X;
      TYPE :=
       CDR ASSOC(TYPE,
                 '((ATOM . 128)
                   (NUMBERP . 192)
                   (LOGP . 208)
                   (FLOATP . 224)
                   (BIGP . 200)));
      X :=
       LIST(LIST('TM,0,
                 IF REG=1 THEN '(A) ELSE IF REG=2 THEN '(Q) ELSE '(M),
                 TYPE),LIST('BC,BCSWITCH,ADR));
      RETURN IF REG<3 THEN X ELSE LIST('L,'M,!*ARG REG,'(R0 R13)) . X
   END;

COMMENT The vectors created by the following function have a series
       of 4-byte words with the H-O bit on.  Thus the end of a
       vector is signalled by a word without the bit on.  Therefore
       these vectors may NOT be followed by an instruction whose
       op-code is 80X or greater:  BXH, BXLE, LM, STM, TM, etc. In
       other words, the SI and SS instructions need a NOPR spacer in
       user-written code, but this is not a problem for the present
       compiler. Note that this H-O bit makes the AC list produced
       here look, when printed by a TRACE, like  'AC . garbage,
       because the print functions think it's an atomhead;

SYMBOLIC PROCEDURE !*FLUIDVEC ALST;
   % ALST is list (var , loc) where loc = 1:128;
   BEGIN SCALAR B,LL;
      WHILE ALST DO
         <<B := 128 - CADAR ALST;
           IF B<256
             THEN LL :=
                  ('AC
                     . (LOGZ(GET(CAAR ALST,'FLUID),B)
                          . (CAAR ALST . B)))
                    . LL
            ELSE LPRIE LIST(CAR ALST,"out of range in FLUIDVEC");
           ALST := CDR ALST>>;
      RETURN LL
   END;

COMMENT one instruction test c-macros for IBM 360;

PUT('ATOM,'COMTST,'TESTFN);

PUT('NUMBERP,'COMTST,'TESTFN);

PUT('LOGP,'COMTST,'TESTFN);

PUT('FLOATP,'COMTST,'TESTFN);

PUT('BIGP,'COMTST,'TESTFN);

COMMENT Additional c-macros defined in IBM 360 implementation;

SYMBOLIC PROCEDURE !*DEALEXT N;
   IF N>0 THEN '((CNOP) (BC 15 60 (R0 R12))) ELSE EXIT();

SYMBOLIC PROCEDURE !*SAVE X;
   LIST('ST . (X . '(0 (R0 PDS))),'(BXH PDS K4 0 (R12)));

SYMBOLIC PROCEDURE !*UNSAVE X;
   LIST('(SR PDS K4),'L . (X . '(0 (R0 PDS))));

SYMBOLIC PROCEDURE !*LM LOC;
   IF LOC>0 THEN LIST LIST('LM,'A,'Q,!*ARG LOC,'(R13))
    ELSE LIST LIST('LM,'A,'Q, - 4*LOC,'(PDL));
```

```
SYMBOLIC PROCEDURE !*STM LOC; LIST LIST('STM,'A,'Q, - 4*LOC,'(PDL));

FLAG('(!*DEALEXT !*SAVE !*UNSAVE !*LM !*STM),'MC);

COMMENT Open coded functions in this version;

DEFLIST('((CAR (LAMBDA (A B) ((L A 0 (R0 B)))))
          (CDR (LAMBDA (A B) ((L A 4 (R0 B)))))
          (LIST2 ((BAL R2 16 (R0 R12))))
          (LIST3 ((BAL R2 44 (R0 R12))))
          (ACONC ((BAL R2 84 (R0 R12))))
          (ATOM ((BAL R2 88 (R0 R12))))
          (CONS ((BALR R2 R12)))
          (FLAGP ((BAL R2 96 (R0 R12))))
          (GET ((BAL R2 100 (R0 R12))))
          (NCONC ((BAL R2 104 (R0 R12))))
          (NUMBERP ((BAL R2 108 (R0 R12))))
          (TERPRI ((BAL R2 112 (R0 R12))))
          (NCONS ((BAL R2 116 (R0 R12))))
          (XCONS ((BAL R2 120 (R0 R12))))
          (REVERSIP ((BAL R2 124 (R0 R12))))
          (IDP ((BAL R2 52 (R0 R12))))
          (STRINGP ((BAL R2 32 (R0 R12))))
          (PUT ((BAL R2 28 (R0 R12))))
          (EQUAL ((BALR R2 R11)))),'OPEN);

DEFLIST('((CAR (LAMBDA (X Y) ((L X Y (R0 PDL)) (L X 0 (R0 X)))))
          (CDR (LAMBDA (X Y) ((L X Y (R0 PDL)) (L X 4 (R0 X))))),
        'ANYREG);

COMMENT peephole optimizations for IBM 360 use;

SYMBOLIC PROCEDURE RETOPT U;
   IF EQCAR(CADR U,'DEALLOC)
     THEN RPLACW(U,('!*DEALEXT . CDADR U) . CDDR U)
    ELSE NIL;


PUT('EXIT,'OPTFN,'RETOPT);
```

B.3 Compiler c-macro definitions for the PDP-10

```
COMMENT Global variable and flag values for PDP-10 version;

!*NOLINKE := NIL;

!*ORD := NIL;

!*PLAP := NIL;

!*R2I := T;

MAXNARGS := 14;

COMMENT general functions;

SYMBOLIC PROCEDURE MKFUNC FN; MKQUOTE FN;

COMMENT c-macros for PDP-10 Implementation;

SYMBOLIC PROCEDURE ALLOC N;
   IF N=0 THEN NIL
    ELSE IF N=1 THEN LIST '(PUSH P 1)
    ELSE LIST(LIST('ADD,'P,LIST('C,0,0,N,N)),'(213 P 85 16));

SYMBOLIC PROCEDURE DEALLOC N;
   IF N>0 THEN LIST LIST('SUB,'P,LIST('C,0,0,N,N)) ELSE NIL;
```

```
COMMENT ENTRY is handled by the loader;

SYMBOLIC PROCEDURE EXIT; LIST '(POPJ P);

SYMBOLIC PROCEDURE STORE(REG,FLOC); % Uses R as extra reg;
   BEGIN SCALAR OP,PQ;
      IF NUMBERP FLOC
        THEN (IF FLOC>5 THEN FLOC := 'EXARG . (FLOC - 6)
              ELSE IF FLOC<1 THEN PQ := '(P))
       ELSE IF EQCAR(FLOC,'GLOBAL) THEN FLOC := 'FLUID . CDR FLOC;
      IF NUMBERP REG AND REG>5
        THEN RETURN IF IDP FLOC OR NUMBERP FLOC AND FLOC>0
                    THEN LOAD(FLOC,REG)
                    ELSE NCONC(LOAD('R,REG),
                               LIST ('MOVEM . ('R . (FLOC . PQ))));
      OP := IF REG THEN 'MOVEM ELSE <<REG := 0; 'SETZM>>;
      RETURN LIST (OP . (REG . (FLOC . PQ)))
   END;

SYMBOLIC PROCEDURE JUMP ADR; LIST LIST('JRST,0,ADR);

SYMBOLIC PROCEDURE JUMPNIL ADR; LIST LIST('JUMPE,1,ADR);

SYMBOLIC PROCEDURE JUMPT ADR; LIST LIST('JUMPN,1,ADR);

SYMBOLIC PROCEDURE JUMPE(ADR,EXP);
   NCONC(!*LOADEXP(1,EXP,'(CAMN . CAIN)),LIST LIST('JRST,0,ADR));

SYMBOLIC PROCEDURE JUMPN(ADR,EXP);
   NCONC(!*LOADEXP(1,EXP,'(CAME . CAIE)),LIST LIST('JRST,0,ADR));

SYMBOLIC PROCEDURE LBL ADR; LIST ADR;

SYMBOLIC PROCEDURE LAMBIND(REGS,ALST);
   %produces the parameter list for binding;
   BEGIN SCALAR X,Y;
      ALST := REVERSE ALST;
      REGS := REVERSE REGS;
      WHILE ALST DO
         <<IF NULL REGS THEN X := 0
            ELSE <<X := CAR REGS; REGS := CDR REGS>>;
           Y := LIST(0,X,LIST('FLUID,CAAR ALST)) . Y;
           ALST := CDR ALST>>;
      RETURN '(CALL 0 (E !*LAMBIND!*)) . Y
   END;

SYMBOLIC PROCEDURE PROGBIND ALST; LAMBIND(NIL,ALST);

SYMBOLIC PROCEDURE FREERSTR ALST; '((CALL 0 (E !*SPECRSTR!*)));

SYMBOLIC PROCEDURE LOAD(REG,EXP); % Uses R as extra reg;
   IF REG=EXP THEN NIL
    ELSE IF NUMBERP REG AND REG>5
     THEN IF IDP EXP OR NUMBERP EXP AND EXP>0 THEN STORE(EXP,REG)
          ELSE IF EXP='(QUOTE NIL) THEN STORE(NIL,REG)
          ELSE NCONC(LOAD('R,EXP),STORE('R,REG))
    ELSE !*LOADEXP(REG,EXP,'(MOVE . MOVEI));

SYMBOLIC PROCEDURE LINK(FN,TYPE,NARGS);
   MKLINK(FN,TYPE,NARGS,-1,'CALL);

SYMBOLIC PROCEDURE LINKE(FN,TYPE,NARGS,N);
   MKLINK(FN,TYPE,NARGS,N,'JCALL);

COMMENT Auxiliary functions used by the c-macros;

SYMBOLIC PROCEDURE !*OPEN U;
   IF CAR U EQ 'LAMBDA THEN SUBPLIS(U,'(1 1)) ELSE U;
```

```
SYMBOLIC PROCEDURE SUBPLIS(X,Y); SUBLIS(PAIR(CADR X,Y),CADDR X);

SYMBOLIC PROCEDURE !*LOADEXP(REG,U,OPS);
   %OPS=(direct . immediate). When not MOVE, uses D as extra reg;
   %REG is always an actual machine register;
   IF ATOM U
     THEN IF IDP U OR U>0 AND U<6 THEN LIST LIST(CAR OPS,REG,U)
          ELSE IF U>5 THEN LIST LIST(CAR OPS,REG,'EXARG . (U - 6))
          ELSE LIST LIST(CAR OPS,REG,U,'P)
    ELSE IF CAR U EQ 'QUOTE THEN LIST LIST(CDR OPS,REG,U)
    ELSE IF CAR U EQ 'GLOBAL THEN LIST LIST(CAR OPS,REG,'FLUID . CDR U)
    ELSE IF CAR U EQ 'FLUID THEN LIST LIST(CAR OPS,REG,U)
    ELSE IF NOT CAR OPS EQ 'MOVE
     THEN NCONC(LOAD('D,U),LIST LIST(CAR OPS,REG,'D))
    ELSE BEGIN SCALAR X,Y,Z;
           X := 'ANYREG;
           IF ATOM (Y := CADR U)
             THEN IF IDP Y THEN X := 'OPEN
                  ELSE IF Y<1 THEN Y := Y . '(P)
                  ELSE IF Y>5 THEN Y := LIST ('EXARG . (Y - 6))
                  ELSE X := 'OPEN
            ELSE IF CAR Y EQ 'GLOBAL THEN Y := LIST ('FLUID . CDR Y)
            ELSE IF CAR Y EQ 'FLUID THEN Y := LIST Y
            ELSE <<X := 'OPEN; Z := LOAD(REG,Y); Y := REG>>;
           IF NOT (X := GET(CAR U,X))
             THEN LPRIE LIST("Incomplete macro definition for",
                             CAR U);
           RETURN NCONC(Z,SUBPLIS(X,LIST(REG,Y)))
         END;

SYMBOLIC PROCEDURE MKLINK(FN,TYPE,NARGS,N,CALL);
   BEGIN SCALAR B,Y;
      B := N<0;
      IF (Y := GET(FN,'OPEN)) AND (B OR NOT FLAGP(FN,'NOPENR))
        THEN <<Y := !*OPEN Y;
               IF NOT B
                 THEN Y :=
                      APPEND(Y,LIST(LIST('DEALLOC,N),'(EXIT)))>>
       ELSE <<Y :=
               LIST LIST(CALL,
                         IF TYPE EQ 'FEXPR THEN 15 ELSE NARGS,
                         LIST('E,FN));
              IF N>0 THEN Y := LIST('DEALLOC,N) . Y>>;
      RETURN Y
   END;

COMMENT Peep-hole optimization tables;

SYMBOLIC PROCEDURE STOPT U;
   %this has to use fact that LLNGTH is offset during code generation;
   IF CDAR U='(1 0) AND CADR U='(ALLOC 0)
     THEN <<RPLACA(U,'(PUSH P 1)); RPLACD(U,NIL)>>
    ELSE IF CDAR U='(2 -1)
              AND CADR U='(STORE 1 0)
              AND CADDR U='(ALLOC -1)
     THEN <<RPLACA(U,'(PUSH P 1));
            RPLACA(CDR U,'(PUSH P 2));
            RPLACD(CDR U,NIL)>>;

PUT('STORE,'OPTFN,'STOPT);

COMMENT Some PDP-10 dependent optimizations;

SYMBOLIC PROCEDURE PAEQUAL(U,VARS);
   (LAMBDA(X,Y);
       IF EQVP X OR EQVP Y THEN 'EQ
        ELSE IF NUMBERP X OR NUMBERP Y THEN 'EQN
        ELSE 'EQUAL)
      (CADR U,CADDR U)
     . PALIS(CDR U,VARS);
```

```
PUT('EQUAL,'PA1FN,'PAEQUAL);

SYMBOLIC PROCEDURE EQP U;
   %EQP is true if U is an object for which EQ can replace EQUAL;
   INUMP U OR IDP U;

SYMBOLIC PROCEDURE EQVP U;
   %EQVP is true if EVAL U is an object for which EQ can
   %replace EQUAL;
   INUMP U OR EQCAR(U,'QUOTE) AND EQP CADR U;

SYMBOLIC PROCEDURE PAMEMBER(U,VARS);
   (LAMBDA(X,Y);
       IF EQVP X THEN 'MEMQ
        ELSE IF NOT EQCAR(Y,'QUOTE) THEN 'MEMBER
        ELSE BEGIN SCALAR A;
                A := (Y := CADR Y);
                WHILE Y AND A DO <<A := EQP CAR Y; Y := CDR Y>>;
                RETURN IF A THEN 'MEMQ ELSE 'MEMBER
             END)
      (CADR U,CADDR U)
       PALIS(CDR U,VARS);

PUT('MEMBER,'PA1FN,'PAMEMBER);

SYMBOLIC PROCEDURE PAASSOC(U,VARS);
   (LAMBDA(X,Y);
       IF EQVP X THEN 'ATSOC
        ELSE IF NOT EQCAR(Y,'QUOTE) THEN 'ASSOC
        ELSE BEGIN SCALAR A;
                A := T;
                Y := CADR Y;
                WHILE Y AND A DO <<A := EQP CAAR Y; Y := CDR Y>>;
                RETURN IF A THEN 'ATSOC ELSE 'ASSOC
             END)
      (CADR U,CADDR U)
     . PALIS(CDR U,VARS);

PUT('ASSOC,'PA1FN,'PAASSOC);

SYMBOLIC PROCEDURE COMAPPLY(EXP,STATUS); % Look for LIST;
   BEGIN INTEGER N,NN; SCALAR FN,ARGS;
      EXP := CDR EXP;
      FN := CAR EXP;
      ARGS := CDR EXP;
      IF CFNTYPE FN EQ 'FEXPR
        THEN LPRIE LIST(FN,"IS NOT AN EXPR FOR APPLY");
      IF NULL ARGS
           OR CDR ARGS
           OR NOT EQCAR(CAR ARGS,'LIST)
           OR (NN := (N := LENGTH CDAR ARGS))>MAXNARGS
        THEN RETURN CALL('APPLY,EXP,STATUS);
      ARGS := REVERSE (FN . REVERSE CDAR ARGS);
      ARGS := COMLIS ARGS;
      STORE1();
      FN := CAR ARGS;
      ARGS := CDR ARGS;
      IF STATUS>0 THEN CLRREGS();
      WHILE N>0 DO
         <<LREG(N,CAR ARGS,CDR ARGS,STATUS);
           ARGS := CDR ARGS;
           N := N - 1>>;
      ATTACH ('LINKF . (NN . LOCATE FN));
      REGS := LIST (1 . NIL)
   END;

PUT('APPLY,'COMPFN,'COMAPPLY);
```

```
SYMBOLIC PROCEDURE COMRPLAC(EXP,STATUS);
   BEGIN SCALAR FN,X,Y;
      FN := IF CAR EXP EQ 'RPLACA THEN '!*RPLACA ELSE '!*RPLACD;
      EXP := COMLIS CDR EXP;
      Y := IF CAR EXP = '(QUOTE NIL) THEN NIL
            ELSE IF Y := RASSOC(CAR EXP,REGS) THEN CAR Y
            ELSE <<LREG('TT,CAR EXP,CDR EXP,STATUS); 'TT>>;
      IF STATUS<2
        THEN <<IF Y=1 THEN LREG(Y := 'TT,CAR EXP,CDR EXP,STATUS);
               LREG1(CADR EXP,STATUS)>>;
      ATTACH (FN . (Y . LOCATE CADR EXP))
   END;

PUT('RPLACA,'COMPFN,'COMRPLAC);

PUT('RPLACD,'COMPFN,'COMRPLAC);

COMMENT Additional c-macros defined in PDP-10 implementation;

SYMBOLIC PROCEDURE LINKF(NARGS,FNEXP);
   !*LOADEXP(NARGS,FNEXP,'(CALLF!@ . CALLF));

SYMBOLIC PROCEDURE !*RPLACA(REG,EXP);
   !*LOADEXP!*(REG,EXP,'((RPLCA!@ . RPLCA) . (HRRZS!@ . HRRZS)));

SYMBOLIC PROCEDURE !*RPLACD(REG,EXP);
   !*LOADEXP!*(REG,EXP,'((RPLCD!@ . RPLCD) . (HLLZS!@ . HLLZS)));

SYMBOLIC PROCEDURE !*LOADEXP!*(REG,EXP,OPS);
 IF REG
   THEN IF NUMBERP REG AND REG>5
          THEN NCONC(LOAD('R,REG),!*LOADEXP('R,EXP,CAR OPS))
          ELSE !*LOADEXP(REG,EXP,CAR OPS)
  ELSE !*LOADEXP(0,EXP,CDR OPS);

FLAG('(LINKF !*RPLACA !*RPLACD),'MC);

FLAG('(LINKF),'UNKNOWNUSE);

COMMENT Open coded functions in this version;

PUT('CAR,'OPEN,'(LAMBDA (X Y) ((HLRZ X 0 Y))));

PUT('CDR,'OPEN,'(LAMBDA (X Y) ((HRRZ X 0 Y))));

FLAG('(RPLACA RPLACD),'NOPENR);

PUT('CAR,'ANYREG,'(LAMBDA (X Y) ((HLRZ!@ X . Y))));

PUT('CDR,'ANYREG,'(LAMBDA (X Y) ((HRRZ!@ X . Y))));
```

APPENDIX C: Additional Functions Needed for Run Time Support of Compiler

In addition to the functions defined in the Standard LISP Report, the code produced by this compiler requires the use of seven additional functions for run time support. These functions are defined below in RLISP syntax, and should therefore be loaded with any compiled code run in a Standard LISP environment. Since these functions are defined as part of RLISP itself, they need not be loaded separately if that language is used.

```
SYMBOLIC PROCEDURE NCONS U; U . NIL;

SYMBOLIC PROCEDURE XCONS(U,V); V . U;

SYMBOLIC PROCEDURE LIST2(U,V); U . V . NIL;
```

```
SYMBOLIC PROCEDURE LIST3(U,V,W); U . V . W . NIL;

SYMBOLIC PROCEDURE LIST4(U,V,W,X); U . V . W . X . NIL;

SYMBOLIC PROCEDURE LIST5(U,V,W,X,Y); U . V . W . X . Y . NIL;

SYMBOLIC PROCEDURE REVERSIP U;
        BEGIN SCALAR X,Y;
           WHILE U DO <<X := CDR U; Y := RPLACD(U,Y); U := X>>;
           RETURN Y
        END;
```

REFERENCES

1. J. McCarthy, 'Recursive functions of symbolic expressions and their computation by machine', *CACM*, **3**, 184–195 (1960).
2. J. R. Allen, *Anatomy of LISP*, McGraw-Hill, New York, 1978.
3. R. A. Saunders, 'The LISP system for the Q-32 computer', in E. C. Berkeley and D. G. Bobrow, (Eds), *The Programming Language LISP: Its Operation and Applications*, Information International Inc., Cambridge, Mass, 220–238 and appendix 290–317, 1964.
4. R. London, 'Correctness of Two Compilers for a LISP Subset', Stanford AI Project Memo No. AIM-151 (1971).
5. F. Blair, 'The structure of the LISP compiler', *Unpublished paper*, IBM Research, Yorktown Heights, 1971.
6. B. S. Greenberg, 'The Multics Maclisp compiler—The basic hackery', *Unpublished paper*, Honeywell Information Systems, Cambridge, Mass, 1977.
7. G. L. Steele, Jr., 'Fast arithmetic in Maclisp', *Proc. 1977 Macsyma Users' Conference*, NASA Report No. CP-2012, 215–224, 1977.
8. F. Motoyoshi, 'A portable LISP compiler for a hypothetical LISP machine', *Tech. Rep. 76–05*, Dept. of Info. Sc., Univ. of Tokyo, 1976.
9. J. Urmi, 'A Machine independent LISP compiler and its implications for ideal hardware', *Dissertation No. 22*, Linkoping, Sweden, 1978.
10. F. E. Allen, 'Bibliography on program optimization', *RC-5767*, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1975.
11. P. B. Schneck and E. Angel, 'A FORTRAN to FORTRAN optimizing compiler', *The Computer Journal*, **16**, 322–330 (1973).
12. J. Marti *et al.*, 'Standard LISP Report', *SIGPLAN Notices*, ACM, New York, **14**, (10), 48–68 (1979).
13. A. C. Hearn, 'REDUCE symbolic mode primer', Utah Symbolic Computation Group, *Operating Note No. 5*, 1973.
14. M. L. Griss and R. R. Kessler, 'A Micro-programmed implementation of standard LISP and REDUCE on the Burroughs B1700/B1800 computer', Utah Symbolic Computation Group, *Report No. UCP-70*, 1979. (Submitted to *IEEE-TC*).
15. H. G. Baker, 'Shallow binding in LISP 1.5', *CACM*, **21**, 565–569, (1978).
16. T. Risch, 'REMREC, a program for automatic recursion removal in LISP', *Datalog. Report No. DLU 73/24*. Uppsala, Sweden, 1973.
17. B. W. Arden, B. A. Galler and R. M. Graham, 'An algorithm for translating Boolean expressions', *JACM*, **9**, 222–239 (1962).
18. M. Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, New Jersey, 1973.
19. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, V2*, Prentice-Hall, New Jersey, 1973.
20. M. L. Griss, 'Enhanced pattern matching in REDUCE', Utah Symbolic Computation Group, *Operating Note No. 22*, 1978.
21. R. R. Kessler, 'PMETA—pattern matching META/REDUCE', Utah Symbolic Computation Group, *Operating Note No. 40*, 1979.
22. C. W. Frazer, 'A compact, machine-independent peephole optimizer', *Proceedings of the Sixth Annual ACM Symposium on Principals of Programming Languages*, ACM, 1, 1979.
23. W. M. McKeeman, 'Peephole optimization', *CACM*, **8**, 443–444, (1965).

24. W. A. Wulf, R. Johnson, C. Weinstock, S. Hobbs and C. Geschke, '*The Design of An Optimizing Compiler*, American Elsevier, New York, 1975.
25. A. C. Hearn, 'A mode analyzing algebraic manipulation program', *Proceedings of ACM 74*, ACM, New York, 722–724, (1974).
26. M. L. Griss, 'The definition and use of data structures in REDUCE', *Proc. SYMSAC 76*, ACM, New York, 53–59, (1976).
27. F. E. Allen and J. Cocke, 'A program data flow analysis', *CACM*, **19**, ACM, New York, 137–147, (1976).
28. I. B. Frick, 'A Portable loader and fast loader for Standard LISP', (in preparation).
29. A. C. Hearn and A. C. Norman, 'A one-pass prettyprinter', *SIGPLAN Notices*, ACM, New York, **14**, (12), 50–58, (1979).
30. A. C. Hearn, 'REDUCE 2 users manual', Utah Computational Physics Group, *Report No. UCP–19* 1973.
31. M. L. Griss and R. R. Kessler, 'REDUCE/1700: a micro-coded algebra system', *Proc. Micro 11*, IEEE, New York, 130–138, (1978).
32. J. P. Fitch, 'Manual for Standard LISP on IBM System 360 and 370', Utah Symbolic Computation Group, *Tech. Rep. No. TR–6*, 1978.
33. I. B. Frick, 'Manual for Standard LISP on DECsystem 10 and 20', Utah Symbolic Computation Group, *Tech. Rep. No. TR-2*, 1978.