Stanford Artificial Intelligence Laboratory
Memo AIM-259

May 1975

Computer Science Department
Report No. STAN-CS-75-498

Automatically Proving the Correctness of Translations Involving Optimized Code

by

Hanan Samet

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

Stanford Artificial Intelligence Laboratory
Memo AIM-259

May 1975

Computer Science Department
Report No. STAN-CS-75-498

Automatically Proving the Correctness of Translations Involving Optimized Code

by

Hanan Samet

ABSTRACT

A formalism is described for proving that programs written in a higher level language are correctly translated to assembly language. In order to demonstrate the validity of the formalism a system has been designed and implemented for proving that programs written in a subset of LISP 1.6 as the high level language are correctly translated to LAP (an assembly language for the PDP-10) as the low level language. This work involves the identification of critical semantic properties of the language and their interrelationship to the instruction repertoire of the computer executing these programs. A primary use of the system is as a postoptimization step in code generation as well as a compiler debugger.

The assembly language programs need not have been generated by a compiler and in fact may be handcoded. The primary restrictions on the assembly language programs relate to calling

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22161.

sequences and well-formedness. The assembly language programs are processed by a program understanding system which simulates their effect and returns as its result a representation of the program in the form of a tree.

The proof procedure is independent of the intermediary mechanism which translates the high level language into the low level language. A proof consists of applying valid transformations to show the equivalence of the forms corresponding to the assembly language program and the original higher level language program, for which there also exists a tree-like intermediate form.

Some interesting results include the ability to handle programs where recursion is implemented by bypassing the start of the program, the detection and pinpointing of a wide class of errors in the assembly language programs, and a deeper understanding of the question of how to deal automatically with translations between high and extremely low level languages.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ACKNOWLEDGEMENTS

I would like to express my thanks to the following people who have contributed in different ways to my thesis.

My adviser, Professor Vinton Cerf, whose help, enthusiasm, understanding, advice, and availability played an immeasuarable role in this work.

Professors Donald Knuth and Terry Winograd who served on the reading committee and made many valuable suggestions. Professor Jerome Feldman who was my initial adviser and who encouraged me to continue this work. Professor John McCarthy who stimulated (via a CS206 class project) and supported the work reported in this thesis.

Many thanks go to Jim Low and John Allen both of whom patiently listened to me as I debugged my ideas at various stages. John Reiser for having done a thorough job of reading the thesis, and Mike Farmwald for help in a last minute proof reading session.

I would also like to thank collectively all of the people at the AI lab who have provided the facilities with which I was able perform the work reported in this thesis. Special thanks go to Brian Harvey and Martin Frost for their help with PUB, the document compiler; to Dave Barstow for creating the graphic fonts which I used to create the figures, and thereby write the entire thesis in PUB; and to Tom Wolpert who helped make the proof system run under the TENEX operating system and the UCILISP environment.

I would be remiss if I did not thank the following people who have played, in various stages of my life, a vital role in enabling me to do this work: Margalit and Seev Berlinger, Harry Greiff, Eytan and Varda Selberg, and Maurice and Toni Weinberger.

Finally and most importantly, I would like to thank my parents Julius and Lotte Samet whose love and sacrifice made it all possible.

TABLE OF CONTENTS

CHAPTER PAGE

PREFACE

This thesis describes a formalism for proving that programs written in a higher level language are correctly translated to assembly language. It is hoped that the reader will go away with more than the thought that a tool has been created. More importantly, we have gained a deeper understanding of the problem of how to deal automatically with translations of programs between high and extremely low level languages. We have implemented a system to handle most of the contents of the thesis. We have used LISP as our higher level language and a variant of PDP-10 assembly language known as LAP as our object language. A proof procedure is presented which is independent of the intermediary mechanism which translates the former into the latter. The system is based on the identification of critical semantic properties of LISP and their interrelationship to the instruction repertoire of the computer executing the LISP programs. The selection of the PDP-10 as the host machine was done to illustrate an application of the ideas; the instruction sets of other computers could easily be incorporated.

The thesis is divided into eight chapters. The purpose of the first chapter is to convey to the reader a notion of the power of the concepts expressed in the work. It is designed to be self contained. Examples are given in an ALGOL-like variant of LISP known as meta-LISP. A brief description of the LISP implementation with respect to the PDP-10 is also given. The examples do not presuppose a familiarity with the assembly language of the PDP-10. Each assembly language encoding is fully annotated in terms of the operations performed, and their relationship to the LISP function being encoded. The basic motivation for this chapter is to set a loose framework for subsequent discussions while providing a brief summary of the type of results that can be expected from the techniques espoused in this work. The reader who is totally unfamiliar with LISP could glance at Chapter 2 first where many of the relevant concepts are outlined. However, there is clearly no need to read the previous chapter in its entirety in order to follow the introductory chapter.

Chapter 2 gives a definition of LISP in addition to an outline of the requirements that an equivalence proving system places on a LISP system. These requirements lead to a design of a LISP system that would satisfy them. Chapter 3 presents the canonical form used for the representation of a LISP program. Chapter 4 is a description of the assembly program understanding system. This includes a method for describing an instruction set of a computer. Chapter 5 binds the results of the previous two chapters to define a proof procedure for equivalence. Chapter 6 indicates the error detection capabilities of the system and applies the proof methods using a rather complicated example. Once the errors are detected, we show how an automatic system could correct them. Thus we outline an automatic debugger. Chapter 7 provides a pair of examples illustrating the mechanics of the proof procedure. Chapter 8 combines a perspective on the previous chapters with suggestions for future research.

Therefore in summary we present the following characterization of a reader. See where you fit in and read the relevant parts which are self contained.

1. Browser: Chapter 1.
2. Curious about LISP: Chapter 2.
3. Formalism: Chapter 3.
4. Hardware: Chapter 4.
5. Iron stomach: Chapters 3,4,5 and Phillips Milk of Magnesia.

6. Hacker: Chapter 6.
7. Non-believer: Chapter 7.
8. Graduate student: You read this far, so you might as well read Chapter 8.

CHAPTER 1

THE OPTIMIZER'S ASSISTANT

This thesis describes a formalism for proving that programs written in a higher level language are correctly translated to assembly language. Through this we hope to gain a deeper understanding of the problem of how to deal automatically with translations of programs between high and extremely low level languages. We implemented a system to handle virtually all of the program examples in the thesis using a subset of LISP[McCarthy60] as our higher level language and a variant of PDP-10[DEC69] assembly language, known as LAP[Quam72], as our object language. The proof procedure is independent of the intermediary mechanism which transforms the former into the latter. The system is based on the identification of critical semantic properties of LISP and their interrelationship to the instruction repertoire of the computer executing these programs. The selection of the PDP-10 as the host machine is merely done to illustrate an application of the ideas; the instruction sets of other computers could be incorporated.

We are interested in proving that programs are correctly translated. A similar problem that has been receiving much attention in the past few years is that of proving programs correct. Most of the attempts have been along the lines of assertions([Floyd67],[King69]) about the intent of the program which are then proved to hold. The difficulties with such methods are numerous. Most notable are the problems encountered in specifying the assertions[Deutsch73] and the actual proof methods. Proofs using such methods reduce to showing that a set of assertions hold. However, when examining such proofs we must allow for the possibility that the assertions are inadequate to specify all of the effects of the program in question. Thus we are led to a belief that the concept of intent is too imprecise for proving correctness of compilation. We feel that it is justifiable in proving equivalence between algorithms. Nevertheless, in the case of computer programs written in a higher level language we are primarily interested in the correctness of the translation. In this case, there is no need for any knowledge about the purpose of the program to be translated. As an example of a problem in which use is made of the purpose of the program, consider two methods of computing the greatest common divisor. In such a case we have defined an input-output pair relationship (i.e. the greatest common divisor) and we wish to determine if the two algorithms actually yield the same results for all possible inputs. The problem of proving the equivalence of two different algorithms is known to be unsolvable in general by use of halting problem-like arguments. We do not deal with such problems in this thesis.

Notice that we prove the correctness of the translation. One method of achieving this is to prove that the translator (e.g. a compiler) is correct – e.g. to prove that there does not exist a program which is incorrectly translated by the compiler. In this case we would revert to the intent characterization of correctness set forth in the previous paragraph. Instead, we prove for each program input to the translation process, that the translated version is equivalent to the original program. Thus, we are not saying anything about the general correctness of the translation process. A proof must be generated for each input to the translation process. However, this has several important advantages, especially when the translator is a compiler. First, as long as the compiler does its job for each case input to it, then its correctness is of a secondary nature – i.e. we have bootstrapped ourselves to the state where we can attribute an effective correctness to the compiler. Second, the proof process is independent of the compiler. The latter means that if another compiler were used, no difference would result. This implies that programs could be hand compiled or translated. This is quite important and identifies our proof as belonging to the semantics of the high and low level languages in which the input and output respectively are expressed rather than to the translation process. Third, any proof method that would prove a compiler correct would be limited with respect to the types of optimizations that it could allow. This is because such a proof would rely on the identification of all the possible input output pairs for code sequences. This is the type of approach taken in the proof of the correctness of LCOM0 and LCOM4 [London71].

Our proof system is not based on an assumption of an existence of a unique relationship between the source code and the object code. We feel that compilation is a many–to–many process – i.e. there is no one–to–one relationship between source code and object code. Thus there is no reflection of the source level syntax in the object code as is common in decompilation[Hollander73] systems which attempt to reconstruct a program from the object code. We make no such attempts at reconstructing the program. Instead we use an intermediate representation of the program which reflects all of the computations and decisions that are performed. In addition, this representation reflects an ordering based on the relative times at which the various computations are executed. This representation is known as the canonical form.

At this point an overview of the proof system is appropriate. The original LISP program is converted to the intermediate form by a set of transformations outlined in Chapter 3. Similarly, the LAP program is converted to the intermediate form by means of a process known as *symbolic execution*[†]. This procedure is outlined in Chapter 4. Next an attempt is made to prove that the two intermediate forms can be transformed into each other. This is the subject of the discussion in Chapter 5. During the proof procedure inequivalence may be detected and the sources of error can often be pinpointed. Chapter 6 provides an insight into the debugging capabilities afforded by the system.

In order to obtain the intermediate form representation of the object program we require an assembly language understanding system. Such a system includes a mechanism for describing a computer instruction set and to some degree its basic architecture. For example, see Figure 1.1 where the MOVE[‡] instruction is illustrated. Furthermore, these descriptions must be related to the semantics of the higher level language. In order to be able to accomplish this task we must reduce the semantics of the source language to a form which is compatible with the assembly language without indicating how each construct is encoded. This is one of the reasons for choosing LISP as our high level language. LISP is one of the simplest languages in terms of its constructs; and one of the most powerful in terms of its capabilities. The simplicity of its basic features and its similarity to case analysis allows the use of the same intermediate representation for the source program and the object program.

```
FEXPR MOVE(ARGS);
LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));
```

Figure 1.1 - MOVE instruction

[†] The meaning of symbolic execution is different from its use in the EFFIGY program testing system[King74]. In that system the term is used to denote a method of trying out various cases of a high level language program by using symbolic values for the parameters.

[‡] The instruction loads the accumulator with the contents of the effective address.

The previous intermediate form can be directly obtained from the original LISP program by a set of transformations as shown in Chapter 3. Moreover, this same intermediate form is used to represent the effect of the object program. A correctness proof is reduced to transforming one of the two intermediate forms into the other. The proof system is capable of detecting a certain class of errors in the optimized program. These errors pertain to programs that are well–formed – i.e. the computations performed in the assembly language program can be described using our primitives. Moreover, well–formedness implies that there is adherence to certain conventions pertaining to proper interfacing between functions (e.g. calling sequences, etc.).

The main motivation for this thesis has been a desire to write an optimizing compiler for LISP. In order to do this properly, it was determined that a proof was necessary of the correctness of the optimization process. Thus we are especially interested in proving that small perturbations in the code leave the effect of the function unchanged. It is our feeling that this is where the big payoff lies in optimizing LISP. This resulted in part from the observation that the primary action of a LISP program is to set up proper linkages between various functions. This view is buttressed by the simplicity of the primitive operations available in LISP. It quickly became clear that formal methods used in previous correctness work (i.e. proofs by use of predicate calculus) would be of little value. Thus we were led to design a representation of the program which would be independent of these perturbations.

The remainder of this chapter presents a scenario of an optimization process in which a user sits at his terminal and interactively applies transformations to his program. During this process, mistakes may be made and if possible they are detected and the user is informed of his errant ways. This is quite similar to a system for achieving the type of results proposed in [Knuth74]. The examples serve a dual purpose. First, they show the reader what PDP–10 assembly language looks like without having to refer to the manual. This is accomplished by annotating each instruction by a verbal description of its effect thereby gently breaking the reader into the notation used in this thesis. In addition, the optimizations used in the examples provide an indication of the power that can be derived from such methods without being an expert assembly language programmer†. Second, many of the concepts that are embodied by the examples will reappear in subsequent discussions and thus we are actually laying a foundation for these discussions. At the conclusion of the chapter we will outline some of the difficult problems that arise in the attempts to achieve our goal of proving the correctness of the translations.

Two examples are given. The first is rather detailed and shows how a function can be transformed step by step into an optimal encoding. We also indicate when such transformations can no longer be applied. At that point we change the algorithm to accomplish further optimizations. The second example is along similar lines, except that the modification of the algorithm is of a more complex nature. During the process, we also show some erroneous transformations which can be detected by the system. Any thoughts of similarity with decompilation should be disspelled by the examples.

Consider the function NEXT which takes as its arguments a list L and an element X. It searches L for an occurrence of X. If such an occurrence is found, and if it is not the last element of the

† The hackers among the readers may recognize even more optimal ways of achieving the desired encodings. They will have to have patience, since one of the intents of the discussion is to introduce the various instructions of the PDP–10 and to see how they can be used in reducing time and space requirements.

list, then the next element in the list is returned as the result of the function. Otherwise, NIL is returned. For example, application of the function to the list (A B C D E) in search of D would result in E, while a search for E or F would result in NIL. One formulation of this function in a dialect of LISP[McCarthy60] known as MLISP[Smith70] (i.e. meta-LISP) is given in Figure 1.2. This formulation of the function will be referred to as Algorithm 1.

```
NEXT(L,X) ≡ if NULL(L) or NULL(CDR(L)) then NIL
            else if CAR(L) EQ X then CADR(L)
            else NEXT(CDR(L),X)
```

Figure 1.2 - Algorithm 1 for NEXT

The symbolic representation of the intermediate form of the function given in Figure 1.2 is shown on the left side of Figure 1.3. Notice the tree-like representation where non-terminal nodes represent predicates and terminal nodes denote results. The left and right subtrees correspond to the true and false cases respectively of the predicate appearing in the root node. In addition, we also record, as part of the intermediate form, a numeric representation which serves to indicate a relative ordering of the sequence of computation. The actual numbers are irrelevant – i.e. only the partial ordering is of any importance. The significance of the numbers will become more apparent in Chapters 3 and 5 when we discuss the rearranging of the order of computing functions. The numeric representation for the NEXT function is given in the right side of Figure 1.3.

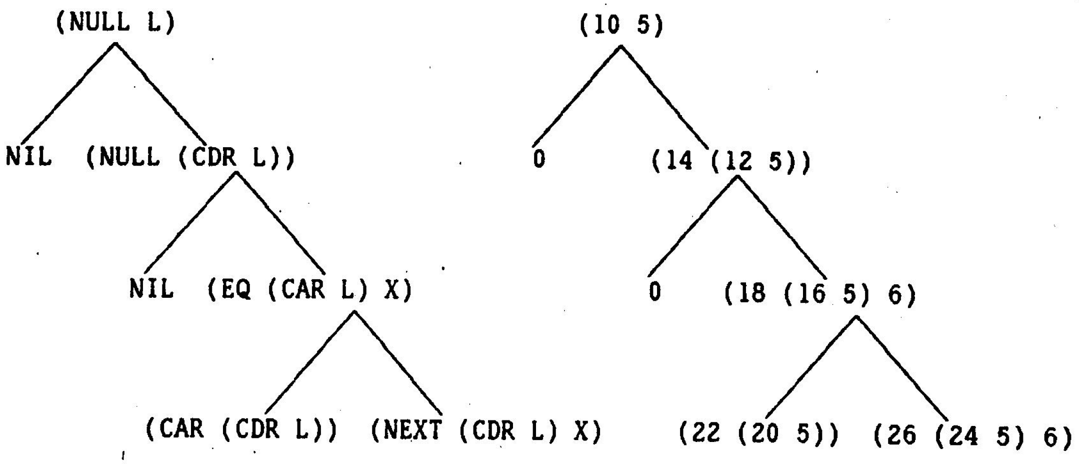


Figure 1.3 - Intermediate Form of Figure 1.2

The LISP 1.6 compiler at Stanford[Quam72] generates the code given in Figure 1.4 for this function. The code is in LAP, a variant of PDP-10 assembly language, which is described in Appendices 1 and 2. Briefly, each PDP-10 word is 36 bits wide and can be partitioned into two 18 bit halves. A LISP cell is represented by a full word whose left and right halves point to CAR and CDR respectively. Addresses of atoms are represented by (QUOTE <atom name>) and by zero in the case of the atom NIL. The left and right halves of an address denoting an atom contain identical values corresponding to inaccessible locations. A list of the form (C 0 0 num1 num2) appearing in the address field of an instruction is interpreted as an address of a word containing num1 and num2 in its right and left halves respectively. The PDP-10 has a hardware stack and LISP assumes that functions return via a return address which has been placed on the stack by the invoking function. A LAP program expects to find its parameters in the accumulators†, and also returns its result in accumulator 1. In our case L and X are in accumulators 1 and 2 respectively. The registers containing the parameters are always of such a form that a 0 is in the left half and the LISP pointer is in the right half. All parameters are assumed to be valid LISP pointers. A program is entered at its first instruction and a return address is situated in the top entry of a stack whose pointer is in register P. Whenever recursion or a function call to an external function (via the CALL or JCALL mechanism) occurs, the contents of all the accumulators are assumed to have been destroyed unless otherwise known. In the examples used here this is true for all functions except for CONS and XCONS (the antisymmetric counterpart of CONS which differs only in the order of the arguments). In this case all of the accumulators except 1 and 2 have their contents preserved.

```
(LAP NEXT SUBR)
        (PUSH P 1)              push L on the stack
PC2     (PUSH P 2)              push X on the stack
PC3     (JUMPE 1 TAG3)          jump to TAG3 if L is NIL
        (HRRZ@ 1 1)             load register 1 with CDR(L)
PC5     (JUMPN 1 TAG2)          jump to TAG2 if CDR(L) is not NIL
TAG3    (MOVEI 1 (QUOTE NIL))   load register 1 with NIL
PC7     (JRST 0 TAG1)           jump to TAG1
TAG2    (HLRZ@ 1 -1 P)          load register 1 with CAR(L)
        (CAME 1 2)              skip if CAR(L) is EQ to X
        (JRST 0 TAG4)           jump to TAG4
        (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
        (HLRZ@ 1 1)             load register 1 with CAR(CDR(L))
        (JRST 0 TAG1)           jump to TAG1
TAG4    (MOVE 2 0 P)            load register 2 with X
        (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
        (CALL 2 (E NEXT))       call NEXT(CDR(L),X)
TAG1    (SUB P (C 0 0 2 2))     undo the first two push operations
        (POPJ P)                return
        NIL
```

Figure 1.4

There are many unnecessary operations in this encoding. For example, there is no need to push X on the stack at location PC2. Therefore, at location TAG1 there is only a need to subtract one from the stack pointer. Likewise register 2 need not be reloaded with X at location TAG4 since it still contains X. The test at location PC3 may go directly to location TAG1 rather than to location TAG3 where NIL is loaded into register 1 which already contains NIL if entered via location PC3. The same line of reasoning holds when TAG3 is entered via PC5. Thus the instruction at location TAG3 is redundant and can be removed. This causes a reevaluation of the necessity of the two instructions at locations PC5 and PC7. The instruction at PC7 can be removed and, by reversing the sense of the test, the instruction at location PC5 can be changed to conditionally branch to location TAG1. At this point we have the encoding given in Figure 1.5.

† All accumulators can be used as index registers on the PDP-10. In our subsequent discussion we will use the terms accumulator and register interchangeably.

```
(LAP NEXT SUBR)
NEXT     (PUSH P 1)              push L on the stack
         (JUMPE 1 TAG1)          jump to TAG1 if L is NIL
         (HRRZ@ 1 1)             load register 1 with CDR(L)
         (JUMPE 1 TAG1)          jump to TAG1 if CDR(L) is NIL
TAG2     (HLRZ@ 1 0 P)           load register 1 with CAR(L)
         (CAME 1 2)              skip if CAR(L) is EQ to X
         (JRST 0 TAG4)           jump to TAG4
         (HRRZ@ 1 0 P)           load register 1 with CDR(L)
         (HLRZ@ 1 1)             load register 1 with CAR(CDR(L))
         (JRST 0 TAG1)           jump to TAG1
TAG4     (HRRZ@ 1 0 P)           load register 1 with CDR(L)
PC12     (CALL 2 (E NEXT))       call NEXT(CDR(L),X)
TAG1     (SUB P (C 0 0 1 1))     undo the first push operation
PC14     (POPJ P)                return
         NIL
```

Figure 1.5

The optimizations performed in the transition from Figure 1.4 to Figure 1.5 have the effect of reducing the size of the program from 18 to 14 instructions†. However, we could do better in terms of speeding up the inner loop of the function. By the term inner loop we mean the execution path when the values of the conditions are such that recursion is in order (i.e. L and CDR(L) are not NIL and CAR(L) is not EQ to X). The size of the inner loop has been decreased from 13 (37.88μs) to 11 (31.44μs) instructions. In addition, we have managed to optimize the base case although this is of limited value since this code is only executed once at function exit. By using accumulators other than 1 and 2 we can achieve a further reduction in the size of the inner loop. In our example this will free us from using the stack, recomputing CDR(L) one of two times (common subexpression elimination), and enable the removal of the instructions at locations NEXT and TAG1. The purge of the latter renders unnecessary the recursion at PC12 which can now be replaced by iteration. We also relabel PC14 with TAG1. At this point we have the encoding given in Figure 1.6.

† In the discussion we also give in parentheses the running time of the inner loop in microseconds (denoted by μs). These values should be viewed with caution since they are somewhat dependent on the memory configuration of the host computer (e.g. memory access time) and whether execution occurs in time-shared or dedicated modes. For our estimates we assume time-shared mode since the effect of reducing the number of memory references is more marked due to the added expense incurred by relocation arithmetic whenever memory fetches occur. Instruction times are given in Appendix 7. Note that for some instructions two times are given. The first is the basic execution time of the instruction while the second indicates the speed when the effective address for a memory fetch is one of the accumulators. We also make the following assumptions: Indexing takes 0.28 microseconds. When function calls occur we only give the execution time of the instruction performing the call. Thus if a CONS operation is performed, then the time that is counted is only that which is required for executing the linking operation. We assume that the UUOs CALL and JCALL are of the same speed as the PUSHJ and JRST instructions respectively. This is not unreasonable since we are primarily interested in compiled code in which case CALL and JCALL are converted to PUSHJ and JRST respectively.

```
(LAP NEXT SUBR)
NEXT     (JUMPE 1 TAG1)           jump to TAG1 if L is NIL
         (HRRZ@ 3 1)              load register 3 with CDR(L)
PC3      (JUMPN 3 TAG2)           jump to TAG2 if CDR(L) is not NIL
         (MOVEI 1 (QUOTE NIL))    load register 1 with NIL
PC5      (JRST 0 TAG1)            jump to TAG1
TAG2     (HLRZ@ 4 1)              load register 4 with CAR(L)
         (CAME 4 2)               skip if CAR(L) is EQ to X
         (JRST 0 TAG4)            jump to TAG4
         (HLRZ@ 1 3)              load register 1 with CAR(CDR(L))
PC10     (JRST 0 TAG1)            jump to TAG1
TAG4     (MOVE 1 3)               load register 1 with CDR(L)
         (JRST 0 NEXT)            call NEXT(CDR(L),X)
TAG1     (POPJ P)                 return
         NIL
```

Figure 1.6

Notice that the size of the function has only decreased by one instruction. However, the size of the inner loop has been decreased from 11 (31.44µs) to 8 (18.12µs) instructions. The reason the program size did not decrease any further is that in order to speed up the inner loop we avoided the use of a stack. This meant that CDR(L) could not reside in register 1 since we are no longer saving the value of L on the stack. Thus we had to restore our previous sequence of instructions for testing OR(NULL(L),NULL(CDR(L))). This is done with no qualms since the end result is still a decrease in execution time of the case when L is not NIL and CDR(L) is NIL although the number of instructions remains the same (this is because stack manipulating instructions take more time than other instructions). The latter case can still be speeded up by one instruction by noting that the unconditional jump at location PC5 to the function exit at TAG1 is not necessary and can be replaced by a function exit at location PC5. Similarly, we may replace the unconditional jump at location PC10 by a function exit. Replacing the target of the conditional branch operation at location NEXT by PC5 renders the function exit at TAG1 inaccessible. Therefore, we remove this instruction. At this point we have the encoding given in Figure 1.7.

```
(LAP NEXT SUBR)
NEXT     (JUMPE 1 PC5)            jump to PC5 if L is NIL
         (HRRZ@ 3 1)              load register 3 with CDR(L)
         (JUMPN 3 TAG2)           jump to TAG2 if CDR(L) is not NIL
         (MOVEI 1 (QUOTE NIL))    load register 1 with NIL
PC5      (POPJ P)                 return
TAG2     (HLRZ@ 4 1)              load register 4 with CAR(L)
PC7      (CAME 4 2)               skip if CAR(L) is EQ to X
PC8      (JRST 0 TAG4)            jump to TAG4
         (HLRZ@ 1 3)              load register 1 with CAR(CDR(L))
         (POPJ P)                 return
TAG4     (MOVE 1 3)               load register 3 with CDR(L)
         (JRST 0 NEXT)            compute NEXT(CDR(L),X)
         NIL
```

Figure 1.7

The present encoding of the function is 12 instructions long with an inner loop of length 8

(18.12μs). We can decrease the length of the inner loop by one instruction (i.e. to 16.85μs) by changing the sense of the test at location PC7. Currently, we jump to TAG4 when CAR(L) is not EQ to X. This causes an added burden in the case of recursion since if the atom bound to X appears as the n'th item in the list, then we need n−1 iterations for its detection. Therefore the instruction at location PC8 is executed n−1 times. Instead we feel that when conditional tests must skip, the case resulting in recursion should be the one causing the condition to skip in the true case. The result is that the terminating case of the function (i.e. we are ready to return the next element) will take one extra instruction. Thus we have gained speed unless the desired atom is the first element in the list in which case the resulting function will take one instruction longer. Another optimization is in the number of memory fetches performed by the function. Indirect addressing (@) is often used when indexing would be cheaper. In fact this is the case in all of the instructions involving indirect addressing in this example. At this point we have the encoding in Figure 1.8.

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 PC5)           jump to PC5 if L is NIL
        (HRRZ 3 0 1)            load register 3 with CDR(L)
PC3     (JUMPN 3 TAG2)          jump to TAG2 if CDR(L) is not NIL
PC4     (MOVEI 1 (QUOTE NIL))   load register 1 with NIL
PC5     (POPJ P)                return
TAG2    (HLRZ 4 0 1)            load register 4 with CAR(L)
        (CAMN 4 2)              skip if CAR(L) is not EQ to X
        (JRST 0 TAG4)           jump to TAG4
        (MOVE 1 3)              load register 1 with CDR(L)
PC10    (JRST 0 NEXT)           compute NEXT(CDR(L),X)
TAG4    (HLRZ 1 0 3)            load register 1 with CAR(CDR(L))
        (POPJ P)                return
        NIL
```

Figure 1.8

The total space occupied by the encoding in Figure 1.8 can be reduced from 12 instructions to 11 instructions by observing that function exit could be accomplished by one instruction. Such an optimization is possible when we note that registers may be loaded with values and control passed to the next instruction via a skip. This is the effect of a SKIPA instruction with a non−zero index field. In case the desired value is zero (i.e. NIL) there is an even more efficient way of accomplishing this result. A TDZA instruction has the effect of using the contents of the address designated by the address field as a mask to zero all corresponding bits in the accumulator designated by the accumulator field (the efficiency is derived from having one less memory reference). Once this operation is performed, control is passed to the next instruction via a skip. Thus when the accumulator and address fields of an instruction are identical, the said accumulator is set to zero. Using this instruction we can accomplish the sequence of two instructions at PC4 and PC5 by means of a TDZA instruction between PC10 and TAG4 in Figure 1.8. We also reverse the sense of the test at location PC3 and cause the branch to proceed to the TDZA instruction. Furthermore, the first instruction must now branch to the last instruction if L is NIL. At this point we have the encoding in Figure 1.9.

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 PC11)        jump to PC11 if L is NIL
        (HRRZ 3 0 1)          load register 3 with CDR(L)
        (JUMPE 3 TAG2)        jump to TAG2 if CDR(L) is NIL
        (HLRZ 4 0 1)          load register 4 with CAR(L)
        (CAMN 4 2)            skip if CAR(L) is not EQ to X
        (JRST 0 TAG4)         jump to TAG4
PC7     (MOVE 1 3)            load register 1 with CDR(L)
        (JRST 0 NEXT)         compute NEXT(CDR(L),X)
TAG2    (TDZA 1 1)            set register 1 to NIL and skip
TAG4    (HLRZ 1 0 3)          load register 1 with CAR(CDR(L))
PC11    (POPJ P)              return
        NIL
```

Figure 1.9

We are now at the end of the road as far as the encoding given in Figure 1.9. The main stumbling block to any further decrease in the length of the inner loop is the loading of register 1 with CDR(L) at location PC7 in Figure 1.9 so recursion can proceed in a valid manner. The problem is that we would like to load CDR(L) directly into register 1 when it is computed, yet we cannot destroy the previous contents of register 1 at that time since we may need it in case CDR(L) is not NIL for the computation of CAR(L). CAR(L) is not precomputed since in case CDR(L) is NIL, the function definition does not call for its computation. However, all is not lost. Recall, our earlier statement about the capability of loading a register and skipping the next instruction. Well, we have the same capability to test the value to be loaded into the register. Thus we can save the contents of register 1 in another register while at the same time testing its value. This is accomplished by use of a SKIPN instruction at the location NEXT which will load register 3 with L while testing if L is NIL. Thus we no longer need the TDZA operation. In effect we are undoing the work performed in going from Figure 1.8 to Figure 1.9. The key property of the skip and test optimization is that it enables us to proceed to recursion as soon as it is determined that CAR(L) is not EQ to X. Thus we may once again reverse the sense of the test at location PC7. Furthermore, we have managed to reduce the number of instructions necessary in the case that neither L nor CDR(L) are NIL and CAR(L) is EQ to X. At this point we have the encoding in Figure 1.10.

```
(LAP NEXT SUBR)
NEXT    (SKIPN 3 1)           load register 3 with L and
                              skip if not NIL
        (POPJ P)              return NIL
PC3     (HRRZ 1 0 1)          load register 1 with CDR(L)
        (JUMPE 1 TAG1)        jump to TAG1 if CDR(L) is NIL
        (HLRZ 4 0 3)          load register 4 with CAR(L)
        (CAME 4 2)            skip if CAR(L) is EQ to X
PC7     (JRST 0 NEXT)         compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)          load register 1 with CAR(CDR(L))
TAG1    (POPJ P)              return
        NIL
```

Figure 1.10

The length of the inner loop has now been reduced to 6 instructions (13.36μs). Moreover, the entire encoding has been reduced to a length of 9. Further reduction in the length of the inner loop can be achieved by noting that once CDR(L) has been found not to be NIL, all subsequent recursive calls need not perform the test of L against NIL. This will be referred to as *loop shortcutting* in the sequel. To a SAIL[VanLehn73] programmer this concept is somewhat analogous to the similarity between a FOR loop and a DO UNTIL loop. It would seem that we could simply jump to location PC3 from PC7 rather than to the start of the program. This is shown in Figure 1.11.

```
(LAP NEXT SUBR)
NEXT    (SKIPN 3 1)          load register 3 with L and
                             skip if not NIL
        (POPJ P)             return NIL
PC3     (HRRZ 1 0 1)         load register 1 with CDR(L)
PC4     (JUMPE 1 TAG1)       jump to TAG1 if CDR(L) is NIL
PC5     (HLRZ 4 0 3)         load register 4 with CAR(L)
        (CAME 4 2)           skip if CAR(L) is EQ to X
PC7     (JRST 0 PC3)         compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)         load register 1 with CAR(CDR(L))
TAG1    (POPJ P)             return
        NIL
```

Figure 1.11

Upon a cursory glance it would seem that we are through with an inner loop of length 5 (11.09μs). However, there is an error in the program. Once recursion occurs, register 3, when referenced at location PC5, will not have the current binding of L. Thus we see that we could not bypass the loading of register 3 with L at location NEXT despite the fact that the condition was redundant. However, we may bypass the test of CDR(L) being NIL at the first instruction. Therefore, interchange the first two instructions with the instruction at location PC4 and the desired function is obtained as shown in Figure 1.12. Note that the inner loop is still of length 5 (11.57μs).

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 TAG1)       jump to TAG1 if L is NIL
PC2     (HRRZ 1 0 1)         load register 1 with CDR(L)
PC3     (SKIPN 3 1)          load register 3 with CDR(L) and
                             skip if not NIL
        (POPJ P)             return NIL
PC5     (HLRZ 4 0 3)         load register 4 with CAR(L)
PC6     (CAME 4 2)           skip if CAR(L) is EQ to X
        (JRST 0 PC2)         compute NEXT(CDR(L),X)
        (HLRZ 1 0 1)         load register 1 with CAR(CDR(L))
TAG1    (POPJ P)             return
        NIL
```

Figure 1.12

Once again we have erred. This time we have managed to destroy L before encountering the last location at which it is needed – i.e. PC5. It should be clear that we must not destroy the value of L before CAR(L) is computed. However, by inserting a temporary storage operation at location

PC2, we may still bypass the testing of the nullness of CDR(L). Moreover, we no longer need a test and load operation at location PC3 – i.e. a test is sufficient. An additional minor optimization is the use of an immediate instruction accompanied by indexing at location PC6 thereby avoiding a memory access. At this point we have the encoding shown in Figure 1.13.

```
(LAP NEXT SUBR)
NEXT     (JUMPE 1 TAG1)             jump to TAG1 if L is NIL
PC2      (MOVE 3 1)                 load register 3 with L
         (HRRZ 1 0 1)               load register 1 with CDR(L)
         (JUMPE 1 TAG1)             jump to TAG1 if CDR(L) is NIL
         (HLRZ 4 0 3)               load register 4 with CAR(L)
         (CAIE 4 0 2)               skip if CAR(L) is EQ to X
         (JRST 0 PC2)               compute NEXT(CDR(L),X)
         (HLRZ 1 0 1)               load register 1 with CAR(CDR(L))
TAG1     (POPJ P)                   return
         NIL
```

Figure 1.13

When performing loop shortcutting we must make sure that all locations are set to their proper values. This criterion is satisfied by the encoding given in Figure 1.13. The length of the inner loop has been reduced from 13 (37.88μs) to 6 (12.84μs) while the overall length of the program has been reduced from 18 to 9 instructions. In fact the length of the inner loop can be further decreased by one instruction if we are willing to accept an increase of two instructions in the total space occupied by the function. This revision is given in Figure 1.14 and was pointed out by Donald Knuth.

```
(LAP NEXT SUBR)
         (JUMPN 1 PC6)              jump to PC6 if L is not NIL
         (POPJ P)                   return
PC3      (HLRZ 4 0 3)               load register 4 with CAR(L)
         (CAIN 4 0 2)               skip if CAR(L) is not EQ to X
         (JRST 0 TAG2)              jump to TAG2 if CAR(L) is EQ to X
PC6      (MOVE 3 1)                 load register 3 with L
         (HRRZ 1 0 1)               load register 1 with CDR(L)
         (JUMPN 1 PC3)              jump to PC3 if CDR(L) is not NIL
         (POPJ P)                   return
TAG2     (HLRZ 1 0 1)               load register 1 with CAR(CDR(L))
         (POPJ P)                   return
         NIL
```

Figure 1.14

The encoding given in Figure 1.14 has an inner loop of length 5 (11.37μs). This is about as good an encoding as we can get for this formulation of the NEXT function because six operations are required for each iteration – although we have managed to reduce this requirement to four operations by noting the redundancy of the test of the nullness of L when recursion occurs, and accomplishing a test operation simultaneously with the iteration step. These operations are the computation of CDR(L), CAR(L), the comparison of CDR(L) with NIL, the comparison of CAR(L) with X, and the iteration step. Thus the length of the inner loop cannot be further reduced without changing the algorithm. This statement is crucial to the remainder of the thesis and we shall have more to say about it at a later point. In fact, one further optimization proposed by Donald Knuth is shown in Figure 1.15.

```
(LAP NEXT SUBR)
        (JUMPN 1 PC5)           jump to PC5 if L is not NIL
        (POPJ P)                return
PC3     (CAIN 4 0 2)            skip if CAR(L) is not EQ to X
        (JRST 0 TAG2)           jump to TAG2 if CAR(L) is EQ to X
PC5     (HLRZ 4 0 1)            load register 4 with CAR(L)
        (HRRZ 1 0 1)            load register 1 with CDR(L)
        (JUMPN 1 PC3)           jump to PC3 if CDR(L) is not NIL
        (POPJ P)                return
TAG2    (HLRZ 1 0 1)            load register 1 with CAR(CDR(L))
        (POPJ P)                return
        NIL
```

Figure 1.15

This encoding is 10 instructions long and has an inner loop of length 4 (9.28μs). Unfortunately, Figure 1.15 encodes a slightly different algorithm. The trouble is that when CDR(L) is known to be NIL our original algorithm does not specify that CAR(L) is to be computed. In other words, CAR(L) has been treated as a common subexpression. In Chapter 5 we shall shed more light on the issue of when we would allow such superfluous computations.

An alternate formulation of the NEXT algorithm is one which recognizes that the test for CDR(L) being not NIL is only necessary prior to the CADR(L) operation. This is because L is assumed to be a list and if it is not NIL, then it cannot be atomic. Therefore the nullness of CDR(L) can be checked when processing the recursive call. The new algorithm is given in Figure 1.16.

```
NEXT(L,X) ≡ if NULL(L) then NIL
            else if CAR(L) EQ X then
                if NULL(CDR(L)) then NIL
                else CADR(L)
            else NEXT(CDR(L),X)
```

Figure 1.16 - Algorithm 2 for NEXT

The intermediate form representation of the new algorithm, known as Algorithm 2, is given in Figure 1.17. Once again, the symbolic and numeric representations are denoted by the left and right sides respectively of the Figure. Note the use of F instead of NIL.

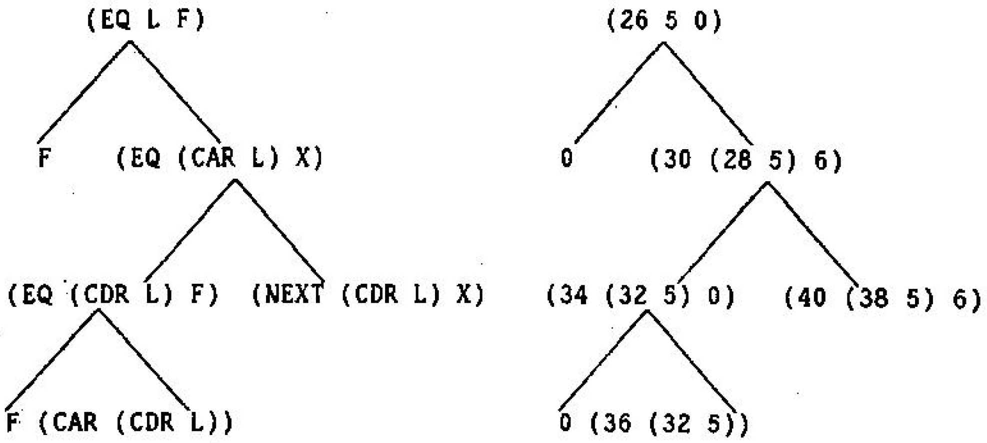


Figure 1.17 - Intermediate Form of Figure 1.16

It is clear that algorithm 2 is better in the higher level because it reduces the number of operations necessary prior to the performance of recursion. This has the effect of reducing the potential length of the inner loop. Our analysis for this algorithm will not go into as great a detail as for Algorithm 1. The encodings that we present serve to demonstrate optimization techniques different from those used in Algorithm 1. The LAP encoding produced by the LISP 1.6 compiler is shown in Figure 1.18.

```
(LAP NEXT SUBR)
PC1      (PUSH P 1)              push L on the stack
         (PUSH P 2)              push X on the stack
         (JUMPE 1 TAG1)          jump to TAG1 if L is NIL
         (HLRZ@ 1 1)             load register 1 with CAR(L)
PC5      (CAME 1 2)              skip if CAR(L) EQ X
         (JRST 0 TAG2)           jump to TAG2
PC7      (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
PC8      (JUMPE 1 TAG3)          jump to TAG3 if CDR(L) is NIL
PC9      (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
         (HLRZ@ 1 1)             load register 1 with CAR(CDR(L))
TAG3     (JRST 0 TAG1)           jump to TAG1
TAG2     (MOVE 2 0 P)            load register 2 with X
PC13     (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
         (CALL 2 (E NEXT))       compute NEXT(CDR(L),X)
TAG1     (SUB P (C 0 0 2 2))     undo the first two push operations
         (POPJ P)                return
         NIL
```

Figure 1.18

This encoding abounds with unnecessary operations. There is no need to save X on the stack since register 2 is never stored into prior to being referenced. Register 1 already contains CDR(L) at location PC9 and thus this instruction is unnecessary. The same line of reasoning holds for register 2 and X at location TAG2. Since CAR(L) is not needed in register 1 (i.e. for subsequent operations), we may place it in another register, say 3. Thus there is no need to save L on the stack since register 1 will not be stored into prior to being referenced for the computation of CDR(L). Therefore, the instructions at locations PC1 and TAG1 may be removed. Since CDR(L) is computed whether or not CAR(L) is EQ to X, we can factor its computation to a point before PC5. This has the effect of rendering the two operations at locations PC7 and PC13 unnecessary and they can be removed. The conditional jump at location PC8 has as its target address an unconditional jump instruction to location TAG1. Thus the destination of the conditional jump at location PC8 is changed to TAG1. Once again no indirect addressing is needed. At this point we have the encoding shown in Figure 1.19.

```
(LAP NEXT SUBR)
        (JUMPE 1 TAG1)           jump to TAG1 if L is NIL
PC2     (HLRZ 3 0 1)             load register 3 with CAR(L)
        (HRRZ 1 0 1)             load register 1 with CDR(L)
        (CAME 3 2)               skip if CAR(L) EQ X
PC5     (JRST 0 TAG2)            jump to TAG2
        (JUMPE 1 TAG1)           jump to TAG1 if CDR(L) is NIL
        (HLRZ 1 0 1)             load register 1 with CAR(CDR(L))
PC8     (JRST 0 TAG1)            jump to TAG1
TAG2    (CALL 2 (E NEXT))        compute NEXT(CDR(L),X)
TAG1    (POPJ P)                 return
        NIL
```

Figure 1.19

The optimizations performed in the transition from Figure 1.18 to Figure 1.19 have the effect of reducing the size of the program from 16 to 10 instructions. Even more significant is the fact that the length of the inner loop has been reduced from 11 (32.26μs) to 7 (17.38μs) instructions. Note that there is no need for recursion – i.e. the call to NEXT can be replaced by an unconditional branch to the start of the program. Therefore, the unconditional jump to TAG2 at PC5 can be replaced by a jump to the start of the program. But now the operation at location TAG2 is inaccessible and it can be removed. The latter causes the unconditional jump at PC8 to be unnecessary and it too can be removed. At this point, we have the encoding shown in Figure 1.20.

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 TAG1)           jump to TAG1 if L is NIL
PC2     (HLRZ 3 0 1)             load register 3 with CAR(L)
        (HRRZ 1 0 1)             load register 1 with CDR(L)
PC4     (CAME 3 2)               skip if CAR(L) EQ X
PC5     (JRST 0 NEXT)            compute NEXT(CDR(L),X)
PC6     (JUMPE 1 TAG1)           jump to TAG1 if CDR(L) is NIL
        (HLRZ 1 0 1)             load register 1 with CAR(CDR(L))
TAG1    (POPJ P)                 return
        NIL
```

Figure 1.20

Notice that the lengths of the program and the inner loop have been reduced from 10 and 7 (17.38μs) to 8 and 5 (11.09μs) instructions respectively. Even further reduction in the size of the inner loop can be achieved. Recall our earlier comment about the redundancy of the test of nullness of L once recursion has started. This is not the case here; however, the concept of bypassing the start of the program, which we called loop shortcutting, is relevant. At location PC5 we branch unconditionally to a conditional branch. We could perform the test for the nullness of L at location PC5 with the sense of the condition reversed. In this case, the condition at PC6 is true and we will proceed to the function exit with the right result since register 1 will contain NIL[†]. An additional minor optimization is the use of an immediate instruction accompanied by indexing in place of the memory access at location PC4. At this point we have the encoding shown in Figure 1.21.

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 TAG1)       jump to TAG1 if L is NIL
LOOP    (HLRZ 3 0 1)         load register 3 with CAR(L)
        (HRRZ 1 0 1)         load register 1 with CDR(L)
        (CAIE 3 0 2)         skip if CAR(L) EQ X
PC5     (JUMPN 1 LOOP)       if CDR(L) is not NIL then
                             compute NEXT(CDR(L),X)
PC6     (JUMPE 1 TAG1)       jump to TAG1 if CDR(L) is NIL
        (HLRZ 1 0 1)         load register 1 with CAR(CDR(L))
TAG1    (POPJ P)             return
        NIL
```

Figure 1.21

We are now through. The intermediate form corresponding to the LAP encoding in Figure 1.21 is given in Figure 1.22. Once again, note the two distinct representations. Our last optimization has resulted in a decrease of the length of the inner loop from 5 (11.09μs) to 4 (9.28μs) instructions. In summary, the length of the original encoding has been reduced from 16 to 8 instructions. More importantly, the length of the inner loop has decreased from 11 (32.26μs) to 4 (9.28μs) instructions. The last encoding can be considered optimal for the following reason. The NEXT function formulated by Figure 1.16 requires the computation of CAR(L), CDR(L), the testing of CAR(L) EQ to X, and the nullness of L, and the iteration step. All in all these comprise five operations. At times a test may be combined with another non–test operation. Since we have two test operations the minimal number of instructions with which we could accomplish our desired computation is three. We have been able to encode the function with four instructions and thus we have almost achieved the lower bound. Further analysis would reveal that on the PDP–10 we have indeed an optimal encoding since actually the only operation that can be simultaneously achieved with a test is a branch and our function only requires one such branch (i.e. the iteration step).

[†] This optimization was pointed out to the author by Steve Savitsky.

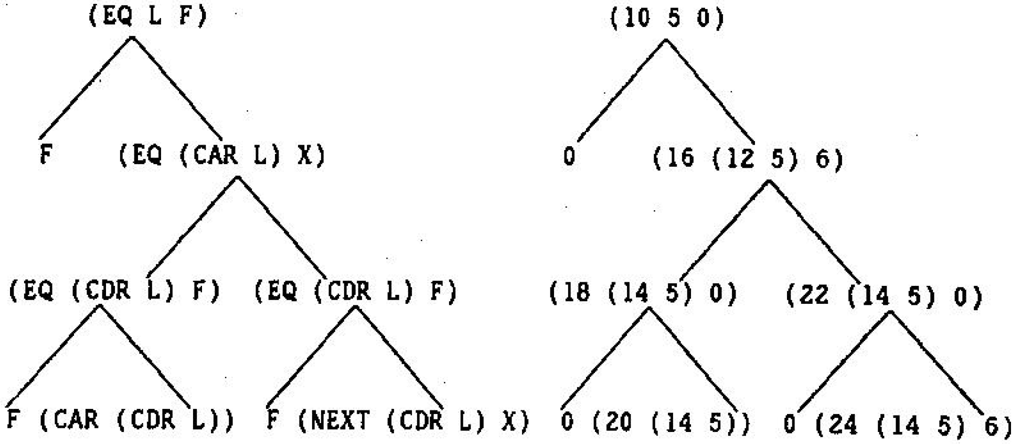


Figure 1.22 - Intermediate Form of Figure 1.21

In summary, we have demonstrated how the function has been optimized step by step. We feel that in the future such refinements could be performed by a postoptimizing program. Notice how for each of the two algorithms we removed the unnecessary steps until the inner loop was reduced to its minimum. When Algorithm 1 was reduced to its minimum, we had to make a change in our original algorithm in order to be able to proceed further. Using our methods we cannot vouch for the equivalence of algorithms 1 and 2†. The proof system in this thesis is designed to prove correct the manipulations performed in the transitions to optimality within each of the two algorithms but not the equivalence of the two algorithms. This is an important point and we hope that the example has demonstrated what we mean by equivalence.

As a second example consider the old standby of correctness work, REVERSE. The function takes one argument, a list L, and returns as its result the reverse of the top elements of its argument. For example, REVERSE applied to (A B C) yields (C B A); similarly, REVERSE((A (B C D) E)) would yield (E (B C D) A). One formulation of this function in meta-LISP is given in Figure 1.23. *APPEND is a function whose arguments are lists that are concatenated to form the result of the function. LIST is a function of an arbitrary number of arguments that returns a list containing these arguments.

```
REVERSE(L) ≡ if NULL(L) then NIL
             else *APPEND(REVERSE(CDR(L)),LIST(CAR(L)))
```

Figure 1.23 - Definition of REVERSE

† In fact the two algorithms are equivalent as can be shown indirectly by noting that the encodings given in Figures 1.15 and 1.21 will have the same rederived forms (more precisely, the two rederived forms can be shown to be equivalent) using our proof methods. This is subject to showing that an unnecessary operation (CAR(L) in Figure 1.15) can be performed safely. This point is discussed in Chapter 5. However, the use of indirection is not very general since it relies on the existence of assembly language encodings that can be shown to be equivalent to different algorithms (in a higher level language sense). In other words we are demonstrating the equivalence by using assembly language encodings rather than the higher level language encodings.

The LAP encoding generated by the Stanford LISP 1.6 compiler is shown in Figure 1.24. NCONS is a function of one argument which is equivalent to a CONS of its argument with NIL. NCONS is known not to destroy any of the registers except for register 1. The encoding and the inner loop are both 12 instructions long.

```
(LAP REVERSE SUBR)
        (PUSH P 1)              push L on the stack
PC2     (JUMPE 1 TAG1)          jump to TAG1 if L is NIL
        (HRRZ@ 1 0 P)           load register 1 with CDR(L)
        (CALL 1 (E REVERSE))    compute REVERSE(CDR(L))
PC5     (PUSH P 1)              push REVERSE(CDR(L)) on the stack
        (HLRZ@ 1 -1 P)          load register 1 with CAR(L)
        (CALL 1 (E NCONS))      compute LIST(CAR(L))
        (MOVE 2 1)              load register 2 with LIST(CAR(L))
        (POP P 1)               pop REVERSE(CDR(L)) from the stack
        (CALL 2 (E *APPEND))    compute *APPEND(REVERSE(CDR(L)),
                                                LIST(CAR(L)))
TAG1    (SUB P (C 0 0 1 1))     undo the first push operation
PC12    (POPJ P)                return
        NIL
```

Figure 1.24

The encoding in Figure 1.24 can be improved upon in several ways. In case the test at PC2 is true, then the previous push operation was not necessary. Therefore, interchange the first two instructions and the jump may now proceed to PC12 rather than TAG1. The length of the inner loop may be reduced by noting that *APPEND requires that REVERSE(CDR(L)) be in register 1. Thus since all results are returned in register 1, it would be preferable if the computation destined for register 1 is computed last. This is possible only if it can be proved that no harm can result from rearranging the order of computation of arguments to a function (i.e. no side–effects are possible). In this case the rearranging is feasible. Another operation that is not necessary is the PUSH instruction at location PC5. Instead we note that the value currently occupying the top of the stack is L whose final use occurs in the next instruction. Thus we may recycle the allocated cell on the stack by use of an EXCH which exchanges the contents of a register with a location in memory. This renders the stack pointer adjustment at location TAG1 unnecessary and it is removed. Such optimizations are quite useful for several reasons. First of all, they reduce the overall stack length required by a factor of two when recursion occurs, since an extra cell must be so allocated for each element in the list that is to be reversed. Secondly, when garbage collection† occurs, we must perform what is known as the marking phase which consists of determining all of the accessible cells in the List Structure. This is done by following the chains of all of the active pointers. The active pointers are defined to be contents of locations that may be subsequently referenced by the function. This includes certain accumulators and the stack. Thus reducing the size of the stack may have an important effect on the efficiency of garbage collection since the marking phase is reduced in length and more of the List Structure may be reclaimed. At this point we have the encoding shown in Figure 1.25.

---

† Reclaiming of storage in the List Structure that is no longer accessible. For more details see Chapter 2 or [Knuth68].)

```
(LAP REVERSE SUBR)
PC1     (JUMPE 1 TAG1)          jump to TAG1 if L is NIL
PC2     (PUSH P 1)              push L on the stack
        (HLRZ@ 1 0 P)           load register 1 with CAR(L)
        (CALL 1 (E NCONS))      compute LIST(CAR(L))
PC5     (EXCH 1 0 P)            exchange LIST(CAR(L)) with L
        (HRRZ@ 1 1)             load register 1 with CDR(L)
        (CALL 1 (E REVERSE))    compute REVERSE(CDR(L))
        (POP P 2)               pop LIST(CAR(L)) from the stack
PC9     (CALL 2 (E *APPEND))    compute *APPEND(REVERSE(CDR(L)),
                                                LIST(CAR(L)))
TAG1    (POPJ P)                return
        NIL
```

Figure 1.25

We have only succeeded in decreasing the size of our program by two instructions. Similarly, for the length of the inner loop† (i.e. the speed of the inner loop is reduced from 38.06μs to 32.50μs). More can be achieved by noting that L need not be saved on the stack at location PC2 since NCONS only destroys register 1. Thus we may temporarily save it in another register while LIST(CAR(L)) is being computed. In fact, we will use a technique mentioned earlier which allows a load and skip test to be performed simultaneously. We replace the instruction at PC1 by a SKIPN operation results in function exit if L is NIL. This modification forces the removal of the EXCH operation at location PC5, and its replacement by a PUSH operation. Moreover, there is no longer a need to enter *APPEND via a recursive call. Instead, we may use a JCALL (same as a call but does not place a return address on the stack) instruction at location PC9. Now, the function exit operation at TAG1 is unreachable and may be removed. We may also change all uses of indirect addressing to indexing. At this point we have the encoding shown in Figure 1.26.

```
(LAP REVERSE SUBR)
        (SKIPN 2 1)             load register 2 with L and
                                skip if not NIL
        (POPJ P)                return NIL
        (HLRZ 1 0 1)            load register 1 with CAR(L)
        (CALL 1 (E NCONS))      compute LIST(CAR(L))
        (PUSH P 1)              push LIST(CAR(L)) on the stack
        (HRRZ 1 0 2)            load register 1 with CDR(L)
        (CALL 1 (E REVERSE))    compute REVERSE(CDR(L))
        (POP P 2)               pop LIST(CAR(L)) from the stack
PC9     (JCALL 2 (E *APPEND))   compute *APPEND(REVERSE(CDR(L)),
                                                LIST(CAR(L)))
        NIL
```

Figure 1.26

† For the REVERSE function the actual times of the inner loops that are given only reflect the time spent in the function being optimized. Recall the cautionary remark made earlier and the example of the CONS operation whose only effect on the inner loop is the time required to perform the linking operation. This is not unreasonable since our goal is to optimize a particular function and not necessarily the functions invoked by it.

The encoding in Figure 1.26 results in an inner loop of length 8 (22.91μs) and overall function length of 9. Thus we have once again succeeded in reducing the execution time by a factor close to 2. We don't see any further obvious optimization that can be done for this formulation of the REVERSE function. This can be seen when we consider that the function definition requires six basic operations – i.e. the computation of CAR(L), CDR(L), LIST(CAR(L)), *APPEND, recursion, and the testing of the nullness of L. In addition we must temporarily save and restore the value of one of the arguments to *APPEND while computing the other one.

Greater reductions in space and time requirements can be achieved by changing the algorithm. The change we propose is a general one which is dependent on the schema of the function definition. In our case the schema corresponding to REVERSE fits into a class of schemas given in Figure 1.27 along with their equivalents. The driving force behind such transformations is a desire to replace recursion by iteration via the use of an additional argument as an accumulator to store temporary results. In the figure, a and b denote expressions and ⊛ is an operation. The transformations are applicable to the schemas provided that the ⊛ operation is associative†.

```
f(x) ≡ if p(x) then a      becomes     h(x,y) ≡ if p(x) then y⊛a
       else b⊛f(g(x))                           else h(g(x),y⊛b)

f(x) ≡ if p(x) then a      becomes     h(x,y) ≡ if p(x) then a⊛y
       else f(g(x))⊛b                           else h(g(x),b⊛y)
```

Figure 1.27 - Transformation Schemas

Once the transformation is performed we still have some unfinished business. The newly transformed function serves as an auxiliary function and must be properly activated with an appropriate initial value for the placeholder argument. For these transformations, we need a redefinition of the function f to invoke the function h as shown in Figure 1.28 where id⊛ is the identity element of the ⊛ operation.

```
f(x) ≡ h(x,id⊛)
```

Figure 1.28 - Identity Transformation

The definition of REVERSE given in Figure 1.23 has the same schema as the second transformation given in Figure 1.27 with the following bindings. a and b are bound to the atom NIL and to LIST(CAR(L)) respectively, and g, p, and ⊛ are bound to the functions CDR, NULL, and *APPEND respectively. In addition, we cite in Figure 1.29 a pair of identities which will be used to obtain a more optimal function definition.

```
*APPEND(NIL,y) ≡ y
*APPEND(LIST(CAR(L)),y) ≡ CONS(CAR(L),y)
```

Figure 1.29 - *APPEND Identities

† The associativity requirement was pointed out to the author by Ashok Chandra.

Use of the second transformation in Figure 1.27, the associativity of *APPEND, the identities given in Figure 1.29, and the fact that the identity element corresponding to *APPEND is NIL, yield the familiar function definition given in Figure 1.30. This formulation of the function is quite efficient because there is no longer any need for the *APPEND function. The latter was a drawback of the previous definition in that *APPEND always creates an extra copy of its first argument. Furthermore, the new algorithm may be implemented by iteration instead of recursion.

```
REVERSE1(L) ≡ REVERSE1A(L,NIL)
REVERSE1A(L,RL) ≡ if NULL(L) then RL
                  else REVERSE1A(CDR(L),CONS(CAR(L),RL))
```

Figure 1.30 - Definition of REVERSE with Two Arguments

Note that no matter how we optimize the algorithm, the result of the REVERSE function must be a new list. In other words, we may not reverse the links in the original list as is done in the algorithm given in Figure 1.31. This algorithm employs what has been referred to as "compile time garbage collection" by [Darlington73]. At first, such an approach seems attractive as it does not result in the use of any free storage[†]. However, since list structures are generally shared, reversal of the links in the original list will have a far reaching effect.

```
REVERSE1A(L,RL) ≡ if NULL(L) then RL
                  else REVERSE1A(CDR(L),RPLACD(L,RL))
```

Figure 1.31 - Destructive REVERSE

Our system cannot detect the equivalence between the definitions given in Figures 1.23 and 1.30. Such work is best done on the level of LISP function definitions. One method of proof is to have a library of such valid schema transformations and to apply them in some reasonable manner. Recently, Boyer and Moore have reported [Boyer73] work on a theorem prover for LISP functions. We feel that such equivalences as necessary for REVERSE in Figures 1.23 and 1.30 fall more into that domain. The transformations in Figure 1.27 can be applied to other functions (e.g. factorial) and more than once as seen in the discussion of the Fibonacci function in Chapter 8.

As indicated earlier, use of such transformations as Figure 1.27 may reduce the need for recursion, yet there is an extra cost in terms of memory space involved in initially activating the function. Furthermore, we must not overlook the extra overhead that could result from the additional argument. The extra argument adds instructions to the program for the purpose of saving arguments prior to function calls as well as another argument to compute on each additional recursive call. The encoding generated by the LISP 1.6 compiler is given in Figure 1.32.

[†] Using optimizations similar to those discussed here, the entire function can be encoded using four instructions and an inner loop of three instructions.

```
(LAP REVERSE1A SUBR)
        (PUSH P 1)              push L on the stack
PC2     (PUSH P 2)              push RL on the stack
        (JUMPN 1 TAG2)          jump to TAG2 if L is not NIL
        (MOVE 1 2)              load register 1 with RL
        (JRST 0 TAG1)           jump to TAG1
TAG2    (MOVE 2 0 P)            load register 2 with RL
        (HLRZ@ 1 -1 P)          load register 1 with CAR(L)
        (CALL 2 (E CONS))       compute CONS(CAR(L),RL)
        (MOVE 2 1)              load register 2 with CONS(CAR(L),RL)
        (HRRZ@ 1 -1 P)          load register 1 with CDR(L)
        (CALL 2 (E REVERSE1A))  compute REVERSE1A(CDR(L),
                                                  CONS(CAR(L),RL))
TAG1    (SUB P (C 0 0 2 2))     undo the first two push operations
        (POPJ P)                return
        NIL
```

Figure 1.32

The lengths of the program and the inner loop are 13 and 11 (33.86μs) instructions respectively. The encoding again suffers from a variety of redundant operations. These include the unnecessary saving of RL on the stack at PC2, the loading of register 2 with RL at TAG2 when the register already contains this value, saving the arguments on the stack prior to testing whether or not the saving is necessary, and other problems that can be remedied using techniques similar to those used earlier. The main drawback of the encoding in Figure 1.32 is the need to perform many data moving operations to make sure items are in the proper locations for recursion to occur. This stems in part from the fact that the value that is to be returned as the result of the function is in the second register. This can be alleviated in two equivalent ways. One way is to redefine the function with the position of the arguments reversed (i.e. L becomes the second argument and RL becomes the first argument as shown in Figure 1.33). The alternative is simply to interchange the arguments in the first two registers and then to perform recursion (actually we can convert the recursive call to iteration).

```
REVERSE1(L) ≡ REVERSE1A(NIL,L)
REVERSE1A(RL,L) ≡ if NULL(L) then RL
                  else REVERSE1A(CONS(CAR(L),RL),CDR(L))
```

Figure 1.33 - Definition of REVERSE with Argument Positions Reversed

We choose to illustrate this transformation by use of the first method. Note that the algorithm has now changed and our system will not be able to recognize the equivalence of the algorithms in Figures 1.30 and 1.33. The LAP encoding produced by the LISP 1.6 compiler is given in Figure 1.34.

```
(LAP REVERSE1A SUBR)
PC1     (PUSH P 1)                  push RL on the stack
        (PUSH P 2)                  push L on the stack
PC3     (JUMPN 2 TAG2)              jump to TAG2 if L is not NIL
PC4     (JRST 0 TAG1)               jump to TAG1
TAG2    (MOVE 2 -1 P)               load register 2 with RL
PC6     (HLRZ@ 1 0 P)               load register 1 with CAR(L)
        (CALL 2 (E CONS))           compute CONS(CAR(L),RL)
        (HRRZ@ 2 0 P)               load register 2 with CDR(L)
PC9     (CALL 2 (E REVERSE1A))      compute REVERSE1A(CONS(CAR(L),RL),
                                                      CDR(L))
TAG1    (SUB P (C 0 0 2 2))         undo the first two push operations
PC11    (POPJ P)                    return
        NIL
```

Figure 1.34

The length of the function and its inner loop are 11 and 10 (31.77μs) instructions respectively. Several optimizations come to mind. The PUSH operation at location PC1 is unnecessary. Similarly, by using an XCONS operation, there is no need to load register 2 with RL at TAG2. The latter implies that register 2 be loaded with CAR(L) at location PC6. The pair of branch instructions at locations PC3 and PC4 can be placed before the PUSH operation at PC1. Moreover, this pair of instructions can be replaced by a JUMPE operation to PC11. The recursion at location PC9 can be replaced by iteration provided that the stack pointer is adjusted first. As was done for the example NEXT, the iterative jump may be replaced by a test having the reverse sense of that performed at location PC3 in Figure 1.34. In other words we will once again bypass the start of the program. At this point we have the encoding given in Figure 1.35.

```
(LAP REVERSE1A SUBR)
        (JUMPE 2 TAG1)              jump to TAG1 if L is NIL
REV     (PUSH P 2)                  push L on the stack
        (HLRZ 2 0 2)                load register 2 with CAR(L)
        (CALL 2 (E XCONS))          compute CONS(CAR(L),RL)
        (HRRZ@ 2 0 P)               load register 2 with CDR(L)
PC6     (SUB P (C 0 0 1 1))         undo the first push operation
        (JUMPN 2 REV)               if CDR(L) is not NIL then compute
                                    REVERSE1A(CONS(CAR(L),RL),CDR(L))
TAG1    (POPJ P)                    return
        NIL
```

Figure 1.35

The length of the function and its inner loop are 8 and 6 (17.92μs) instructions respectively. This can be improved upon by noting that the PUSH operation at location REV has the effect of recycling a stack location which was just released at PC6. The length of the inner loop could be decreased by two instructions if we would place the value of CDR(L) on the stack as well as in register 2. This would mean that the PUSH operation at REV could be bypassed. Moreover, the stack adjustment operation performed at PC6 would be moved out of the inner loop. One possible way of achieving this effect is to use a HRRZS instruction which stores its result in the register specified by the accumulator field and in the location addressed by the effective address of the instruction. This would result in the encoding given in Figure 1.36.

```
(LAP REVERSE1A SUBR)
        (JUMPE 2 TAG1)          jump to TAG1 if L is NIL
PC2     (PUSH P 2)              push L on the stack
REV     (HLRZ 2 0 2)            load register 2 with CAR(L)
        (CALL 2 (E XCONS))      compute CONS(CAR(L),RL)
PC5     (HRRZS@ 2 0 P)          load register 2 and the top of
                                the stack with CDR(L)
PC6     (JUMPN 2 REV)           if CDR(L) is not NIL then compute
                                REVERSE1A(CONS(CAR(L),RL),CDR(L))
PC7     (SUB P (C 0 0 1 1))     undo the first push operation
TAG1    (POPJ P)                return
        NIL
```

Figure 1.36

At a first glance it seems that we have succeeded in reducing the length of the inner loop to four (11.88μs) instructions. However, use of the equivalence proving system finds that we have erred. Unfortunately, the HRRZS instruction places its result back in the location designated by the effective address. Thus instead of having CDR(L) in the location on top of the stack, we have succeeded in changing the contents of the location pointed at by L from CAR(L) and CDR(L) in the left and right halves respectively to NIL and CDR(L) in the left and right halves respectively. Again we see the potential usefulness of the equivalence proving system. We have to revert to our previous means of computing CDR(L); and if we wish to recycle the stack location, then we have to store the value back on the stack via a MOVEM operation between locations PC5 and PC6 in the encoding given in Figure 1.36. We can still improve on this encoding by recalling that the CONS (and XCONS) operations only destroy the contents of registers 1 and 2. Thus instead of using a location on the stack to temporarily store the value of L, we can use another accumulator, say 3. In this case, the PUSH operation at PC2 is no longer necessary and likewise for the stack pointer adjustment at PC7. In fact, we merely need to initialize register 3 with the value of L and then iterate with CDR(L) placed in register 3. Recalling our encoding of NEXT in Figure 1.10, we may achieve this while simultaneously testing the nullness of L with a SKIPN instruction. At this point we will have the encoding given in Figure 1.37.

```
(LAP REVERSE1A SUBR)
        (SKIPN 3 2)             load register 3 with L and
                                skip if not NIL
        (POPJ P)                return NIL
REV     (HLRZ 2 0 3)            load register 2 with CAR(L)
        (CALL 2 (E XCONS))      compute CONS(CAR(L),RL)
        (HRRZ 3 0 3)            load register 3 with CDR(L)
        (JUMPN 3 REV)           if CDR(L) is not NIL then compute
                                REVERSE1A(CONS(CAR(L),RL),CDR(L))
TAG1    (POPJ P)                return
        NIL
```

Figure 1.37

The encoding given in Figure 1.37 is minimal in the following sense. The function definition of REVERSE1A requires the computation of CAR, CDR, CONS, recursion, and a test for the nullness of L. All in all, these comprise five operations. At times a test may be combined with another non–test operation. Since we have four non–test operations and one test operation, the minimal number of instructions that can accomplish our desired computation is four. Therefore our encoding is indeed minimal. In summary, we have succeeded in reducing the length of the program from 13 to 7 instructions. More significantly, the length of the inner loop has been decreased from 11 (33.86μs) to 4 (10.32μs) instructions. When considering the type of operations involved, the factor of optimization is greater than 3 since the number of memory references has been greatly reduced. The latter includes the elimination of stack accessing operations which are inherently slow due to the extra memory reference.

In conclusion, we have seen how a number of optimizations can be performed and recognized by our system. These include rearranging of the order of computation of arguments to a function call, bypassing the start of a program, making use of the results of tests, changing a calling sequence, and others. Hopefully, we have also clarified the meaning of equivalence. In particular, when the basic algorithm does not change, equivalence can be proved. We cannot prove equivalence in cases where changes in the algorithm occur. This was illustrated by the transition from Algorithm 1 to Algorithm 2 for NEXT, and the addition of an argument to REVERSE as well as the rearranging of the parameter positions. For this type of modification it is not impossible to prove equivalence. However, it should be proved on a higher level which does not involve any optimization in the assembly language encoding of an algorithm.

Now that the examples have been presented, we are better able to indicate some of the more difficult problems that arise in proving the correctness of optimizations. First, we must have a model of the computation. We have seen the use of an intermediate form as shown in Figures 1.3, 1.17, and 1.22. This model must be useful for representing both the higher level and lower level formulations of an algorithm. This includes the capability for expressing a temporal relationship between the various components of the computation. Second, a system is necessary for describing the instruction set of a computer. Third, a proof procedure is required for demonstrating the equivalence of representations of the higher level and lower level encodings of the algorithm. This proof must allow for the rearranging of the order of computation†, substitution of equals for equals, as well as loop shortcutting. Loop shortcutting is particularly important as shown by the difference in the intermediate forms corresponding to Algorithm 2 for NEXT which are given in Figures 1.17 and 1.22.

The series of encodings that have been presented have been either proved correct or have had their errors detected by our system. The need for such optimizations should be quite evident from the examples. Specifically, the use of the proof procedure in the process of inner loop length reduction is quite important since such a procedure can aid one in analyzing algorithms to detect where a change in the formulation can lead to a substantial reduction in space and time requirements. The basic process underlying the optimizations that we have shown is one of stepwise refinement. Admittedly, many of the optimizations are of a heuristic nature. However, it is our feeling that the big payoff in optimization is to be derived from such methods. The heuristic nature of such a procedure requires a means of proving the correctness of the various attempts at gaining optimality. The work reported in this thesis provides a framework for this verification process as well as demonstrating its feasibility by presenting an implementation. The interactive nature of the procedure could also be used as a means of debugging. However, the main intent of such a system is an incorporation into an optimizing compiler to serve as a final validation step.

† Important in the handling of computations having side–effects.

CHAPTER 2

CMPLISP

2.A Introduction to LISP

LISP is a symbol manipulation language developed by John McCarthy [McCarthy60] having its roots in recursive function theory and the lambda calculus[Church41]. It is distinguished by a small number of primitive operations, and, most strikingly, by the indistinguishability of the representation of program and data. This duality implies that LISP functions may be both created and executed by other LISP functions.

The basic data structure in LISP is the s–expression. The syntax of s–expressions is given in Figure 2.1 by means of BNF notation[Naur60]. Note that the primary unit is the atom which, practically speaking, denotes identifiers, numbers, and strings as well as the pre–defined distinguished identifiers NIL and T. In the sequel we will often use the terms identifier and atom interchangeably.

```
<s-expression> ::= ( <s-expression> . <s-expression> )
               ::= <atom>
        <atom> ::= <identifier>
               ::= <number>
               ::= <string>
               ::= NIL
               ::= T
```

Figure 2.1 - Syntax of S-expressions

As a programming language, LISP has been in existence for a number of years with a variety of implementations. As originally designed, it is an interpretive language. However, for efficiency considerations a number of attempts have been made to implement a compiled version. Further efficiency considerations have resulted in implementations[Quam72] which, in certain cases, are characterized by yielding different results for identical functions, depending on whether the functions are interpreted or compiled prior to being executed.

In this work we focus our attention on compiled LISP with certain restrictions. Rather than itemize the restrictions, we define CMPLISP (denoting compilable LISP) – a subset and variant of LISP 1.6. In the course of the definition, we do not hesitate to stray into the implementation domain in order to motivate the presence and absence of certain properties of LISP programs. However, it should be clear that no attempt is made to discuss all of the properties of LISP functions and implementations in this exposition. For a more complete description and definition the reader is directed to the references [McCarthy62],[AllenJ74].

CMPLISP often defines functions in terms of their actions, assuming that the data is valid. The meaning of *valid* is further explored in the section containing differences between CMPLISP and other LISPs (see Section 2.C). Some of the differences are in features, and others are of a more fundamental nature. Some inconsistencies present in certain compiled LISP implementations will be discussed, as well as the remedies offered by CMPLISP. The remedies are mainly in the form of providing protection from undesirable side–effects of certain operations. However, it should be noted that CMPLISP has been designed in such a manner that all programs capable of being executed by the CMPLISP system are also executable by the LISP 1.6 system. Moreover, results of CMPLISP functions are not dependent on whether the function is interpreted or compiled prior to being executed.

The ensuing definition of the syntax and semantics of a CMPLISP program uses a descriptive variant of BNF. This definition uses list notation rather than s–expression notation. Figure 2.2 illustrates the conversion process from list notation to s–expression notation by means of examples, while Figure 2.3 gives a more precise definition of the translation procedure. Note the use of NIL as an atom indicating the last atomic element in the s–expression formulation of a list. The rightward pointing arrow in Figure 2.3 indicates that when the reduction on the left is made, the sentence on the right yields the translation. An asterisk (*) on the right indicates that no translation action is to be taken.

```
          () ≡ NIL
         (A) ≡ (A . NIL)
       (A B) ≡ (A . (B . NIL))
   (A (B) C) ≡ (A . ((B . NIL) . (C . NIL)))
```

Figure 2.2 - S-expression and List Notation Equivalents

```
      <list> ::= ( <rest list> → *
 <rest list> ::= ) → NIL
             ::= <atom> <rest list> → ( <atom> . <rest list> )
             ::= <list> <rest list> → ( <list> . <rest list> )
```

Figure 2.3 - Conversion from S-expression to List Notation

2.A1 Functions and Special Forms

A CMPLISP program is a collection of global variables (known as SPECIAL variables) and functions of the following form:

```
( DEFPROP <function name>
  ( LAMBDA <parameter list> <function body sequence> )
  <function type> )
```

<function name> is the name of the function.

<parameter list> is a sequence of zero or more variable names surrounded by parentheses. These variable names act as placeholders for computations and their names do not exist as atoms.

```
<function body sequence> ::= <function body>
                             <function body sequence>
                         ::= <function body>
           <function body> ::= <atom>
                         ::= <non-atomic fbody>
      <non-atomic fbody> ::= <function call>
                         ::= <conditional form>
                         ::= <internal lambda>
                         ::= <prog>
           <function call> ::= ( <function name> <arglist> )
                 <arglist> ::= <function body> <arglist>
                         ::= empty
      <conditional form> ::= ( COND <cond pairs> )
              <cond pairs> ::= ( <function body>
                               <function body sequence> )
                             <cond pairs>
                         ::= empty
       <internal lambda> ::= ( ( LAMBDA ( <pairs> )
                   <pairs> ::= <varname> <pairs> <function body>
                         ::= ) <function body sequence> )
                    <prog> ::= ( PROG <parameter list> <prog body> )
        <parameter list> ::= ( <varlist> )
                 <varlist> ::= <varname> <varlist>
                         ::= empty
               <prog body> ::= <prog statement> <prog body>
                         ::= <prog statement>
          <prog statement> ::= <atom> <non-atomic fbody>
                         ::= <non-atomic fbody>
```

Figure 2.4 – Syntax of LISP

<function type> is one of the following:

1. EXPR – indicates that the function is invoked by the call by value mechanism – i.e. all arguments have been evaluated prior to the invocation of the function. The function may have an arbitrary number of arguments.

2. FEXPR – indicates that the function is invoked by the call by name mechanism – i.e. the argument has not been evaluated prior to the invocation of the function. The function has exactly one argument.

<function body sequence> is a sequence of one or more <function body>s which is defined below (for a more precise description see the BNF given in the box in Figure 2.4).

1. atom or function call of the form (fname arg1 arg2 . . . argn) where argi are elements of <function body>.

2. internal lambda of the form:

```
( ( LAMBDA (var1 var2 . . . varn )
  <function body sequence> )
  <function body of var1_binding>
  <function body of var2_binding>
                 .
                 .
                 .
  <function body of varn_binding> )
```

This construct indicates that var1, var2, . . . , varn (having the same properties as elements of <parameter list>) are to be bound to their respective bindings and are to serve as formal parameters to <function body sequence>. The value of the last <function body> is returned as the result of the form. Its primary purpose is to avoid recomputing common subexpressions as well as to temporarily store results of functions whose values may differ due to side–effects depending on the instance of computation.

3. condition of the form:

```
( COND ( P1 <function body sequence1> )
       ( P2 <function body sequence2> )
                    .
                    .
                    .
       ( Pn <function body sequencen> ) )
```

COND is a special form indicating that P1 through Pn, elements of <function body>, are to be evaluated in order until encountering the first Pi which returns a value not equal to NIL. If no non–NIL Pi is found, then NIL is returned; otherwise, <function body sequencei> is evaluated and the value of the last <function body> in the sequence is returned as the result of the form. Note that often the final Pn is the atom T which is always non–NIL.

4. program feature of the form (PROG <progvar list> <prog body>) . PROG indicates that elements of <progvar list> are to serve as the local variables for <prog body> – a list of atoms interpreted as labels and non–atomic function bodies interpreted as statements. Basically PROG is a procedure in the FORTRAN[ASI66] sense. In addition, PROGs may contain the constructs GO and RETURN which may only appear at the top level of a PROG or in a COND at the top level of a PROG. GO causes the sequence of control within the PROG to be transfered to the next statement following its atomic argument. RETURN causes the PROG to exit with the value of its argument.

PROGs pose a restriction on the occurrence of a label, GO, and RETURN. This can be justified by viewing a PROG as a sequence of function definitions (henceforth referred to as *pseudofunctions*) having <parameter list> and <progvar list> as their local variables. <prog body> is broken up into pseudofunctions as follows. Each COND appearing at the top level of the PROG is a pseudofunction with the label associated with the COND as the function name (if COND is unlabeled, then a unique label is generated). Also, each sequence of prog statements between a COND and a label is a pseudofunction (determination of function names is identical to that proposed for the COND). In addition, in each pseudofunction, except for the last one, each terminal computation that is not a GO or RETURN is replaced by a call to the next sequential pseudofunction with the current bindings of the pseudofunction variables. The PROG can now be replaced by the first pseudofunction which only has <parameter list> as its local variables. Note that in this discussion we have attributed a functional nature to the GO construct. Namely, we interpret it as a FEXPR with the target label as an unevaluated functional parameter. An example of the conversion of REVERSE in PROG notation to a sequence of functions REVERSE, REVERS1, and REVERS2 is given in the box in Figure 2.5[†].

```
(DEFPROP REVERSE (LAMBDA (SLIST)
                  (PROG (RESULT)
                        (SETQ RESULT NIL)
                   TAG1 (COND ((NULL SLIST) (RETURN RESULT)))
                        (SETQ RESULT (CONS (CAR SLIST) RESULT))
                        (SETQ SLIST (CDR SLIST))
                        (GO TAG1)))
 EXPR)

yields:

(DEFPROP REVERSE (LAMBDA (SLIST) (REVERS1 SLIST NIL))
 EXPR)

followed by:

(DEFPROP REVERS1 (LAMBDA (SLIST RESULT)
                  (COND ((NULL SLIST) RESULT)
                        (T (REVERS2 SLIST RESULT))))
 EXPR)

followed by:

(DEFPROP REVERS2 (LAMBDA (SLIST RESULT)
                 (REVERS1 (CDR SLIST) (CONS (CAR SLIST) RESULT)))
 EXPR)
```

Figure 2.5 – Example of PROG Elimination

2.A2 SPECIAL Variables

In our definition of CMPLISP programs we have seen two types of variables, elements of <parameter list> and SPECIAL. The former have been noted to act merely as names associated with certain computations. The latter have the previous property in addition to being atoms. We will soon see that the characterization of a variable as an atom implies certain important properties. The primary reason for the existence of SPECIAL variables is to provide a means for the communication of values across function boundaries and lifetimes.

[†] (SETQ A B) indicates that variable A is assigned the value of B.

2.A3 Atoms

In order that occurrences of atoms with the same symbolic representation (henceforth known as *printname*) have the same internal address, there exists a table associating the internal address of the atom with its description. This table, known as the OBLIST, contains all atoms and SPECIAL variables (not elements of <parameter list>).

Until now we have seen atoms created by the SPECIAL construct and those that exist in the compiled function (henceforth known as the LAP program). The OBLIST is initialized to contain them. All other atoms (such as those appearing in data) are entered in the OBLIST upon their initial encounter if they have not been seen before. The special function GENSYM is yet another means of creating atoms. These atoms are not automatically entered in the OBLIST. Instead, the programmer has at his disposal the functions INTERN and REMOB which are used respectively to enter their unevaluated argument in the OBLIST, if not already there, and to remove their unevaluated argument from the OBLIST. Note that the atoms which are in the OBLIST prior to the start of execution (i.e. immediately after compilation) cannot be removed from the OBLIST.

2.A4 Property Lists

We have mentioned that associated with each atom is a description (known as the *property list*) containing data we may wish to associate with the atom. The property list is organized in terms of pairs of entries where one part of the entry indicates a name (henceforth known as the property) that we wish to associate with the data, and the other part is the actual value of the data. The property list always contains the property PNAME which has as its corresponding value the printname (i.e. symbolic representation) of the atom. All other properties have s–expressions as their values. Another property alluded to previously is VALUE, which is associated with SPECIAL variables and other distinguished atoms (i.e. NIL and T) and contains a pointer to their s–expression values. In fact, whenever an atom is evaluated it is the VALUE property that is obtained.

As in the case of the OBLIST, there exist functions for accessing and modifying elements of the property list. They are defined below and further elaborated upon in Section 2.C where their differences from the LISP 1.6 definition are examined. Note that these functions have s–expressions as both their arguments and results.

GET(IDENTIFIER,VAL): Search the property list of IDENTIFIER looking for the property name VAL. If such a property is found, then return the value associated with it; otherwise NIL is returned.

PUTPROP(IDENTIFIER,VAL,PROPERTY): The function is an EXPR which enters the property name PROPERTY with property value VAL into the property list of IDENTIFIER. If the property name PROPERTY is already in the property list, then the old property value is replaced by the new one; otherwise the new property name PROPERTY and its value VAL are placed on the property list of IDENTIFIER. PUTPROP returns VAL.

DEFPROP(IDENTIFIER,VAL,PROPERTY): The function is identical to PUTPROP except that it does not evaluate its arguments (i.e. a FEXPR) and returns IDENTIFIER as its value. This form, as previously seen, is useful in creating function definitions.

REMPROP(IDENTIFIER,PROPERTY): Remove the property PROPERTY and its value from the property list of IDENTIFIER. REMPROP returns T if such a property was found, and NIL otherwise.

2.A5 Other Functions

QUOTE is a FEXPR whose value is the unevaluated argument. It is primarily used for preventing evaluation. Note that the FEXPR function type relieves, to some extent, (insofar as the function has only one argument) the need for QUOTE since the argument to a FEXPR is not evaluated. Recall that whenever an atom is evaluated, its VALUE property is obtained. The prevention of this is one of the motivations for the QUOTE construct. The only atoms which do not require this mechanism are T and NIL (which is often denoted by F).

EVAL is the heart of the CMPLISP system. It evaluates s–expressions. In some LISP systems there exists a construct named EVALQUOTE which is very much like EVAL except that its arguments are not evaluated. In such systems the user is talking to EVALQUOTE only at the top level of his program while at all other times he is talking to EVAL. This is quite confusing and provides more than an ample excuse for its abandonment.

APPLY is a function of two arguments; a function name, fname, and a list of arguments ARGS. The function stipulates that each s–expression in ARGS is to be evaluated and bound to the corresponding argument of fname and returns as its result the value of fname applied to its arguments.

OR is a function of an arbitrary number of arguments which evaluates them sequentially until one of them has a non–NIL value. The function returns T if such an argument is found and NIL otherwise. Note that the function only evaluates as many arguments as are necessary to establish the desired result.

AND is a function of an arbitrary number of arguments which evaluates them sequentially until one of them has a value of NIL. The function returns NIL if such an argument is found and T otherwise. Note that the function only evaluates as many arguments as are necessary to establish the desired result.

2.B The CMPLISP Environment

In the previous section we have seen a description of the syntax of LISP and its subset CMPLISP. In addition, the semantics of the syntactic constructs were presented along with the basic data structure (i.e., s–expressions). In this section we proceed to expand further on s–expressions in terms of their implementation and functions that access, create, and modify them. Our approach is to describe a data structure relevant to the semantics of CMPLISP. Next a mapping (or implementation) is outlined from CMPLISP and s–expressions into the data structure. The mapping defines what is henceforth known as the CMPLISP environment. Finally, a series of basic functions which operate on the CMPLISP environment is defined. These functions and their interrelationships form a basis for the analysis and proofs in the remainder of the thesis.

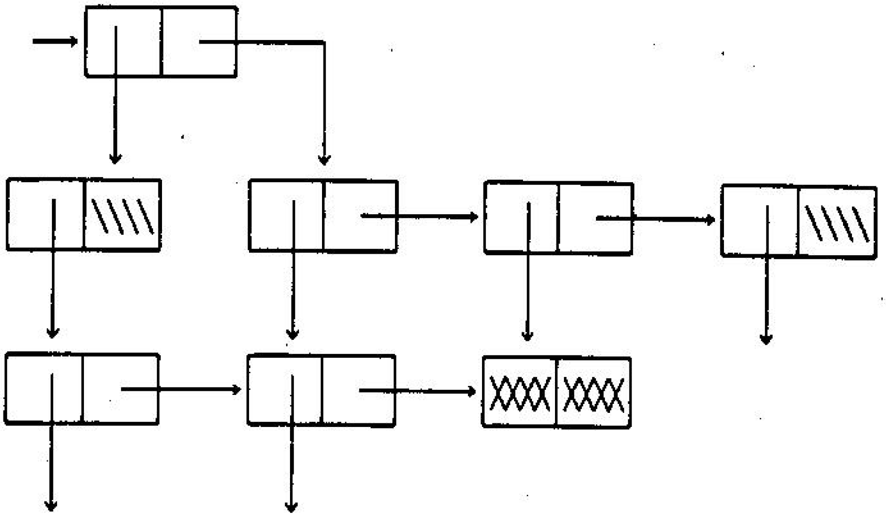


Figure 2.6 - Sample List Structure

2.B1 Data Structure

(1) List Structure - a set of two-element cells (see Figure 2.6) of which each element is an address of another cell in the List Structure or an address outside of the data structure. In the Figure cross-hatched elements indicate an address outside of the data structure while hatched elements denote a specific address in the List Structure.

(2) Free Storage list - an area of two-element cells of which one element contains the address of the next cell in the list. The last cell in the list is marked in a unique manner.

(3) Free Space - the cells comprising the List Structure and the Free Storage list.

(4) Free Word Space - a set of cells containing an arbitrary number of elements. Cells or elements of cells in this space are not directly accessible from cells in the List Structure. Elements of cells in this space containing addresses of cells in the List Structure fall into two classes:

(a) name space - possess a symbolic name known to the List Structure

(b) pointer space - possess a symbolic name known to the host (i.e. the computer) of the data structure.

(5) Binary Program Space - a set of four-element cells containing the machine encoding (i.e. LAP) of operations on the List Structure. Some of the cells may contain as elements addresses of cells in the List Structure or Free Word Space.

(6) Garbage - those cells in the List Structure which are not accessible (directly or indirectly) from outside of Free Space.

Associated with the data structure is a procedure known as Garbage Collection whose task is to purge the List Structure of all cells which fall into the Garbage category. The removed cells are placed on the Free Storage list. The procedure is invoked whenever it is necessary to add a cell to the List Structure and the Free Storage list is found to be empty. The List Structure is guaranteed to be free of Garbage only between the most recent invocation of this procedure and the first instance of addition (or allocation) of a cell from the Free Storage list into the List Structure.

2.B2 Implementation

Each s-expression corresponds to a two-element cell in the List Structure. Atoms are represented by a two-element cell containing identical marker symbols in both parts. These marker symbols are a distinguished address outside of the CMPLISP environment and are denoted in the figures by a cross-hatched element. The distinguished atom NIL is represented graphically by a hatched element and internally by 0. Non-atomic s-expressions are represented by pointers to two-element cells in the List Structure. Symbolically, if S1 and S2 denote s-expressions, then we have the mapping S1 x S2 →→ pointer to cell . Henceforth, the element containing S1 is known as the *head*, and the element containing S2 is known as the *tail*. Figure 2.7 contains a detailed illustration of the implementation.

Free Word Space contains the following special constructs which are not directly accessible from within the List Structure.

(1) Simple pointers reside in temporary locations not accessible from any element in the data structure. These locations are machine dependent - i.e. accumulators, stack, and memory.

(2) SPECIAL pointers (also known as SPECIAL or global variables) have a home base (i.e. a name associated with them) and a value. The existence of a name implies that a permanence is associated with the value. This permanence is expanded on in the sequel.

(3) The OBLIST is a set of cells pointing into the List Structure and acts as a symbol table to insure that all atoms with the same printname are uniquely represented. This statement is qualified by the earlier discussion of GENSYM.

(4) Property lists contain descriptions of atoms.

Binary Program Space contains the compiled code corresponding to certain LISP functions. These code sequences are accessed either directly from other locations in Binary Program Space or via the properties SUBR and FSUBR (for EXPR and FEXPR respectively) attached to the atomic function name that is to be invoked. Binary Program Space also contains pointers to elements of the List Structure for QUOTEd lists and pointers to Free Word Space for SPECIAL cells.

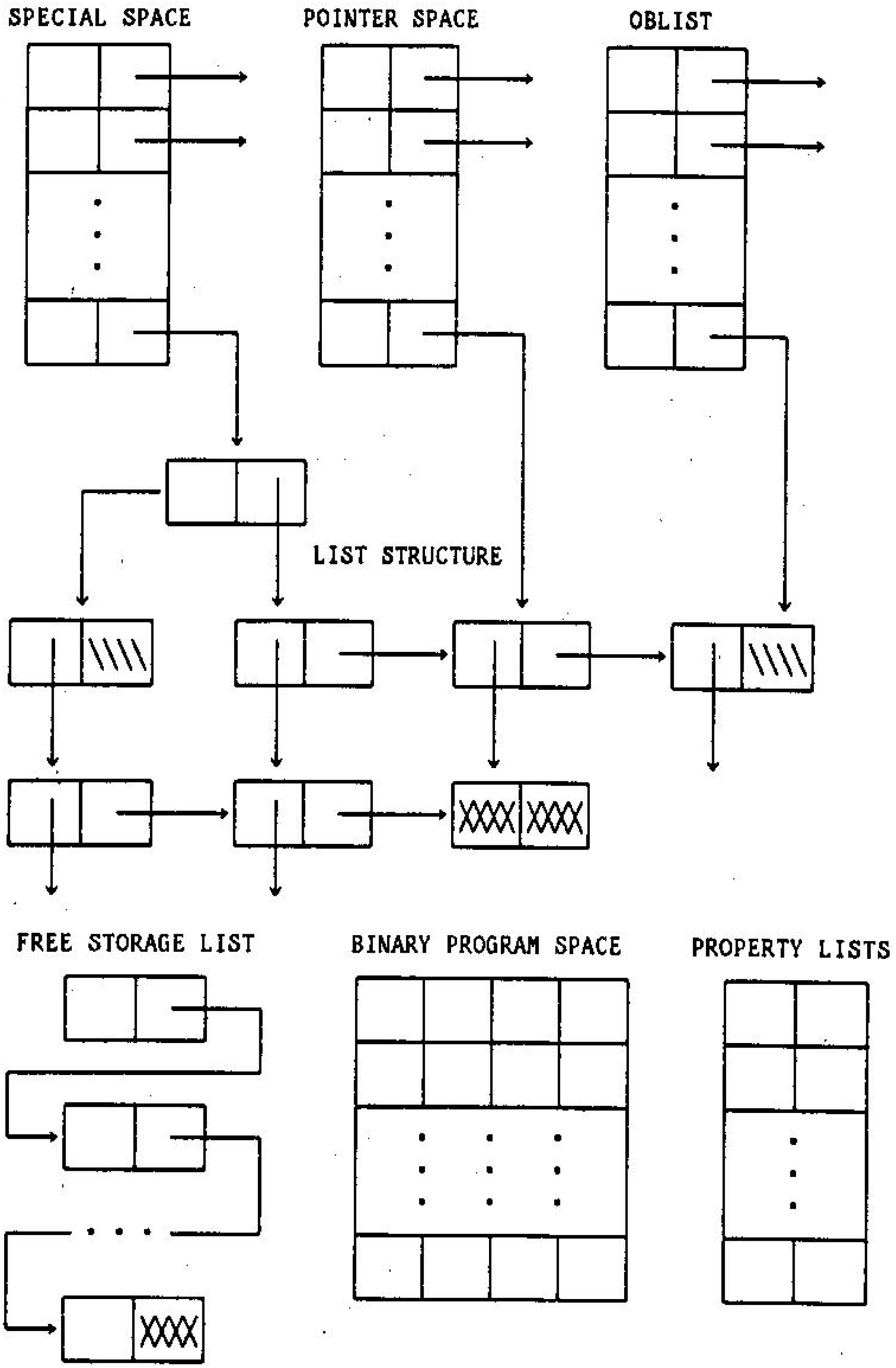


Figure 2.7 - LISP Implementation

2.B3 Functions

Functions evaluated in the CMPLISP environment have as their arguments and results s-expressions which have been transformed by the mapping into pointers. The domain is a subset of $S_1 \times S_2 \times \ldots \times S_n$ where n is the number of arguments and S_i is an s-expression. Similarly, the range is a subset of the set of all s-expressions. Functions are characterized as *primitive* if all of the following conditions are satisfied and non-primitive otherwise. We shall soon see that many functions will satisfy some or most of these criteria, but that only a very small number will satisfy them all.

(1) The function terminates.

(2) The function does not have side-effects - i.e. it does not modify the CMPLISP environment.

(3) No misapplication of the function is possible - i.e. no error can result from the application of the function. This is the case if the domain of the function is the set of all s-expressions.

(4) The function only accesses pointers to the List Structure. In other words, the function does not access cells in the List Structure (or for that matter, cells having symbolic names known to the List Structure).

(5) The result of the function is repeatable. This means that the function will yield the same result at all times if given the same arguments. This criterion only holds (and holds automatically) for functions that do nothing but access pointers into the List Structure.

2.B4 Pre-Defined Functions

Ten basic functions are defined below as mappings. The definitions are coupled with observations on whether or not the function is primitive. Whenever this is not the case, an attempt is made to indicate which of the criteria for primitiveness hold and under what conditions.

1. ATOM : <s-expression> →→ {NIL,T}

This primitive function returns NIL or T depending on whether or not the argument is a non-atomic s-expression. A typical implementation finds the atom property denoted by a marker of value -1 (i.e. all bits in the word are 1). The function is primitive because of the sanctity of atoms - i.e. atom is the most basic s-expression and once an s-expression represents an atom it does so forever.

2. CAR : (<s-expression1> . <s-expression2>) →→ <s-expression1>

This function accesses the *head* of a cell in the List Structure. It is a non-primitive operation because it is undefined when given an atomic argument.

3. CDR : (<s-expression1> . <s-expression2>) →→ <s-expression2>

This function accesses the *tail* of a cell in the List Structure. It is a non-primitive operation because it is undefined when given an atomic argument.

Composition of CAR and CDR operations is common and a special shorthand notation exists. For example, (CAR (CDR A)) ≡ (CADR A).

4. EQ : <s-expression> x <s-expression> ↠ {NIL,T}

This primitive transitive function (e.g. if A EQ B and B EQ C, then A EQ C) returns NIL or T depending on whether or not its two s-expression arguments are identical. It derives its usefulness primarily when its arguments are atomic since atoms are uniquely represented. In case the two arguments are both non-atomic, then they must be identical (i.e. not copies of one another). In all other cases the function returns NIL. Note that, unlike some definitions of LISP[Allen74], EQ is defined for non-atomic arguments.

There are several variants on EQ which are sufficiently common to warrant their definition at this time. NEQ yields the negation of the EQ function. NOT and NULL are identical functions, of one argument which return the value T if the argument is EQ to NIL, and NIL otherwise. NOT, NULL, and NEQ are all primitive.

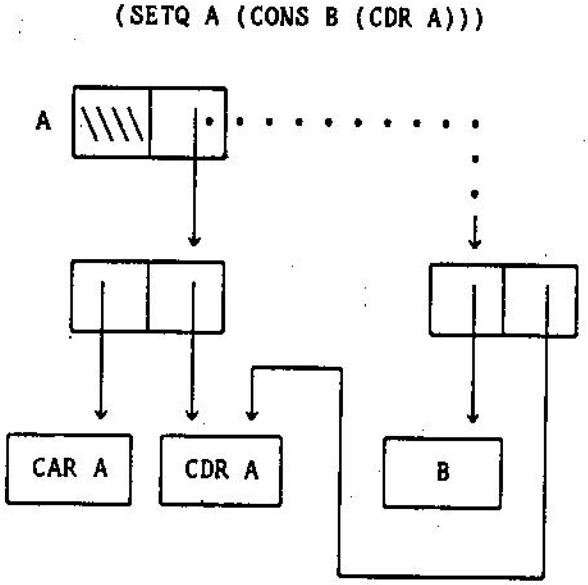


Figure 2.8a

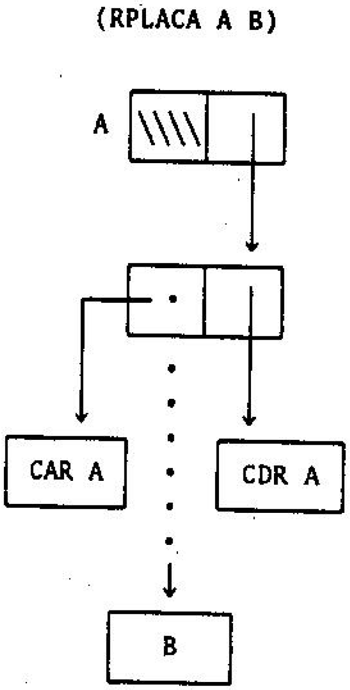


Figure 2.8b

5. RPLACA : (<s-expression1> . <s-expression2>) x <s-expression3>
↠ (<s-expression3> . <s-expression2>)

This non-primitive function results in the modification of the List Structure - i.e. the *head* of the specified cell is replaced. However, note that the value of the result is the same as the value of the first argument - i.e. the pointer to the modified cell has not changed, instead the contents of the cell has changed. The function owes its non-primitive nature to the fact that atoms cannot be destroyed and consequently the function is not defined when its first argument is atomic. For example, Figure 2.8b illustrates (RPLACA A B) before and after its execution. The dotted line represents the state of the List Structure after the operation has been performed.

Whenever a RPLACA operation is performed an EQ relation between certain pointers becomes true. For example, RPLACA(A,B) implies that CAR(A) EQ B is true for all subsequent CAR(A) operations until another operation is performed that results in the modification of the *head* of the same cell. However, in general, we do not know whether other pointers are EQ to the target of the RPLACA operation. Thus we modify our stipulation to read that no modification can occur of the *head* part of any cell. In fact, this is one of the motivations for a dual representation of functions - i.e. symbolic as well as numeric where the numeric representation indicates the instance of computation. In the following discussions we will repeatedly see a reference to a temporal property of functions.

Consecutive RPLACA operations with the same first argument and no intermediate access of the *head* of any other cell in the List Structure result in the first instance being redundant. The reason for the use of *any* is that more than one cell may contain a pointer to the modified cell. This situation can be likened to using the location in the list structure pointed at by the first argument of the RPLACA operation as a temporary location.

Consecutive RPLACA operations with the same first and second arguments, and no intermediate modification of the *head* of any cell in the List Structure, result in the second instance of the operation being redundant. The reason for the use of *any* is the same as in the previous paragraph.

6. RPLACD : (<s-expression1> . <s-expression2>) x <s-expression3>
→→ (<s-expression1> . <s-expression3>)

The RPLACD operation is analogous to RPLACA, with *tail* and *CDR* substituted for *head* and *CAR* respectively. For example, Figure 2.9b illustrates (RPLACD A B) before and after its execution. The dotted line represents the state of the List Structure after the operation has been performed.

7. SET : <special s-expression1> x <s-expression2> →→ <s-expression2>

This non-primitive operation assigns <s-expression2> to the SPECIAL variable name to which <special s-expression1> evaluates. In addition, the function returns <s-expression2> as its value. Thus we see that any one of the SPECIAL variables may be modified as a result of the function.

Consecutive SET operations with the same first argument and no intermediate access of any SPECIAL variable result in the first instance being redundant. The reason for the use of *any* is that we do not know which SPECIAL variable was modified. This situation can be likened to using the SPECIAL variable as a temporary location.

Consecutive SET operations with the same first and second arguments, and no intermediate modification of any SPECIAL variable result in the second instance of the operation being redundant. The reason for the use of *any* is the same as in the previous paragraph.

Note the duality between SET and RPLACA and RPLACD. Namely, the latter two imply that any *head* and *tail* respectively may have been destroyed while SPECIAL variables maintained their values, whereas SET implies that any SPECIAL variable may have been destroyed while *heads* and *tails* remained invariant.

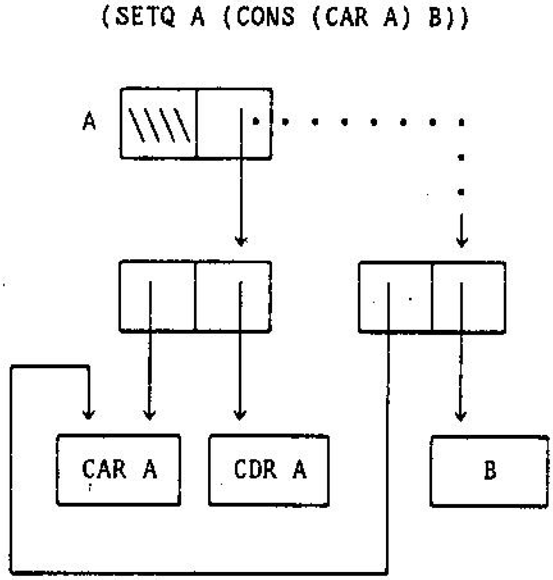

(SETQ A (CONS (CAR A) B))

Figure 2.9a

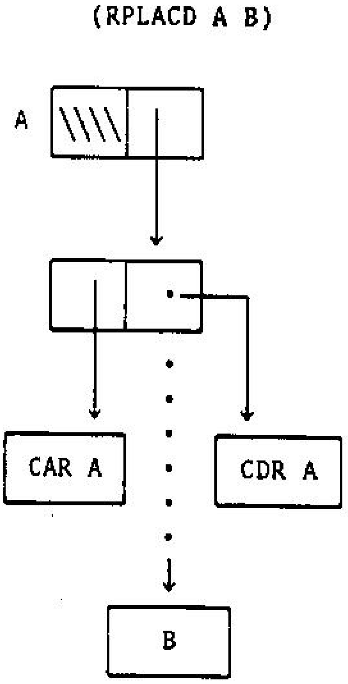

(RPLACD A B)

Figure 2.9b

8. SETQ : <special atom> x <s-expression1> →→ <s-expression1>

This non-primitive operation assigns <s-expression1> to the SPECIAL variable represented by <special atom>. Note that in contrast with SET, SETQ does not evaluate its first argument. The function returns as its result the value of <s-expression1>.

Consecutive SETQ operations with the same first argument (symbolically without making use of EQ operations) and no intermediate access of the said variable result in the first instance being redundant. This situation can be likened to using the SPECIAL variable as a temporary location.

Consecutive SETQ operations with the same first and second arguments (only the first arguments must be symbolic matches, the second arguments may also be the same by virtue of certain EQ relationships), and no intermediate modification of the said variable results in the second instance of the operation being redundant.

Note the distinction between the SETQ operation and RPLACA and RPLACD. Specifically, compare Figures 2.8a and 2.8b which correspond to (SETQ A (CONS B (CDR A))) and (RPLACA A B) respectively. Similarly, Figures 2.9a and 2.9b correspond to (SETQ A (CONS (CAR A) B)) and (RPLACD A B) respectively.

9. CONS : <s-expression1> x <s-expression2> →→
(<s-expression1> . <s-expression2>)

This non-primitive operation obtains a cell from the Free Storage list, sets its *head* and *tail* to <s-expression1> and <s-expression2> respectively, adds the cell to the List Structure, and returns as its result the address of the newly allocated cell.

The operation has several implications. First of all, we note that its result is always non-atomic. Secondly, CONS(A,B) implies that A and CAR(CONS(A,B)) are EQ, and similarly for B and CDR(CONS(A,B)). A related function is XCONS which is the same as CONS only the arguments are reversed - i.e. CONS(A,B) and XCONS(B,A) both return pointers to a cell containing A as its *head* part and B as its *tail* part. This type of relationship is denoted as antisymmetry and has been seen in the more familiar domain of arithmetic; A < B is equivalent to B > A (the CMPLISP formulation of the relation is (EQ *LESS(A,B) *GREAT(B,A)) = T where A and B evaluate to numbers). A still more familiar relation is commutativity which is a subset of antisymmetry and holds for such operations as addition and multiplication (whose CMPLISP analogues are *PLUS and *TIMES respectively).

CONS is almost a primitive operation in the sense that it terminates and results in a pointer. However, our implementation of CONS does not check if a pointer with identical *head* and *tail* parts already exists prior to allocating a new cell. The latter is the case with a system utilizing hashing as a means of keeping track of CONSs already performed. Thus the operation is not repeatable - i.e. two CONS operations with identical arguments yield pointers to s-expressions that are copies of one another rather than identical pointers. This implies a need for a different equality test than EQ since non-atomic s-expressions, unlike atoms, are not uniquely represented in terms of their atomic components. Thus CONS(CAR(A),CDR(A)) is not EQ to A.

The CONS operation may be considered redundant if its result is not pointed at by any addressable location in the CMPLISP environment. In this case the cell is a member of Garbage and the act of its removal from the List Structure is equivalent to not performing the operation. However, in order for the operation not to have any side-effects we must assume an infinite Free Storage list. Otherwise, the performance of the operation may yield different results since in one case we may run out of Free Storage while no problem would arise in the other case.

10. EQUAL : <s-expression1> x <s-expression2> →→ {NIL,T}

This non-primitive operation returns T if its two s-expression arguments are identical or copies of one another, otherwise NIL. EQUAL is non-primitive because list modification is allowed. In such a case an occurrence of the predicate may possibly not terminate, since a circular list may have been constructed. When the two arguments to the predicate are circular lists which are copies of one another, then application of the predicate will result in an infinite loop. This can be seen by examining the CMPLISP definition of EQUAL given in Figure 2.10.

Application of the predicate in two distinct instances with identical arguments may yield different results if any List Structure modification has occurred between the two instances of the application (compare with EQ and ATOM which always return the same result). This can be seen by examining our model of the CMPLISP environment and noticing that EQUAL must access the list structure. Unlike EQ, the function EQUAL is not necessarily transitive. The relations A EQUAL B and B EQUAL C being true at times t0 and t1 respectively may not imply that A EQUAL C at time t2 (t0 < t1 < t2). List Structure modification may have taken place between t0 and t2. However, if two items A and D are known to be EQUAL to two items B and C respectively, then if B and C are EQ, then A EQUAL D is also true.

The definition of EQUAL in Figure 2.10 has several implications. If two s-expressions are EQ, then they are also EQUAL. The contrapositive of the previous also holds and thus if two s-expressions are not EQUAL (define NEQUAL to be the negation of EQUAL), then they are also NEQ. The unique representation of atoms implies that if an s-expression is NEQ to an atom, then it is also NEQUAL to the atom. Similarly, if an s-expression is EQUAL to an atom then it is also

EQ to the atom. In the next section we will see that if two s-expressions, say A and B, are EQUAL and NEQUAL or if they are EQUAL and NEQ at various times, then neither A nor B can be an atom.

```
(DEFPROP EQUAL (LAMBDA (A B)
        (COND ((EQ A B) T)
              ((ATOM A) NIL)
              ((ATOM B) NIL)
              ((EQUAL (CAR A) (CAR B)) (EQUAL (CDR A) (CDR B)))))
 EXPR)
```

Figure 2.10 - Definition of EQUAL

2.B5 Implications of EQ, EQUAL, ATOM, and List Structure Modification

Theorem: If at different times A EQUAL B and A NEQUAL B, then neither A nor B is an atom.

Proof: We prove the result for the two cases depending on which of the two operations was encountered first.

(1) EQUAL before NEQUAL. In this case NEQUAL can only be true if List Structure modification occurred between the two computations. However, List Structure modification can only affect contents of cells in the List Structure and not the values of pointers to elements in the List Structure. Thus if A was an atom, then B must have also been an atom. Moreover, the unique representation of atoms implies that A EQ B. However, List Structure modification cannot be applied to atoms. Therefore the NEQUAL relationship would be impossible. Hence neither A nor B is an atom.

(2) NEQUAL before EQUAL. Since EQ implies EQUAL, we can use the contrapositive which yields NEQUAL implies NEQ. The unique representation of atoms coupled with A NEQUAL B implies that at least one of A and B can not be an atom. However, A EQUAL B means that neither A nor B is an atom by virtue of the unique representation of atoms.

Q.E.D.

Theorem: If A EQUAL B and A NEQ B, then both A and B are not atoms.

Proof: We prove the result for the two cases depending on which of the two operations was encountered first.

(1) EQUAL before NEQ. If A EQUAL B is true, then A NEQ B being true implies that the two list structures A and B are not represented uniquely. However, since atoms have a unique representation we have the result that A and B are both not atoms.

(2) NEQ before EQUAL. If A NEQ B is true, then if A and B are both atoms, then they are different. But A EQUAL B means that both A and B cannot be atoms since otherwise the unique representation of atoms would imply that A EQ B which is a contradiction.

Q.E.D.

Two computations are EQ if transitivity can be invoked, or the two functions represent two instantiations of the same function computed at different times satisfying the following criteria.

(1) The arguments to the two functions must be EQ. Note that we apply the same procedure to the arguments (i.e. transitivity or criteria 1-8).

(2) If the function accesses SPECIAL variables, then the variables accessed must have identical values prior to the two instances of computation of the function. Note that this is a stronger criterion than the more obvious stipulation that there be no modification of the said variables by the function or between its two instances of computation.

(3) If the function modifies any SPECIAL variables, then the variables modified must have identical values immediately after the first instantiation of the function and immediately prior to the second instantiation of the function. Note that this is a stronger criterion than the more obvious stipulation that there be no accessing of the said variables by the function, and no modification of the said variable between the function's two instances of computation.

(4) If the function accesses *head* parts of the List Structure, then there must be no modification of the *head* parts by the function or between its two instances of computation.

(5) If the function accesses *tail* parts of the List Structure, then there must be no modification of the *tail* parts by the function or between its two instances of computation.

(6) If the function modifies *head* parts of the List Structure, then there must be no accessing of the *head* parts by the function, and no modification of *head* parts between the function's two instances of computation.

(7) If the function modifies *tail* parts of the List Structure, then there must be no accessing of the *tail* parts by the function, and no modification of *tail* parts between the function's two instances of computation.

(8) The function does not involve the performance of a CONS operation.

Note the similarity between the various criteria as well as the complementary nature of the SPECIAL variable and List Structure conditions. We have seen these conditions expressed in a less general manner in the description of SET, SETQ, RPLACA, and RPLACD. If criteria (1) through (7) are satisfied for two instances of the same function, but the function involves the performance of a CONS operation, then the two instances are said to be EQUAL. If the function in the previous discussion is QUOTE, and if the arguments are symbolically equivalent then the two instances are EQ if the arguments are atomic, and EQUAL otherwise. The relationships hold because of the unique representation of atoms. Finally, if an operation involves the incrementing of a SPECIAL variable, then a second instance of the operation fails to satisfy criterion (2) even though criterion (3) may have been satisfied.

2.C Differences from Other Versions of LISP

In this section we attempt to relate our implementation decisions and definition of CMPLISP to other versions of LISP. In the course of the discussion, certain problems inherent to LISP will be posed and our solutions will be explained. Most of the comparisons made are with the LISP 1.6 system at Stanford. However, the issues are generally common to all implementations of compiled LISP. Perhaps a more appropriate title is *why do we make such statements about properties of functions?*

We have shown an implementation defined in terms of the semantics of functions and an environment. This is important since otherwise the equivalence between the source and object programs will be somewhat meaningless. This is especially true with respect to possible error conditions and their detection. More importantly, it allows us to prove equivalence, subject to certain properties of functions which must hold in an implementation. This can be seen in the soon-to-be-proposed assumptions about the permanence of the bindings of certain function names to their code, protection, etc.

One of the primary reasons for the existence of CMPLISP is the inconsistency of the implementation of compiled LISP 1.6 with the definition (or, shall we say spirit) of LISP. This is most glaring in the handling of atomic s-expressions. From the formal definition of LISP we expect that atoms are represented in a uniform manner. However, in LISP 1.6 atomic s-expressions have different representations which depend on whether or not the atom corresponds to a small integer. In LISP 1.6 atomic s-expressions that do not denote small integers are represented by a two-element cell whose *head* part contains the atom marker (-1) and the *tail* part contains a pointer to the property list associated with the atom. When the atom corresponds to a small integer, then the atom marker is a 1 in the position of the most significant bit (i.e. the sign bit). The remaining bits are used to represent small integers using an excess code. This is unfortunate since it leads to a more complicated test for the atom property, and more importantly a non-uniform representation of atoms (e.g. the value of an atom denoting a small integer does not reside on the property list). Another inconsistency results from the fact that the CAR and CDR operations are not defined for atomic s-expressions. Yet due to the implementation, CDR of an atom yields the property list rather than an invalid address. Similarly, RPLACA and RPLACD are not defined when their first argument is an atom, yet a check is not made for the misapplication of the operations. Improper use of these operations could lead to irreparable harm. A case in point is the destruction of the *head* of an atom which is the atom marker. All subsequent references to the modified cell may yield errors since the sanctity of atoms is no longer preserved.

All functions must return as their results valid s-expressions. This is in addition to any other effects they may have. The primary villains in LISP 1.6 are property list accessors. Since property lists may contain values which are not s-expressions (e.g. strings), care must be exercised in operations performed on such values. Basically, the programmer must not be allowed to manipulate these non-s-expression values. Instead, he should have (and does have) at his disposal pre-defined system functions which operate on this data and perform the necessary operations. For example, in order to concatenate two strings, say A and B, a LISP programmer simply does:

```
READLIST(APPEND(EXPLODE(A),EXPLODE(B)))
```

where EXPLODE transforms its s-expression argument into a list of single character identifiers corresponding to the printname associated with the s-expression (note that the printname of a non-atomic s-expression is its representation as a function of its constituent atoms).

APPEND forms a list comprising all of the elements in its arguments, which must also be lists.

READLIST transforms a list of single character identifiers into an s-expression identical to that which would be produced by reading those characters as data. All identifiers in the resulting s-expression are INTERNed.

Granted, this is slightly inefficient; however, it is the price we pay for having a type-less language, and there is no justification for butchering the original language for the sake of efficiency at the price of validity of results.

The type of protection considerations undertaken for atoms finds its analog in the domain of functions. Built-in or basic functions such as those described in the previous section cannot be redefined. This is a must when dealing with compiled code, since the compiler must know at compile-time such function properties as the number of arguments expected by the function, possible side-effects, etc. Thus we do not allow the form (operation fname prop) where operation is PUTPROP, DEFPROP, or REMPROP, prop is EXPR, FEXPR, SUBR, or FSUBR, and fname is the name of a function that has been compiled or seen in the original program. Similarly, user defined functions cannot be redefined.

Definition of functions *on the fly* is permissible in CMPLISP, but it has not yet been implemented. Such a definition can only be done via DEFPROP, which must check that the s-expression corresponding to function is a valid function (syntactically). The existence of DEFPROP makes unnecessary the LABEL construct (see [McCarthy62]), which allows the definition of recursive functions *on the fly*.

The previous restrictions do not mean that the s-expression formulation of a function is inaccessible to the programmer. In fact, the programmer has access to copies of the s-expression formulation of all EXPRs and FEXPRs. Thus when modifications are made the only restriction is that the modified function must be given a new name and a DEFPROP is performed on the new function definition. Notice that the latter plus the fact that a copy was modified rather than the original insures compatibility with the restriction on redefinition of functions.

One difference between various LISP systems is in the manner in which variables are accessed. Some use deep binding and others use shallow binding. Deep binding is characterized by an association-list (known as the a-list) containing pairs of entries where the first element is a variable name while the second element of the pair is the current binding of the variable. This list is updated at function entry by placing the values of the variables at the front of the list. Similarly, the appropriate values of the variables are deleted from the list at function exit. Whenever a variable name is encountered, the a-list is searched for the first pair having the said variable as its first entry and the second entry is returned as the value of the variable. Shallow binding is based on the premise that the lookup operations associated with the deep binding mechanism are too slow. Instead, upon function entry, all local variables are bound to their appropriate values. If the variable already had a value, then the old value is saved and restored upon function exit. The work of binding and restoring is done by the routines BIND and UNBIND respectively.

In CMPLISP all variables are either local to the function instantiating them or global to the entire set of function definitions (i.e. all variables occurring free are global variables - where free takes on the same meaning it has in the lambda calculus). In other words if variables A and B are local to functions F and G, and function F calls function G, then variable B is not accessible to function G. These global variables are called SPECIAL in CMPLISP. They are analogous to PUBLIC variables in LISP70[Tesler73].

Our interpretation of SPECIAL is as a placeholder for a pointer to an s-expression which transcends the period of time during which the function is active (i.e. the s-expression is still pointed at after function exit). This is different from the SPECIAL construct in the LISP 1.6 system where it is only used in the compiler. The reason is that the use of shallow binding coupled with the non-existence of names renders the routines BIND and UNBIND meaningless, and thus they must be given explicit instructions by the programmer as to which variables are to be accessible to inner functions. In fact, in compiled LISP 1.6 the SPECIAL construct is used to indicate that the variable can be accessed as a free variable by all functions invoked by the function in which it is declared to be SPECIAL. Note that this construct is not necessary in the interpreter where shallow binding with its accompanying BIND and UNBIND routines are used.

Functional arguments to functions are allowed, but they can only reference their own local variables and the SPECIAL variables. This eliminates the FUNARG problem which is characterized by a functional argument whose function definition contains free variables. The problem is what should be the bindings of the free variables once the function is invoked. There are two choices. One solution is that the variables ought to be bound to their values at the time the function was passed in as a parameter, while the alternative solution is to use the bindings of the variables at the time the function is invoked. The first choice was intended by the original LISP definition but not achieved by it. This caused a need for passing in, along with the function name, the bindings of the free variables occurring in the function definition. For more details see [McCarthy62],[AllenJ74]. Note that in CMPLISP the second choice is the case by definition of SPECIAL and the question of which of the two choices to implement is moot.

A concept that is vital to a shallow binding implementation is the VALUE cell. This is an element on the property list of atoms which serve as variables. The term *VALUE cell* is derived from the fact that the cell containing the value of the variable is identified by the property name VALUE. The distinguished atoms NIL and T are characterized by VALUE cells pointing to the atoms NIL and T respectively. The concept has lost most of its meaning in terms of CMPLISP since variables do not exist and references to SPECIAL variables in compiled code are made directly to the VALUE cell. Thus we see that the programmer in CMPLISP has no need for the VALUE property or cell and thus he is not allowed access to it. This means that the property list accessing and modifying functions, when invoked by a user function, may not have as their property parameter the VALUE property. However, functions such as EVAL can access and modify this property as part of their functional definition.

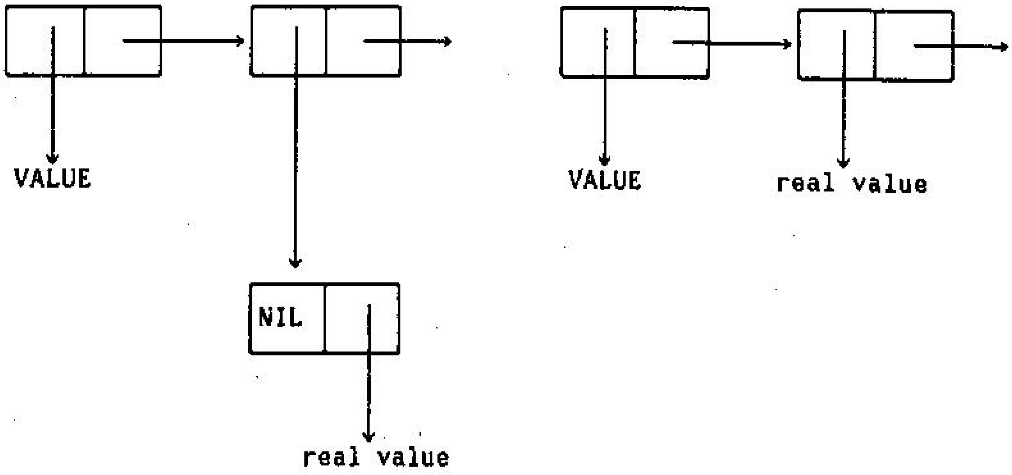


Figure 2.11 - VALUE Cell Implementation

Examination of the implementation of the property VALUE in LISP 1.6 (Figure 2.11) finds it to be sufficiently perverse to make it inadvisable for programmer access. Note that the value link is to a cell containing the value rather than to the value itself as is done for all other properties. Thus a GET(IDENTIFIER,VALUE) operation will return (NIL . <value>) as shown in the left part of the figure rather than the expected <value> as shown in the right part of the figure. The reason for this implementation is to enable compiled code to access a cell that is the same as a word containing a pointer rather than merely access a pointer in the left half of a cell. Thus the cell has the same appearance as a cell containing a parameter upon function entry. This example provides a graphic demonstration why a programmer should not have direct access to the VALUE property, or in fact to any property, since the exact structure of the property list is extremely implementation dependent.

At this point we have covered enough ground so that a new problem can be presented. Executing a compiled function and interpreting the same function may yield different results. This is the main motivation for the *careful* definition of the CMPLISP environment and functions. First of all, the property list of an atom should not be accessible to a program except via predefined system functions. We want to preserve the integrity of the property list by controlling the access and modifications to it. This will prevent the LISP 1.6 obscenities[†] associated with a combination of RPLACA and RPLACD operations that result in the replacement of the value cell associated with a SPECIAL variable with another cell. In this case, compiled code will refer to a value cell different than that done by the interpreter. A similar problem occurs when the PNAME associated with an atom is changed. This is why the GETL operation of LISP 1.6 is not available in CMPLISP; its result is defined to be a pointer into the property list. Secondly, there must be some protection against certain acts of modification of the OBLIST. In particular, arbitrary entries cannot be removed. A problem can occur if an atom that has been removed from the OBLIST is later referenced in the LAP code of a function. A similar problem will occur for SPECIAL variables.

atom marker	atom marker
REMOB marker	PNAME pointer
SPECIAL marker	rest of property list pointer

Figure 2.12 - New ATOM Property List Implementation

So far, our implementation of atoms and property lists has been revealed in fragments. We have seen various caveats, but no unified solution has yet been presented. In order to allow access to property lists of atoms with protection, along with no degradation in access times, we propose that property list information associated with the atom occupy cells in memory immediately following the cell containing the atom markers. This has several advantages. First of all, the property list cannot be directly addressed by any cell in the List Structure; yet knowledge of the cell corresponding to an atom implies that the property list location is also known. Secondly, certain properties belonging to all atoms such as printname, SPECIAL variable designation, and whether or not the atom can be removed from the OBLIST can occupy fixed locations (via an offset). Thus for each atom we allocate three contiguous two-element cells where the first is considered a member of the List Structure (Free Space) and the remaining two are elements of Free Word Space.

[†] "LISP is a four letter word." [Cerf]

For a sample implementation and assignment of meaning to the cells see Figure 2.12. We also mention that all atoms are represented uniformly. This means that the LISP 1.6 method of representing small integers is abandoned. When an atom marker of all bits 1 is used, then the ATOM predicate becomes exceedingly simple and lends itself to inline execution on most computers. Actually, for the sake of efficiency and history, the atom NIL is represented internally by 0 and does not fall into the three contiguous cells representation of atoms. However, the property list accessing and modifying functions functions can be made aware of this fact. Other implementations may abandon the 0 internal representation of the atom NIL.

The key to our solution is the realization that entries in Free Space and Free Word Space need not be contiguous. This has some effect on the process of Garbage Collection. However, atoms and their property lists can be identified by examining the OBLIST. Thus Garbage Collection must take into account an area consisting of Free Space and property list cells in Free Word Space as one contiguous area. Judicious use of bit tables can facilitate the process.

The restrictions on the use of predefined functions proposed here (e.g. REMPROP, sanctity of atoms, etc.) are necessary if equivalence is to be proved. If the property list functions were not so modified and likewise for the implementation, then operations could occur which might result in errors that may or may not be detected. The main thrust is to make sure that effects of functions are known. If illegal operations were to occur, then they would be detected. We are especially concerned with changes to functions whose definitions are known and serve as a basis for the proof. By making our modifications to LISP 1.6, we in fact insure that our assumptions about the effect of certain functions are valid. A case in point is the RPLACD operation. From our definition, we know that pointers to the property list of an atom, or to parts of it, cannot be accessed or modified by any function written by the programmer. Thus the VALUE cell corresponding to a SPECIAL variable cannot be modified by a RPLACD operation. Therefore, we may assume that RPLACD operations do not modify SPECIAL variables. This type of analysis is critical to the achievement of any optimizations through knowledge of the behavior of functions. In fact with the restrictions posed here we see that RPLACA and RPLACD are no longer as dangerous as previously believed. Namely, they will now have only the obvious effect on shared s-expressions, and no unanalyzable ramifications.

The previous discussion was concerned primarily with the effects of modifications of locations whose accessing is undesirable. There are also locations whose accessing is illegal, yet no undetected harm can arise from subsequent operations. That is to say any harm caused by subsequent operations on these illegal accesses will be detected. We speak of the accessing of illegal addresses in the List Structure. An example is when a function expecting an s-expression is presented with an atom. In this case the result will be an illegal address not in the List Structure and subsequent attempts to use it as an s-expression to further access the List Structure will be detected. On the PDP-10 system this will be recognized as an ILLEGAL MEM REF FROM CAR message. In other implementations the CAR and CDR functions may be implemented with such checks in mind. Nevertheless, it should be clear why accessing such addresses is not harmful in the sense of the previous discussion - i.e. no important information will be stored or be accessible via the illegal addresses (they can be likened to either traps on a system with protection capabilities, or to deferred interrupts).

CHAPTER 3

THE CANONICAL FORM

The cornerstone of this thesis is the work done by McCarthy[McCarthy63] in showing the existence of a canonical form for the theory of conditional expressions and its use in proving equivalence. This theory corresponds to analysis by cases in mathematics and is basically a generalization of propositional calculus. In this chapter we define conditional forms and their properties as they relate to a canonical form to be used in proving equivalence. This work is primarily a restatement of McCarthy's presentation. Next we show how a canonical form is obtained and prove its existence. The algorithms have been somewhat modified from those to be found in [McCarthy63] in that certain of the original axioms are proved to be unnecessary. Finally, we show how to adapt the notions of a canonical form and equivalence to programs written in LISP[†].

3.A Conditional Forms

Consider expressions, known as generalized Boolean forms (gbf), formed as follows:

(1) Variables are divided into propositional variables p,q,r, etc. and general variables x,y,z, etc.

(2) $(p \to x,y)$ is called an elementary conditional form of which p,x, and y are called the premise, conclusion, and alternative respectively.

(3) A variable is a gbf, and if it is a propositional variable, then it is called a propositional form (pf).

(4) If π is a pf and α and β are gbfs, then $(\pi \to \alpha,\beta)$ is a gbf. If, in addition, α and β are pfs, so is $(\pi \to \alpha,\beta)$.

The value of a gbf α for given values (T,F, or undefined) of the propositional variables will be T or F in case α is a pf and a general variable otherwise. This value is determined for a gbf $(\pi \to \alpha,\beta)$ according to the table given in Figure 3.1.

value(π)	value(($\pi \to \alpha,\beta$))
T	value(α)
F	value(β)
undefined	undefined

Figure 3.1 - Conditional Form Values

[†] Henceforth we shall be referring to LISP rather than CMPLISP although it should be clear that we mean the latter.

Two gbfs are said to be strongly equivalent (denoted by $\equiv$) if they have the same value for all values of the propositional variables in them including the case of undefined propositional variables. The gbfs are weakly equivalent (denoted by $\equiv_w$) if they have the same values for all values of the propositional variables when these are restricted to T and F.

There are two equivalence rules which enable the use of equivalences to generate other equivalences†.

(a) If $\alpha\equiv\beta$ and α_1, β_1 are the results of substituting any gbf for any variable in $\alpha\equiv\beta$, then $\alpha_1\equiv\beta_1$. This is known as the *Rule of Substitution*. This enables the use of the about-to-be-presented axioms as schemas.

(b) If $\alpha\equiv\beta$ and α is a subexpression of λ and ∂ is the result of replacing occurrences of α in λ by an occurrence of β, then $\lambda\equiv\partial$. This is known as the *Rule of Replacement*. Note the similarity to substitution of equals for equals.

Equivalence can be tested by the method of truth tables as in propositional calculus, and also by using the following eight equations as axioms to transform any gbf into an equivalent one. This transformation is aided by using the rules of substitution and replacement.

(1) $(p\to a,a) \equiv_w a$

(2) $(T\to a,b) \equiv a$

(3) $(F\to a,b) \equiv b$

(4) $(p\to T,F) \equiv p$

(5) $(p\to(p\to a,b),c) \equiv (p\to a,c)$

(6) $(p\to a,(p\to b,c)) \equiv (p\to a,c)$

(7) $((p\to q,r)\to a,b) \equiv (p\to(q\to a,b),(r\to a,b))$

(8) $(p\to(q\to a,b),(q\to c,d)) \equiv (q\to(p\to a,c),(p\to b,d))$

Note that all are strong equivalences with the exception of the first which is a weak equivalence. Thus our previous statement about transforming a gbf into an equivalent one should be reworded to preclude the use of axiom (1) in proving strong equivalence.

In fact these rules and axioms can be used to transform any gbf into a canonical form defined as follows:

If $p_1, p_2, \ldots, p_n$ are the variables of the gbf, π, taken in some arbitrary order, then π can be transformed into the form:

$(p_1\to a_0,a_1)$ where each a_i has the form:

$a_i \equiv (p_2\to a_{i0},a_{i1})$ and in general for each $k=1,\ldots,n-1$

$a_{i_1\ldots i_k} \equiv (p_{k+1}\to a_{i_1\ldots i_k 0}, a_{i_1\ldots i_k 1})$

and each $a_{i_1\ldots i_n}$ is a truth value or a general variable.

† The rules hold for weak and strong equivalences.

Thus in this canonical form, the 2^n cases of the truth or falsity of $p_1, p_2, \ldots, p_n$ are explicitly exhibited. Another way of viewing the canonical form is to think of it as a binary tree[Knuth68] whose non-terminal nodes are propositional variables and whose terminal nodes represent computations. This conceptualization will be used in the sequel where the general term *tree* will be used instead.

The algorithm for obtaining a canonical form for weak equivalence is as follows:

(1) Use axiom (7) repeatedly until in every subexpression the π in $(\pi \rightarrow \alpha, \beta)$ consists of a single propositional variable. Also apply axioms (2) and (3) whenever possible.

(2) The propositional variable p_1 is moved to the front by repeated application of axiom (8). There are three cases:

(a) $(q \rightarrow (p_1 \rightarrow a,b),(p_1 \rightarrow c,d))$ to which axiom (8) is directly applicable.

(b) $(q \rightarrow a,(p_1 \rightarrow c,d))$ where axiom (8) is applicable after axiom (1) is used to make it $(q \rightarrow (p_1 \rightarrow a,a),(p_1 \rightarrow c,d))$.

(c) $(q \rightarrow (p_1 \rightarrow a,b),c)$ which is handled in the same manner as case (b) - i.e. axiom (8) is applicable after axiom (1) is used to yield the form $(q \rightarrow (p_1 \rightarrow a,b),(p_1 \rightarrow c,c))$.

(3) Once the main expression has the form $(p_1 \rightarrow \alpha, \beta)$, then all p_1's which occur in α and β are moved to the front of α and β by using the same procedure. The p_1's which have been moved are then eliminated by using axioms (5) and (6). p_2 is then moved to the front of α and β, using axiom (1) if necessary to insure at least one occurrence of p_2 in each of α and β. This process is continued until the canonical form is achieved.

There is also a canonical form for strong equivalence. The difference is that the propositional variable p_1 may not be chosen arbitrarily, but instead must be an inevitable variable of the gbf a. An inevitable variable of a gbf $(\pi \rightarrow \alpha, \beta)$ is defined to be either the first propositional variable or else an inevitable variable of both α and β. Note that once again the canonical form is of the form $(p_1 \rightarrow \alpha, \beta)$ where α and β do not contain p_1 and are themselves in canonical form.

The algorithm for the derivation of the canonical form for strong equivalence is identical to the algorithm given for weak equivalence. This statement is in contrast with the algorithm given in [McCarthy63] where two axioms were added in addition to the restriction that axiom (1) could not be used. The axioms that were unnecessarily added were:

$$(9) \quad (p \rightarrow (q \rightarrow a,b),c) \equiv (p \rightarrow (q \rightarrow (p \rightarrow a,a),(p \rightarrow b,b)),c)$$
$$(10) \quad (p \rightarrow a,(q \rightarrow b,c)) \equiv (p \rightarrow a,(q \rightarrow (p \rightarrow b,b),(p \rightarrow c,c)))$$

The algorithm for obtaining a canonical form for weak equivalence was modified to be valid for strong equivalence by use of these axioms so that occurrences of an inevitable variable, say p_1, in the conclusion or alternative can be eliminated when substitution and replacement are used. This was given as an alternate solution to the obvious use of the general rule that any occurrence of the premise in the conclusion can be replaced by T and occurrences in the alternative by F. The motivation behind the proposed solution is a possible reluctance to make use of the meta-notion of T and F and to work strictly by using formulas not involving the introduction of T or F. The revised algorithm stated that it is desired to replace all occurrences of the premise in the conclusion by T, and occurrences in the alternative by F. This is accomplished by finding the clause (i.e. conclusion or alternative) which contains the objectionable atom. If it is in the conclusion, then axiom (9) is used; and if it is in the alternative, then axiom (10) is used. Next, axioms (9) and (10) are applied in the manner described in the previous statement until the objectionable atom, say p, occurs as the inner p of one of the forms $(p\rightarrow(p\rightarrow a,b),c)$ or $(p\rightarrow a,(p\rightarrow b,c))$. In either case the objectionable p is removed by using axioms (5) or (6) and p's that were introduced by applications of axioms (9) and (10) are removed by repeated application of axiom (8).

Actually, the algorithm differs from that given for the weak equivalence case in that step (2) now states: choose any inevitable variable, say p_1, and put the gbf in the form $(p1\rightarrow\alpha,\beta)$ by using axiom (8). Note that axioms (9) and (10) were added for the specific reason that axiom (1) could not be used. In fact, there is no need at all for axioms (9) and (10) since, for example, axiom (9) can be shown to be true in the following manner:

$$\begin{aligned}(p\rightarrow(q\rightarrow(p\rightarrow a,a),(p\rightarrow b,b)),c) &\equiv (p\rightarrow(p\rightarrow(q\rightarrow a,b),(q\rightarrow a,b)),c) && \text{by axiom (8)}\\ &\equiv (p\rightarrow(q\rightarrow a,b),c) && \text{by axiom (5)}\end{aligned}$$

The same can be said for axiom (10).

Therefore the algorithm is revised as follows:

(1) Use axiom (7) to get all premises as propositional variables.

(2) Choose any inevitable variable, say p_1, and put the gbf in the form $(p_1\rightarrow\alpha,\beta)$ by using axiom (8).

(3) Eliminate occurrences of p_1 in α and β. If p_1 is in the conclusion, then introduce in the alternative clause, say alt, $(p1\rightarrow alt,alt)$. Similarly, if p_1 is in the alternative, then introduce in the conclusion clause, say conc, $(p_1\rightarrow conc,conc)$. Following this step axiom (8) and the above substitution or its alternative are used in an alternating manner until one of axioms (5) or (6) is applicable.

<u>Proof of the validity of the change in step (3):</u>

Since we have passed the point where p_1 is undefined, axiom (1) - i.e. $(p_1\rightarrow a,a) \equiv a$ is valid since p_1 is now defined. Remember the latter was the only reason that axiom (1) was not applicable to strong equivalence. The use of the equivalence $(p_1\rightarrow a,a) \equiv a$ is valid according to the rule of replacement.

Thus it is seen that actually there is no difference in the method of proof for strong equivalence and weak equivalence and that the canonical forms can be the same if inevitable variables are used. This leads to the following:

Theorem: If two gbfs are strongly equivalent, then they are also weakly equivalent.

Proof: Order the variables according to inevitability. This is one of the acceptable orders and weak equivalence follows.

Q.E.D.

As an example of the process of determining the equivalence of two gbfs we show $(p\to(q\to(p\to x,y),b),c) \equiv (p\to(q\to x,b),c)$ by means of axioms (9) and (10) and also by means of the revised algorithm.

Axioms (9) and (10) method:

$$\begin{aligned}
&(p\to(q\to(p\to x,y),b),c) \\
&\quad\equiv (p\to(q\to(p\to(p\to x,y),(p\to x,y)),(p\to b,b)),c) && \text{by axiom (9)} \\
&\quad\equiv (p\to(q\to(p\to x,(p\to x,y)),(p\to b,b)),c) && \text{by axiom (5)} \\
&\quad\equiv (p\to(q\to(p\to x,y),(p\to b,b)),c) && \text{by axiom (6)} \\
&\quad\equiv (p\to(p\to(q\to x,b),(q\to y,b)),c) && \text{by axiom (8)} \\
&\quad\equiv (p\to(q\to x,b),c) && \text{by axiom (5)}
\end{aligned}$$

Revised algorithm method:

$$\begin{aligned}
(p\to(q\to(p\to x,y),b),c) &\equiv (p\to(q\to(p\to x,y),(p\to b,b)),c) && \text{by axiom (1)} \\
&\equiv (p\to(p\to(q\to x,b),(q\to y,b)),c) && \text{by axiom (8)} \\
&\equiv (p\to(q\to x,b),c) && \text{by axiom (5)}
\end{aligned}$$

The canonical form algorithm for strong equivalence can be further simplified by revising steps (2) and (3) as follows. Once an inevitable propositional variable, say p_1, of the gbf $(\pi\to\alpha,\beta)$ has been found we simply replace the gbf by an application of axiom (1) - i.e. $(\pi\to\alpha,\beta) \equiv (p_1\to(\pi\to\alpha,\beta),(\pi\to\alpha,\beta))$. To the quantity on the right we apply axioms (5) and (6) until they can be applied no further. It should be clear that this procedure is equivalent to the previously given algorithm. The only difference is that the latter proceeds to propagate the inevitable variable out from inside the gbf; while the former first brings the variable out and then applies the redundant predicate removal axioms (i.e. (5) and (6)). In the future any discussion of an algorithm for proving strong equivalence for canonical forms will refer to the revised algorithm.

At this point we come to the main result:

Theorem: Two gbfs are equivalent (weak or strong) iff they have the same (weak or strong) canonical form.

Proof: One direction (the only if case) is true by definition since if two gbfs have the same canonical form, then they can be transformed into each other by the canonical form transformations. The proof of the other direction (the if case) is proved separately for weak and strong equivalence.

weak equivalence: We prove the contrapositive. Suppose two gbfs have different canonical forms when the propositional variables are taken in the same order. Then values can be chosen for the propositional variables yielding different values for the form thereby proving non-equivalence.

strong equivalence: Suppose that two gbfs, say a and b, do not have the same propositional variables. Let p be inevitable in a but not in b. Now, if the other propositional variables are assigned suitable values, then b will be defined with p undefined. However, a will be undefined since p is inevitable in a which proves non-equivalence. Therefore, strongly equivalent gbfs have the same inevitable variables; so, let one of them be put in front of both gbfs. The procedure is then repeated in the conclusion and alternative etc. This corresponds to an inductive proof where we induct on the number of conditions.

Q.E.D.

The relation of functions to gbfs is given by the distributive law:

$$\begin{aligned} f(x_1,\ldots,x_{i-1},(p\rightarrow q,r),x_{i+1},\ldots,x_n) \equiv (p\rightarrow & f(x_1,\ldots,x_{i-1},q,x_{i+1},\ldots,x_n),\\ & f(x_1,\ldots,x_{i-1},r,x_{i+1},\ldots,x_n)) \end{aligned}$$

A general conditional form is defined to be $(p_1\rightarrow e_1,p_2\rightarrow e_2,\ldots,p_n\rightarrow e_n)$ which is equivalent to $(p_1\rightarrow e_1,(p_2\rightarrow e_2,\ldots(p_n\rightarrow e_n,u)\ldots))$ where u is a special undefined variable. The properties of general conditional forms are identical to those of gbfs.

The rule of replacement can be extended in the case of conditional forms. Suppose α is a subexpression of an expression β. A propositional expression π called the premise of α in β is defined as follows:

(1) The premise of α in α is T.

(2) The premise of α in $f(x_1,\ldots,x_i,\ldots x_n)$ where α is part of x_i is the premise of α in x_i.

(3) If α occurs in e_i and the premise of α in e_i is π, then the premise of α in $(p_1\rightarrow e_1,\ldots,p_i\rightarrow e_i,\ldots p_n\rightarrow e_n)$ is $(\neg p_1\wedge\ldots\wedge\neg p_{i-1})\wedge p_i\wedge\pi$.

(4) If α occurs in p_i and the premise of α in p_i is π, then the premise of α in $(p_1\rightarrow e_1,\ldots,p_i\rightarrow e_i,\ldots,p_n\rightarrow e_n)$ is $\neg p_1\wedge\ldots\wedge\neg p_{i-1}\wedge\pi$.

The *Extension to the Rule of Replacement* is that an occurrence of α in β may be replaced by α' if $(\pi\rightarrow\alpha) \equiv (\pi\rightarrow\alpha')$ where π is the premise of α in β. Thus in a subcase we need only prove equivalence under the premise of the subcase. In other words, α can be replaced by α' whenever π is satisfied. We shall later see the importance of this rule.

3.B Adaptation to LISP

In order for the previous ideas to be useful in proving correctness of translation of LISP programs to assembly language programs, we must show how to adapt them to LISP functions. We are primarily interested in proving strong equivalence and the more general notion of functions rather than variables and likewise for functional predicates instead of propositional variables. Our adaptation consists of defining some new constructs and using them to extend the canonical form ideas so that our LISP programs will fit into the framework.

In this section we are only interested in obtaining a representation of the function in some normal form with no rearranging of conditions (axiom (8)) or application of axiom (1). This is because we want to prove later that the compiled program yields the same result. This process will be called matching. At that point we will see the invocation of these axioms. The use of substitution of items EQ to one another further contributes to the use of a normal form. In the course of the discussion we will refer to the representation as the *canonical form*; however, the previous caveat should clear up any resulting confusion. Another cautionary note is that we will present a number of examples; some will be written in the more familiar infix functional notation while others will be written in prefix notation. The meaning should be obvious.

We first drop the requirement that the premise part of a gbf must be a propositional variable. Instead, the premise may be an arbitrary function which is a predicate.

Let FL be a function of one or more arguments which returns as its result the value of its final argument.

A COND is mapped onto a general conditional form (see above definition) which is in turn replaced by its equivalent gbf. Symbolically, we have:

$$(\mathrm{COND}\ (p_1\ e_1)\ (p_2\ e_2\ e_3)\ \ldots\ (p_n\ e_n)) \equiv (p_1 \rightarrow e_1, (p_2 \rightarrow \mathrm{FL}(e_2, e_3), \ldots (p_n \rightarrow e_n, F) \ldots))$$

The undefined term corresponds to NIL which is henceforth represented by F. Note the use of the FL construct to denote a sequence of computations whose result is the value of the final computation. We define the base predicates in LISP to be the functions EQ,ATOM, and EQUAL which are known to return T or F. All gbfs whose predicate part is not one of the previous, are replaced using the following transformation:

$$(\mathrm{predicate} \rightarrow \mathrm{conclusion}, \mathrm{alternative}) \equiv (\mathrm{EQ}(\mathrm{predicate}, F) \rightarrow \mathrm{alternative}, \mathrm{conclusion})$$

All occurrences of these predicates in the premise position of the gbf are termed explicit occurrences. All other occurrences are termed implicit occurrences and are replaced by their equivalent via use of axiom (4) - i.e. predicate p is replaced by (p→T,F). This is motivated by the definition of a canonical form where the propositional variables have now been replaced by the more general concept of a predicate.

LISP also has associated with it the predicates AND, NEQ, NEQUAL, NOT, NULL, and OR which are combinations of EQ, EQUAL, as well as negations. Since the canonical form is in terms of the base predicates, we convert these to their definitions in terms of EQ, EQUAL, T, and F and generate conditional forms as follows:

$$\begin{aligned}
\mathrm{AND}(A,B) &\equiv (A \rightarrow B, F) \\
\mathrm{AND}(A_1, A_2, \ldots, A_n) &\equiv (A_1 \rightarrow (A_2 \rightarrow (\ldots (A_{n-1} \rightarrow A_n, F), \ldots), F), F) \\
\mathrm{NEQ}(A,B) &\equiv (\mathrm{EQ}(A,B) \rightarrow F, T) \\
\mathrm{NEQUAL}(A,B) &\equiv (\mathrm{EQUAL}(A,B) \rightarrow F, T) \\
\mathrm{NOT}(A) &\equiv (\mathrm{EQ}(A,F) \rightarrow T, F) \\
\mathrm{NULL}(A) &\equiv (\mathrm{EQ}(A,F) \rightarrow T, F) \\
\mathrm{OR}(A,B) &\equiv (A \rightarrow T, B) \\
\mathrm{OR}(A_1, A_2, \ldots, A_n) &\equiv (A_1 \rightarrow T, A_2 \rightarrow T, \ldots, A_n \rightarrow T)
\end{aligned}$$

An internal lambda of the form:

```
((LAMBDA (var$_1$ var$_2$ . . . var$_n$) <function body sequence>)
 <function body of var$_1$ binding>
 <function body of var$_2$ binding>
                 .
                 .
                 .
 <function body of var$_n$ binding>)
```

is represented by the form:

```
FL(SETQ(var$_1$,<function body of var$_1$ binding>),
   SETQ(var$_2$,<function body of var$_2$ binding>),
                 .
                 .
                 .
   SETQ(var$_n$,<function body of var$_n$ BINDING>),
   FL(<function body sequence>))
```

Note that all lambda variables are given unique names to avoid errors at a later stage when bindings will be used instead of the variable names.

A PROG, which may contain RETURN and a restricted GO, is represented in a similar manner to an internal lambda once the GO and RETURN constructs have been eliminated. PROG variables are treated in the same manner as the lambda variables (i.e. generate unique names and all other properties attributed to lambda variables), and are initialized to NIL (actually F) via the SETQ construct. Also, the final physical statement of the body of the PROG is NIL (actually F), unless the final PROG statement is RETURN, which by definition is the value returned by a PROG when a normal exit is taken rather than by means of a RETURN. The restriction on GO is that the target label must not have occurred physically in the body of the PROG prior to the occurrence of the GO to the label. In this case, the GO is replaced by the remainder of the PROG starting at the specified label. Handling of the more general case of GO would be along the lines of a function call as outlined in the definition of a PROG. This is left for future work by a more sophisticated system. At this point an example is in order. Consider:

```
(DEFPROP EXAMPLE (LAMBDA (A B)
                   (PROG (X Y)
                         (SETQ X A)
                         (COND (B (GO TAG1))
                               ((EQ A B) (RETURN A))
                               ((EQUAL A B) (SETQ Y B)))
                         (SETQ X (CONS X Y))
                    TAG1 (SETQ Y (CONS Y X))))
 EXPR)
```

is replaced by:

```
(DEFPROP EXAMPLE (LAMBDA (A B)
                  (PROG (X Y)
                        (SETQ X A)
                        (COND (B (SETQ Y (CONS Y X))
                                 NIL)
                              ((EQ A B) A)
                              ((EQUAL A B) (SETQ Y B)
                                           (SETQ X (CONS X Y))
                                           (SETQ Y (CONS Y X))
                                           NIL)
                              (T (SETQ X (CONS X Y))
                                 (SETQ Y (CONS Y X))
                                 NIL))))
 EXPR)
```

Notice how a COND at the outer level has appended to each of its results, which is not a GO or a RETURN, the code that will be executed once the COND terminates. We also point out the use of T to handle the situation when none of the conditions hold.

Another feature present in LISP which does not have a parallel in the presentation of the canonical form is the concept of a variable and assignments made to it. In proving equivalence we will want to make sure that SPECIAL variables are assigned their appropriate values; however, local variables and variables associated with internal lambdas (henceforth referred to as *lambda variables*) exist only as placeholders for computations. The act of assignment is only temporary and thus is not part of the equivalence - i.e. in proving equivalence we wish to show that the functions perform the same computations on the LISP environment which means that identical conditions are tested and identical side-effects occur. In the case of local and lambda variables, we will simply use their bindings in the canonical form and ignore the act of assignment. This corresponds to invoking the *Rule of Replacement*. In the case of SPECIAL variables we will use their bindings as well as record the act of assignment.

Note that SPECIAL variables may be assigned new values by use of the SET and SETQ functions as well as via side-effects of invoked functions. Thus we need to know which functions have as their possible side-effects the modification of SPECIAL variables.

As an example, consider the meaningless function given in Figure 3.2 expressed in meta-LISP. The same function is given in Figure 3.3 using LISP. The name of the function is FF and it has three local variables, A, B, and C. H is a SPECIAL variable. The general boolean form representation of the function is given in Figure 3.4. Notice the use of the FL construct to denote a function whose value is the value of its final argument.

```
EXPR FF(A,B,C);
IF NULL CDR(C) THEN A
ELSE IF ATOM(A) THEN LAMBDA(D);
        IF A EQUAL B THEN
                IF ATOM(B) THEN C
                ELSE D
        ELSE H←D CONS H;
        (FF(A,B,H))
ELSE FF(A←CDR A,
        IF NULL A THEN A
        ELSE CDR C CONS CDR C,
        A);
```

Figure 3.2 - Example Encoded in MLISP

```
(DEFPROP FF
 (LAMBDA(A B C)
  (COND ((NULL (CDR C)) A)
        ((ATOM A)
         ((LAMBDA(D)
           (COND ((EQUAL A B) (COND ((ATOM B) C) (T D)))
                 (T (SETQ H (CONS D H)))))
          (FF A B H)))
        (T
         (FF (SETQ A (CDR A))
             (COND ((NULL A) A) (T (CONS (CDR C) (CDR C))))
             A))))
EXPR)
```

Figure 3.3 - Figure 3.2 Encoded in LISP

```
(((EQ (CDR C) F)→F,T)
 →(((ATOM A)→T,F)
   →(FL (SETQ D (FF A B H))
        (FL (((EQUAL A B)→T,F)
             →(((ATOM B)→T,F)
               →C,
                D),
              (SETQ H (CONS D H))))),
    (FF (SETQ A (CDR A))
        (((EQ A F)→F,T)
         →(CONS (CDR C) (CDR C)),
          A)
        A)),
  A)
```

Figure 3.4 - General Boolean Form for Figure 3.3

3.C Flow Analysis

In the previous section we saw the need for knowledge as to which SPECIAL variables are modified by a function. As we shall see later in the duplicate computation phase of the canonical form algorithm, this is only a small part of the information that we shall require. In fact, we wish to have the following information for all functions (directly or indirectly):

(1) Does the function access *head* parts of s-expressions?
(2) Does the function access *tail* parts of s-expressions?
(3) Does the function modify *head* parts of s-expressions?
(4) Does the function modify *tail* parts of s-expressions?
(5) Does the function perform a CONS operation?
(6) Which SPECIAL variables are accessed by the function?
(7) Which SPECIAL variables are modified by the function?

This information is obtained directly for each function by examining its LISP definition. We also record the names of all the functions invoked by each function. At this point we take the transitive closure of our data to obtain the information specified by (1)-(7) above for each function. This corresponds to a worst case analysis.

As seen in the previous paragraph, the type of flow analysis performed poses limitations on the knowledge of the effects of functions. Some of these limitations are further outlined below:

(1) When a function reads and modifies SPECIAL variables or s-expressions, no distinction is made as to which operation was performed first.

(2) When determining reading and modification of s-expressions and or SPECIAL variables, then no distinction is maintained between the conditions under which the operations were performed. In fact, it could be the case that the operations are mutually exclusive - i.e. only one of them could occur.

(3) Whenever a function performs a CONS operation, then a subsequent call to the function with the same parameters is assumed to return a different value since one of the effects of a CONS operation is to grab a cell from the free storage list. In a system where CONS is hashed, such a problem does not exist. Thus we see, that in order to maintain generality, we have once again assumed that the worst could happen.

(4) No knowledge is used of the types or structure of arguments to a function - i.e. the arguments could be known to be of an arithmetic nature, lists, etc.

(5) No knowledge is used of the conditions under which the function is called - i.e. no information is known about the caller's state.

3.D Numbering Scheme

In the process of obtaining a canonical form we will be using the distributive law for functions and conditions. This will mean that certain computations, namely conditions, will be moved so that physical position will no longer indicate the sequence of computation. In order to maintain a record of the original sequence of computation we need a representation of the LISP program in terms of the order in which computations are performed. What is really desired is a numbering scheme having the characterization that associated with each computation is a number with the property that all of the computations predecessors have lower numbers and the successors have higher numbers (i.e. a partial ordering). This property can be achieved by numbering a conditional form in the following fashion where each time a number is assigned, it is higher than any number previously assigned. Actually when the numbers are assigned we will start with an initial number equal to the number of local parameters+8. Also numbers are assigned in increments of two. The motivation for the latter will become clear as the rest of the discussion develops while the former is somewhat arbitrary. Furthermore, atoms which do not correspond to variable names (e.g. T and F) are assigned a computation number of zero and likewise for all arguments to QUOTE.

(1) an atomic symbol is assigned a number.

(2) a function f(arg_1,arg_2,. . . ,arg_n) is numbered in the order arg_1, arg_2, . . . , arg_n, followed by assigning a number to f.

(3) a general boolean form (p→q,r) is numbered in the order p, q, r.

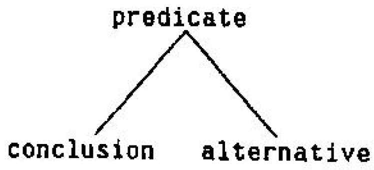
As an example consider the sample function given in Figure 3.2. The numeric representation corresponding to the symbolic representation given in Figure 3.4 is shown in Figure 3.5.

```
(((15 (13 11) 0)→0,0)
 →(((19 17)→0,0)
  →(57 (29 0 (27 21 23 25))
        (55 (((35 31 33)→0,0)
             →(((39 37)→0,0)
               →41,
                43),
              (53 45 (51 47 49))))),
    (85 (65 59 (63 61))
        (((69 67 0)→0,0)
         →(79 (73 71) (77 75)),
          81)
        83)),
  87)
```

Figure 3.5 - Numeric Representation Corresponding to Figure 3.4

3.E Revised Canonical Form Algorithm

At this point we are ready to present the revised canonical form algorithm. In describing the algorithm, we recall our characterization of the canonical form as a tree whose non-terminal nodes correspond to predicates and whose terminal nodes are computations that denote results. In fact, we will often refer to a conclusion part of a gbf as the left subtree and the alternative part as the right subtree. Graphically, (predicate→conclusion,alternative) ≡



The algorithm can be decomposed into two parts, each of which uses a certain set of axioms. The first part corresponds to the application of step (1) of the original strong equivalence canonical form algorithm along with the distributive law for functions while simultaneously binding variables to their proper values. During this process axioms (2), (3), and (7) are used. The second part of the algorithm corresponds to an extension of step (3) of of the original strong equivalence canonical form algorithm to get rid of duplicate occurrences of predicates as well as redundant computations. The latter corresponds to computations whose equivalents have already been encountered. This will be seen to be useful in the matching phase of the proof procedure as discussed in Chapter 5. This process makes use of axioms (2), (3), (5), and (6). As noted earlier, axioms (1) and (8) are not necessary in this phase.

Note that axiom (4) has not been used in either the original or revised algorithms. This is not surprising since it is only applicable to predicates, and from our tree characterization of the canonical form we see that we do not want to replace a subtree of the form (p→T,F) by p. However, during the first part of the algorithm, this axiom might be useful in making sure that every subexpression π in (π→α,β) consists of a single predicate. Nevertheless, the axiom need not be used here since axiom (7) used in combination with axioms (2) and (3) is equivalent to axiom (4) as shown in the following example:

```
((p→T,F)→a,b) ≡ (p→(T→a,b),(F→a,b)) by axiom (7)
              ≡ (p→a,(F→a,b))       by axiom (2)
              ≡ (p→a,b)             by axiom (3)
```

Therefore, the first part of the algorithm accomplishes the elimination of assignments to local and LAMBDA variables by always using the most recent bindings of these variables. We also try to use the most recent binding of SPECIAL variables. However, in this case we may not disregard the act of assignment as in the case of local and internal LAMBDA variables. If via flow analysis it is determined that a function, say FH, may cause an assignment to occur to a SPECIAL variable, then associated with any subsequent occurrence of the said variable (until the next function that may cause an assignment to the said variable) is a number equal to one plus the computation number associated with FH.

For example, the symbolic and numeric results of the application of the first part of the canonical form algorithm to Figure 3.4 are given in Figures 3.6 and 3.7 respectively using a tree-like form. Notice that assignments to local and internal LAMBDA variables have been eliminated. The distributive law for functions has been invoked. The computation numbers associated with the SPECIAL variable H in the assignment (SETQ H (CONS D H)) are one higher than the highest computation number associated with the local variables to the function for the first occurrence, and one higher than the computation number associated with (FF A (CDR B) H) for the second occurrence. The former is because there is no evaluation in progress - i.e. we are merely obtaining an address (see Section 2.B4). The latter is because the last assignment to H is made by (FF A (CDR B) H). Also observe that the computation number associated with H in (FF A (CDR B) H) is one higher than the highest computation number assigned to a local variable to the function because this is the value of H upon function entry.

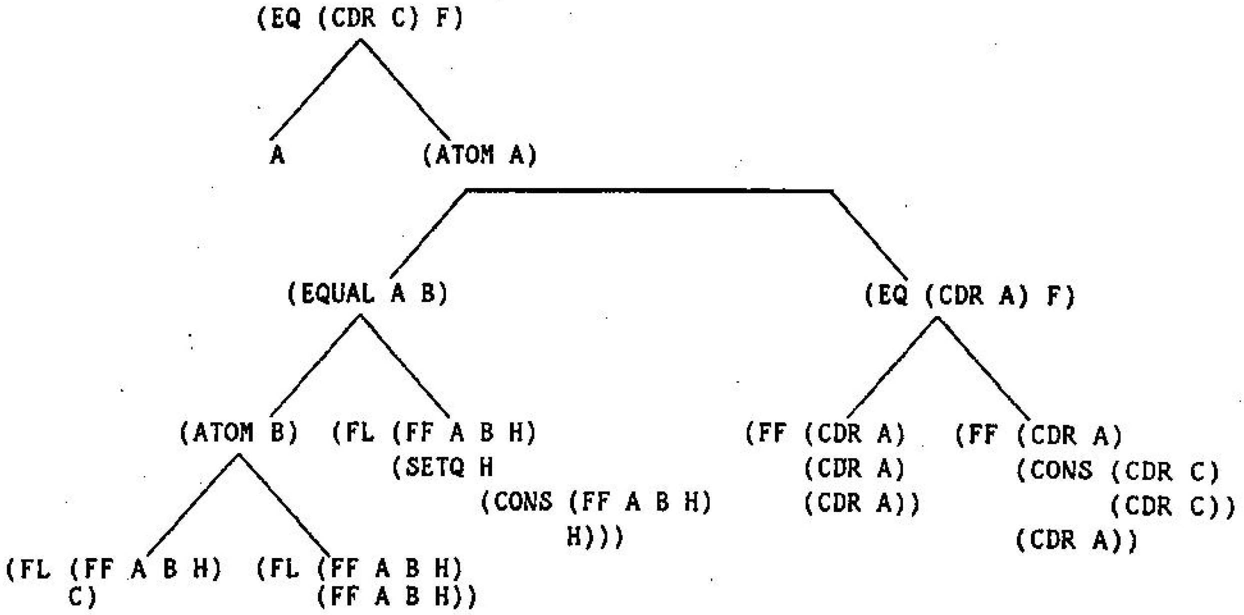

(EQ (CDR C) F)
A
(ATOM A)
(EQUAL A B)
(EQ (CDR A) F)
(ATOM B)
(FL (FF A B H) (SETQ H (CONS (FF A B H) H)))
(FF (CDR A) (CDR A) (CDR A))
(FF (CDR A) (CONS (CDR C) (CDR C)) (CDR A))
(FL (FF A B H) C)
(FL (FF A B H) (FF A B H))

Figure 3.6 - Symbolic Result of Part 1 of the Canonical Form Algorithm

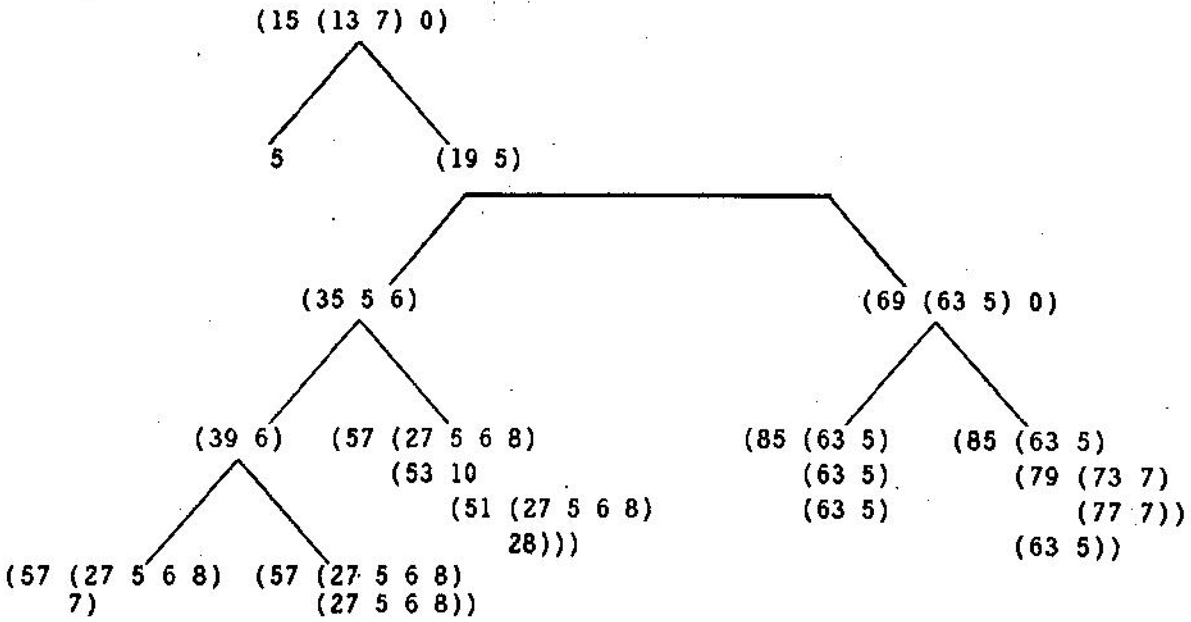


Figure 3.7 - Numeric Result of Part 1 of the Canonical Form Algorithm

Before proceeding to develop the duplicate computation removal algorithm, we need to further elaborate on some of the constructs still present in the result of the binding and condition-expanding algorithm. As noted earlier, the concept of assignment is twofold - there is an act of assignment and also an act of returning a result. For the duplicate computation removal phase we would like to decouple the two. This stems from a realization that the acts of assignment may in fact be redundant as indicated in the definition of SET and SETQ, and to a lesser extent RPLACA and RPLACD. Therefore, the previous four operations are replaced by the following FL constructs where each FL is assigned a computation number of 0 since it does not denote any actual act of computation.

```
    (SET A B) ≡ (FL (FS A B) B)
   (SETQ A B) ≡ (FL (FG A B) B)
 (RPLACA A B) ≡ (FL (FA A B) B)
 (RPLACD A B) ≡ (FL (FD A B) B)
```

The first argument to the FG construct is assigned a computation number one higher than the highest number associated with the local variables to the function whose canonical form is being obtained. This should be familiar as the computation number assigned to the variable initially and is done in this manner because the address of the variable has not changed.

Of course, there are other functions that have side-effects that can be decoupled from their results. For example, PUTPROP returns as its result the value of its second argument. However, in this analysis we are only concerned with the previous four functions since the likelihood of their duplication is much greater than other functions.

The output of the binding and condition expansion algorithm has the form of a tree where all non-terminal nodes correspond to predicates. However, we may have FL constructs interspersed within our computations. We wish to replace all FL constructs not appearing as the outermost function in a terminal node, say N, by their value (i.e. the last argument) and place the remaining arguments in the outer level of FL constructs in all terminal nodes of the subtrees of N. For example:

```
((EQ A (FL (G A) B)→F,(G B))) would be replaced by
((EQ A B)→(FL (G A) F),(FL (G A) (G B)))
```

Next, solely for efficiency and cosmetic considerations, we would like to replace each occurrence of the construct FL by the function FN - a function of two or more arguments whose result is the value of its first argument. We arbitrarily assign zero as the computation number associated with FN since it does not denote any actual act of computation.

Application of the previous transformations to the symbolic and numeric representations given in Figures 3.6 and 3.7 results in Figures 3.8 and 3.9 respectively.

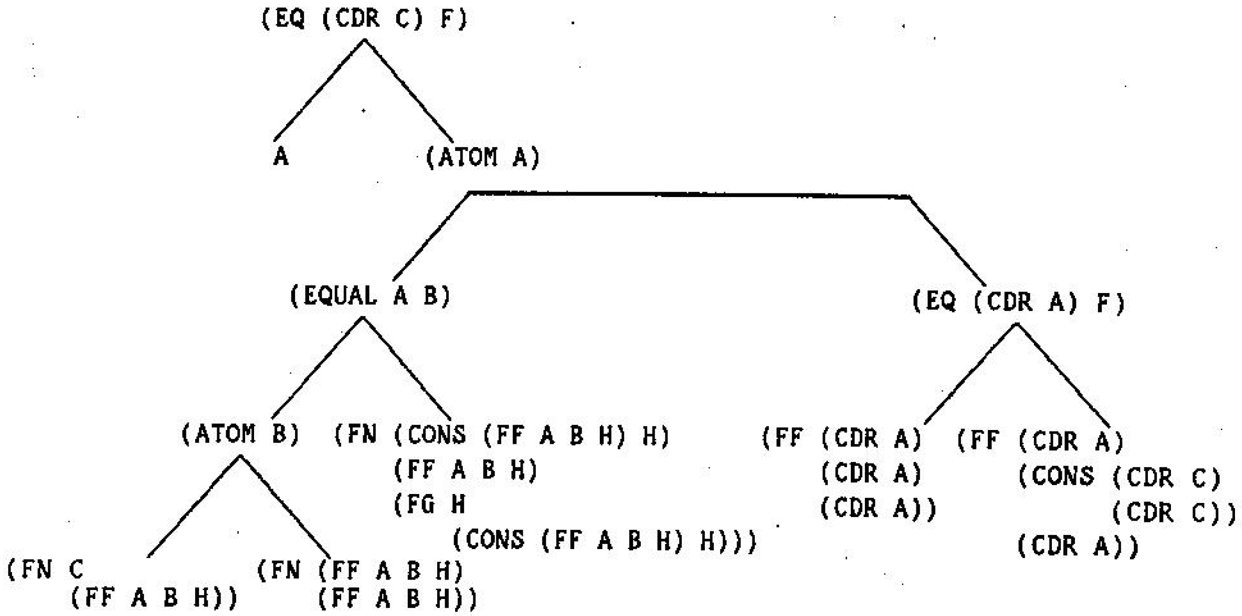


Figure 3.8 - Symbolic Result of Assignment Removal and FN Insertion

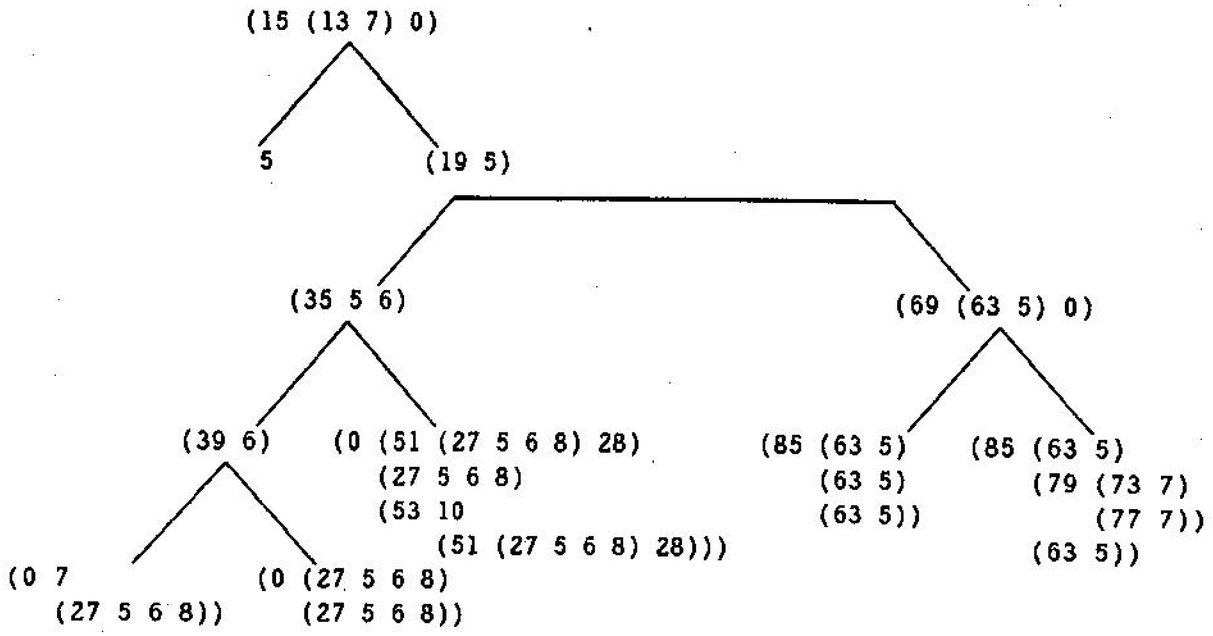


Figure 3.9 - Numeric Result of Assignment Removal and FN Insertion

We have seen that the numeric representation of a LISP function has the property that associated with each constituent computation is a number which is greater than the numbers associated with the computation's predecessors and at the same time less than the numbers associated with the computation's successors. This approach, henceforth referred to as *breadth first*, was necessary in order to properly execute the binding and condition expansion part of the canonical form algorithm (also the only possible numeric representation prior to its execution). The duplicate computation removal phase, and more importantly the matching phase (see Chapter 5), requires an even stronger criterion. We wish the numeric representation to have the afore-mentioned properties plus the property that all computations with the same computation number have been computed simultaneously. By simultaneously, we do not necessarily require computation at the same location. This criterion is not strong enough. Basically, what we are after is the following: If two identical computation numbers appear in two different subtrees, then the functions associated with them must have been computed with the same input conditions and equivalent arguments. For example, consider the following conditional form where FUNC is a function; and A,B,C,D are atoms (i.e. local parameters).

```
(FUNC((EQ A B)→C,D))
```

A possible numeric representation assigned to this form is:

```
(22((16 12 14)→18,20)).
```

After applying the distributive law for functions (i.e. the first part of the canonical form algorithm) we have:

```
((EQ A B)→(FUNC C),(FUNC D))
```

with the numeric representation:

```
((16 5 6)→(22 7),(22 8)).
```

Note that the same computation number, 22, is associated with the two instances of FUNC. However, the function FUNC yields different results for the two instances since in one case the argument is C and in the other case the argument is D. Thus we wish to have different computation numbers for the two instances of FUNC. Of course, if C and D were equivalent, then the two instances of FUNC could have the same computation number; this type of question is dealt with in the matching phase.

Recalling our characterization of the canonical form as a tree, we see that the numbering scheme that we require, known as *depth first*, has the property that all computations performed solely in the right subtree have a higher computation number associated with them than the numbers associated with computations performed solely in the left subtree. In fact, this is the basis of the algorithm given in Appendix 8 to convert a breadth first numeric representation to a depth first numeric representation.

Thus for the above example we would want the following numeric representation:

((16 5 6)→(22 7),(24 8)).

As a more substantial example see Figure 3.10 which is a depth first renumbering of the numeric representation given in Figure 3.9. Note that the absolute values of the numbers are somewhat meaningless (we are basically interested in a partial ordering).

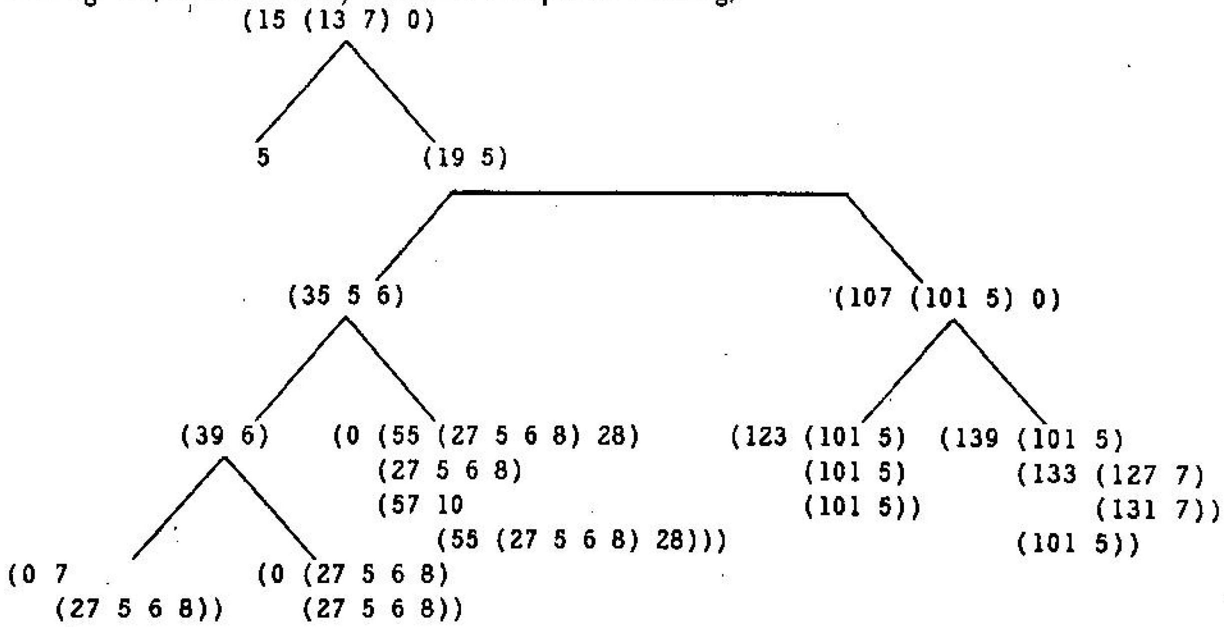


Figure 3.10 - Depth First Renumbering of Figure 3.9

The duplicate computation removal algorithm processes the symbolic and numeric representations of the function in order of increasing computation numbers. The tree-like nature of our representations coupled with the property that all computations performed solely in the left subtree have lower computation numbers associated with them than with those performed solely in the right subtree greatly facilitates our work. We also make use of the fact that application of axiom (7) coupled with the manner in which the distributive law for functions was applied preserved the order in which conditions were tested - i.e. each predicate has a lower computation number associated with it than is associated with the predicates computed in its subtrees. A rough description of the algorithm is given below.

Algorithm:

(1) If the node does not correspond to a terminal node, then go to step (4).

,2) In the order of increasing computation numbers do the following for all computations that have not yet been encountered.

Replace computations that are duplicates of earlier computations by their equivalents.

By duplicate we allow for such concepts as substitution of items that are known to be EQ to one another. This is valid by the *Extension to the Rule of Replacement.*

(3) Exit.

(4) The node corresponds to a predicate.

(4.1) Let HIGH be the computation number associated with the predicate.

(4.2) Process the node and its subtrees in the same manner as step (2) with the added proviso that all computation numbers ≥ HIGH are ignored.

(4.3) Determine if the value of the predicate is already known. This procedure uses information based on all previous tests, properties of functions set forth in Section 2.B, the facts that T and F are atoms, the result of the ATOM predicate is T or F, and the truth of T NEQ F. The actual steps of this procedure with respect to the various predicates are outlined below in steps (4.3.1)-(4.3.3). The algorithms used to maintain and update the data base in an efficient and complete manner are to be found in [Samet74].

(4.3.1) If the predicate is ATOM(arg_1), then ATOM(arg_1) is T if arg_1 is known to be an atom. ATOM(arg_1) is F if arg_1 is known not to be an atom, or if the assumption that arg_1 is an atom leads to a contradiction. The second case is a result of the analysis, performed in Chapter 2, of the implications of EQ, EQUAL, and ATOM.

(4.3.2) If the predicate is EQ(arg_1,arg_2), then EQ(arg_1,arg_2) is T if arg_1 is known to be EQ to arg_2. EQ(arg_1,arg_2) is F if arg_1 is known to be NEQ to arg_2, or if the assumption that arg_1 is EQ to arg_2 leads to a contradiction. Again, the second case is a result of the analysis, performed in Chapter 2, of the implications of EQ, EQUAL, and ATOM.

(4.3.3) If the predicate is EQUAL(arg_1,arg_2), then if ATOM(arg_1) is T then use the EQ test given in step (4.3.2). If ATOM(arg_2) is T, then use the EQ test given in step (4.3.2). EQUAL(arg_1,arg_2) is T if arg_1 is known to be EQUAL to arg_2, and F if arg_1 is known to be NEQUAL to arg_2.

If the value of the predicate is already known, then the symbolic and numeric representations of the arguments to the predicates are attached to the end of the argument lists to the FN constructs at all of the terminal nodes belonging to the tree having the known predicate at its root (see the following paragraph for a justification). The tree having the predicate as its root is replaced by its left subtree in case the value of the predicate was T, or by the right subtree in case the value of the predicate was F. This corresponds to an application of axioms (5) and (6) respectively. In either case reinvoke the algorithm to process the remaining subtree and exit when through.

(5) Reinvoke the algorithm to process the left and right subtrees and exit when through.

Arguments to redundant predicates are placed on the FN argument list because we do not want to lose a record of the act of computing them. An example is the test (EQ (G A) (G B)) where A is known to be EQ to B and G is a function which is known to return the same value when invoked twice with the same argument (see Section 2.B5). In this case, the predicate is redundant, yet a record of the act of computing (G A) must still be kept since the function could have had side-effects or perhaps not terminate. Actually, we only need to place the arguments of a redundant condition on the FN list when the predicate was EQ or EQUAL, and the arguments were not already in the corresponding FN lists. Moreover, they need only be added to the FN lists which do not already contain them.

In step (5) we also update the data base to reflect the value of the condition for each subtree as follows:

(ATOM A): In the true case (ATOM A) is set EQ to T and all computations EQ or EQUAL to A become EQ to each other. In the false case (ATOM A) is set EQ to F.

(EQ A B): In the true case A is set EQ to B and all computations EQ to A or B become EQ to each other. In the false case A is set NEQ to B.

(EQUAL A B): In the true case A is set EQUAL to B and the limited transitivity property of EQUAL, as outlined in the definition of EQUAL in Section 2.B4, is applied. In the false case A is set NEQUAL to B.

Briefly, the data base for keeping track of predicate values consists of equivalence classes for the items known to be EQ and likewise for EQUAL. The negations of these relations are stored in terms of pairs. Contradiction is used to prove inequality. Thus the functions EQ and EQUAL never appear as entries in the data base. Whenever an EQ relation is true, then transitivity is applied as well as functional implication which is defined by example as follows: If the computations G(A) and G(B) have been encountered at some time prior to encountering A EQ B, then the truth of A EQ B may imply the truth of G(A) EQ G(B) provided conditions for EQ of functions of identical arguments computed at different instances hold (see Section 2.B5).

As an example of the duplicate computation removal algorithm, see Figures 3.11 and 3.12 which demonstrate the result of the application of the algorithm to the symbolic and numeric representations given in Figures 3.8 and 3.10 respectively. Notice that the predicate (ATOM B) was found to be redundant in the case that (CDR C) was not EQ to F, A was an atom, and A was EQUAL to B. This is because of the unique representation of atoms. We should also be aware that (CDR C) is only computed once when computing (CONS (CDR C) (CDR C)) as well as in the test (EQ (CDR C) F).

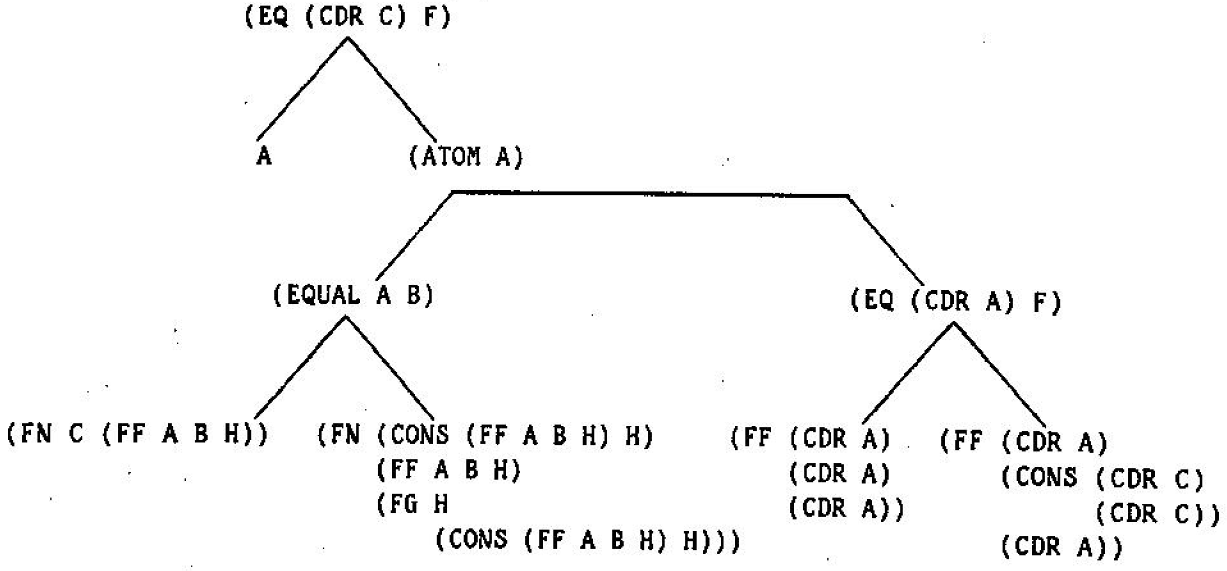


Figure 3.11 - Symbolic Result of Duplicate Computation Removal

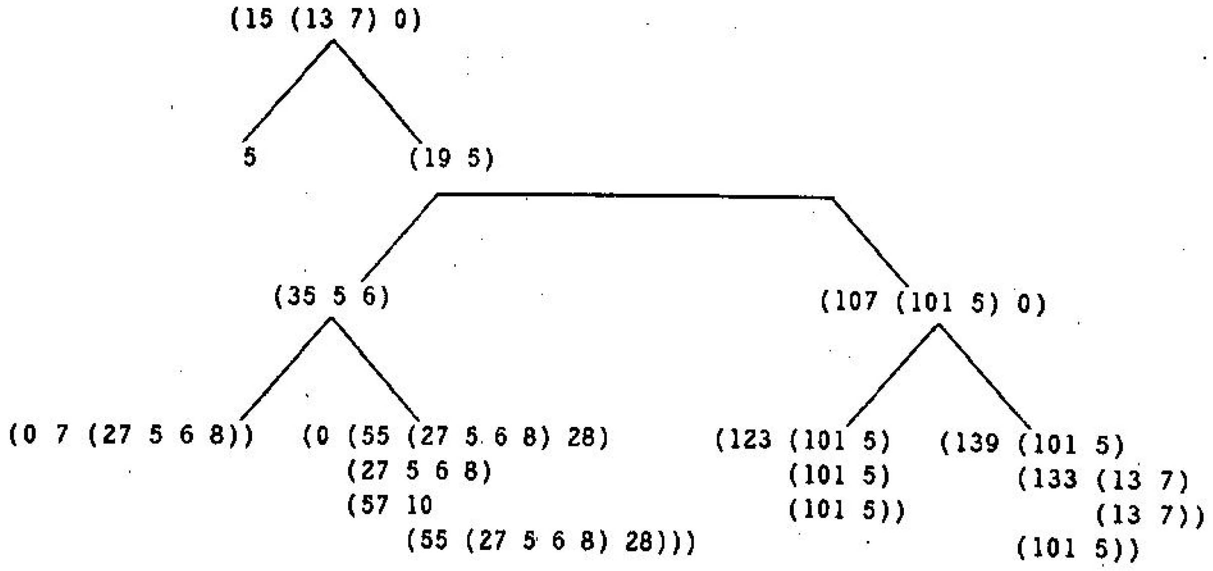


Figure 3.12 - Numeric Result of Duplicate Computation Removal

In the definition of the assignment operators SET, SETQ, RPLACA, and RPLACD we mentioned the possibility of redundant first cases as well as redundant second cases of the act of assignment. The duplicate computation removal algorithm disposes of the latter but not the former. This is alleviated by performing a variant of the duplicate computation removal algorithm which will assemble the various assignment operations and remove first occurrences only if there exist assignment operations in all subsequent computation paths which will make this first instance redundant. For example, consider the pair of forms below where A is a SPECIAL variable:

```
((EQ (SETQ A B) C)→(SETQ A D),E)                (1)
((EQ (SETQ A B) C)→(SETQ A D),(SETQ A E))       (2)
```

Here we see that in (1) the first act of assignment cannot be considered redundant in all subsequent paths, while in (2) the first act of assignment can be considered redundant. This phase of the canonical form algorithm need only be performed once in the derivation of the canonical form. In all subsequent manipulations, such as the application of axiom (8) in rearranging the order of application of conditions during the matching phase, this step is unnecessary because it failed to be applicable once, and it can not become applicable by the act of rearranging the testing of conditions since this act does not result in the creation of any new computations. Thus the first act of assignment will still fail to be redundant.

Once the duplicate computation phase is done, we remove all computations occurring as non-result arguments to FN that are subexpressions of other computations appearing in predicates or as arguments to the FN function. By subexpression we mean a symbolic and numeric equivalence - i.e. it is not enough for the two symbolic representations to match. Remember, all duplicate computations have been replaced by their earlier occurrences by virtue of step (2) and thus a numeric match is a must. At the same time, we also remove all occurrences of atoms occurring as non-result arguments to the FN construct. This work is done merely to simplify the canonical form and to insure that an FN construct only denotes computations not seen elsewhere in the path for which the FN node serves as a terminal node.

For example, application of this step to Figures 3.11 and 3.12 results in the elimination of the occurrence of (FF A B H) as an argument to the FN construct when (CDR C) is not EQ to F, A is an atom, and A is not EQUAL to B. This is shown in Figures 3.13 and 3.14.

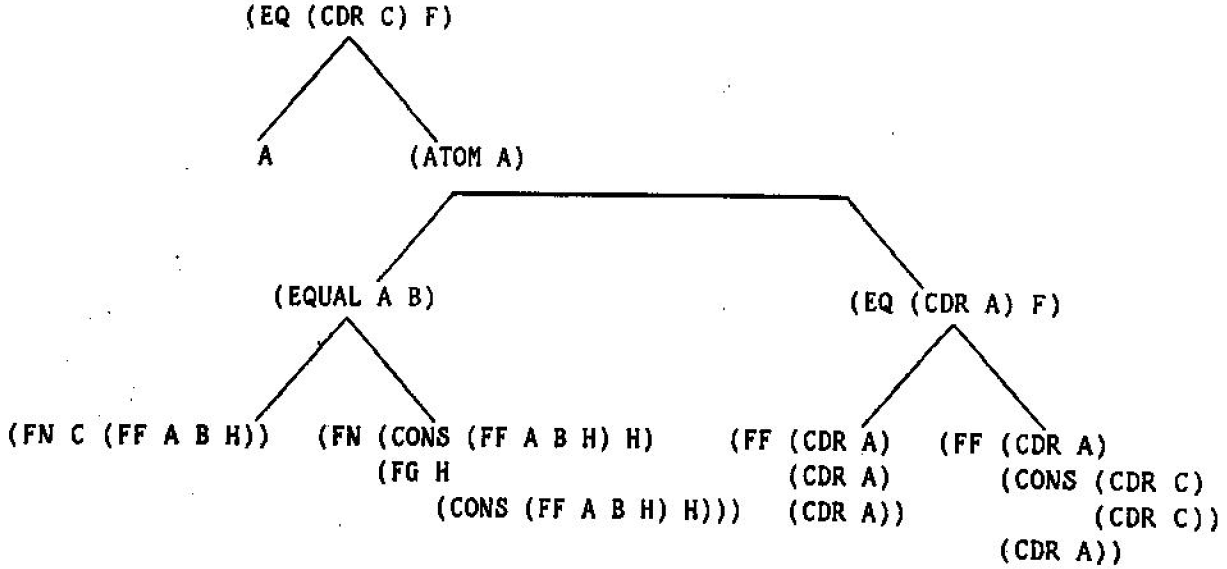

(EQ (CDR C) F)
A
(ATOM A)
(EQUAL A B)
(EQ (CDR A) F)
(FN C (FF A B H))
(FN (CONS (FF A B H) H) (FG H (CONS (FF A B H) H)))
(FF (CDR A) (CDR A) (CDR A))
(FF (CDR A) (CONS (CDR C) (CDR C)) (CDR A))

Figure 3.13 - Symbolic Result of FN Removal

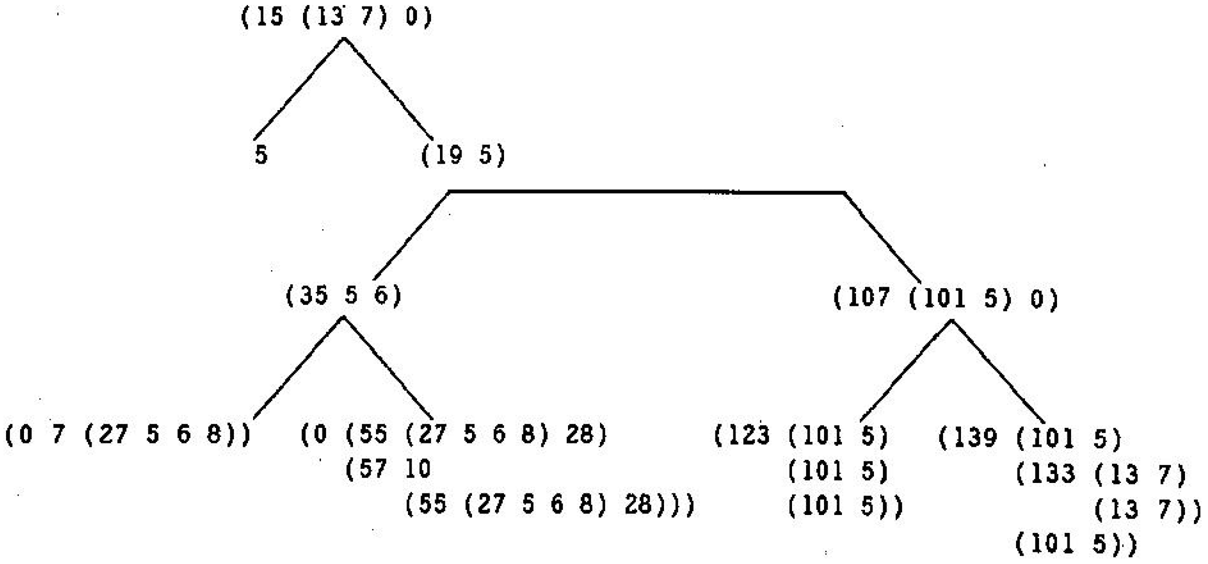


Figure 3.14 - Numeric Result of FN Removal

CHAPTER 4

THE PROGRAM UNDERSTANDER

4.A Introduction

In order to prove the equivalence of the translation of a LISP program to a LAP program we must be able to *understand* the LAP program. *Understanding* is a term which can be abused, and it is also one of the main distinctions between our system and what is commonly known as decompilation. The distinction that we wish to make is that no use is made of syntax in the assembly language encoding to indicate how some constructs in the LISP are encoded in assembly language. By *understanding* we mean a representation of the program, as executed by the computer, in an environment dictated by the source program. This representation is nothing more than a transcript of all computations performed by the assembly language program that have an analog in LISP. This rules out the presence in the representation of address computation, data shuffling, and noise operations such as the accessing of locations about whose contents nothing is known.`

The environment, LISP in our case, poses many restrictions on the type of operations that can be performed by the system. These restrictions range from the definitions of functions and protection of the List Structure outlined in Chapter 2 to what will be termed well–formedness of the program. Well–formedness includes certain stipulations on the flow of control, the structure of the assembly language code, and, most importantly, proper interfacing with other functions (i.e., the observance of calling sequence conventions).

The representation of the assembly language program is basically in the form of a tree identical to the canonical form, which was also a tree. However, in order to distinguish the two, we shall call the process of *understanding* a program *rederivation*, and the output of the rederivation procedure will be known as the *rederived form*. The actual rederivation process consists of a symbolic execution of the program as set forth by the LAP encoding of the LISP function.

Symbolic execution consists of activating a set of procedures which correspond to the instructions in the LAP program (quite similar to interpretation). These procedures specify how each instruction affects what is known as the computation model (e.g. procedural embedding[Winograd71]). This model reflects the contents of various data structures relevant to the execution of the program, as well as the values of conditions tested. Thus we see a need for a capability to describe a computer instruction set. This description must provide for data types as well as a control structure for the symbolic execution procedure. By control structure we mean the ability to invoke various parts of the assembly language program, as is the case when processing a condition, a branch, or a function call.

The use of symbolic execution is the distinguishing factor between our system and decompilation methods[Hollander73]. The latter are designed to return a representation of the object program in a format identical to the source program. Such methods operate by looking for a syntax in the assembly language program. This is much akin to pattern recognition. The trouble with such methods is that they imply that the decompiling program must know how the various constructs of the high level language encoded in the low level language. This sets a limit to the variation of the object code presented to such a system. A more serious flaw is the fact that compilation is a many to many process. Namely, the object program corresponding to a program written in a high level language can be encoded in many different equivalent ways. Similarly, to an object program there corresponds more than one equivalent source program. This is because most high level programming languages have built in redundancies that allow duplication or non–unique program specification (i.e. internal lambda construct in LISP). We shall see that in our system, there is no need for specifying how a particular construct is encoded, since our internal representation is simply a record of the computations performed. In other words our system is built on the semantics of the various assembly language instructions in terms of their effect on a computation model.

In the process of describing the rederivation process we try whenever possible to shy away from depending on a particular computer or internal representation of a LISP program. This is quite difficult from both pedagogical and feasibility standpoints. In our case, we will use the PDP–10 and its instruction set (see Appendix 3 for a description of the instructions used in the examples) as examples of a computer implementation. In fact, this is the machine for which the system was designed. However, the reader should not misconstrue this as indicating that such a system would not work on another computer. In order to lay such misunderstandings to rest we present a set of constraints on our discussion from the standpoint of computer architecture. In other words, if the architecture constraints are satisfied, then the system can easily be changed to handle an architecturally compatible system. It will be seen that these architecture constraints are quite reasonable. Also recall, that LAP is merely a way of expressing the format of the instruction and does not need to be dependent on a particular computer (see Appendix 1 for a description of the LAP format).

The remainder of this chapter expands on the concepts presented in the previous paragraphs. Examples of optimizations are used to clarify the discussion. The use of optimizations is meant both as a motivation and as an illustration of the power of the rederivation process. We first describe the architectural and implementation constraints. Second, the various data types necessary to keep track of the effects of the instructions are defined. Next the machine description process is presented, followed by an explanation of the symbolic execution process and its limitations, and by a motivated outline of restrictions placed on the structure of the LAP code. We culminate our expansion with a description of the actual procedure for obtaining the rederived form. This procedure will typically involve at least two passes over the LAP program and we indicate the conditions for successful or unsuccessful termination.

4.B Architecture and Implementation Constraints

We assume that the computer has two's complement arithmetic and has a hardware stack (or at least that a stack can be simulated via suitable operations in LAP). The existence of a stack also implies that there exists a data structure known as a stack descriptor, in our case a stack pointer. In the examples we shall use accumulator P (numerically 12) as the location containing the stack pointer. Note that other locations may also contain a stack pointer; however, accumulator P will be used for interfacing between the various functions. Thus all consistency checks with respect to the well–formedness of stack operations are performed on accumulator P. We shall henceforth refer to it as the stack pointer.

Our implementation of LISP uses a stack pointer format such that when the stack is initially allocated, the left half of the stack pointer contains the negative of the size of the contiguous data area associated with the stack, and the right half contains a pointer to a location one below the base of the stack. For example see Figure 4.1 which shows the stack pointer immediately after the allocation of an N element stack whose first entry is location A+1.

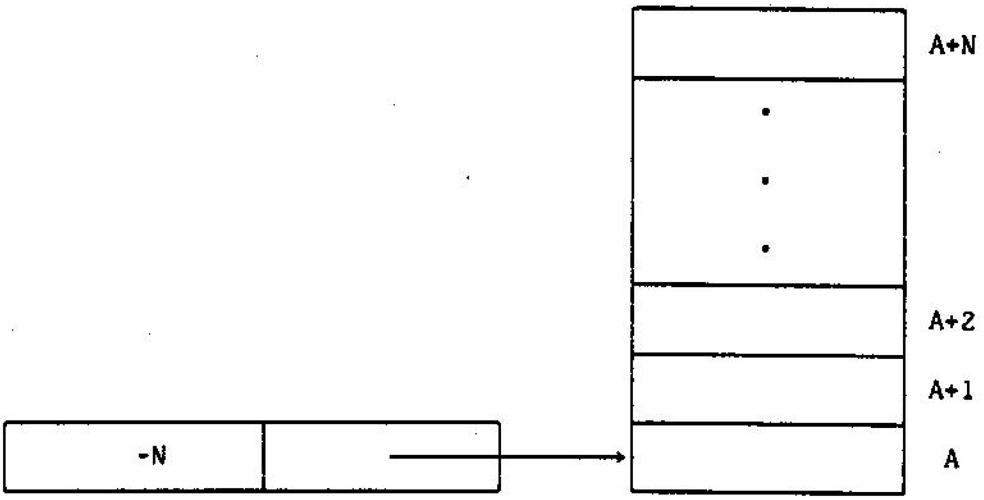


Figure 4.1 - Sample Stack Pointer

Whenever a stack entry is allocated (by use of stack allocating instructions such as PUSH or PUSHJ), both halves of the stack pointer are increased by one. Similarly, whenever a stack entry is deallocated (by use of stack deallocating instructions such as POP or POPJ), both halves of the stack pointer are decremented by one. Thus we see that a count value of zero in the left half can be used for detection of stack overflow or underflow (but not both). We have opted for stack overflow detection by virtue of the negative count in the left half of the stack descriptor. Clearly, the use of a positive count would yield stack underflow detection.

All function calls use the stack as a means of implementing recursion. In other words, whenever a function call occurs, the return address is pushed on the stack. This concept is extremely useful and allows for interesting constructions as shown in the following two examples.

Suppose that a function tests a number of conditions, and that based on these conditions, other functions are executed (similar to a case statement). Moreover, upon termination, all of these functions must return to a common point and execute a segment of code. For example, see Figure 4.2 where a function epilogue is illustrated. This can be implemented by executing a branch to the desired location of the common code sequence after each of the function calls. However, a more efficient approach is to push the address of the common code sequence on the stack prior to testing the first condition, and to invoke the functions in the various cases via simple branch (non stack) instructions. When the invoked functions terminate, they will return via the stack. Thus the size of the program has been reduced by a number of instructions equal to one less than the number of conditions, with virtually no increase in execution time. Actually the difference in execution times is the difference between a PUSH plus an unconditional jump and a PUSHJ plus an unconditional jump.

Another example occurs when it is desired to enter a function at a point, say TAG, other than its starting address, and at TAG the stack is known to contain some entries. The lowest entry is a return address. In order to have correct operation, the code sequence responsible for the entry at midstream must make sure that the return address is in its proper place. This is done by entering the return address on the stack prior to placing the other elements on the stack and by using a branch instruction to enter the function. The usefulness of such a feature is quite obvious when employing a calling sequence with parameters on the stack. In fact we will use this as an optimization for recursive functions that need not go to the start of the function in certain instances of recursion (see the optimized encoding of the function HIER in Chapter 6).

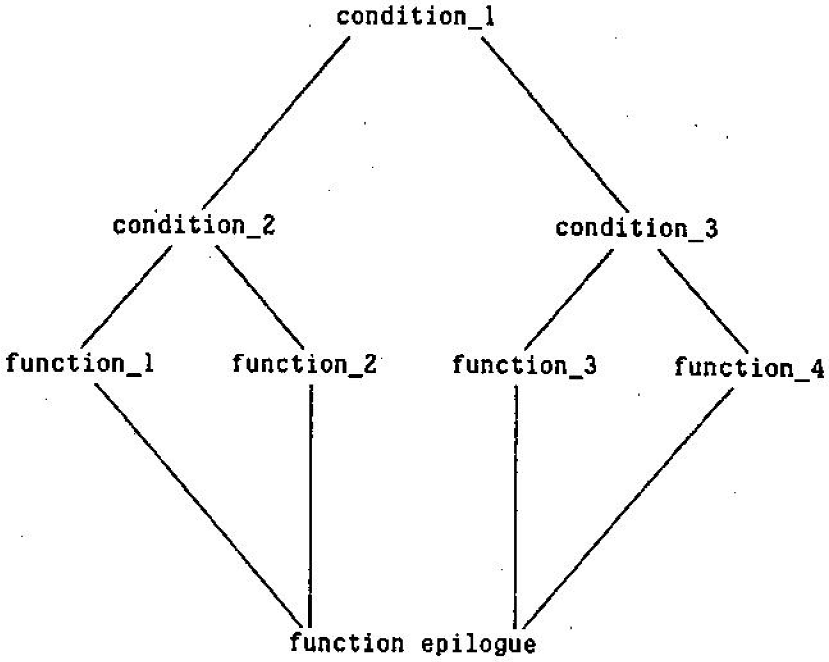


Figure 4.2 - Function Epilogue

The description and implementation assume the capability of breaking up a computer word into halfwords. Moreover, each computer word represents one cell in the List Structure where one half denotes CAR and the other denotes CDR. The exact match of halfword to CAR and CDR is a parameter to the rederivation system, as is the length of a word in bits. In this discussion we assume that the left half corresponds to CAR and the right half to CDR, and that the length of a word is thirty-six bits. For future work we would like to be able to allow the use of two words to represent a cell in the List Structure (i.e. one word for CAR and another word for CDR). If contiguous locations were used, then CDR could be recognized by an address of CAR+1. This would mean that limited arithmetic on LISP pointers would be allowed. This issue is further explored in Section 4.F.

Finally, our description and implementation assumes that NIL is represented by 0. This is not crucial, but it is extremely useful in the optimization of tests. The lack of such a representation means that tests for NULLness would require a comparison against the atom NIL and would typically require an extra instruction. Thus 0 has a dual representation as both a LISP pointer and as a data pointer (see the discussion on data types in the next section). However, it will always be interpreted in the manner most amenable to a legal rendering of the context in which it appears.

4.C Data Types

In order to make sense of the various instructions we must have some knowledge of their operands. Specifically, when compositions of operations are performed, we must have a useful way of expressing the intermediate results. We solve this problem by stipulating that each halfword has a data type. These types allow us to describe the contents of various locations in a manner which renders subsequent operations using them as data meaningful. For example, certain locations are known to contain LISP pointers. In such a case, we would like to keep track of the operations performed on the contents of these locations. Other useful types are related to internal data structures such as stack pointers, bits, labels, address constants, half words, instructions, numbers, and unknown (locations about whose contents nothing is known). In the case of stack pointers, labels, and unknown, the type *data* is particularly useful since it allows us to keep track of arithmetic operations. Basically, in these cases the value associated with the entity of type data acts as a relocation constant and the normal conventions governing the behavior of assemblers with respect to relocation arithmetic also hold in our system[†]. In the case of bits, half words, and instructions the value associated with a specific data type merely acts as a different representation of a number with an allowance for conversion between the synonyms. Address constants are somewhat dual to unknown since they represent a location whose address is unknown while at the same time its contents are known. Finally, a very useful property of types, of which we shall be making frequent use, is the capability it gives us to detect illegal operations by virtue of a mismatch of types of operands.

4.C1 LISP

The values of the parameters to a LISP function at function entry, and the contents of the SPECIAL cells are represented by the type LISP, denoted by LP. These values are known as LISP pointers. As the LISP program is symbolically executed all LISP functions of LISP pointers result in LISP pointers. Each cell pointed at by a LISP pointer, say A, contains CAR A in the left half and CDR A in the right half. Thus whenever we obtain or access the left half of a cell pointed at by a LISP pointer, then we are computing CAR of the LISP pointer, and similarly for a right half and the CDR operation.

4.C2 Stack

As indicated in the previous section, a stack pointer contains a count in its left half and an address in its right half. We denote a stack count by the type SC and a stack address by the type SA. Upon function entry the value of both the stack count and stack address is (relative) zero. This enables us to detect illegal accesses to locations belonging to the caller (i.e. negative stack address), locations not yet allocated (i.e. keep track of maximum number of stack entries allocated), and to insure that upon function exit the stack is of the same depth as it was upon function entry. Furthermore, we are able to perform computations involving stack addresses. This is particularly useful when a stack is deallocated via use of arithmetic operations rather than the customary POP operation, since we are able to detect resulting illegal stack pointers (i.e. negative values).

[†] The algorithm for the addition and subtraction of full words in terms of the various data types is given in Appendix 5 where two's complement arithmetic is assumed. This includes a treatment of carry and borrow between half words.

4.C3 Data

Non-LISP numbers and symbolic addresses of instructions are represented by the type data, denoted by DP. These entities are useful in computing new addresses, in which case a new symbol is generated. Note that addresses cannot be used as numbers except in subtraction of addresses. The same rules used in assemblers for computing relocatable addresses apply here. The only addition is a special treatment of the numeric constant -1 which corresponds to all bits being 1 in a word and is useful in handling borrow terms in subtraction. The address constant, which represents a pointer to an unknown location with a known value given in the format of a LAP instruction (i.e. three or four fields), is also of type data. Recall that numbers occurring in the LISP program appear QUOTEd since they correspond to atoms which are not to be evaluated.

4.C4 Half Word

The half word type, denoted by LH, is used for describing a half word in terms of three fields corresponding to the fields of a LAP instruction that originally appear in the left half of the instruction. These fields correspond to the opcode (with an @ symbol suffixed if the indirect address bit is on), accumulator, and index fields.

4.C5 Bit

The bit type, denoted by DB, is useful for describing the contents of words for bit testing operations. Bit tests using a mask are basically no different than regular tests (e.g. a test for equality against zero). The only difference is that the effective address or its contents indicates which bits are to be zero. With this in mind, an operation making use of a mask containing ones in the positions which are to be tested, is represented as a sequence of NIL and 0. The latter indicates that the bit is to be zero while the former indicates a don't care condition. Now, it is clear that a bit testing operation is analogous to a test against 0 with the selected bits tested rather than an entire word or half word. Note that the two's complement representation of numbers implies added significance for -1, since -1 corresponds to a mask with all bits being 1.

4.C6 Unknown

The unknown type, denoted by NIL, represents locations whose contents are unknown. The value of such a location is a list containing the name of a location whose contents are unknown (possibly the same location), a halfword identifier (i.e. left or right), and a unique number. Note that the location, say A, may contain a value of type unknown which is different than the name A. This results when entities of type unknown are moved from one location to another.

4.D Machine Description

The basic function of a computer is to operate on data. These operations are a combination of retrieval, test, modification, and storage. Our description is decomposed into two parts. We first describe what we loosely term the memory. This corresponds to the source and destination of the retrieval and storage operations. It includes both internal and environmental data structures. The former correspond to machine related constructs such as accumulators, index registers and the stack, while the latter reflect the constructs which owe their existence to the LISP environment, such as the List Structure and SPECIAL cells. We next present a formalism, in the form of a programming language, for describing the operations of the computer in terms of the internal data structures. Some of the basic primitives are examined as well as the procedure for extending the system to include instructions not initially implemented. These extensions are followed by examples of their usage in optimization which indicate some interesting properties of the machine description process.

4.D1 Memory

Memory can be viewed as consisting of two parts, locations that can be overwritten and those that cannot. The latter part consists of the area containing the program to be executed. Clearly, overwriting should be forbidden since we assume that a program remains the same. Otherwise an equivalence proof is somewhat meaningless, due to the recursive nature of the functions with which we are dealing. Thus we do not allow self-modifying code. Note that the contents of all locations in memory may be read.

The locations that can be overwritten include the accumulators (all of which serve as index registers in the implementation that we have considered), List Structure, SPECIAL cells, and the stack. The exact restrictions on overwriting are further clarified in a section describing restrictions placed on the LAP code. Briefly, we note that elements in the List Structure may only be overwritten with LISP pointers and the affected locations cannot be atomic. Thus a location in the List Structure that is being overwritten by an inline operation must be provably non-atomic. This places a rather severe restriction on inline RPLACA and RPLACD operations. Since in our implementation location 0 is both an accumulator and a pointer to the atom NIL, we see that it cannot be overwritten at all.

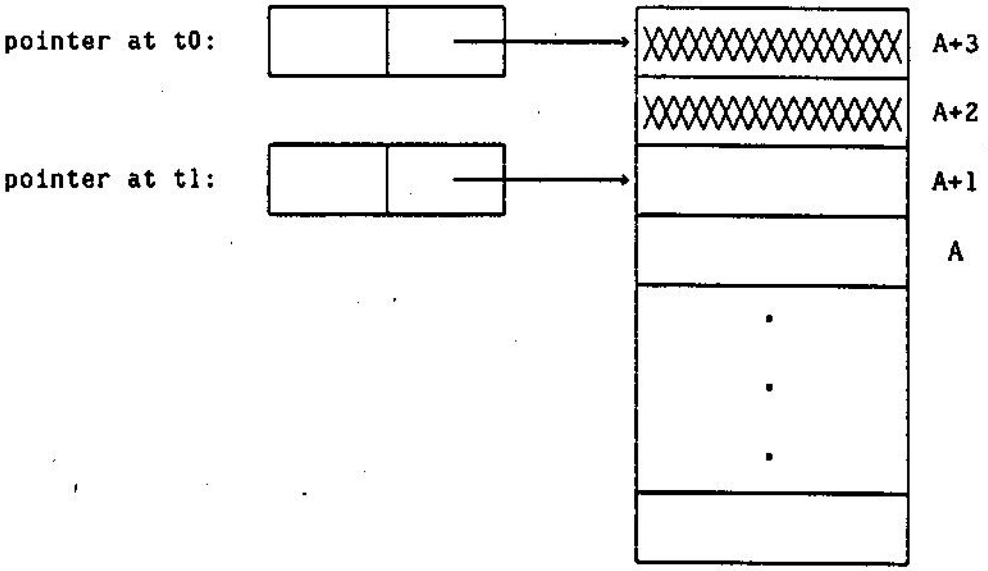


Figure 4.3 - Sample TOPSTACK Data Area

The stack and some of the accumulators are useful as a temporary data areas. We also introduce an additional temporary data area known as TOPSTACK. This area consists of locations that have been allocated on the stack and have later been deallocated. In other words, these are the positions between the current top of the stack as indicated by the stack pointer and the maximum address that has been pointed at by the stack pointer. These locations, unlike those below the current value of the stack pointer, do not have their contents preserved when external function calls occur. However, the concept is quite useful within a function and will be seen as an aid in a subsequent discussion of recursive calls to locations other than the starting address of the function. Figure 4.3 illustrates this concept by means of two stack pointers, at times t0 and t1 (t0<t1), pointing at locations A+3 and A+1 respectively. In this case at time t1 locations A+2 and A+3, shown cross hatched, denote the current TOPSTACK data area.

Associated with all locations that can be overwritten is a five-field descriptor indicating vital information regarding the contents of the location. Note that the smallest unit of memory is a halfword and thus we have a descriptor for each halfword. Each field, and a motivation for its existence, is described below:

1. TYPE: Indicates the data type contained in the half word.

2. DATA: Indicates the value of the the data contained in the halfword. This includes the symbolic representation for LISP pointers and the name of a location for unknown pointers. All other data types have as their data field entry the appropriate values mentioned in the section dealing with data types.

3. NUMBER: This field is only applicable to LISP pointers and unknown pointers. In the former case it contains the numeric representation of the LISP pointer (recall the dual representation of a LISP computation discussed in Chapter 3). In the latter case it contains a unique number generated when the unknown value was placed in the location. This number is a property of the unknown value in the location. Thus if the contents of the location is moved to another location, then the number is also moved.

4. REFERENCE: This field is only applicable to a location whose content is a LISP pointer. It is used to indicate whether or not the content of the location has been referenced. By referenced we mean that it has been used in some computation. Such information is useful in detecting when computations involving LISP pointers have been performed purely for their side-effects - i.e. their value is not used as an argument to a subsequent function. The maintenance of a reference field is necessary so that we can maintain a record of all LISP computations performed along an execution path. The following describes the mechanics of keeping track of reference information. Each time an instruction is symbolically executed a unique number is generated higher in value than all previous numbers. This number is placed in the reference field of the descriptor of the location that is being referenced. The reference value is a property of a location as long as its contents do not change. Whenever a value is stored into a location, the location's reference field is set to NIL which denotes that the location has not yet been referenced with the new value as its contents. This treatment compensates for the movement of data between locations since the motion involves a source and a destination and the act of transfer causes a reference to be made of the source, while the destination gets the new unreferenced value.

5. STORAGE HISTORY: The storage history field belongs to the location. Each time any value is stored in the location, the value of the program counter is recorded. This is useful in detecting certain errors in the well-formedness of the program, since it allows the detection of the instant when a location receives its illegal value. Some examples are invalid stack pointer formats, missing return addresses on the stack, and errors relating to improper interfacing with other functions.

Many of the errors result from a violation of the criteria set forth in a subsequent section detailing restrictions on the structure of the LAP program. Appendix 4 contains a listing of some of the errors that we are currently able to detect.

4.D2 Machine Description

The machine instructions are described via the use of procedures in a programming language quite similar in appearance to the class of register transfer languages[Bell71] common in hardware descriptions. Our language is a subset of MLISP[Smith70] which is a parentheses free LISP (also known as meta-LISP). In appearance it is very much like ALGOL[Naur60] with the exception that the underlying data structure is the set of s-expressions. The user is provided a number of primitives for describing such aspects as the operand fetch cycle, data transfer, control, predicate testing and the sense of a test. In addition, there exists the concept of a variable. Variables are declared in a routine via the construct NEW and are local to the declaring procedure. There is also a variable, PCG, which is used to keep track of the program counter. The symbolic execution system is aware of this variable and insures that it is incremented as well as kept up to date when executing various branches of condition testing instructions. Predeclared constants are also available. These include X11 which is a word containing a data pointer of value 1 in each halfword, and ZEROCNST which is a word containing a data pointer of value 0 in each halfword.

An instruction is described by a one argument procedure of type FEXPR which has the same name as the instruction. The argument represents a list bound to the CDR of the LAP "word" containing the instruction. In other words the parameter to each procedure is a list comprising the accumulator, address, and index fields of the LAP word. Hence there is one procedure with indirect addressing and one without. The value of the accumulator field is accessed by the function ACFIELD, and the combined value of the address field modified by the contents of the accumulator denoted by the index field (if nonzero) is accessed by the function EFFECTADDRESS. In case the instruction is one of the special UUO's† (i.e. CALL, JCALL) used for function calls, then the name of the called function is obtained by use of the function CALLADDRESSFIELD. The function calling UUOs refer implicitly to the stack pointer which is located in an accumulator denoted by REGSTKPTR.

The description of instructions that test conditions is aided by the primitive CHECKTEST which checks if the value of the condition is already known. CHECKTEST returns NIL if the value of the condition is unknown and the ordered pairs (CONS T T) or (CONS T NIL) in case it is known to be true or false respectively. The sense of the test performed is denoted by the primitives TRUEPREDICATE and FALSEPREDICATE. In case the value of the condition is unknown, CHECKTEST will set a special variable, inaccessible to the user, to the predicate being tested. We shall later see that at this point, the only possible test is for equality (against a LISP pointer or zero which represents NIL).

The primitives UNCONDITIONALJUMP, UNCONDITIONALSKIP, and NEXTINSTRUCTION specify unconditional flow of control with the obvious meaning. The first has one argument, the address to which control is to flow. Condition testing instructions imply a transfer of control based upon a condition value. In this case we provide for the symbolic execution of the two paths. CONDITIONALJUMP and JUMPALTERNATIVE are used to indicate respectively the case that the condition is true and the case that it is false. The two

† UUO (Undefined User Operation) is a PDP-10 acronym for a supervisor call.

primitives must be applied in the order described for reasons to become clear when the symbolic execution process is examined. The arguments indicate the original parameter to the instruction and the name of the routine used to execute the remainder of the condition. The routine is necessary for the purposes of indicating flow of control as well as any further properties of the instruction that depend on the result of the test. Similarly, we have the constructs CONDITIONALSKIP and SKIPALTERNATIVE.

In order to demonstrate the description process, we show how the instructions MOVE, HLRZ@, PUSHJ, SKIPE, JUMPN, and TDZN are handled. Immediately preceding each instruction we also present a verbal description of its effect. We let AC denote the name of the accumulator specified by the accumulator field of the instruction.

MOVE: The same as a load operation, which moves the contents of the effective address into accumulator AC.

```
FEXPR MOVE(ARGS);
LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));
```

HLRZ@: Load the left half of the contents of the effective address using indirect addressing into the right half of AC and clear the left half of AC.

```
FEXPR HLRZ@(ARGS);
LOADSTORE(ACFIELD(ARGS),
          EXTENDZERO(LEFTCONTENTS(INDIRECT(CONTENTS(EFFECTADDRESS(ARGS))))));
```

PUSHJ: Add octal 1 000 001 to AC to increment both halves by one, and place the result back in AC. If addition causes the count in AC left to reach zero, then set the Pushdown Overflow flag. Store the contents of PCG (i.e. the program counter which has already been incremented) in the location now addressed by the right half of AC and take the next instruction from the effective address. Actually, this instruction saves the incremented value of PCG on the stack and resumes execution at the location denoted by the effective address.

```
FEXPR PUSHJ(ARGS);
BEGIN
        NEW ADDRESS;
        ADDRESS←EFFECTADDRESS(ARGS);
        ALLOCATESTACKENTRY(ACFIELD(ARGS));
        ADDX(<ACFIELD(ARGS),X11>);
        LOADSTORERIGHT(RIGHTCONTENTS(ACFIELD(ARGS)),FORMRETURNADDRESS(PCG));
        LOADSTORELEFT(RIGHTCONTENTS(ACFIELD(ARGS)),FLAGSPOINTER());
        UNCONDITIONALJUMP(ADDRESS);
END;
```

SKIPE: Skip the next instruction if the contents of the effective address is equal to zero. If the accumulator name specified by the AC field is non-zero, then load the accumulator specified by AC with the contents of the effective address.

```
FEXPR SKIPE(ARGS);
BEGIN
        NEW MEMG,TST;
        MEMG←CONTENTS(EFFECTADDRESS(ARGS));
        IF ACFIELD(ARGS) NEQ 0 THEN LOADSTORE(ACFIELD(ARGS),MEMG);
        TST←CHECKTEST(MEMG,ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                          ELSE NEXTINSTRUCTION());
        TRUEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION SKIPETRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION SKIPEFALSE);
END;

FEXPR SKIPETRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR SKIPEFALSE(ARGS);
NEXTINSTRUCTION();
```

JUMPN: Jump to the location indicated by the effective address if the contents of the accumulator specified by AC is unequal to zero; otherwise continue with the next instruction.

```
FEXPR JUMPN(ARGS);
BEGIN
        NEW TST;
        TST←CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                          ELSE UNCONDITIONALJUMP(EFFECTADDRESS(ARGS)));
        FALSEPREDICATE();
        CONDITIONALJUMP(ARGS,FUNCTION JUMPNTRUE);
        JUMPALTERNATIVE(ARGS,FUNCTION JUMPNFALSE);
END;

FEXPR JUMPNTRUE(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

FEXPR JUMPNFALSE(ARGS);
NEXTINSTRUCTION();
```

TDZN: Zero the bits in the accumulator specified by the AC field corresponding to the bits that are 1 in the contents of the location specified by the effective address and skip the next instruction if any of the bits in the AC were one.

```
FEXPR TDZN(ARGS);
BEGIN
        NEW ACG,MEMG,TST;
        MEMG←CONTENTS(EFFECTIVEADDRESS(ACFIELD(ARGS)));
        ACG←CONTENTS(ACFIELD(ARGS));
        LOADSTORE(ACFIELD(ARGS),SETMASKEDBITS(ACG,MEMG,0));
        TST←CHECKTEST(ACG,MAKEMASK(MEMG));
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                           ELSE NEXTINSTRUCTION());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,TDZNTRUE);
        SKIPALTERNATIVE(ARGS,TDZNFALSE);
END;

FEXPR TDZNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR TDZNFALSE(ARGS);
NEXTINSTRUCTION();
```

Thus we see that the user works with primitives which are formulated in terms of data types relevant to the environment in which the machine is executing. He need not worry about coding the primitives. This should enable such a correctness system to be more easily transported to an architecturally similar computer. Also the user need not indicate what features the instructions are used to encode. Appendix 2 contains a description of the primitives, and Appendix 3 includes the encoding in terms of these primitives of each of the instructions which we have implemented. We also give a verbal description of the instructions as a means of documenting each procedure or set of procedures.

At this point we give an example of how the system can be extended to handle two additional instructions, JRA and TLNN. We also demonstrate how they can be used without indicating in the instruction description that this is the manner of use.

JRA has the following description:

```
FEXPR JRA(ARGS);
BEGIN
    LOADSTORE(ACFIELD(ARGS),CONTENTS(LEFTCONTENTS(ACFIELD(ARGS))));
    UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));
END;
```

The effect of the instruction is to place the contents of the location addressed by the left half of the accumulator denoted by the AC field into the same accumulator. The next instruction is taken from the effective address and operation continues from there.

Now, suppose the instruction (JRA AC LABEL) has been seen, where LABEL is the immediately following instruction and accumulator AC contains the contents of a LISP cell, say A. In other words AC contains pointers to CAR A and CDR A in its left and right halves respectively. The effect of this particular instance of the instruction is to load the left and right halves of AC with CAAR A and CDAR A respectively and process the following instruction. However, nowhere in our description of the instruction did we make any provision for its use in such a manner.

TLNN has the following description:

```
FEXPR TLNN(ARGS);
BEGIN
        NEW TST;
        TST←CHECKTEST(LEFTCONTENTS(ACFIELD(ARGS)),
                      MAKEMASKHALF(EFFECTADDRESS(ARGS)));
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                           ELSE NEXTINSTRUCTION());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,TLNNTRUE);
        SKIPALTERNATIVE(ARGS,TLNNFALSE);
END;

FEXPR TLNNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR TLNNFALSE(ARGS);
NEXTINSTRUCTION();
```

The effect of the instruction is to skip the next instruction in sequence if the bits in the left half of the accumulator denoted by the AC field corresponding to 1s in the effective address are not all equal to zero. Otherwise, the next instruction in sequence is processed.

Now, suppose the instruction (TLNN AC -1) has been seen where accumulator AC contains the contents of a LISP cell, say A. In other words AC contains pointers to CAR A and CDR A in its left and right halves respectively. The effect of this particular instruction is to test if CAR A is EQ to NIL and skip the next instruction if true. Once again we see that nowhere in our description of the instruction did we make provision for its use in such a manner.

4.E Symbolic Execution

Symbolic execution corresponds to executing a procedure for each instruction. The procedure updates the computation model for the effect of the instruction and also performs control operations. Instructions that test conditions check if the condition value is already known based on results of previous tests and flow analysis. If this is the case, then the appropriate path is processed. Otherwise, both paths are processed. Therefore there is an implicit control structure for trying all possible paths from the start of the program. The procedure will terminate because whenever we get to a location that we have previously seen on the path we interpret it as recursion. This solves the looping problem. Otherwise a path terminates when the function is exited either normally or abnormally. Actually, if we get to a location previously encountered, then we try to prove that there exists some path from the beginning of the program leading to the location which can be bypassed. If no such path exists, then the result of the previous path is unknown (treated as an abnormal exit). The correctness of the symbolic execution process is a question of the correctness of the procedures encoding the various instructions.

All computations involving LISP functions are recorded since they can generally be distinguished from overhead computations. The only stumbling block is in distinguishing between calculations of addresses and calculations of data. However, numbers have a distinct representation in LISP which is not the raw number (i.e. represented as an atom), and appears as (QUOTE number). This means that there is a separation between program and data which sort of runs counter to the Von Neumann concept[Burks71] of indistinguishability between the two.

The result of the symbolic execution process is a representation of the program in the same format as the canonical form - i.e., a symbolic and numeric representation of predicates followed by the result and all computations not in tests or results (non result arguments to FN constructs are denoted by UNREFERENCED). The computation model consists of the accumulators, stack, SPECIAL cells, values of conditions tested in the form of an equality data base as outlined in Chapter 3, and the list UNREFERENCED.

Elements are added to UNREFERENCED when computations with side-effects such as RPLACA, RPLACD, SET, and SETQ are executed. In this case we record their act of computation in UNREFERENCED with the appropriate FA, FD, FS, and FG constructs as was done in the canonical form. UNREFERENCED also has added to it the contents of memory locations which at one time contained LISP pointers (i.e. results of computations) but were overwritten before any other computation was able to use the LISP pointer previously residing in the location. These computations are typically of a side-effect nature although they are not restricted to this class. They are detected via the use of the reference field of the memory descriptor whenever a location is destroyed that has not been previously referenced. By not previously referenced we mean that either the location has not been referenced with its current value; or if yes, then the reference could not be done by an instruction which is simultaneously destroying the contents of the cell. For example (TDZA 1 1) will set accumulator 1 to zero by referencing it and storing into it. In order to be able to detect that despite such references the previous results of the said location must still be placed on the UNREFERENCED list, unique numbers are generated for each instruction that is symbolically executed. Once the entire program has been rederived, we will apply the same algorithm used in the canonical form to purge an FN list of computations that appear as subexpressions in the predicates or in the result of the path being symbolically executed.

The computation model is updated by the primitives that are used to describe an instruction. There are also constructs which are invisible to the user which modify the model. Upon a function call, all accumulators but those that are known to be preserved by the function, and elements of TOPSTACK are considered to be destroyed This situation causes their contents to be added to UNREFERENCED. Upon return from a function call, certain SPECIAL cells known to have been possibly modified receive new bindings. Similarly, the stack pointer is updated. Upon program exit, the contents of all unreferenced locations is added to UNREFERENCED. The equality data base may also be updated when certain computations take place that imply EQ or EQUAL relations. These include RPLACA, RPLACD, and their implications as well as other operations possessing such properties as antisymmetry and commutativity.

The basic type of non-arithmetic test that can be performed by a computer is a check for equality. All other non-arithmetic tests are modifications of this primitive using certain data structures. This equality is a test against another value or zero. These tests translate quite easily into their corresponding LISP construct EQ. Thus as far as the program rederivation process is concerned, all tests are EQ. We shall soon see how we transform various instances of the EQ predicate into our base predicates EQ, EQUAL, and ATOM. We mentioned earlier that the result of the symbolic execution process is identical in format to the canonical form. The actual construction occurs in the JUMPALTERNATIVE and SKIPALTERNATIVE primitives which upon termination will return a conditional form (predicate→conclusion,alternative). The procedure is recursive so that at the end of the symbolic execution process, we will have a conditional form representing the function.

The symbolic execution process treats the true and false cases of predicates by first determining if any of their arguments are known to be NIL (i.e. F) or T. If this is true and the predicate is identical to the predicates 1-4 in Figure 4.4 (the meaning of EQSUBI in entry 7 is explained in

Section 4.F), then we attempt to recursively apply evaluation followed by cases 1-4 to the argument which is not T or F, say arg, and reversing the sense of the test in case arg was compared against F. When this procedure can no longer be applied we have a predicate in the form of cases 1-8. If the sense of the predicate has not been changed by the process of predicate evaluation, then use columns 2 and 3 of Figure 4.4 to process the true and false cases respectively of the predicate. If the sense of the predicate has been reversed, then use columns 2 and 3 of Figure 4.4 to process the false and true cases respectively of the predicate. The processing, with respect to updating the equality data base, is identical to that performed upon encountering non-redundant conditions in the duplicate computation removal algorithm of the canonical form process.

	predicate	true case	false case
1.	(EQ (EQ A B) F)	(NEQ A B)	(EQ A B)
2.	(EQ (EQ A B) T)	(EQ A B)	(NEQ A B)
3.	(EQ (EQUAL A B) F)	(NEQUAL A B)	(EQUAL A B)
4.	(EQ (EQUAL A B) T)	(EQUAL A B)	(NEQUAL A B)
5.	(EQ (ATOM A) F)	(NOT (ATOM A))	(ATOM A)
6.	(EQ (ATOM A) T)	(ATOM A)	(NOT (ATOM A))
7.	(EQ (EQSUBI A B) F)	(EQ A B)	(NEQ A B)
8.	(EQ A B)	(EQ A B)	(NEQ A B)

Figure 4.4

Results of predicates can only be used in detecting redundancies or other equivalences once the predicates have been tested. Until then, predicate-like functions such as EQ, EQUAL, and ATOM are treated just as if they were normal functions. In other words, the data base is not updated to reflect any possible inferences from these functions, except for the indication that the results of these operations are atoms (remember that their range is T and NIL which are atoms). The inferences that could possibly be made are a result of the predicates being two-valued functions rather than normal multi-valued functions. The two-valued property means that inequality is a far more powerful notion. Specifically, two items not EQ to the same item are EQ to each other, whereas in a multi-valued system this is not true. Hence we see that inequalities (i.e. NEQs) must be propagated fully in order to take advantage of the inferable EQ relations.

Recall from Chapter 3 that the data base for keeping track of predicate values consists of equivalence classes for the items known to be EQ and likewise for EQUAL. The negations of these relations are stored in terms of pairs. Contradiction is used to demonstrate inequality. Thus the functions EQ and EQUAL never appear as entries in the data base

As an example of an undetectable relation suppose FUNC is a function of two arguments with no side-effects and that we wish to determine if EQ(C,D) is EQ to EQ(I,J) given that the following three relations are known to be true:

```
1.  FUNC(EQ(A,B),EQ(C,D)) NEQ FUNC(EQ(E,F),EQ(G,H))
2.  EQ(A,B) EQ EQ(E,F)
3.  EQ(G,H) NEQ EQ(I,J)
```

The answer to this query is unknown since an entry was not made in the data base for the inequality EQ(C,D) NEQ EQ(G,H) after step 2 even though it is implied by the data. This is because we rely on contradiction to detect this fact for us. However, with a two-valued logic, handling of inequalities cannot be deferred. Thus since the EQ(C,D) NEQ EQ(G,H) entry was not made after the establishment of the validity of relation 2, we are unable to derive the fact that EQ(C,D) EQ EQ(I,J) even though it is true. This deficiency cannot be easily alleviated since its remedy would involve examination of all possible arguments of functions and detecting if any elements of the equivalence classes in which they are members are two valued functions. Moreover, the combinatorics are much larger when a function involves more than two arguments. Our main objection to handling such constructs is that it destroys the efficiency provided by the use of contradiction. Namely, contradiction acts as a check on the size and growth of the data base.

The problems associated with unevaluated EQ, EQUAL, and ATOM functions did not arise in the canonical form since we converted each instance of these functions to a predicate followed by application of the distributive law for functions. In fact we will do this again once the entire assembly language program has been rederived (i.e., prior to the matching phase). It should be noted that only inferences from untested predicates are not made. However, whenever an EQ, EQUAL, or ATOM function is evaluated, whether it has been tested or not, then we check if its result is known by the same method outlined in step (4.3) of the duplicate computation removal algorithm in Chapter 3. In the affirmative case we return T or F but no update of the data base is made (i.e. propagate transitivity, etc.). For example, consider the two identities EQ(A,B) EQ EQ(C,D) and EQ(A,B) EQ T where EQ(C,D) has not been tested. If we desire to know if EQ(C,D) is T, then the answer is yes; but, C and D are not in the same equivalence class. In other words no update of the data base in terms of operands of the EQ operator is performed. Thus if E and F were known to be EQ to C and D respectively, then the equality of E and F will not be detected.

4.F Restrictions on Program Structure

We have several restrictions on labels. All labels are local to a function and are not known outside of the function (i.e. they are not accessible to any function except the one in which they appear). Thus each function may only have one entry point for outside functions. Also, we must always be able to determine the target of a jump operation (i.e. the label to be executed next). This rules out a computed GO which is often implemented by use of a jump table.

Whenever the symbolic execution process encounters a label that it has previously seen along a specific path, then it is assumed that recursion has taken place and that the start of the program has been bypassed (e.g. the encoding of NEXT given in Figure 1.21). The system will try to prove that this was the case by showing that the instruction at label LOOP would have been arrived at anyway if in fact the recursive call had been made to the start of the program (i.e to label NEXT). This means that the condition values along the path need not be tested since their values were known. If such a path from the start of the program exists, then it is unique since a predicate cannot be both true and false. All labels which can occur as recursive call entry points must satisfy the property that all forward paths (i.e. paths from the start of the program to the label that encounter the label for the first time) enter the label with the same stack depth. This is not crucial in general, but it does allow us to implement the rederivation procedure without backtracking[Golomb65]. More will be said about the handling of previously encountered labels when the various passes made over the data are discussed in Section 4.G.

```
(LAP START SUBR)
START (PUSH P 1)
      (PUSH P 2)
      (PUSH P 3)
      (PUSH P 4)
        .
        .
        .
      Compute argument for reg.1
      (PUSH P 1)
      Compute argument for reg.2
      (PUSH P 1)
      Compute argument for reg.3
      (PUSH P 1)
      Compute argument for reg.4
      (MOVE 4 1)
      (MOVE 1 -2 P)
      (MOVE 2 -1 P)
      (MOVE 3 0 P)
      (SUB P (C 0 0 3 3))
      (CALL 4 (E START))
         ≡ (PUSHJ P START)
      Continue with rest of
         computation
        .
        .
        .
END   (SUB P (C 0 0 4 4))
      (POPJ P)
```

```
        (LAP START SUBR)
START   (PUSH P 4)
        (PUSH P 3)
        (PUSH P 2)
NEWSTRT (PUSH P 1)
          .
          .
          .
        (PUSH P (C 0 0 RESUME))
            *return address
        Compute argument for reg.4
        (PUSH P 1)
        Compute argument for reg.3
        (PUSH P 1)
        Compute argument for reg.2
        (PUSH P 1)
        Compute argument for reg.1
        (JRST 0 NEWSTRT)
RESUME  Continue with rest of
           computation
          .
          .
          .
END     (SUB P (C 0 0 4 4))
        (POPJ P)
```

Figure 4.5 – Comparison of Accumulator and Mixed Calling Sequences

We assume that a function always returns its result in accumulator 1. The arguments to the function are found in accumulators 1-5 and thus we have a limit on the number of parameters (i.e. five). Another scheme which places them on the stack has no such limitation. The type of calling sequence is not critical to the rederivation process; the system can work for other types of calling sequences. An ideal calling sequence is a combination of accumulators and stack. This is most evident when we examine recursive calls that bypass the start of a program. The trouble with a calling sequence that only uses accumulators is the need to save the contents of accumulators (which may contain the parameters) on the stack while executing function calls. This causes quite a bit of data shuffling. However, a stack calling sequence also has its share of problems not the least of which is a need for more memory to hold the stack which may have to be greatly expanded in size. Extra memory operations are needed for access of computations, and data shuffling is still a characterization when functions call other functions with many of the same variables in the same argument positions. The trouble is that room must be made for the return address. An alternate solution placing the arguments below the return address does not further relieve the problem.

Another solution is to have a separate parameter stack and control stack. This has the disadvantage that much space must be used as well as a constant need to have the parameters on the stack, whereas in a calling sequence using accumulators only the parameters needed for future reference are saved on the stack. The reason a calling sequence making use of accumulators appears so poor is that only rarely does one find compliance with the previous criterion. Most compilers fail to make the distinction between what should and should not be saved on the stack. Thus it is quite common to find that the first thing done by a function is to save all of its arguments on the stack prior to any further execution. Another problem is that arguments are generally computed in the same order in which they are stipulated in the function call, rather than in an order that minimizes saving and restoring from the stack.

For example, consider the function START of four arguments shown in the box in Figure 4.5. On the left is given the compiler generated version with the drawbacks mentioned in the previous paragraph. On the right we demonstrate how a mixed calling sequence can be used to make this function run more efficiently. In this case mixed means that a combination of the accumulators and the stack was used for internal recursion. Also note the fact that the order of computing arguments was rearranged. Such rearranging is only allowed if it can be proved that the a equivalence holds with the original function definition. This topic is the subject of Chapter 5.

In the previous example we see a shift in the location of the parameters to the function at function entry. Thus in the case of entry from outside of the function, accumulators 1-4 will contain the parameters, whereas if the entry was from within the function, then only accumulator 1 contains a parameter. Thus when performing such optimizations, we must prove that accumulators 2-4 were never referenced past the label NEWSTRT with the assumption that they contained the parameters to the function. Moreover, note the rearranging of the order of computing the arguments on the internal recursive call. This may not always be a valid optimization; however, our proof procedure can determine if no harm will result from this type of optimization. The need for rearranging the order of computation was not demonstrated here since the example is in the form of a schema. Nevertheless, it should be quite clear that often such rearranging can save needless saving and restoring of computations from the stack. For example, consider a call to a function requiring two arguments. It is preferable to compute the second argument first, since the result of the first argument is needed for accumulator 1. Figure 4.6 demonstrates the two alternatives. On the left we see that the arguments are computed in the order specified by the function definition; whereas on the right we see that the order of computing the arguments has been reversed.

```
1.  compute first argument               1.  compute second argument
2.  save result on the stack             2.  save result on the stack
3.  compute second argument              3.  compute first argument
4.  move to accumulator 2                4.  restore second argument
5.  restore argument 1 from the stack
```

Figure 4.6 - Computing Sequences for Arguments to a Function Call

If the computation of the first argument could have been done inline, and if the order of computation of the arguments could be reversed, then the operations of saving and restoring of the stack as shown on the right of Figure 4.6 are not necessary (i.e. only steps 1 and 3 and a load of accumulator 2) would be necessary). The power of such calling sequence optimizations is clear, especially in light of the fact that LISP functions spend most of their time in linking. It is for this reason, among others, that a verification system such as that presented and implemented in this thesis is needed:

Several functions have been hand compiled using these methods as a simulation of the type of optimizations that we envision. The savings in space and time were quite substantial. One example, HIER1, which is to be found in Chapter 6, showed that the compiler generated encoding required 40 percent more space for the object program than the hand coded version. Moreover, a timing system was constructed[Samet73] and used to measure the execution time spent within a function other than that in external function calls. At the same time, the contribution of the execution of the function to the stack size in recursive instantiations was also recorded. When the program was executed with some sample data it was found that the maximum stack size was cut in half, and the hand compiled version was 40 percent faster than the compiler generated program.

All offline functions are considered to be LISP functions and thus must have as their parameters valid LISP pointers. Functions performed inline (machine instructions) on LISP pointers must, in general, result in valid LISP pointers. We relax the previous restriction in some cases where the result may be a valid LISP pointer or a valid LISP predicate. We next present a pair of examples of the latter as well as instances where the limitations of our system prevent proper recognition of the operation.

Due to the analogy between an equality test of two items (EQ) and a subtraction followed by a test against 0 we allow the subtraction of LISP pointers. This results in a pair of special LISP functions named EQSUBD and EQSUBI which denote the result of the subtraction of two LISP pointers. The former is used for the borrow element when the two operands of the subtraction operation are full words that contain LISP pointers in their right halves and zeros in their left halves. Otherwise, the result of the subtraction operation in the half word not containing LISP pointers is considered unknown. The suffixes I and D in the functions denote an independent test and a dependent test. This can be seen by noting that when EQSUBI(A,B) is EQ to 0 (i.e. A EQ B), then EQSUBD(A,B) is also EQ to 0. In an implementation where the atom NIL is represented by 0, the result of the subtraction operation in the true case is a valid LISP pointer; whereas in the false case, and also when NIL is not implemented by 0, the result of the operation is not a valid LISP pointer. This means that, generally speaking, a LISP pointer representing the result of an EQSUBI or EQSUBD operation may not be used as a parameter to any other LISP function or returned as a result of a LISP function. In fact, in order to make sure that the EQSUB construct has not been used improperly, we verify this is so once the rederivation process is complete. This means that the EQSUBI construct may only appear as one of two arguments to an EQ, with the other argument known to be EQ to NIL. In this case we replace (EQ (EQSUBI A B) NIL) by (EQ A B). In all other cases, occurrences of EQSUBI and EQSUBD must be capable of being shown EQ to NIL and in fact are replaced by NIL.

In the case of bit manipulation we allow the use of tests with masks which will result in valid LISP pointers. One example is the use of a mask containing 1 in every bit position. This is identical to a test against 0 and can be used to test for NIL in an implementation where NIL is represented by 0, and even to implement an EQSUB type test (i.e. TLNN, TLNE, etc.). Another example is to zero all bits that are 1 in a word as is done with a TDZA operation whose mask is equivalent to the contents of the location to be zeroed. Finally, consider the TDZN instruction which combines the previous two operations. When used with a mask containing a 1 in every bit position TDZN will set the tested location to zero as well as skip the next instruction if the previous contents of the accumulator were non-zero. This is equivalent to setting a location to NIL and branching if it was previously non-NIL. Clearly, the instruction is useful in implementing a NOT test. Note that the reason for our ability to detect these operations is the observed analogy between bit testing and checking equality to zero.

In these examples we see that the operations may, under certain conditions, yield valid LISP pointers or results meaningful in LISP. However, they all have one feature in common; they are not compositions of non-LISP functions. This is an important characterization, since the main reason for making them valid is the duality in representation of the constant 0 and the atom NIL. This can be checked for rather easily when there are no compositions. However, more complicated operations which could have a deeper consequence than the NIL duality would require a theorem prover for the domain. In the case of arithmetic operations, a comparison of the result of two EQSUB operations is equivalent to an *equivalence* operation. For example, EQSUBI(A,B)=EQSUBI(C,D) is the same as EQ(A,B)=EQ(C,D). A more potent example is the exchange of the contents of two locations via the use of three exclusive or operations (we use the associativity and commutativity of the exclusive or operation to obtain this result). Clearly, this would require some type of theorem prover for the domain of logical operations.

Predicates are restricted in the sense that all tests that have unknown results must have LISP pointers as their operands. Tests whose operands are non-LISP pointers will always have their results known - i.e. they are redundant because non-LISP data is assumed to be known. This is why we use such concepts as offsets for stack pointers. They enable us to do arithmetic without worrying about relocation. This is a necessary restriction since we are aiming at obtaining a representation of a LISP function. Thus predicates operating on non-LISP data have no corresponding analog in LISP. In such a case equivalence between the LAP program and the LISP program fails to hold.

Any instruction may make only one LISP-like test at a time. Thus no compound tests are currently allowed. For example, a test of an entire word for 0 cannot be a test of whether or not both pointers in the word (i.e. the left and right halves of the word) are NIL. The test is valid if one of the halves is known to already be 0. The basic problem is that the AND construct in LISP is defined as follows:

```
(AND A B) ≡ (COND (A B) (T NIL))
```

This is different from our compound test since we have forced both A and B to be computed whereas in LISP, the arguments to AND are evaluated one at a time. Actually, we could allow the test by defining a new function, ANDC, of two arguments and no side-effects which is used as follows: (EQ (ANDC A B) NIL). In the true case, we add the equalities A EQ NIL, B EQ NIL, and ANDC(A,B) EQ NIL to our data base; while in the false case, we add the inequalities: ANDC(A,B) NEQ NIL and ANDC(A,B) NEQ ANDC(NIL,NIL) to our data base. The reason for the addition of the second inequality is to enable the detection of a contradiction when it is desired to know if both A and B can be NIL simultaneously. When the rederivation procedure is through we process the rederived form in the same manner as was done for EQSUBI - i.e. make sure that the ANDC construct is never used as a LISP pointer. This means that it does not appear as a result of a computation or as an argument to any LISP function other than EQ; and in the latter it is compared against a LISP pointer known to be NIL. In other words it must appear in a predicate slot of a conditional form. Note that if the construct appears as an argument to a LISP function other than EQ, then it must be in fact EQ to NIL and is replaced by NIL. All occurrences of EQ with an ANDC as one argument and NIL as the other argument are treated as follows:

```
((EQ (ANDC A B) NIL)→CONCLUSION,ALTERNATIVE) ≡
        ((EQ A NIL)→((EQ B NIL)→CONCLUSION,ALTERNATIVE),
                    (FN ALTERNATIVE B))
```

Note the placement of the computation of B as one of the non-result arguments of an FN construct in order to record its act of computation. Actually, if ALTERNATIVE is a conditional form, then B is placed in a non-result slot of FN constructs in all the terminal nodes of the tree corresponding to ALTERNATIVE.

Noise operations are defined as machine instructions whose results are unknown. This is caused by unknown operands or a mismatch of data types. Note the distinction between noise operations and LISP functions applied to unknown or non-LISP operands. Namely, in the case of LISP functions, the result is considered to be an error and processing along the path is terminated. However, this is only true for machine instructions when attempting to store into either unknown locations or known locations with improper data (i.e. store non-LISP pointers into elements of the List Structure). Whenever an attempt is made to store an unknown value not associated with any location in a specific destination, a data descriptor of type unknown is generated and placed in the appropriate location. This enables the detection of equality of operations involving unknowns to one level. For example, we can keep track of the movement of unknowns between legal locations (i.e. accumulators or stack locations) which is useful when testing equality of locations containing unknowns. By useful, it is meant that we can detect if two locations have identical contents. However, just because the latter test can not be determined in the affirmative does not mean that the two locations do not have identical contents. Moreover, since equality is considered a LISP function when it is not redundant, an equality test with unknown arguments which are not known to be EQ causes an error result and processing along the path is terminated. Of course, this limited amount of EQ detection is only a drop in the bucket, since it is not true for functions of unknowns or compositions of such functions. For instance, we would also like to keep track of arithmetic and logical operations performed on unknown computations. In order to do this, we would need to conceptualize the unknown in the same manner as relocatable constants and offsets as is done for stack pointers. In our case the relocatable constant is the descriptor of type unknown. However, this is not done for the same reason as given for bit manipulation as well as arithmetic involving EQSUB operations; namely, we have no theorem prover for the arithmetic or logical domain. We leave such work for the future.

Recall that our implementation of LISP uses a stack pointer format of [-allocated length of stack,base address of stack] which means that only stack overflow will be detected. Let count denote the number of items in the stack as indicated by the left half of the stack pointer and let offset be the difference between the base address of the stack and the value in the right half of the stack pointer. If count > offset, then stack overflow could be detected in a called function when in fact there was no stack overflow. Similarly, if offset>count, then stack overflow could fail to be detected in a called function. Thus we see that a valid stack pointer (offset=count) is one of the conditions that must be satisfied upon function calls, function exit, and of course it is assumed to hold at function entry.

When a function is called originally, a word is pushed on the stack containing the return address and a set of processor flags (FLAGS) in its right and left halves respectively. The placement of the flags is only important from the point of view that the index and indirect address fields are zero. We allow the left half to contain zero or the flags FLAGS where each instance of FLAGS is of a different value than other instances.

Prior to a function call several criteria must be satisfied. Parameter accumulators must contain LISP pointers in their right halves and data pointers of value zero in their left halves. All SPECIAL cells must contain valid LISP pointers in their right halves and data pointers of value zero in their left halves. All accumulators which were not to have been overwritten must have the same value that they had upon function entry. We also require a valid stack pointer and valid left and right halves in the stack cell containing the return address.

Prior to function exit the following criteria must be satisfied. The result of the function is in accumulator 1 which must contain a LISP pointer in its right half and a data pointer of value zero in its left half. All SPECIAL cells must contain LISP pointers in their right halves and data pointers of value zero in their left halves. Moreover SPECIAL cells which according to flow analysis were not to be modified by the function must contain the same values that they had upon function entry. A similar condition must be satisfied for accumulators which were not to have been overwritten by the function. We also require a valid stack pointer and valid left and right halves in the stack cell containing the return address.

4.G Summary of the Rederivation Procedure

The process of obtaining the rederived form requires several steps. These include information gathering and preparation of the input LAP program into a format suitable for symbolic execution. The remaining work consists of making several passes over the the data using symbolic execution to obtain the rederived form. Finally, the result is converted to a form ready for the matching phase of the proof procedure.

Initially, the user is asked several questions regarding the LAP program and the function it is purported to encode. These include a listing of the commutative and antisymmetric operations, the accumulator used to contain the stack pointer, the names of the functions which do not result in the destruction of the contents of all of the accumulators when they exit, as well as the highest numbered accumulator that they do destroy. This is quite useful in optimization. The user must also indicate the manner in which functions are to be invoked. There exists a special instruction named CALL which is used to call functions via traps. These traps are useful for such purposes as monitoring execution of a function (e.g. TRACE). Thus the user must specify the names of the functions which must be invoked via the CALL mechanism and which can be invoked directly. The rederivation process will make sure that the latter is satisfied. All other properties of functions discussed in Chapter 2 still hold (e.g. implications of operations such as EQ, CONS, RPLACA).

The next step is to transform the LAP program into a format suitable for symbolic execution. This consists of partitioning the the original program into code blocks (i.e. [Lowry69]) such that all instructions that can be entered via jumps, skips, or the false parts of condition testing instructions start a code block. A label is created for each unlabeled code block. Each code block is terminated by a branch to the next code block to be executed (we use the construct PLAB which is equivalent to the UNCONDITIONALJUMP primitive previously described in Section 4.D2). Thus we see that we must also specify the names of the branch and skip instructions in our machine description. Code blocks for skips are terminated by two PLAB constructs rather than one with the obvious meaning. The physical end of the program is denoted by the pseudo instruction PNIL which upon being symbolically executed denotes that an error has occurred. Function exit is detected when a branch is made to the label PCALLER which denotes the return address and is found at the bottom of the stack. Figure 4.7 is an example of the internal representation of the optimal encoding of algorithm 2 of NEXT which was first seen in Figure 1.21†. Note that the original function is given on the left and the internal representation on the right. Also the LISP definition of the algorithm was given in Figure 1.16.

† Notice the use of a SKIPE instruction. A more optimal encoding would use a JUMPE. However, we wish to illustrate the internal representation of a skip instruction.

```
(LAP NEXT SUBR)                  (LAP NEXT SUBR)
NEXT    (JUMPE 1 DONE)           NEXT    (JUMPE 1 DONE)
                                         (PLAB LOOP)
LOOP    (HLRZ 3 0 1)             LOOP    (HLRZ 3 0 1)
        (HRRZ 1 0 1)                     (HRRZ 1 0 1)
        (CAIE 3 0 2)                     (CAIE 3 0 2)
                                         (PLAB TAG1)
                                         (PLAB TAG2)
        (JUMPN 1 LOOP)           TAG1    (JUMPN 1 LOOP)
                                         (PLAB TAG2)
        (SKIPE 0 1)              TAG2    (SKIPE 0 1)
                                         (PLAB TAG3)
                                         (PLAB DONE)
        (HLRZ 1 0 1)             TAG3    (HLRZ 1 0 1)
                                         (PLAB DONE)
DONE    (POPJ P)                 DONE    (POPJ P)
        NIL                              (PNIL)
```

Figure 4.7 - Sample Internal Representation

4.G1 Pass One

The first pass over the data using symbolic execution will attempt to obtain a rederived form. During this process, some instructions which did not originally appear labeled may become accessible from other instructions due to the occurrence of branches that employ indexing to compute the target label. In this case new code blocks will be created. A record is always kept of the labels encountered along the path being executed and it is dynamically updated if intermediate labels are created. This is necessary so that we can detect when a label is entered that was previously encountered. Recall that in such a case we will try to prove that recursion has taken place.

Whenever a label is entered that was previously encountered along the path being processed, then the label is recorded as being an element of a set denoted as targets of backward branches. In this case we leave the parameters unspecified since we no longer know the calling sequence being used - i.e. a mixture of stack and accumulators. This means that at least two more passes need be made. When a recursive call has occurred to an external function or to the function being processed, we search the stack for the nearest label to the top and continue execution. If this label was also previously encountered, then we apply the same procedure again. This is done until a return address is found which was not previously encountered. When all possible paths have been examined and no backward branches were encountered, then the resulting rederived form is valid. Otherwise, we need to perform Pass Two and Pass Three.

An example of the output of Pass One is given in Figure 4.8 for the encoding of NEXT given in Figure 4.7. Note the use of LX4 and LX5 to indicate unspecified parameters for the recursive call via LOOP. The symbolic representation is given on the left while the numeric representation is given on the right. Notice that we have purged the UNREFERENCED list in the interest of conciseness although this is not necessary.

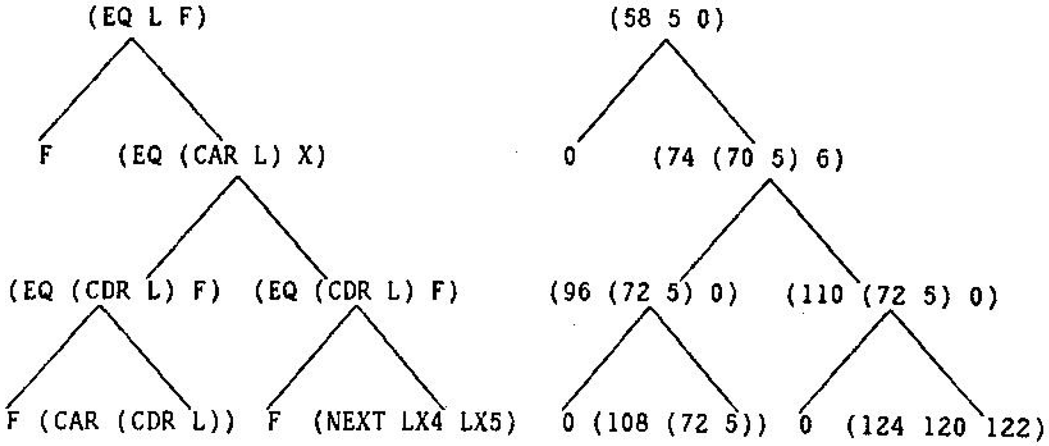


Figure 4.8 - Pass One Output for Figure 4.7

4.G2 Pass Two

The main purpose of Pass Two is to collect information relevant to the detection of recursion for backward branches not to the start of the program. We are interested in determining for each label branched to in the backward direction the set of paths leading into the label from the start of the program (known as *forward paths*), and the locations that are referenced prior to being destroyed (i.e. overwritten). The former is necessary for determining the redundant path that was bypassed by the backward branch. The latter is necessary for determining the parameters of the recursive call in addition to insuring that the locations referenced in the future have the appropriate contents (see the discussion related to Figure 4.5 in Section 4.F). We recall, that the calling sequence is no longer known, and thus the values of the parameters are not found in fixed locations. The actual process of determining the values of the parameters for such recursive calls is known as symbolic matching and is done during Pass Three.

Thus, in Pass Three we will perform a symbolic match of the contents of the locations indicated by Pass Two as containing information that must be properly set prior to the backward jump. The main result of the process of symbolic matching is a detection of the necessary parameter bindings. More will be said about this matter in the discussion of Pass Three. Pass Two is quite similar to Pass One with the following modifications. No new code blocks (e.g. labels) can be created. Record all accumulators and stack locations (in terms of halfwords) that are referenced prior to being destroyed past labels branched to in the backward direction. In addition, all of these labels must have the property that all entry paths into them are of a uniform stack depth.

For example, the NEXT function in Figure 4.7 has only one label that is entered in the backward direction, LOOP. This label is only entered in one way, namely after label NEXT. Pass Two indicates that accumulators 1, 2, and P (the accumulator containing the stack pointer) have their left and right halves referenced subsequent to the encounter of the label LOOP and prior to being destroyed. Also, the right half of the top entry on the stack (i.e. the return address) is referenced prior to being destroyed.

4.C3 Pass Three

The main purpose of Pass Three is to prove that backward jumps actually do correspond to a recursive call. The process consists of using the results of Pass Two (forward paths to the label and the locations referenced subsequent to the label) to determine a redundant path to the label plus a set of parameter bindings. Our presentation is preceded by some motivated restrictions on the type of backward jumps that can be handled.

We do not have an arithmetic identities theorem prover. This means that we cannot detect redundancies in predicates due to properties of the domain in question. For example:

```
f(x) ≡ if x=0 then 1
       else if x=1 then 1
       else f(x-2)+f(x-1)
```

In this case f(x-1) need not perform the test x=0 since it is impossible by virtue of the x=1 test being false on the previous instantiation of f(x). In other words x≠1 implies x-1≠0. In some sense our failure to allow use of such identities is unfortunate; however, it can be quickly seen that an adequate treatment for identities in a particular domain requires a theorem prover for the domain. We choose to leave this problem for future work. Note that the full analysis of the interrelationships between EQ, ATOM, and EQUAL does correspond to a theorem prover for the predicates of LISP. However, we go no further.

In the normal sense, LISP functions have only one entry point, namely the first instruction. When a jump occurs to a point already seen, then it is interpreted as a recursive call. Now, just as one has a uniform calling sequence for functions we will use the previous analogy to stipulate that all paths enter uniformly a label branched to in the backward direction. This uniformity manifests itself in the restriction that the label in question is entered with the same stack depth by all paths encountering it for the first time. This allows us to determine the stack depth at the time of the jump that bypasses the start of the program. Otherwise, without knowing the exact path that was bypassed, we do not know where to continue the symbolic execution if the call was of a recursive nature (more work to do when done) rather than of an iterative nature.

We leave it for future work to handle the general case of arbitrary stack depth at labels already encountered. Implementation of such a feature would be of a backtracking nature. Namely, a specific bypassed path would be assumed and rederivation would continue. If a contradiction arises, then a backtrack is made to the last decision point and an alternate bypassed path chosen until success is obtained. Otherwise the correctness proof fails. In summary, the stipulation on stack depth is not overly restrictive since compilers and programmers will for the most part use uniform stack depths at labels. The stack depth uniformity can be checked in Pass Two.

In some sense the uniformity criterion is analogous to attributing a functional nature to the label with a mixed calling sequence. In fact this is how we would handle PROGs with a GO construct to labels previously physically encountered. This would implement the pseudofunction representation of a PROG that was presented in Chapter 2.

When a backward jump is encountered to a label, say LABEL, other than the start of the program, and if the jump has not yet been proven to be a recursive call, then for each of the forward paths to LABEL we do the following. Determine what the state of the computation model is when this particular path is symbolically executed. From this computation model form a set, say CALLED, containing the values of the SPECIAL cells, all locations that were determined by Pass Two to be referenced subsequent to LABEL, and the value of the stack pointer. Similarly, form a set, say CALLER, containing the values of the corresponding cells in the computation model valid at the time of the encounter of the backward jump. At this point perform a symbolic match (not to be confused with the matching phase of Chapter 5) between the corresponding elements in CALLED and CALLER.

(1) All elements that are not LISP pointers or stack pointers (i.e. offsets) must be matched exactly.

(2) Stack pointer data type offsets must differ by an amount equal to the difference between the stack size in CALLER and the stack size in CALLED.

(3) All LISP pointers that are QUOTEd and addresses of SPECIAL cells must also be matched exactly. The bottom entry in the stack in CALLER must contain FLAGS or zero in its left half and PCALLER (the return address indicator) in its right half.

(4) LISP pointers are matched by performing a pattern match between the value in CALLER and the value in CALLED. The latter serves as the template and the names of the local parameters serve as the variables in the template. Whenever a match results in bindings for any variables, we check if any of the following occur. All parameters receive a binding in which case we exit. A parameter receives more than one binding in which case we indicate that the path is not worth further pursuit. Otherwise continue matching. For example, in the example NEXT we have the right half of accumulator 1 in CALLED containing L while the same location in CALLER contains (CDR L). In this case we say that parameter L in CALLED is matched or bound to (CDR L). If this procedure terminates without obtaining a binding for all of the parameters, then they are left as unspecified. This is generally a result of parameters not being necessary, or only referenced prior to the label in question. We shall see in the discussion of the matching phase how this type of problem is resolved.

Once the match has resulted in values for the parameters, we have both a symbolic and a numeric binding for each parameter. At this point we initialize the accumulators (since we use a calling sequence employing the accumulators) to the values of the parameters, initialize the computation model to that valid at the jump to LABEL, and apply the symbolic execution process. If the following two conditions hold, then the path that we have just processed is redundant.

(1) All the conditions on the path to LABEL will have known values.

(2) All LISP pointers that were computed along the path are EQ to computations performed before the path was executed. In other words, no new information was added to the computation model.

If the above conditions did not hold, then we attempt another path. If no more paths remain unprocessed, then we declare that the backward jump could not be resolved.

Prior to presenting the control loop of Pass Three we define the concept of *level*. This will be seen to have important ramifications in proving termination of our algorithm, and will also act as a pruning device. As each path in a program is being rederived, a count is kept of the number of calls to the start of the program or to labels past the start of the program that have been proved to correspond to recursion (referred to as recursive calls in the rest of the discussion). This procedure will be known as keeping track of levels where a level of 2 indicates that two recursive calls have already been encountered along the path. For each forward path entering a label in the set of backward labels, we also record the level at which the label is first encountered along the path (this can be done during Pass Two). For example the LAP program in Figure 4.9 exhibits a jump to label LOOP at the instruction at label PC13. In this case the jump is said to occur at level 2.

Pass Three is a multipass process which means that we may attempt to rederive the program more than once. In fact we will make at most as many passes as one plus the highest level at which a backward jump appears. Each pass will resolve all backward jumps of level one less than the pass number. This is true because a backward jump at level n could only bypass a path of level less than or equal to n. Thus, if NMAX is the maximum level of any backward jump, then if after execution of the NMAX+1 pass all of the backward jumps are not resolved, then subsequent passes will not yield any more information and we stop. Thus we have proved that the rederivation procedure will terminate.

```
(LAP FN1 SUBR)
        (JUMPE  1 OUTPOP)
        (PUSH   P 1)
        (HRRZ   1 0 1)
        (PUSHJ  P FN1)
        (JUMPN  1 CONT)
LOOP    (HRRZ@  1 0 P)
        (HRRZ   1 0 1)
LOOP1   (PUSHJ  P FN1)
        (JUMPN  1 CONT)
        (HRRZ@  1 0 P)
        (JUMPE  1 OUT)
        (MOVEM  1 0 P)
PC13    (JRST   0 LOOP)
CONT    (MOVE   1 0 P)
        (MOVE   2 1)
        (CALL   2 (E *TIMES))
OUT     (SUB    P (C 0 0 1 1))
OUTPOP  (POPJ   P)
        NIL
```

Figure 4.9 - Sample Program Illustrating Levels

A recursive call to a point past the start of the program is represented in terms of the parameter bindings that were found during the process of determining a redundant path. Thus we represent a backward jump as if it were a recursive call to the start of the program.

From this discussion it should be quite clear that we will not be able to handle backward branches in the general case (if there is one). Moreover, our methods of handling backward jumps are driven to a large extent by heuristics. However, our restrictions are quite reasonable and in fact lead to some interesting results.

4.G4 Postprocessing

Once the previous passes are complete we apply a postprocessing step to the resulting rederived form so that it will have undergone the same transformations as the canonical form.

(1) Replace all predicates that have not been tested by (pred→T,F).

(2) Apply a variant of the algorithm used in the first part of the canonical form procedure. The difference is that no binding need be made. The main purpose is to apply axiom (7) and the distributive law of functions to the rederived form. This is necessary due to the replacement of implicit conditions by conditional forms.

(3) Sort the predicates of the resulting form according to increasing computation numbers of the predicates. This is necessary because of the previously untested predicates which have now been converted into tests. This process uses the depth first numbering property of the rederived form - i.e. all computations generated in the right subtree have a higher number associated with them than those generated in the left subtree, and if two functions have identical computation numbers, then they are identical (see Chapter 3). The actual sorting procedure is valid because the predicate is known to be inevitable by virtue of the depth first numbering property. Thus we may apply steps (2) and (3) as indicated in the revised canonical form algorithm for strong equivalence. However, once this is done we will have to apply the depth first numbering algorithm to the resulting form since application of axiom (1) yields identical conclusion and alternative clauses.

(4) Eliminate occurrences of EQSUBI and EQSUBD by their appropriate equivalents. This procedure follows the restrictions placed on the use of this construct in Section 4.F. In addition, whenever an instance of these constructs appears as a non-result argument to an FN list, then the construct is replaced in the list by its arguments.

(5) Apply the duplicate computation removal algorithm of Chapter 3.

(6) Remove all redundant first cases of assignment as was done in the canonical form algorithm.

(7) Purge the FN list of all non-result arguments appearing as subexpressions in other computations. Again, this is the same algorithm applied to the canonical form in Chapter 3.

Figure 4.10 is the rederived form after postprocessing of the NEXT function used in this chapter. The symbolic and numeric representations are given in the left and right halves respectively of the figure. Note that the recursive call which had unknown parameters in Figure 4.8 has been replaced by (NEXT (CDR L)) X).

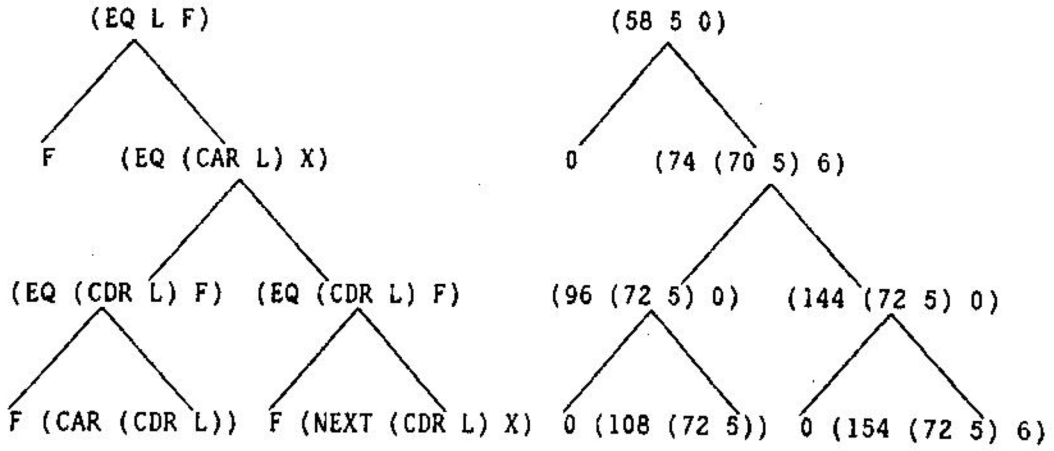


Figure 4.10 - Rederived Form Corresponding to Figure 4.7

CHAPTER 5

THE PROOF SYSTEM

5.A Introduction

Once the original LISP program has been converted to the canonical form and the LAP program converted to the rederived form, we are ready to attempt to prove the equivalence of the two. This procedure is termed *matching*. From previous discussions and examples it should be clear that there is no valid reason for assuming that the outputs of the two procedures are identical (symbolically and to a lesser degree numerically).

One possible solution is to find some common representation of the two forms which is in terms of a lexicographic minimum. This would remove the problem that is associated with the use of substitution of equals for equals (we will use this term to denote substitution of items known to be EQ for each other). Such a solution would imply that we could find a base representation for the rederived form and likewise for the canonical form and then simply perform a strict equality match. However, such a solution fails to take into account the possibility of rearranging the order of computation of predicates and other functions. When such rearranging occurs, a solution of a lexicographic minimum nature would have to examine all possible orders of computation before arriving at a lexicographic minimum. Moreover, the lexicographic minimum does not adequately deal with the concept of recursion via jumps to points past the start of the program.

In the process of proving equivalence we will have to show that neither of the canonical and rederived forms reflect computations not performed in the other. We have already addressed ourselves in part to this problem in the previous chapters when we discussed the removal of duplicate predicates and computations from the canonical and rederived forms. Recall that at the end of the rederivation and canonical form procedures, we removed all redundant first instances of assignment. This was needed to insure that assignments into SPECIAL cells and elements of the List Structure were not done for the purpose of temporary storage of results – i.e. the location could not be subsequently referenced with the specific value as its contents. In the case of a SETQ the reference is only relevant if it occurs in an external function call; not in the function being processed since in such a case we may use the binding of the SPECIAL variable. In the case of a RPLACA or RPLACD operation, the reference is only valid if it occurs in an external function call or is a reference of a LISP pointer not known to be EQ to the first argument of the unnecessary RPLACA or RPLACD operation in question (see Section 2.B4). Occurrences of functions in one form and not in the other will lead to inequivalence unless the functions are of a primitive nature since only they can be introduced at will.

For example, consider the case of a computation of CAR(<expr>) which cannot be matched even though CDR(<expr>) was matched or vice versa. Such an example was illustrated in the encoding given in Figure 1.15 for the NEXT algorithm formulated in Figure 1.2. In this case we currently say that inequivalence is the case since a computation is performed in one form and not in the other. Actually, we should say that if CAR (or CDR) is legal then so is CDR (or CAR). In other words the operation becomes primitive in such a case. This is because no side–effect can occur – i.e. recall that the problem was that CAR and CDR are not defined when their argument is an atom. But the act of computing one of them safely, implies that the other can also be computed safely. This problem is quite common in a LISP implementation where a word represents more than one LISP entity – i.e. a pointer to CAR in the left halfword and a pointer to CDR in the right halfword. In such a case we may access an entire word when in fact we only desire a specific half. A similar consideration is if ATOM(A) is known not to be true, then CAR(A) and CDR(A) are also safe provided that they are performed after the ATOM test. These statements are all justified by the *Extension to the Rule of Replacement.*

One solution is to place CDR(A) on the FN list (and on UNREFERENCED) whenever CAR(A) is seen and similarly for CAR(A) and CDR(A). Also place CAR(A) and CDR(A) on the FN list (and on UNREFERENCED) whenever ATOM(A) is known to be NIL (i.e. false). The only problem occurs when we remove redundant first instances of computation (see the canonical form and also the postprocessing stage of the rederived form) involving modification of heads and tails of elements of the List Structure. In this case, if an access to the head or tail of an element of the List Structure only appears as an argument in a non–result slot to an FN construct, then it should not be considered as an access operation.

We propose the following solution for a CAR(A) operation that cannot be matched. However, first note that the operation must occur in a non–result argument of an FN construct since otherwise there is no hope of proving equivalence as the two intermediate forms will not even be identical in their predicates and results (this will become clearer at the end of Section 5.B). Examine the non–result arguments of FN constructs for instances of CAR(A). If all such occurrences appear as non–result arguments of FN constructs, and if CDR(A) appears in a predicate or in the result of a path containing CAR(A), then CAR(A) can be safely removed from the FN list. By safely removed we mean that its computation qualifies as a primitive operation. We leave the case of ATOM for future work although it is clear that it would be handled in much the same manner. This procedure is applied prior to removing redundant first instances of a computation involving modification of heads or tails of elements of the List Structure at the end of the canonical form process described in Chapter 3 and during the postprocessing stage of the rederivation process. It should be noted that the same solution would hold for CDR(A) (actually we need to replace CAR by CDR and CDR by CAR in the discussion).

Our solution to the equivalence problem is to transform one form into the other. We have two choices of operation. We can show via the use of equivalence preserving transformations (the axioms and substitution) that the canonical form can be transformed into the rederived form or vice versa. However, this is not enough; we must also demonstrate that neither form reflects any computations not performed in the other. Thus we will perform the matching process twice; once manipulating the canonical form to match the rederived form, and next manipulating the rederived form to match the result of the previous manipulation of the canonical form.

The remainder of the chapter expands on the previous notions. We first present the matching algorithm in terms of transforming the canonical form to match the rederived form. We next discuss the issues relevant to proving equivalence when recursion has been implemented via jumps to points other than the start of the program. This is coupled with modifications to the rederivation process as well as to the matching procedure just presented. Throughout the discussion we give examples in which only the symbolic representation is used. In these cases it is assumed that requirements placed on equivalence and matching by the numeric representation are satisfied.

5.B Matching Procedure

The matching procedure consists of manipulating the canonical form to obtain an identical form to the rederived form. This is done by examining the rederived form in order of increasing computation numbers and finding what will be known as *matching* computations. We use a variant of the canonical form algorithm for strong equivalence described in Chapter 3. The difference is that in addition to the predicates being rearranged via axiom (8), the two forms may differ because computations may be computed out of sequence as well as due to the use of substitution of equals for equals. Thus we see that the concept of inevitable predicate must be extended to include all computations. Basically, for each computation performed in the rederived form, we must insure that it is also computed in the canonical form. This means that if a computation appears in a condition (non–terminal node) in the rederived form, then it must either appear in the corresponding condition in the canonical form, or in each of the subtrees of the canonical form. Of course the same criterion holds for computations appearing solely in terminal nodes.

Recall that the numerical representation of a function serves to indicate a relative ordering between the various computations in terms of their instance of computation. Thus the magnitudes of the numbers are seen to be insignificant. We will make use of the following numbering scheme which will be seen to have certain desirable properties. In order to perform the matching process we start with all computations in the canonical form having higher numbers than those in the rederived form. The only exception are the local parameters and the atoms T and F which are assigned identical computation numbers in the two representations.

Each computation in the rederived form must be proved to be inevitable in the canonical form. This means that the computation or its equivalent must be shown to be computed in the canonical form. Moreover, its instance of computation in the canonical form must be shown to yield an equivalent result to the value computed in the rederived form. This process is termed *finding a matching instance*. The criteria for matching are given below.

(1) If the function reads and modifies a SPECIAL variable, then there must be no intervening reading or modification of the said variable.

(2) If the function only reads a specific SPECIAL variable, then the said variable must have the same values prior to the two instances of the function.

(3) If the function only modifies a specific SPECIAL variable, then there must be no intervening reading or modification of the said variable.

(4) If the function reads and modifies *heads* (or *tails*) of elements of the List Structure, then there must be no intervening reading or modification of the said part.

(5) If the function only reads *heads* (or *tails*) of elements of the List Structure, then there must be no intervening modification of the said part.

(6) If the function only modifies *heads* (or *tails*) of elements of the List Structure, then there must be no intervening reading or modification of the said part.

(7) The function may perform a CONS operation.

The criteria for matching are quite similar to those used to determine duplicate computations. However, there are several differences. While matching, operations resulting in the modification of SPECIAL variables and the List Structure may also read the said items, but intervening reading is prohibited. Moreover, a CONS operation is permissible. Without the previous differences, constructs employing such functions could not be matched up.

The key to the matching procedure is that whenever a computation, say A, is matched with a computation not previously encountered, say B, then all occurrences of B in the canonical form are replaced by A. Similarly for the SPECIAL variables modified by A – i.e. the computation number associated with SPECIAL variables modified by A replaces the computation number associated with SPECIAL variables modified by B. This means that during the matching process we are searching the canonical form which has the property that all computations in the canonical form with computation numbers higher than those in the rederived form have not yet been matched. Moreover, if equivalence holds, then at the end of the procedure we will have managed to transform the canonical form into the rederived form.

The process of finding a matching computation is quite straightforward. We process the rederived form in increasing order of computation number. For each function, say A, search the canonical form for a matching instance. Recall that all computations in the canonical form with a computation number less than the number associated with A have already been matched. The matching instance can be encountered in two ways. An *explicit occurrence* is defined to be a function which has not been previously encountered in the canonical form. An *implicit occurrence* is a function or an atom which has already been previously encountered when matching other computations. This results from the use of equality information and subsequent substitution of equals for equals. For example, consider the rederived form given in Figure 5.1. Suppose that we are looking for (CDR B), and (CDR A) has already been computed and matched. In the true subtree (i.e. left) of the condition (EQ A B) we see that (CDR A) is EQ to (CDR B) and thus (CDR A) is a matching instance of an implicit nature. Thus all that remains is to search the false subtree (i.e. right) for an explicit occurrence of (CDR B) or another implicit equivalent.

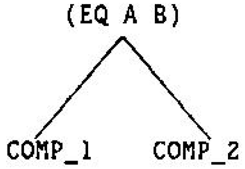


Figure 5.1

As soon as all of the arguments of a predicate, say p, are matched, we check if the predicate is a primitive function. If yes, then apply the following transformations to the canonical form, say CAN, corresponding to the part of the rederived form currently being processed.

(1) Replace CAN by (p→CAN,CAN).

Step (1) corresponds to application of axiom (1).

(2) Apply the breadth first to depth first renumbering algorithm to the result of step (1).

Step (2) is necessary because the conclusion and alternative clauses have the same numeric representation after application of step (1). Moreover, subsequent applications of matching assume that the canonical form has the properties that all functions with the same computation number have been computed simultaneously, and that all computations computed in the right subtree have a higher computation number associated with them than those computed in the left subtree.

(3) Apply the duplicate computation removal algorithm to the result of step (2).

Step (3) corresponds to the application of axioms (5) and (6) as indicated in the canonical form algorithm for strong equivalence given in Chapter 3.

We did not need to match primitive predicates because by definition primitive functions can be considered as inevitable computations. However, if the predicate does not correspond to a primitive function, then we must insure that it is indeed inevitable. We choose a different method for proving inevitability. We attempt to prove that at each terminal node of the canonical form, the predicate is redundant. If the latter is true, then clearly the predicate is inevitable; otherwise, it is not. The reason for use of such a procedure is best exemplified by the predicate EQUAL. In this case, if we were to look for an occurrence of the predicate in the same manner as was done for functions, then the situation may arise that we cannot match it. For example, consider the symbolic canonical and rederived forms as in Figures 5.2 and 5.3 respectively. Moreover, we wish to manipulate the canonical form to match the rederived form.

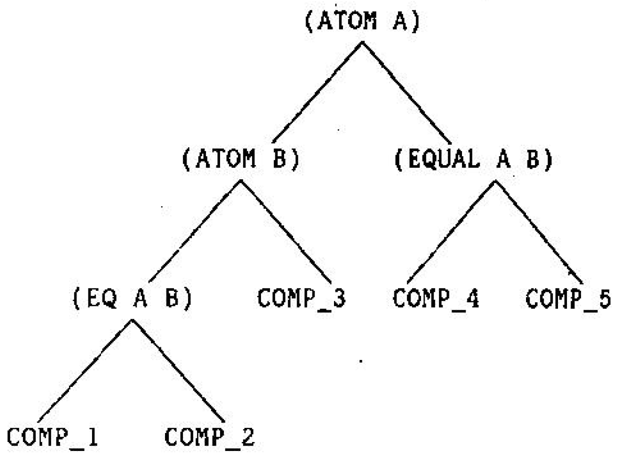


Figure 5.2 - Symbolic Canonical Form

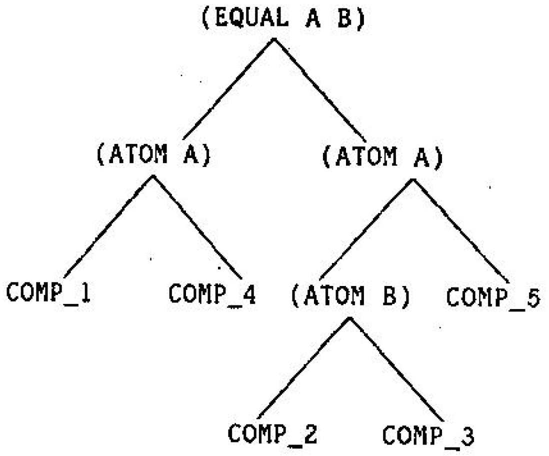


Figure 5.3 - Symbolic Rederived Form

The first step is to show that the predicate (EQUAL A B) is inevitable. This is clearly the case by the time that COMP_4 and COMP_5 are performed. By the time that COMP_1 and COMP_2 are performed the fact that A and B are both atoms implies that the predicate is inevitable since the EQ test is identical to the EQUAL test when the arguments are atoms. In the case that A is an atom and B is not an atom, the predicate is again inevitable by virtue of the unique representation of atoms – i.e. if A is an atom and B is not an atom, then A is neither EQ nor EQUAL to B. However, if we would have used our matching algorithm to detect matching instances of the predicate, then we would lose. Note that we have made use of the implications of the various predicates as outlined in Chapter 2. Actually, our statement about a loss is not quite correct since a value of the predicate of T or F can be considered as an implicit occurrence. Nevertheless, we shall stick with our treatment. Finally, we point out that this example illustrates how problems which might arise in the lexicographic minimum approach to matching do not surface when equivalence is proved using our matching techniques. Specifically, the lexicographic minimum approach implies making direct use of axiom (8). In such a case, examples such as this one do not indicate in an obvious manner the inevitability of the (EQUAL A B) operation.

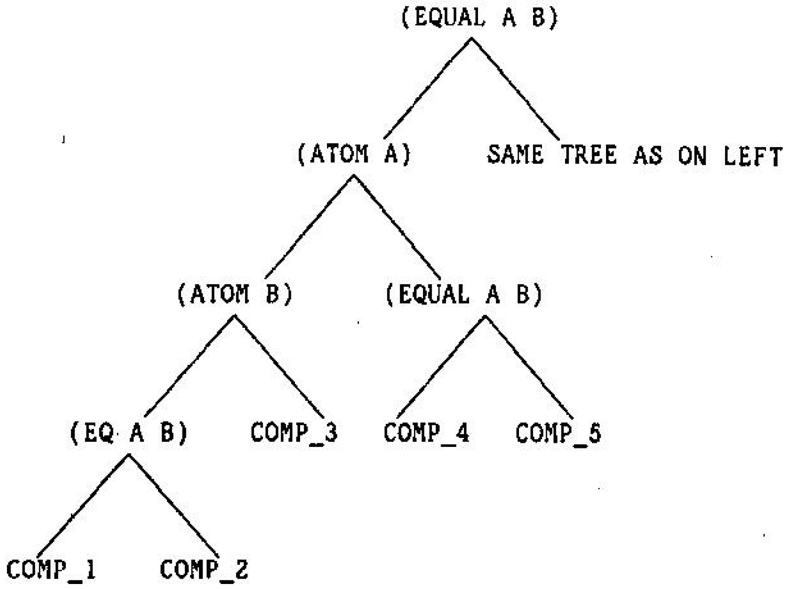


Figure 5.4

Continuing with our example, we apply axiom (1) to obtain Figure 5.4. Application of duplicate predicate removal leads to Figures 5.5a and 5.5b for the true and false subtrees of (EQUAL A B) respectively. In Figure 5.5a the truth of (EQUAL A B) and (ATOM A) imply that (ATOM B) and (EQ A B) are true by virtue of the unique representation of atoms. In Figure 5.5b the falseness of (A EQUAL B) and the truth of (ATOM A) and (ATOM B) imply the falseness of (EQ A B). In fact since Figures 5.5a and 5.5b correspond to the left and right subtrees respectively of the predicate (EQUAL A B) we see that Figure 5.4 has been reduced to be equivalent to Figure 5.3.

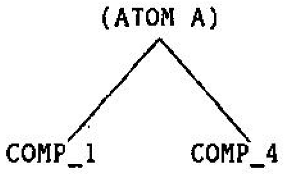


Figure 5.5a

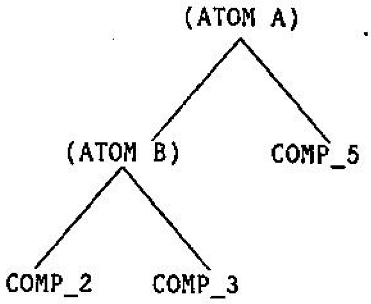


Figure 5.5b

Leaving our example and returning to the matching procedure, once a predicate has been matched, we proceed to apply the matching procedure to the conclusion and alternative clauses of the rederived form. Once the entire rederived form has been processed by the canonical form algorithm, we see that all computations performed in the rederived form have been matched (i.e. shown to be equivalent) with computations in the canonical form. However, we have not yet proved that all computations performed in the canonical form have also been performed in the rederived form. This can be done by applying the matching procedure with the roles of the canonical form and rederived form reversed – i.e. manipulate the rederived form to match the canonical form. This problem will disappear once we present, in a subsequent section, the method for handling matching of functions with recursive calls implemented as jumps to points other than the start of the program.

Upon termination of the previous, if no failure has been indicated, then check to see if the canonical and rederived forms are symbolically and numerically identical. If yes, then a successful match has occurred and the LAP program is said to be equivalent to the LISP program. This final procedure is necessary in order to make sure that all terminal nodes are identical. An example where the computations performed in the two forms are identical, yet the terminal nodes return different results is given in Figure 5.6.

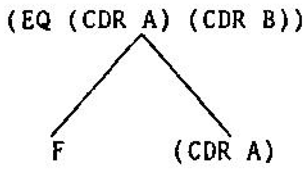


Figure 5.6a

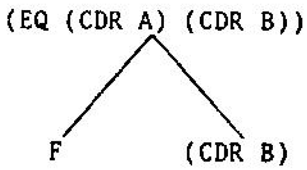


Figure 5.6b

An interesting question results from the discussion of matching and duplicate computation removal. When determining argument values for recursive calls that bypass the start of the program during the rederivation procedure, should we use matching or stick to our present technique of proving the various computations redundant? It turns out that the method of redundant computations is the right solution since if a computation is repeated in the beginning of a function as well as prior to the recursive call, then its elimination (or bypassing) may be wrong. For example, suppose that a function known to read and modify a SPECIAL variable fits this criterion. Moreover, the arguments are identical. In this case, its bypassing would be a mistake since the function could be of a nature that increments the SPECIAL variable.

From the discussion we see that the proof procedure is machine independent. Also notice how the various properties of the depth first numbering scheme are used to enable the matching procedure to correspond quite closely to the duplicate predicate removal process. In fact many of the same routines are used in the implementation.

5.C Recursive Calls Bypassing the Start of the Program

In Chapter 4 we saw that the rederivation system was able to handle recursive calls implemented by jumps which bypassed the start of the program. In such a case we said that an attempt is made to prove that if the jump had indeed gone to the start of the program, then we would have fallen through to the label in question. The representation of such a construct is a recursive call to the start of the program with the parameters having the values that they would have had if the call had indeed gone to the start of the program. There was one exception to the latter; parameters that could not be determined were left unspecified.

We shall use the term *loop shortcutting* to denote recursive calls not to the start of the program. An example is Algorithm 2 for NEXT whose LAP encoding is given in Figure 4.7. The canonical and rederived forms are shown in Figures 5.7a and 5.7b respectively using a tree–like representation.

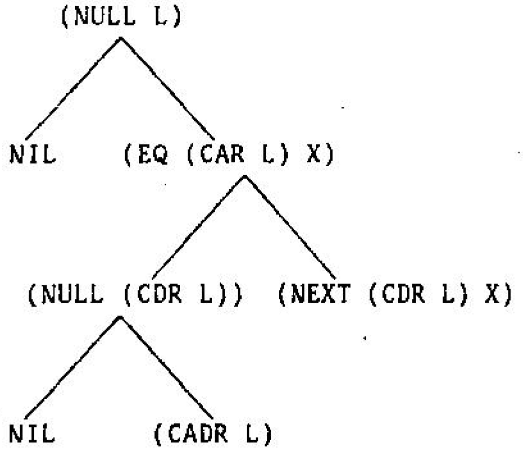


Figure 5.7a - Canonical Form

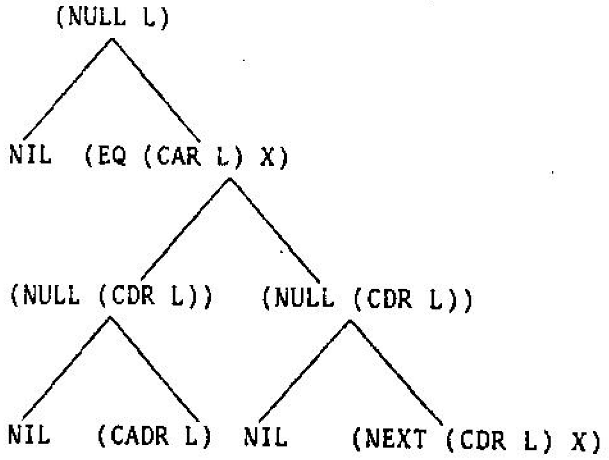


Figure 5.7b - Rederived Form

Note that the two forms differ in the expansion of the rightmost terminal node of the canonical form. This is why we term the procedure loop shortcutting. The rederived form representation has one deficiency. It does not give us any inkling that a recursive call was computed by bypassing the start of the program. Thus an encoding of the function which did not implement the recursive call by bypassing the start of the program (see Figure 5.8) would have the same representation. This is unfortunate because the two functions are not identical. The encoding in Figure 5.8 performs shortcutting and then proceeds to do recursion where it recomputes the part bypassed by the shortcut. However, the encoding in Figure 4.7 performs shortcutting followed by recursion without recomputing the part that was bypassed by shortcutting.

```
(LAP NEXT SUBR)
NEXT    (JUMPE 1 DONE)          jump to DONE if L is NIL
        (HLRZ 3 0 1)            load register 3 with CAR(L)
        (HRRZ 1 0 1)            load register 1 with CDR(L)
        (CAIE 3 0 2)            skip if CAR(L) EQ X
        (JUMPN 1 NEXT)          if CDR(L) is not NIL then
                                compute NEXT(CDR(L),X)
        (SKIPE 0 1)             skip if CDR(L) is NIL
        (HLRZ 1 0 1)            load register 1 with CAR(CDR(L))
DONE    (POPJ P)                return
        NIL
```

Figure 5.8

Figure 5.7b should be decomposed into the two parts given in Figures 5.9a and 5.9b which represent the functions NEXT(L,X) and NEXT1(L,X) respectively. Closer scrutiny will reveal that Figure 5.9a is almost identical to the difference between Figures 5.7a and 5.7b. The only difference is in the use of (CDR L) in Figure 5.7a and L in Figure 5.9a. This similarity will be exploited in the proof procedure given in the next section.

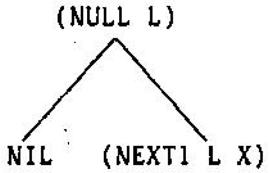


Figure 5.9a

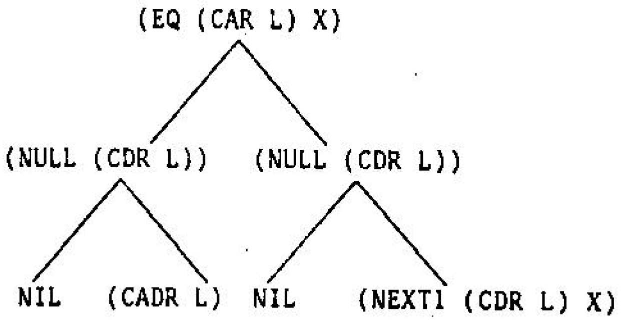


Figure 5.9b

When applying the matching procedure, loop shortcutting will cause a mismatch. This happens because the canonical form expects a recursive call while the rederived form will contain an expansion of the call as in Figure 5.7b. However, recall our previous observation that the difference between Figure 5.7a and 5.7b is that Figure 5.9a has been used with the values of the parameters of the recursive call (i.e. (CDR L) for L). This is not a coincidence. In fact the matching procedure will attempt to substitute all the possible functions having a form similar to Figure 5.9a. The set of such functions is finite and has at most one less element than the size of the powerset[†] of the bypassed paths.

The previous statement needs further clarification. Consider the case when only one label serves as a target of a backward branch and there is only one bypassed path (i.e. the LAP encoding of NEXT given in Figure 4.7). In this case apply the rederivation procedure in a similar manner to that outlined in Chapter 4. The crucial difference is that whenever we encounter the label along the bypassed path, we cease the rederivation process along the path and return the name of the function applied to the names of its local parameters. For our example of NEXT, Figure 5.9a would be the result returned. If the rederivation process had encountered more than one bypassed path, then the same procedure must be applied for each bypassed path. However, this is not sufficient. Suppose the LAP program has bypassed via shortcutting a recursive call via more than one backward jump. This could occur when loop shortcutting yields a branch to one label, say L1, in one case and to another label, say L2, in the other case (in this hypothetical example we are also assuming only one bypassed path per label). Upon detection of a mismatch during the matching procedure, we will attempt to substitute the two rederived forms corresponding to the bypassed paths associated with L1 and L2 and failure will result. The reason is obvious – we needed a rederived form for the combination of the bypassed paths. This can be extended to the case where there are n bypassed paths which implies 2^n-1 different combinations. But the latter is the powerset of the bypassed paths minus the empty set. Actually the empty set can be interpreted as the original rederived program which, of course, corresponds to the situation that no bypassed paths cause the cessation of the rederivation process. Thus the rederivation process extension must be further extended to read:

> For each element, say A, of the powerset (except the empty set) of bypassed paths apply the rederivation procedure in such a manner that whenever a bypassed path which is an element of A is processed, cease the rederivation process along the path and return as the result of the path the name of the function applied to the names of its local parameters along with the contents of UNREFERENCED.

[†] The powerset is defined to be the set of subsets of a set of items. For example, the powerset of {a,b,c} is { }, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, and {a,b,c}.

The postprocessing procedure applied to the rederived form at the end of Pass Three in Chapter 4 is applied to each of the new rederived forms which are said to comprise the set SHORTCUT. Note that there was no need for more than one pass to create the elements of SHORTCUT since multiple passes are only necessary when there are jumps to points previously encountered whose parameters are undetermined. However, in our case the primary rederived form was previously computed, and thus all backward jumps have already been resolved.

We mentioned earlier that the set of rederived forms SHORTCUT has at most one less element than the size of the the powerset of the bypassed paths. The following are some examples of when this upper bound fails to be attained. Whenever a bypassed path has as its result the function name applied to the parameter names and nothing else, then we can simply remove it. Such a situation occurs when we bypass overhead computations such as pushing elements on the stack as in HIER1 (see optimized version of HIER1 in Chapter 6) or in Figure 4.5. When a bypassed path, say A, is a subpath of another bypassed path, say B, then all subsets of the powerset of bypassed paths containing both A and B, say C, are identical to the subset C−B (i.e. B is left out). This should be obvious when the extension to the rederivation process is examined more closely.

At this point we return to a comment made earlier about the similarity between the rederived program corresponding to the LAP program in Figure 5.8 and that presented in Figure 4.7. When rederiving the latter we proved that a certain recursive call can be implemented via loop shortcutting. This means that the shortcutting is legal whether or not we bypass the start of the program. Thus we need only prove the shortcutting legal once for a recursive call. From now on, we may replace other recursive calls using any of the rederived forms in the set of SHORTCUT whether or not these recursive calls have been implemented via shortcutting. In fact this is how the proof procedure works when it can not match a recursive call in the rederived form -- i.e. it will attempt to replace the recursive call in a legal manner using the various elements in SHORTCUT until success occurs or none yield a match. The latter implies failure. Thus we cannot prove the equivalence of the LAP program in Figure 5.8 with the NEXT function because we have no elements in SHORTCUT. In other words if we would have a function which shortcuts in the same manner by bypassing the start of the program in one case and not bypassing in the other case, then we could still prove equivalence since we are able to prove the shortcutting legal in one of the cases. Whereas when we never prove the shortcutting legal, we have no bypassed path with which to work when attempting shortcutting. Such cases will be termed *loop economy* where in the example given in Figure 5.8 we say that the canonical form manifests loop economy (or alternatively we may say that the rederived form exhibits unprovable loop shortcutting). Loop economy may also be manifested by the rederived form. For example, consider the function NEXT2 whose definition, LAP encoding, canonical form, and rederived forms are given in Figures 5.10a, 5.10b, 5.10c, and 5.10d respectively. Note the duality between loop economy and loop shortcutting.

```
NEXT2(L,X) ≡ if NULL(L) then NIL
             else if CAR(L) EQ X then
                if NULL(CDR(L)) then NIL
                else CADR(L)
             else if NULL(CDR(L)) then NIL
             else NEXT2(CDR L,X)
```

Figure 5.10a - MLISP Definition of NEXT2

```
(LAP NEXT2 SUBR)
NEXT2   (JUMPE 1 DONE)        jump to DONE if L is NIL
        (HLRZ 3 0 1)          load register 3 with CAR(L)
        (HRRZ 1 0 1)          load register 1 with CDR(L)
        (CAIE 3 0 2)          skip if CAR(L) EQ X
        (JRST 0 NEXT2)        compute NEXT(CDR(L),X)
        (SKIPE 0 1)           skip if CDR(L) is NIL
        (HLRZ 1 0 1)          load register 1 with CAR(CDR(L))
DONE    (POPJ P)              return
        NIL
```

Figure 5.10b - LAP Encoding of NEXT2

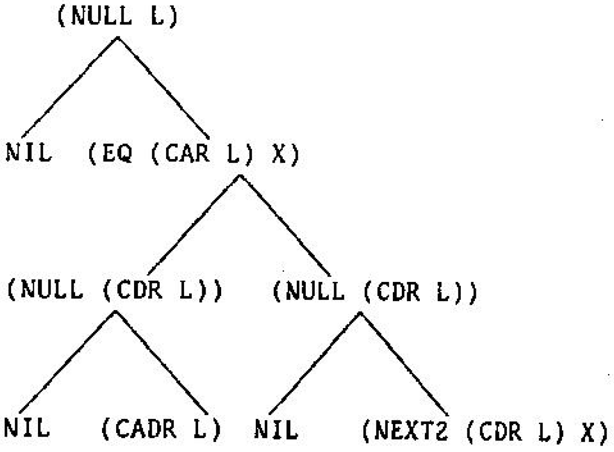


Figure 5.10c - Canonical Form

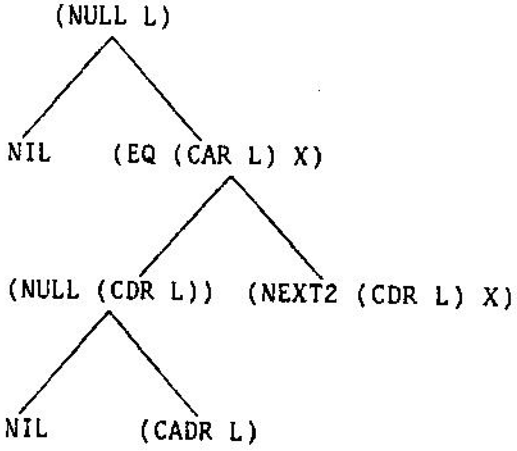


Figure 5.10d - Rederived Form

Clearly, the interesting cases are when the rederived form exhibits loop economy or provable loop shortcutting. This stems from our interest in optimization. In this work we only handle provable loop shortcutting. Loop economy is left for future work. This is a more difficult problem because the bypassed paths act as a pruning device in the search for the path that was bypassed by shortcutting in the matching procedure. However, for loop economy (as well as unprovable loop shortcutting), we would have mismatches in recursive calls that would require the recursive call in the rederived form to be expanded. This task is rendered more difficult since we no longer have the luxury of knowing the possible expansions.

5.D Matching Recursive Calls Bypassing the Start of the Program

The problem of proving equivalence when recursive calls in the LAP program have been implemented by jumps to points other than the start of the program is quite complex. The algorithm that we present reflects our implementation and is complete for a certain subcase of these backward jumps. We also discuss several extensions to the algorithm which will make it complete. However, in most cases that we would encounter, the algorithm is more than adequate. The discussion of extensions merely serves to indicate the type of techniques necessary to solve the basic problem.

(1) Perform as much matching as possible by manipulating the canonical form to match the rederived form. By as much as possible we mean that if a mismatch occurs on one path, then process the alternate path in the same manner. The result of the procedure is a transformed canonical form, say CANRESULT.

(2) If no mismatches occurred, then we have successfully managed to prove that the canonical form can be transformed in a legal manner to match the rederived form.

(3) At least one mismatch has occurred. There are several possible causes.

(a) Computations were encountered that could not be matched. This could have been caused by the expansion of a recursive call as in the case of loop shortcutting or by an error. The exact reason is resolved by the remainder of the algorithm.

(b) An attempt was made to match a result computation (i.e. a terminal node) in the rederived form against a condition (i.e. a non terminal node) in the canonical form. If the condition is a predicate of a primitive nature having identical conclusion and alternative clauses, then the remainder of the algorithm will apply axiom (1).

Renumber both the original rederived form and the canonical form resulting from step (1) – i.e. CANRESULT. However, this time the rederived form receives the higher numbers. Renumbering is accomplished by using the algorithm mentioned in the previous section. At this point we attempt to manipulate the rederived form to match the canonical form. Note that this is the reverse of the procedure applied in step (1) because we wish to detect the exact instance of the mismatches so that we can determine if they are caused by recursive calls in the canonical form which have been expanded in the rederived form via loop shortcutting. If all mismatches are of this nature, then proceed to step (4). Otherwise, the algorithm terminates with a result of inequivalence or an inability to prove equivalence.

All mismatches that were caused by case (b) with a primitive predicate and identical conclusion and alternative clauses no longer exist because the matching algorithm applies axiom (1) followed by axioms (5) and (6). For example, suppose we try to match (CDR A) with the form in Figure 5.11. In this case, the equivalence should hold. If the former is in the rederived form and the latter is in the canonical form, then when attempting to manipulate the canonical form to match the rederived form, the matching procedure will fail and declare a mismatch because of case (b). The next attempt at matching will manipulate the rederived form to match the canonical form. We take advantage of the fact that EQ is a primitive predicate and that the computations (CDR A) and F are inevitable to replace (CDR A) in the rederived form by the form in Figure 5.12. The equivalence should be obvious. Note that no special handling was required; this is simply a part of the matching algorithm since it involves the application of axiom (1). If in fact the condition was in the rederived form and the result in the canonical form, then step (1) would not have detected any mismatch since axiom (1) would have been applied as indicated here.

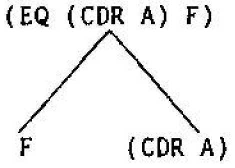


Figure 5.11

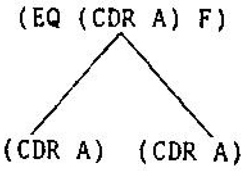


Figure 5.12

Step (3) is necessary since in the case of mismatches due to loop shortcutting, step (1) will detect a mismatch when a computation is performed in the rederived form and not in the canonical form or vice versa. Most often, the mismatch is of the former type and is usually in the condition although it may also be in the result clause. We now give an example of a mismatch occurring because a computation performed in the canonical form was not performed in the rederived form. Consider the optimal encoding for the function NEXT. In this case we will detect a mismatch when attempting to match the form given in Figure 5.13 occurring in the rederived form with (NEXT (CDR L) X) in the canonical form. The first step is to apply axiom (1) to (NEXT (CDR L) X) in the canonical form since (CDR L) and F are inevitable computations and EQ is a primitive predicate. The resulting form is given in Figure 5.14. We now must match the conclusion and alternative clauses. The alternative clause is seen to match; however, the conclusion clause does not match – i.e. F is not matched by (NEXT (CDR L) X). Application of step (3) will result in an inability to match (NEXT (CDR L) X) with the condition in the rederived form. Thus we see that step (3) will indicate if the cause of the mismatch was loop shortcutting of a recursive call.

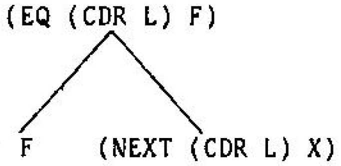


Figure 5.13

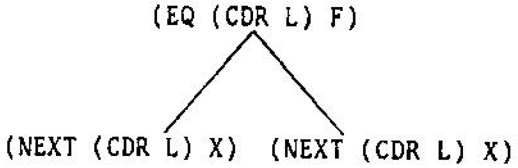


Figure 5.14

(4) All mismatches have been caused by recursive calls in the the canonical form which have not been matched in the rederived form. We now prepare to determine the elements of SHORTCUT that can replace the mismatching recursive calls. For each mismatching recursive call in the canonical form, remove all computations in its subpaths of higher computation number. Thus the canonical form is modified so that wherever a mismatch occurred, the mismatching recursive call is a terminal node. Next, perform matching by manipulating the rederived form to match the canonical form. For each recursive call encountered in the canonical form, say MISCALL with computation number MISNUM, that cannot be matched do the following until a match is found or SHORTCUT is exhausted.

(4.1) Obtain a candidate rederived form from SHORTCUT, say CAND.

(4.2) Assign new computation numbers in CAND to all computations other than non–variable atoms, local parameters, and initial SPECIAL variable bindings. The new numbers are the old numbers incremented by MISNUM. Let HIGHNUM denote the highest reassigned computation number.

(4.3) Repeat step (4.2) for the part of the rederived form which we will be manipulating to match the canonical form. The difference is that only computation numbers greater than MISNUM will be incremented, and the increment is HIGHNUM–MISNUM. This insures that the matching procedure will be manipulating a form having higher computation numbers.

(4.4) Replace all local variables in CAND by the binding assigned to them by MISCALL.

(4.5) Replace all initial SPECIAL variable bindings in CAND by their bindings immediately prior to MISCALL.

(4.6) Replace MISCALL by the form CAND which has undergone the modifications specified in steps (4.2), (4.4), and (4.5). If MISCALL was part of an FN list, then add its non-result arguments to all terminal nodes in CAND.

(4.7) Apply the duplicate predicate removal procedure to CAND.

(4.8) Attempt to match the result of step (4.7) against the rederived form by manipulating the latter.

(5) When all mismatches involving recursive calls have been resolved, we need to reinvoke the algorithm (i.e. at step (1)). However, we first substitute the appropriate elements of SHORTCUT for the mismatching recursive calls in the canonical form and do the following.

(5.1) Assign computation numbers to the canonical form so that computations performed subsequent to the recursive calls that have been replaced by elements of SHORTCUT will have higher computation numbers associated with them.

(5.2) Apply steps (2) and (3) of the postprocessing stage of the rederivation process (see Section 4.G4) to the canonical form.

(5.3) Make sure that all SPECIAL variables have the appropriate bindings if the function that has been replaced by an element of SHORTCUT modifies SPECIAL variables. This is necessary because the function has been replaced by an element of SHORTCUT and thus subsequent occurrences of the SPECIAL variables whose bindings refer to the function most recently executed that could have modified them must be updated to refer to the most recent value of the variable.

(5.4) Apply the duplicate computation removal procedure to the resulting canonical form.

Computation numbers corresponding to unspecified parameters are ignored in the matching process since the latter are atoms rather than functions. When searching the canonical form for a matching instance of a computation having unspecified parameters, then failure will result. However, this is no cause for concern since unspecified parameters denote that loop shortcutting has taken place. When loop shortcutting occurs, the rederived form is manipulated to match the canonical form. Thus whenever a recursive call in the canonical form, say A, is explicitly matched by a recursive call in the rederived form containing unspecified parameters; these unspecified parameters take on their corresponding bindings in A. This is legal when we recall the definition of unspecified parameters – i.e. they are analogous to a don't care. Thus we see that the procedure of loop shortcutting may also yield the bindings of unspecified parameters. The fact that the canonical form does not contain any unspecified parameters means that there must be no occurrences of unspecified parameters in the result of the application of step (4.7) (duplicate predicate removal) to a candidate rederived form. If the latter criterion is violated, then we cease processing the offending candidate rederived form.

Closer examination of the algorithm should reveal a similarity to the rederivation process. Termination can be shown by observing that whenever step (1) is reinvoked, we are processing recursive calls that have been replaced by an element of SHORTCUT at a higher level than previously. Moreover, since the rederived form does not grow, termination is guaranteed.

As the algorithm currently stands, there are several implicit restrictions on the structure of shortcutting that can be proved. Step (4.1) which obtains an arbitrary candidate rederived form means that sometimes more than one candidate could possibly match the rederived form with the failure being detected only at a subsequent level of processing. This failure will not be followed by backtracking and thus we will fail to detect equivalence. Thus step (4.1) will handle correctly at all times only shortcutting at a terminal node (i.e. no further computation along the path). This can be alleviated by assigning an order to the paths.. This order is determined by examining the bypassed paths corresponding to each element of SHORTCUT. We note that if a set of backward paths is a subset of another set or if individual paths in one element of SHORTCUT are subpaths of paths in other elements of SHORTCUT, then in either of these examples the containing elements of SHORTCUT are to be attempted first. Thus we see a partial ordering among the elements of SHORTCUT. The precedence relations can be converted into an ordering for the elements of SHORTCUT via topological sorting[Knuth68].

It should also be noted that during the matching of step (4.8) we will not attempt any further shortcutting to resolve any more unmatched recursive calls occurring in the canonical form. Thus it could be said that we only process shortcutting to one level. Another restriction is that if any of the elements of SHORTCUT refer to unspecified parameters, the latter are not updated as bindings for these parameters are ascertained. We leave these two problems for future work.

An ideal algorithm would simply assume a specific element of SHORTCUT and continue matching with backtracking upon failure. Nevertheless, this is quite costly in a computational sense and we feel that our algorithm, although at times a heuristic, deals adequately with the majority of cases that will be encountered since most often there is only one bypassed path. However, the addition of topological sorting for ordering the elements of SHORTCUT would get rid of some of the present taint.

As a final remark we observe that we used the rederived form (i.e. elements of SHORTCUT) to replace recursive calls in the canonical form. This is permissible because once the right element of SHORTCUT is determined, say A, its substitution is legal by induction. Notice that A has been proved to be capable of being bypassed, and from Chapter 4 we recall that it is of lower level than the recursive call that it is replacing. The latter coupled with our earlier comment about the subsequent invocation of step (1) of the matching algorithm being analogous to processing a higher level, show that A represents a series of computations that has already been matched and thus its substitution into the canonical form is valid. An argument against using the canonical form for expanding recursive calls (see Figure 5.15) is that conditions may have been rearranged and thus partial expansions such as those available for the rederived form would not even be correct. The question of their availability alone is the main problem associated with loop economy – i.e. there exists no set of candidate forms for expansion into locations where loop economy occurs.

```
NEXT(L,X) ≡ if NULL(L) then NIL
            else if CAR(L) EQ X then
               if NULL(CDR(L)) then NIL
               else CADR(L)
            else if NULL(CDR(L)) then NIL
            else if CAR(CDR(L)) EQ X then
               if NULL(CDR(CDR(L)))  then NIL
               else CADR(CDR(L))
            else NEXT(CDR(CDR(L)),X)
```

Figure 5.15

CHAPTER 6

DEBUGGING

In the previous chapter we have shown how a program is proved to be equivalent to another program provided that the algorithms were identical. When a proof cannot be constructed, an error may have occurred. In this chapter we discuss the type of errors that can be detected by the proof procedure. In addition, we present several encodings of a rather complex function, HIER1, which are used to demonstrate that equivalence can be proved on a large scale as well as show how errors are detected. The latter is done by examining an incorrect LAP encoding of the algorithm and showing how our proof system pinpoints the errors. This procedure is accompanied by a description of an error correction procedure which we believe could be implemented with some degree of success. In other words a semi-automatic debugging procedure is illustrated in addition to a proposal for an automatic debugging procedure.

6.A Errors

In Chapter 4 we discuss restrictions on the LAP program which pertain to well- formedness as well as to what constitutes valid results. Appendix 4 contains an enumeration of some of the errors that can be detected during the rederivation process. This detection mechanism includes, via use of the storage history field in the memory descriptor, a means of detecting the history of storage operations in the cell whose invalid contents may have caused the error. In addition to returning a symbolic and numeric representation of the program, the rederivation procedure yields a dictionary whose elements are the computation numbers used in the numeric representation. For each such computation number, the dictionary contains the instruction number at which it was computed and a list of ordered pairs corresponding to the path along which the error occurred. Each element of the list consists of a label and a number denoting the stack depth at the label. Note that the return address is always the bottom-most entry and is considered to be at a stack depth of 1.

The input to the proof procedure presented in Chapter 5 is a well-formed program in the sense that all entries in the symbolic representation part of the rederived form represent LISP functions or are NIL. The latter denotes an invalid conclusion or alternative (i.e. an error occurred on the path). The matching procedure also enables the detection of errors. These errors fall into four general classes.

(1) An unknown conclusion or alternative clause was encountered in the rederived form. This is a result of an error in the well-formedness of the program and was actually detected during the rederivation process. Information about the nature of the error is available with the output from the rederivation process. The reason for its appearance in the matching phase is that we did not wish to cease processing the remainder of the program just because an error occurred in one path. In other words we wish to detect as many errors as possible.

(2) All computations along a path in the rederived form were matched with computations in the canonical form but some of the computations in the canonical form do not appear in the rederived form. This is caused by either non-result arguments of an FN construct in the canonical form not appearing in the rederived form, or a predicate present in the canonical form is not present in the rederived form. In our case this predicate would have to be non-primitive (i.e. EQUAL). The same type of error can also occur when all computations

along a path in the canonical form were matched with computations in the rederived form but some of the computations in the rederived form do not appear in the canonical form.

(3) The result of a specific path (or the result argument of an FN construct) in the canonical form is not equivalent to the corresponding element in the rederived form, or the result of a specific path in the rederived form is not equivalent to the corresponding element in the canonical form. For example consider the two trees given in Figures 6.1a and 6.1b.

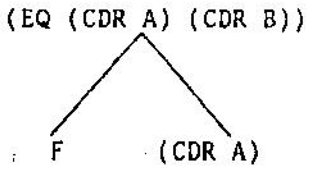


Figure 6.1a

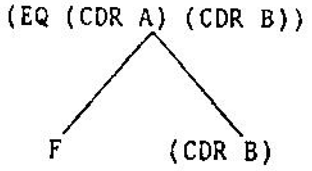


Figure 6.1b

Clearly, all computations performed in Figure 6.1a are also performed in Figure 6.1b. However, the results of the two right subtrees are not equivalent (i.e. (CDR A) is not equivalent to (CDR B)).

(4) A function in the rederived form cannot be matched up with a function in the canonical form. This is caused by such factors as invalid rearranging of computations, mistakes in the LAP program, invalid optimizations, etc. Some of the errors that were detected when an invalidly translated LISP to LAP program (i.e. HIER1) was presented to the system are discussed in the next section. These errors included use of wrong accumulators, erroneous assumptions about the contents of certain accumulators, misuse of antisymmetry, misspelling of opcodes and operands thereby causing the wrong instruction to be executed, and testing the wrong sense of an instruction.

When errors of type (1)–(3) occur, the system will return a message indicating the error type. In types (2) and (3) we indicate whether the error occurred in the canonical form or in the rederived form. In all three types we indicate the erroneous computation (somewhat meaningless for type (1) errors) as well as what should have been computed according to the canonical form. In addition, the values of the conditions in terms of truth values are given so that the offending path can be identified.

When errors of type (4) occur, the system returns the invalid computation along with the computation dictionary entry corresponding to the computation number of the outermost function – i.e. the address of the instruction computing this function and the pairs of labels and stackdepths associated with the path. The actual error is caused by either the wrong function applied to a set of arguments or the function applied to the wrong set of arguments. For example, consider an error in *LESS(A,B). The error could be that we desire *GREAT(A,B) or possibly *LESS(A,C). The matching system indicates that an error has occurred when attempting to match the computation *LESS(A,B). In addition, it also returns the address of the instruction corresponding to the *LESS function which is denoted as the location of error as well as the path along which the error was detected. Thus when debugging the program we must ascertain whether the error was in the function or in the arguments.

In the future we would like to be able to have the program infer the errors and attempt to correct them. This may not be possible for all of the errors, but clearly some of the errors of type (4) could possibly be corrected. These include misspelling of opcodes and operands, testing the wrong sense of the instruction, misuses of antisymmetry, misuse of the contents of accumulators and others. Such error correction would probably have to be done using such techniques as hypothesis and test[Newell73] and would invariably involve backtracking[Golomb65].

6.B HIER1

We will examine the error detection capabilities of the system by using a rather complex function known as HIER1. This algorithm originated in the FOL[Weyhrauch74] system where it is used extensively. We will not dwell to any length on the actual effect of the function except for the following brief summary. The result of application of the function is to convert a list representing an expression with prefix and infix operators to a tree-like representation. The primary driving force in the determination of the operands corresponding to each of the operators is a set of binding powers (operator precedence). The second argument to the function denotes the binding power of the operator corresponding to the expression in question.

The motivation for using this example is that it is large, can be optimized using many of the ideas that have interlaced our previous discussions, and generally serves to indicate the potential use for a verification system. We do not present the proof procedure for this function. Instead, we give its encoding in MLISP and LISP, and several LAP encodings. The first LAP encoding indicates the code generated by the Stanford LISP 1.6 compiler with one exception. Namely, multiple CAR-CDR operations such a (CDDR L) are performed inline. The second encoding denotes a hand optimized program. The final encoding is an erroneous version of the second LAP program. The errors were not intentional. They occurred during the hand optimization procedure. Their inclusion here serves to show the error detection and pinpointing capability of the system. In fact our discussion will dwell on the error detection capability, and we will demonstrate how the system detected and pinpointed the location of each error. During this process we will successively make the corrections deemed necessary by the error detection mechanism. Once all of the errors have been detected and corrected, we will have a program identical to the correct hand optimized version of the program. The compiler generated and hand optimized encodings were proved equivalent by the system; however, inclusion of the proof would not reveal anymore than can be gleamed from the examples to be given in Chapter 7.

6.B1 LISP Encoding of HIER1

```
(DEFPROP HIER1
 (LAMBDA (L RBP)
  (COND ((AND (NULL (CAR L)) (NULL (CDDR L))) L)
         ((NULL (CDDR L)) (HIER1 (LIST (CDR (CAR L))
                                       (CONS (CAAR L) (CADR L)))
                                 RBP))
         ((NULL (CAR L))
          (COND ((NOT (*LESS RBP (BP1 (CAADDR L) (QUOTE LEFT&)))) L)
                (T (HIER1 (CONS F
                                (CONS (CONS (CAADDR L)
                                            (CONS (CADR L)
                                                  (CADR
                                                   (SETQ L
                 (HIER1 (CONS (CAR (CDADDR L))
                              (CONS (CADR (CDADDR L)) (CDDDR L)))
                        (BP1 (CAADDR L) (QUOTE RIGHT&)))))))
```

```
                                       (CDDR L)))
                          RBP))))
         ((NOT (*LESS (BP1 (CAAR L) (QUOTE PRIGHT&))
                      (BP1 (CAADDR L) (QUOTE LEFT&))))
          (HIER1 (CONS (CDR (CAR L))
                       (CONS (CONS (CAAR L) (CADR L)) (CDDR L)))
                 RBP))
         (T (HIER1 (CONS (CAR L)
                         (CONS (CADR (SETQ L
                            (HIER1 (CONS F (CDR L))
                                   (BP1 (CAAR L) (QUOTE PRIGHT&)))))
                               (CDDR L)))
                   RBP))))
EXPR)

(DEFPROP BP1
 (LAMBDA (X Y) (GET X Y))
EXPR)
```

6.B2 MLISP Encoding of HIER1

Note the use of square brackets. This is an MLISP construct which is very useful in visualizing the structure of a list. Each index indicates a number, say num, which is interpreted as being equivalent to num−1 CDR operations followed by a CAR operation. The brackets can be likened to a function whose arguments indicate a sequence of CDR and CAR operations applied from left to right. For example L[2,1] is equivalent to (CAADDR L). Angle brackets are used to indicate a list consisting of the elements separated by commas within the angled brackets. For example, <A,B,C> is equivalent to LIST(A,B,C). We also mention the use of the single quote symbol instead of the word QUOTE. The symbol ? indicates quotation of a single character for scanning purposes due to the character having a predefined meaning in MLISP.

```
EXPR HIER1 (L, RBP);
        IF NULL L[1] & NULL CDDR L THEN L
        ELSE IF NULL CDDR L THEN HIER1(<CDR L[1], L[1,1] CONS L[2]>,
                                       RBP)
        ELSE IF NULL L[1] THEN
                IF RBP ≥ BP1(L[3,1],'LEFT?&) THEN L
                ELSE HIER1(NIL CONS (L[3,1] CONS (L[2] CONS (L ←
                        HIER1(L[3,2] CONS L[3,3] CONS CDDDR L,
                              BP1(L[3,1],'RIGHT?&)))[2]))
                        CONS CDDR L, RBP)
        ELSE IF BP1(L[1,1],'PRIGHT?&) ≥ BP1(L[3,1],'LEFT?&) THEN
                HIER1(CDR L[1] CONS (L[1,1] CONS L[2]) CONS CDDR L,
                      RBP)
        ELSE HIER1(L[1] CONS (L ←
                    HIER1(NIL CONS CDR L, BP1(L[1,1],'PRIGHT?&)))[2]
                    CONS CDDR L, RBP);

EXPR BP1 (X, Y);
        GET(X,Y);
```

6.B3 Compiler Generated Code for HIER1

```
(LAP HIER1 SUBR)
        (PUSH P 1)
        (PUSH P 2)
        (HLRZ@ 1 1)
        (JUMPN 1 TAG2)
        (HRRZ@ 1 -1 P)
        (HRRZ@ 1 1)
        (JUMPN 1 TAG2)
        (MOVE 1 -1 P)
        (JRST 0 TAG1)
TAG2    (HRRZ@ 1 -1 P)
        (HRRZ@ 1 1)
        (JUMPN 1 TAG4)
        (HRRZ@ 2 -1 P)
        (HLRZ@ 2 2)
        (HLRZ@ 1 -1 P)
        (HLRZ@ 1 1)
        (CALL 2 (E CONS))
        (CALL 1 (E NCONS))
        (HLRZ@ 2 -1 P)
        (HRRZ@ 2 2)
        (CALL 2 (E XCONS))
        (MOVE 2 0 P)
        (CALL 2 (E HIER1))
        (JRST 0 TAG1)
TAG4    (HLRZ@ 1 -1 P)
        (JUMPN 1 TAG5)
        (MOVEI 2 (QUOTE LEFT&))
        (HRRZ@ 1 -1 P)
        (CALL 1 (E CAADR))
        (CALL 2 (E BP1))
        (MOVE 2 0 P)
        (CALL 2 (E *GREAT))
        (JUMPN 1 TAG7)
        (MOVE 1 -1 P)
        (JRST 0 TAG6)
TAG7    (HRRZ@ 2 -1 P)
        (HRRZ@ 2 2)
        (HRRZ@ 2 2)
        (HRRZ@ 1 -1 P)
        (CALL 1 (E CDADR))
        (CALL 1 (E CADR))
        (CALL 2 (E CONS))
        (HRRZ@ 2 -1 P)
        (HRRZ@ 2 2)
        (HLRZ@ 2 2)
        (HRRZ@ 2 2)
        (HLRZ@ 2 2)
        (CALL 2 (E XCONS))
        (PUSH P 1)
        (HRRZ@ 1 -2 P)
        (HLRZ@ 1 1)
        (PUSH P 1)
        (HRRZ@ 1 -3 P)
        (CALL 1 (E CAADR))
        (MOVEI 2 (QUOTE RIGHT&))
        (PUSH P 1)
        (HRRZ@ 1 -4 P)
```

```
          (CALL 1 (E CAADR))
          (CALL 2 (E BP1))
          (MOVE 2 1)
          (EXCH 1 -2 P)
          (CALL 2 (E HIER1))
          (HRRZ@ 2 1)
          (HLRZ@ 2 2)
          (EXCH 1 -1 P)
          (CALL 2 (E CONS))
          (POP P 2)
          (CALL 2 (E XCONS))
          (HRRZ@ 2 0 P)
          (HRRZ@ 2 2)
          (CALL 2 (E CONS))
          (MOVEI 2 (QUOTE NIL))
          (CALL 2 (E XCONS))
          (MOVE 2 -2 P)
          (CALL 2 (E HIER1))
          (POP P -3 P)
          (SUB P (C 0 0 1 1))
TAG10
TAG6      (JRST 0 TAG1)
TAG5      (MOVEI 2 (QUOTE PRIGHT&))
          (HLRZ@ 1 -1 P)
          (HLRZ@ 1 1)
          (CALL 2 (E BP1))
          (MOVEI 2 (QUOTE LEFT&))
          (PUSH P 1)
          (HRRZ@ 1 -2 P)
          (CALL 1 (E CAADR))
          (CALL 2 (E BP1))
          (POP P 2)
          (CALL 2 (E *GREAT))
          (JUMPN 1 TAG12)
          (HRRZ@ 2 -1 P)
          (HLRZ@ 2 2)
          (HLRZ@ 1 -1 P)
          (HLRZ@ 1 1)
          (CALL 2 (E CONS))
          (HRRZ@ 2 -1 P)
          (HRRZ@ 2 2)
          (CALL 2 (E CONS))
          (HLRZ@ 2 -1 P)
          (HRRZ@ 2 2)
          (CALL 2 (E XCONS))
          (MOVE 2 0 P)
          (CALL 2 (E HIER1))
          (JRST 0 TAG1)
TAG12     (HRRZ@ 2 -1 P)
          (MOVEI 1 (QUOTE NIL))
          (CALL 2 (E CONS))
          (PUSH P 1)
          (HLRZ@ 1 -2 P)
          (MOVEI 2 (QUOTE PRIGHT&))
          (PUSH P 1)
          (HLRZ@ 1 -3 P)
          (HLRZ@ 1 1)
          (CALL 2 (E BP1))
          (MOVE 2 1)
          (EXCH 1 -1 P)
          (CALL 2 (E HIER1))
```

```
         (HRRZ@ 2 1)
         (HRRZ@ 2 2)
         (MOVEM 1 -3 P)
         (CALL 1 (E CADR))
         (CALL 2 (E CONS))
         (POP P 2)
         (CALL 2 (E XCONS))
         (MOVE 2 -1 P)
         (CALL 2 (E HIER1))
         (SUB P (C 0 0 1 1))
TAG13
TAG1     (SUB P (C 0 0 2 2))
         (POPJ P)
         NIL
```

6.B4 Hand Optimized Code for HIER1

The optimized encoding makes use of several optimizations which are briefly described. We first note that recursion is accomplished by bypassing the start of the program via use of the label HIER1A. Moreover, for the recursive calls the second argument need not be present in accumulator 2 because we have proved (i.e. in Pass Two of the rederivation process) that accumulator 2 is never referenced prior to being destroyed. Thus in the recursive call we do not need to follow the calling sequence which makes use of accumulators. Moreover, we notice that in several cases the second argument is already on the stack and thus there is no need to place it on the stack again. Hence, the first instruction may be bypassed. Other optimizations include common subexpression elimination and a wide use of accumulators to store temporary values across functions that do not destroy the contents of all the accumulators (CONS and XCONS in this example). Finally, note the efficient compilation of the conditions so that redundant tests are avoided. This was a problem in the original LAP program due to the use of the AND operation in some of the conditions.

```
(LAP HIER1 SUBR)
         (PUSH P 2)
HIER1A   (HLRZ 5 0 1)
         (JUMPN 5 TAG2)
         (HRRZ 4 0 1)
         (HRRZ 3 0 4)
         (JUMPE 3 TAGA)
         (JRST 0 TAGB)
TAG2     (HRRZ 4 0 1)
         (HRRZ 3 0 4)
         (JUMPN 3 TAGC)
         (HLRZ 1 0 5)
         (HLRZ 2 0 4)
         (CALL 2 (E CONS))
         (CALL 1 (E NCONS))
         (HRRZ 2 0 5)
         (CALL 2 (E XCONS))
         (JRST 0 HIER1A)
TAGB     (PUSH P 1)
         (HLRZ 1 0 1)
         (MOVEI 2 (QUOTE LEFT&))
         (CALL 2 (E BP1))
         (MOVE 2 -1 P)
         (CALL 2 (E *GREAT))
         (JUMPN 1 TAG7)
```

```
        (POP P 1)
        (JRST 0 TAGA)
TAG7    (PUSH P (C 0 0 TAGX))
        (HRRZ@ 1 -1 P)
        (HRRZ 1 0 1)
        (HLRZ 1 0 1)
        (HLRZ 1 0 1)
        (MOVEI 2 (QUOTE RIGHT&))
        (CALL 2 (E BP1))
        (PUSH P 1)
        (HRRZ@ 5 -2 P)
        (HRRZ 4 0 5)
        (HLRZ 3 0 4)
        (HRRZ 5 0 3)
        (HRRZ 1 0 4)
        (HRRZ 4 0 5)
        (HLRZ 2 0 4)
        (CALL 2 (E XCONS))
        (HLRZ 2 0 5)
        (CALL 2 (E XCONS))
        (JRST 0 HIER1A)
TAGX    (HRRZ 5 0 1)
        (HLRZ 2 0 5)
        (HRRZ@ 4 0 P)
        (HLRZ 1 0 4)
        (CALL 2 (E CONS))
        (HRRZ 3 0 4)
        (HLRZ 2 0 3)
        (HLRZ 2 0 2)
        (CALL 2 (E XCONS))
        (HRRZ 2 0 5)
        (CALL 2 (E CONS))
        (MOVEI 2 (QUOTE NIL))
        (CALL 2 (E XCONS))
        (SUB P (C 0 0 1 1))
        (JRST 0 HIER1A)
TAGC    (PUSH P 1)
        (HLRZ 1 0 3)
        (HLRZ 1 0 1)
        (MOVEI 2 (QUOTE LEFT&))
        (CALL 2 (E BP1))
        (PUSH P 1)
        (HLRZ@ 1 -1 P)
        (HLRZ 1 0 1)
        (MOVEI 2 (QUOTE PRIGHT&))
        (CALL 2 (E BP1))
        (POP P 2)
        (CALL 2 (E *LESS))
        (JUMPN 1 TAG12)
        (HLRZ@ 5 0 P)
        (HLRZ 1 0 5)
        (HRRZ@ 4 0 P)
        (HLRZ 2 0 4)
        (CALL 2 (E CONS))
        (HRRZ 2 0 4)
        (CALL 2 (E CONS))
        (HRRZ 2 0 5)
        (CALL 2 (E XCONS))
        (SUB P (C 0 0 1 1))
        (JRST 0 HIER1A)
TAG12   (PUSH P (C 0 0 TAGY))
```

```
          (HLRZ@ 1 -1 P)
          (HLRZ 1 0 1)
          (MOVEI 2 (QUOTE PRIGHT&))
          (CALL 2 (E BP1))
          (PUSH P 1)
          (HRRZ@ 2 -2 P)
          (MOVEI 1 (QUOTE NIL))
          (CALL 2 (E CONS))
          (JRST 0 HIER1A)
TAGY      (HRRZ 5 0 1)
          (HLRZ 1 0 5)
          (HRRZ 2 0 5)
          (CALL 2 (E CONS))
          (HLRZ@ 2 0 P)
          (CALL 2 (E XCONS))
          (SUB P (C 0 0 1 1))
          (JRST 0 HIER1A)
TAGA      (SUB P (C 0 0 1 1))
          (POPJ P)
          NIL
```

6.B5 Erroneous Hand Optimized Code for HIER1

A number of errors were made unintentionally in the process of optimizing the function, These errors were saved and later the erroneous version was presented as input to the proof system. In this discussion we present the successive encodings of the function as errors are discovered and corrected as well as the reason for the errors. The actual encoding is presented in such a manner that the instruction labels (original and internally generated) are shown along with the value of the program counter. The first time the function was processed, errors were detected by both the rederivation and proof procedures. In all subsequent steps, the errors were of a nature that could only be detected by the proof procedure (i.e. the program satisfied well–formedness).

For each error detected, we indicate the symbolic and numeric representation associated with the computations involved. In order to facilitate the interpretation of these numerical quantities, we give their corresponding computation number dictionary entries. The program counter value at which the error is detected is also indicated. Finally, recall our discussion of errors in the previous section where we pointed out that when an error is detected, the instruction number given by the error detector corresponds to either an error in the said instruction or in the arguments to the function invoked by the said instruction.

Notice that we do not yet have a mechanism of indicating how the error is to be fixed. Thus our analysis merely will show that an error did indeed occur at the location indicated by the error message. The proof that we are right unfolds as we notice that after each correction the encoding grows more and more similar to the correct version in the previous section. Thus the analysis is basically a step by step debugging of a program using the information given by the error messages. There is probably not enough information given here for the reader to fully understand what is going on. It would be preferable if the canonical form and an indication as to the computation that is expected were given. However, this is not done due to space limitations. Nevertheless, in the analysis we indicate the desired computations. Hopefully, this information will ease the process. In any case, the main point of each *bug* is to indicate the type of error that was encountered. Thus the reader can immerse himself in the examples to the depth of his comprehension without fear of drowning since there is always a next example acting as a life saver.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1             1            (PUSH 12 2)
     HIER1A            2            (HLRZ 5 0 1)
                       3            (JUMPN 5 TAG2)
     LX001             4            (HRRZ 4 0 1)
                       5            (HRRZ 3 0 4)
                       6            (JUMPE 3 TAGA)
     LX002             7            (JRST 0 TAGB)
     TAG2              8            (HRRZ 4 0 1)
                       9            (HRRZ 3 0 4)
                       10           (JUMPN 3 TAGC)
     LX003             11           (HLRZ 1 0 5)
                       12           (HLRZ 2 0 4)
                       13           (CALL 2 (E CONS))
     LX004             14           (CALL 1 (E NCONS))
     LX005             15           (HRRZ 2 0 4)
                       16           (CALL 2 (E XCONS))
     LX006             17           (JRST 0 HIER1A)
     TAGB              18           (PUSH 12 1)
                       19           (HLRZ 1 0 3)
                       20           (HLRZ 1 0 1)
                       21           (MOVEI 2 (QUOTE LEFT&))
                       22           (CALL 2 (E BP1))
     LX007             23           (MOVE 2 -1 12)
                       24           (CALL 2 (E *GREAT))
     LX008             25           (JUMPN 1 TAG7)
     LX009             26           (POP 12 1)
                       27           (JRST 0 TAGA)
     TAG7              28           (HRRZ@ 1 0 12)
                       29           (HRRZ 1 0 1)
                       30           (HLRZ 1 0 1)
                       31           (HLRZ 1 0 1)
                       32           (MOVEI 2 (QUOTE RIGHT&))
                       33           (CALL 2 (E BP1))
     LX010             34           (PUSH 12 1)
                       35           (HRRZ@ 5 -1 12)
                       36           (HRRZ 4 0 5)
                       37           (HLRZ 3 0 4)
                       38           (HLRZ 5 0 3)
                       39           (HRRZ 1 0 4)
                       40           (HLRZ 4 0 5)
                       41           (HLRZ 2 0 4)
                       42           (CALL 2 (E CONS))
     LX011             43           (MOVE 2 4)
                       44           (CALL 2 (E XCONS))
     LX012             45           (PUSHJ 12 HIER1A)
     TAGX              46           (HRRZ 5 0 1)
                       47           (HLRZ 2 0 1)
                       48           (HRRZ@ 4 0 12)
                       49           (HLRZ 1 0 4)
                       50           (CALL 2 (E CONS))
     LX013             51           (HRRZ 3 0 4)
                       52           (HLRZ 2 0 3)
                       53           (HLRZ 2 0 2)
                       54           (CALL 2 (E XCONS))
     LX014             55           (HRRZ 2 0 5)
                       56           (CALL 2 (E CONS))
     LX015             57           (MOVEI 2 (QUOTE NIL))
```

```
                 58        (CALL 2 (E XCONS))
    LX016        59        (SUB 12 (C 0 0 1 1))
                 60        (JRST 0 HIER1A)
    TAGC         61        (PUSH 12 1)
                 62        (HLRZ 1 0 3)
                 63        (HLRZ 1 0 1)
                 64        (MOVEI 2 (QUOTE LEFT&))
                 65        (CALL 2 (E BP1))
    LX017        66        (PUSH 12 1)
                 67        (HLRZ@ 1 -1 12)
                 68        (HLRZ 1 0 1)
                 69        (MOVEI 2 (QUOTE PRIGHT))
                 70        (CALL 2 (E BP1))
    LX018        71        (POP 12 2)
                 72        (CALL 2 (E *GREAT))
    LX019        73        (JUMPE 1 TAG12)
    LX020        74        (HLRZ@ 5 0 12)
                 75        (HLRZ 1 0 5)
                 76        (HRRZ@ 4 0 12)
                 77        (HLRZ 2 0 4)
                 78        (CALL 2 (E CONS))
    LX021        79        (HRRZ 2 0 4)
                 80        (CALL 2 (E CONS))
    LX022        81        (HRRZ 2 0 5)
                 82        (CALL 2 (E XCONS))
    LX023        83        (SUB 12 (C 0 0 1 1))
                 84        (JRST 0 HIER1A)
    TAG12        85        (HLRZ@ 1 0 12)
                 86        (HLRZ 1 0 1)
                 87        (MOVEI 2 (QUOTE PRIGHT&))
                 88        (CALL 2 (E BP1))
    LX024        89        (PUSH 12 1)
                 90        (HRRZ@ 2 -1 12)
                 91        (MOVEI 2 (QUOTE NIL))
                 92        (CALL 2 (E CONS))
    LX025        93        (PUSHJ 12 HIER1A)
    TAGY         94        (HRRZ 5 0 1)
                 95        (HLRZ 1 0 5)
                 96        (HRRZ 2 0 5)
                 97        (CALL 2 (E CONS))
    LX026        98        (HLRZ@ 2 0 12)
                 99        (CALL 2 (E XCONS))
    LX027        100       (SUB 12 (C 0 0 1 1))
                 101       (JRST 0 HIER1A)
    TAGA         102       (SUB 12 (C 0 0 1 1))
                 103       (POPJ 12)

***
COMPUTATION NUMBER 64 AT INSTRUCTION 2
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
***
COMPUTATION NUMBER 730 AT INSTRUCTION 8
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
***
COMPUTATION NUMBER 1138 AT INSTRUCTION 11
```

```
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             LX003              2
***
COMPUTATION NUMBER 1140 AT INSTRUCTION 12
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             LX003              2

***
COMPUTATION NUMBER 1148 AT INSTRUCTION 13
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             LX003              2
***
COMPUTATION NUMBER 1154 AT INSTRUCTION 14
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             LX003              2
             LX004              2
***
COMPUTATION NUMBER 1158 AT INSTRUCTION 16
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             LX003              2
             LX004              2
             LX005              2
***
COMPUTATION NUMBER 1250 AT INSTRUCTION 69
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             TAG2               2
             TAGC               2
             LX017              3
*****
REDERIVATION ERRORS
*****
***
RETURN ADDRESS ON THE STACK MUST BE A LABEL
HAS BEEN DETECTED AT INSTRUCTION 45
ALONG PATH:  LABEL         STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
```

```
                 LX010            3
                 LX011            4
                 LX012            4
***
***
RETURN ADDRESS ON THE STACK MUST BE A LABEL
HAS BEEN DETECTED AT INSTRUCTION 93
ALONG PATH:      LABEL        STACK DEPTH
                 HIER1            1
                 HIER1A           2
                 TAG2             2
                 TAGC             2
                 LX017            3
                 LX018            4
                 LX019            3
                 TAG12            3
                 LX024            3
                 LX025            4
***

*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE REDERIVED OUTPUT FOR THE EXACT INSTRUCTION AND PATH.
*****
***
(CONS F
        (CONS (CONS (CAR (CAR L))
                    (CAR (CDR L)))
              F))
(1158 0
        (1154 (1148 (1138 (64 5))
                    (1140 (730 5)))
              0))
AT INSTRUCTION 16
ALONG PATH:      LABEL        STACK DEPTH
                 HIER1            1
                 HIER1A           2
                 TAG2             2
                 LX003            2
                 LX004            2
                 LX005            2
***
***
(QUOTE PRIGHT)
(1250 0)
AT INSTRUCTION 69
ALONG PATH:      LABEL        STACK DEPTH
                 HIER1            1
                 HIER1A           2
                 TAG2             2
                 TAGC             2
                 LX017            3
***
```

Analysis:

Two errors were detected during the rederivation process. Both resulted from an invalid return address on the stack. This is because when a recursive call occurs at instructions 45 and 93 we bypass the start of the program. Thus we must prove that all locations referenced prior to being destroyed must have the appropriate values. However, the contents of the stack are wrong. Namely, the return address (i.e. locations 46 and 94) appears in the stack at a position where the binding of the second argument is expected (i.e. the top of the stack). Thus when a return will be made from the recursive call, we will not be at locations 46 or 94. Also, all references to the top of the stack will fetch the return address rather than the binding of RBP. The solution is to place the return address on the stack before the binding of RBP. In the case of the error at instruction 45, the binding of RBP is (BP1 (CAR (CAR (CDR (CDR L)))) (QUOTE RIGHT&)) which is computed starting at location 28 and pushed on the stack at location 34. Thus we may place the return address on the stack anywhere after location 27 and before location 34. We choose to do this between locations 27 and 28. In the case of the error at instruction 93, the binding of RBP is (BP1 (CAR (CAR L)) (QUOTE PRIGHT&)) which is computed starting at location 85 and pushed on the stack at location 89. Thus we may place the return address on the stack anywhere after location 84 and before location 89. We choose to do this between locations 84 and 85. However, we are not yet through. First of all, since the return addresses should no longer be placed on the stack at locations 45 and 93, we must only do a jump (i.e. JRST) rather than a push of a return address and a jump (i.e. PUSHJ) at these locations. Secondly, placing the return address on the stack earlier has caused the stack to contain an extra entry between locations 27 and 45 and 84 and 93. Thus all references to stack entries below the position holding the new return address must be incremented by one. Therefore we make the following changes:

```
location 28:    (HRRZ@ 1  0 12)     becomes    (HRRZ@ 1 -1 12)
location 35:    (HRRZ@ 5 -1 12)     becomes    (HRRZ@ 5 -2 12)
location 45:    (PUSHJ 12 HIER1A)   becomes    (JRST 0 HIER1A)
location 85:    (HLRZ@ 1  0 12)     becomes    (HLRZ@ 1 -1 12)
location 90:    (HRRZ@ 2 -1 12)     becomes    (HRRZ@ 2 -2 12)
location 93:    (PUSHJ 12 HIER1A)   becomes    (JRST 0 HIER1A)
```

The remaining two errors were detected during the proof procedure. The first error was detected at location 16 in the computation of:

```
(CONS F
      (CONS (CONS (CAR (CAR L))
                  (CAR (CDR L)))
            F))
(1158 0
      (1154 (1148 (1138 (64 5))
                  (1140 (730 5)))
            0))
```

Using the path information we know that the error was detected when (CAR L) was not NIL and (CDDR L) was NIL. Refering to our original function definition we see that at this point we want the following:

```
(CONS (CDR (CAR L))
      (CONS (CONS (CAR (CAR L))
                  (CAR (CDR L)))
            F))
```

Therefore, the error is in the arguments to the function being computed at location 16. The first argument to the CONS operation with computation number 1158 is F which is identical to (CDDR L). Looking at the code we find that at location 15 we perform (HLRZ 2 0 4) which has the effect of loading accumulator 2 with (CDR (CDR L)) rather than the desired (CDR (CAR L)). However, (CDR (CAR L)) can be achieved by changing the instruction to refer to accumulator 5 instead of accumulator 4. Thus we see that there are several possible causes for the error. Among them are a confusion about the contents of certain accumulators, and mistyping of a 4 for a 5. We make the following modification:

```
location 15:    (HRRZ 2  0 4)      becomes    (HRRZ 2  0 5)
```

The second error was detected at location 69 in the computation of:

```
(QUOTE PRIGHT)
(1250 0)
```

Using the path information we know that the error was detected when both (CAR L) and (CDDR L) were not NIL. Referring to our original function definition we see that at this point we want the following:

```
(QUOTE PRIGHT&)
```

Therefore, the error is in the argument to the function being computed at location 69. This time there is no doubt that the cause of the error was misspelling of the atom PRIGHT&. We make the following modification:

```
location 69:    (MOVEI 2 (QUOTE PRIGHT))  becomes
                (MOVEI 2 (QUOTE PRIGHT&))
```

Once the previous errors have been fixed we use the following program as an input to the proof procedure and try to analyze the errors.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1             1            (PUSH 12 2)
     HIER1A            2            (HLRZ 5 0 1)
                       3            (JUMPN 5 TAG2)
     LX001             4            (HRRZ 4 0 1)
                       5            (HRRZ 3 0 4)
                       6            (JUMPE 3 TAGA)
     LX002             7            (JRST 0 TAGB)
     TAG2              8            (HRRZ 4 0 1)
                       9            (HRRZ 3 0 4)
                      10            (JUMPN 3 TAGC)
     LX003            11            (HLRZ 1 0 5)
                      12            (HLRZ 2 0 4)
                      13            (CALL 2 (E CONS))
     LX004            14            (CALL 1 (E NCONS))
     LX005            15            (HRRZ 2 0 5)
                      16            (CALL 2 (E XCONS))
     LX006            17            (JRST 0 HIER1A)
     TAGB             18            (PUSH 12 1)
                      19            (HLRZ 1 0 3)
                      20            (HLRZ 1 0 1)
                      21            (MOVEI 2 (QUOTE LEFT&))
                      22            (CALL 2 (E BP1))
     LX007            23            (MOVE 2 -1 12)
                      24            (CALL 2 (E *GREAT))
     LX008            25            (JUMPN 1 TAG7)
```

```
LX009          26          (POP 12 1)
               27          (JRST 0 TAGA)
TAG7           28          (PUSH 12 (C 0 0 TAGX))
               29          (HRRZ@ 1 -1 12)
               30          (HRRZ 1 0 1)
               31          (HLRZ 1 0 1)
               32          (HLRZ 1 0 1)
               33          (MOVEI 2 (QUOTE RIGHT&))
               34          (CALL 2 (E BP1))
LX010          35          (PUSH 12 1)
               36          (HRRZ@ 5 -2 12)
               37          (HRRZ 4 0 5)
               38          (HLRZ 3 0 4)
               39          (HLRZ 5 0 3)
               40          (HRRZ 1 0 4)
               41          (HLRZ 4 0 5)
               42          (HLRZ 2 0 4)
               43          (CALL 2 (E CONS))
LX011          44          (MOVE 2 4)
               45          (CALL 2 (E XCONS))
LX012          46          (JRST 0 HIER1A)
TAGX           47          (HRRZ 5 0 1)
               48          (HLRZ 2 0 1)
               49          (HRRZ@ 4 0 12)
               50          (HLRZ 1 0 4)
               51          (CALL 2 (E CONS))
LX013          52          (HRRZ 3 0 4)
               53          (HLRZ 2 0 3)
               54          (HLRZ 2 0 2)
               55          (CALL 2 (E XCONS))
LX014          56          (HRRZ 2 0 5)
               57          (CALL 2 (E CONS))
LX015          58          (MOVEI 2 (QUOTE NIL))
               59          (CALL 2 (E XCONS))
LX016          60          (SUB 12 (C 0 0 1 1))
               61          (JRST 0 HIER1A)
TAGC           62          (PUSH 12 1)
               63          (HLRZ 1 0 3)
               64          (HLRZ 1 0 1)
               65          (MOVEI 2 (QUOTE LEFT&))
               66          (CALL 2 (E BP1))
LX017          67          (PUSH 12 1)
               68          (HLRZ@ 1 -1 12)
               69          (HLRZ 1 0 1)
               70          (MOVEI 2 (QUOTE PRIGHT&))
               71          (CALL 2 (E BP1))
LX018          72          (POP 12 2)
               73          (CALL 2 (E *GREAT))
LX019          74          (JUMPE 1 TAG12)
LX020          75          (HLRZ@ 5 0 12)
               76          (HLRZ 1 0 5)
               77          (HRRZ@ 4 0 12)
               78          (HLRZ 2 0 4)
               79          (CALL 2 (E CONS))
LX021          80          (HRRZ 2 0 4)
               81          (CALL 2 (E CONS))
LX022          82          (HRRZ 2 0 5)
               83          (CALL 2 (E XCONS))
LX023          84          (SUB 12 (C 0 0 1 1))
               85          (JRST 0 HIER1A)
TAG12          86          (PUSH 12 (C 0 0 TAGY))
```

```
                     87            (HLRZ@ 1 -1 12)
                     88            (HLRZ 1 0 1)
                     89            (MOVEI 2 (QUOTE PRIGHT&))
                     90            (CALL 2 (E BP1))
      LX024          91            (PUSH 12 1)
                     92            (HRRZ@ 2 -2 12)
                     93            (MOVEI 2 (QUOTE NIL))
                     94            (CALL 2 (E CONS))
      LX025          95            (JRST 0 HIER1A)
      TAGY           96            (HRRZ 5 0 1)
                     97            (HLRZ 1 0 5)
                     98            (HRRZ 2 0 5)
                     99            (CALL 2 (E CONS))
      LX026          100           (HLRZ@ 2 0 12)
                     101           (CALL 2 (E XCONS))
      LX027          102           (SUB 12 (C 0 0 1 1))
                     103           (JRST 0 HIER1A)
      TAGA           104           (SUB 12 (C 0 0 1 1))
                     105           (POPJ 12)
```

```
*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 64 AT INSTRUCTION 2
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
***
COMPUTATION NUMBER 748 AT INSTRUCTION 4
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
***
COMPUTATION NUMBER 750 AT INSTRUCTION 5
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
***
COMPUTATION NUMBER 774 AT INSTRUCTION 19
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
***
COMPUTATION NUMBER 776 AT INSTRUCTION 20
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
***
COMPUTATION NUMBER 986 AT INSTRUCTION 41
ALONG PATH:  LABEL          STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 1152 AT INSTRUCTION 8
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
***
COMPUTATION NUMBER 1154 AT INSTRUCTION 9
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
***
COMPUTATION NUMBER 1776 AT INSTRUCTION 63
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
***
COMPUTATION NUMBER 1778 AT INSTRUCTION 64
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
***
COMPUTATION NUMBER 1780 AT INSTRUCTION 65
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
***
COMPUTATION NUMBER 1788 AT INSTRUCTION 66
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
***
COMPUTATION NUMBER 1844 AT INSTRUCTION 69
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
***
COMPUTATION NUMBER 1846 AT INSTRUCTION 70
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC                2
             LX017               3
***
COMPUTATION NUMBER 1854 AT INSTRUCTION 71
ALONG PATH:  LABEL         STACK DEPTH
             HIER1               1
             HIER1A              2
             TAG2                2
             TAGC                2
             LX017               3
***
COMPUTATION NUMBER 1908 AT INSTRUCTION 73
ALONG PATH:  LABEL         STACK DEPTH
             HIER1               1
             HIER1A              2
             TAG2                2
             TAGC                2
             LX017               3
             LX018               4
***

*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE FILE CONTAINING THE REDERIVED OUTPUT FOR THE EXACT
INSTRUCTION AND PATH

*****
***
(CAR (CAR (CAR (CDR (CDR L)))))
(986 (776 (774 (750 (748 5)))))
AT INSTRUCTION 41
ALONG PATH:  LABEL         STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
             LX007               3
             LX008               3
             TAG7                3
             LX010               4
***
***
(*LESS (BP1 (CAR (CAR (CDR (CDR L))))
            (QUOTE LEFT&))
       (BP1 (CAR (CAR L))
            (QUOTE PRIGHT&)))
(1908 (1788 (1778 (1776 (1154 (1152 5))))
            (1780 0))
      (1854 (1844 (64 5))
            (1846 0)))
AT INSTRUCTION 73
ALONG PATH:  LABEL         STACK DEPTH
             HIER1               1
             HIER1A              2
             TAG2                2
             TAGC                2
             LX017               3
             LX018               4
```

***

Analysis:

Two errors were detected during the proof procedure. The first error was detected at location 41 in the computation of:

```
(CAR (CAR (CAR (CDR (CDR L)))))
(986 (776 (774 (750 (748 5)))))
```

Using the path information we know that the error was detected when (CAR L) was NIL, (CDDR L) was not NIL, and RBP < (BP1 (CADDAR L) (QUOTE LEFT&)). Referring to our original function definition, we see that the argument to the CAR operation at location 41 has already been matched up. Moreover, at this point a CDR operation is required as shown below:

```
(CDR (CDR (CAR (CDR (CDR L)))))
```

We temporarily disregard the fact that the argument to the CDR operation is wrong – i.e. it has not yet been matched. The next sequence of debugging will find this error. Looking at the code we find that at instruction 41 we perform (HLRZ 4 0 5) which has the wrong effect. Moreover, the result of this operation, i.e. (CAR (CAR (CDR (CDR L)))), was not matched and thus it can be changed to a (HRRZ 4 0 5) instruction. The cause for this error can be confusion as to the contents of a location or again misspelling. However, we lean toward the former since the error is of a compound nature as we will see at the next stage of debugging. We make the following modification:

```
location 41:    (HLRZ 4  0 5)      becomes     (HRRZ 4  0 5)
```

The second error was detected at location 73 in the computation of:

```
(*LESS (BP1 (CAR (CAR (CDR (CDR L))))
            (QUOTE LEFT&))
       (BP1 (CAR (CAR L))
            (QUOTE PRIGHT&)))
(1908 (1788 (1778 (1776 (1154 (1152 5))))
            (1780 0))
      (1854 (1844 (64 5))
            (1846 0)))
```

Using the path information we know that the error was detected when both (CAR L) and (CDDR L) were not NIL. One of the features of the canonical form is that operations known to be antisymmetric are always represented by only one of the two possible choices. Thus CONS and XCONS are represented by CONS and similarly, *LESS and *GREAT are represented by *LESS. Referring to our original function definition (actually the canonical form in this case since all *GREAT have been properly replaced by *LESS with the arguments reversed), we find that at this point we want the computation:

```
(*LESS (BP1 (CAR (CAR L))
            (QUOTE PRIGHT&))
       (BP1 (CAR (CAR (CDR (CDR L))))
            (QUOTE LEFT&)))
```

In other words, the error is in the order of the arguments to the *LESS function. Looking at the code we find that at location 73 we perform (CALL 2 (E *GREAT))) rather than the necessary (CALL 2 (E *LESS)). An equivalent interpretation of the error is that the contents of accumulators 1 and 2 (which must contain the arguments to the function) have been permuted. Nevertheless, we opt for the first interpretation since less code need be changed. Clearly, the source of this error is a misunderstanding by the programmer of the antisymmetric properties of the arithmetic relations less than, greater than, less than or equal, and greater than or equal. We make the following modification.

```
location 73:    (CALL 2 (E *GREAT))   becomes   (CALL 2 (E *LESS))
```

Once the previous errors have been fixed we use the following program as an input to the proof procedure and try to analyze the errors.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1             1             (PUSH 12 2)
     HIER1A            2             (HLRZ 5 0 1)
                       3             (JUMPN 5 TAG2)
     LX001             4             (HRRZ 4 0 1)
                       5             (HRRZ 3 0 4)
                       6             (JUMPE 3 TAGA)
     LX002             7             (JRST 0 TAGB)
     TAG2              8             (HRRZ 4 0 1)
                       9             (HRRZ 3 0 4)
                       10            (JUMPN 3 TAGC)
     LX003             11            (HLRZ 1 0 5)
                       12            (HLRZ 2 0 4)
                       13            (CALL 2 (E CONS))
     LX004             14            (CALL 1 (E NCONS))
     LX005             15            (HRRZ 2 0 5)
                       16            (CALL 2 (E XCONS))
     LX006             17            (JRST 0 HIER1A)
     TAGB              18            (PUSH 12 1)
                       19            (HLRZ 1 0 3)
                       20            (HLRZ 1 0 1)
                       21            (MOVEI 2 (QUOTE LEFT&))
                       22            (CALL 2 (E BP1))
     LX007             23            (MOVE 2 -1 12)
                       24            (CALL 2 (E *GREAT))
     LX008             25            (JUMPN 1 TAG7)
     LX009             26            (POP 12 1)
                       27            (JRST 0 TAGA)
     TAG7              28            (PUSH 12 (C 0 0 TAGX))
                       29            (HRRZ@ 1 -1 12)
                       30            (HRRZ 1 0 1)
                       31            (HLRZ 1 0 1)
                       32            (HLRZ 1 0 1)
                       33            (MOVEI 2 (QUOTE RIGHT&))
                       34            (CALL 2 (E BP1))
     LX010             35            (PUSH 12 1)
                       36            (HRRZ@ 5 -2 12)
                       37            (HRRZ 4 0 5)
                       38            (HLRZ 3 0 4)
                       39            (HLRZ 5 0 3)
                       40            (HRRZ 1 0 4)
                       41            (HRRZ 4 0 5)
```

```
                    42          (HLRZ 2 0 4)
                    43          (CALL 2 (E CONS))
 LX011              44          (MOVE 2 4)
                    45          (CALL 2 (E XCONS))
 LX012              46          (JRST 0 HIERIA)
 TAGX               47          (HRRZ 5 0 1)
                    48          (HLRZ 2 0 1)
                    49          (HRRZ@ 4 0 12)
                    50          (HLRZ 1 0 4)
                    51          (CALL 2 (E CONS))
 LX013              52          (HRRZ 3 0 4)
                    53          (HLRZ 2 0 3)
                    54          (HLRZ 2 0 2)
                    55          (CALL 2 (E XCONS))
 LX014              56          (HRRZ 2 0 5)
                    57          (CALL 2 (E CONS))
 LX015              58          (MOVEI 2 (QUOTE NIL))
                    59          (CALL 2 (E XCONS))
 LX016              60          (SUB 12 (C 0 0 1 1))
                    61          (JRST 0 HIERIA)
 TAGC               62          (PUSH 12 1)
                    63          (HLRZ 1 0 3)
                    64          (HLRZ 1 0 1)
                    65          (MOVEI 2 (QUOTE LEFT&))
                    66          (CALL 2 (E BP1))
 LX017              67          (PUSH 12 1)
                    68          (HLRZ@ 1 -1 12)
                    69          (HLRZ 1 0 1)
                    70          (MOVEI 2 (QUOTE PRIGHT&))
                    71          (CALL 2 (E BP1))
 LX018              72          (POP 12 2)
                    73          (CALL 2 (E *LESS))
 LX019              74          (JUMPE 1 TAG12)
 LX020              75          (HLRZ@ 5 0 12)
                    76          (HLRZ 1 0 5)
                    77          (HRRZ@ 4 0 12)
                    78          (HLRZ 2 0 4)
                    79          (CALL 2 (E CONS))
 LX021              80          (HRRZ 2 0 4)
                    81          (CALL 2 (E CONS))
 LX022              82          (HRRZ 2 0 5)
                    83          (CALL 2 (E XCONS))
 LX023              84          (SUB 12 (C 0 0 1 1))
                    85          (JRST 0 HIERIA)
 TAG12              86          (PUSH 12 (C 0 0 TAGY))
                    87          (HLRZ@ 1 -1 12)
                    88          (HLRZ 1 0 1)
                    89          (MOVEI 2 (QUOTE PRIGHT&))
                    90          (CALL 2 (E BP1))
 LX024              91          (PUSH 12 1)
                    92          (HRRZ@ 2 -2 12)
                    93          (MOVEI 2 (QUOTE NIL))
                    94          (CALL 2 (E CONS))
 LX025              95          (JRST 0 HIERIA)
 TAGY               96          (HRRZ 5 0 1)
                    97          (HLRZ 1 0 5)
                    98          (HRRZ 2 0 5)
                    99          (CALL 2 (E CONS))
 LX026             100          (HLRZ@ 2 0 12)
                   101          (CALL 2 (E XCONS))
 LX027             102          (SUB 12 (C 0 0 1 1))
```

```
                        103          (JRST 0 HIER1A)
     TAGA               104          (SUB 12 (C 0 0 1 1))
                        105          (POPJ 12)

*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 64 AT INSTRUCTION 2
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
***
COMPUTATION NUMBER 748 AT INSTRUCTION 4
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
***
COMPUTATION NUMBER 750 AT INSTRUCTION 5
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
***
COMPUTATION NUMBER 774 AT INSTRUCTION 19
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
***
COMPUTATION NUMBER 776 AT INSTRUCTION 20
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
***
COMPUTATION NUMBER 986 AT INSTRUCTION 41
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 1152 AT INSTRUCTION 8
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
***
COMPUTATION NUMBER 1844 AT INSTRUCTION 69
ALONG PATH:  LABEL        STACK DEPTH
```

```
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
***
COMPUTATION NUMBER 1846 AT INSTRUCTION 70
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
***
COMPUTATION NUMBER 1854 AT INSTRUCTION 71
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
***
COMPUTATION NUMBER 2046 AT INSTRUCTION 94
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
             LX018            4
             LX019            3
             TAG12            3
             LX024            4
***
COMPUTATION NUMBER 2414 AT INSTRUCTION 78
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             TAG2             2
             TAGC             2
             LX017            3
             LX018            4
             LX019            3
             LX020            3
***

*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE REDERIVED OUTPUT FOR THE EXACT INSTRUCTION AND PATH.
*****
***
***
(CDR (CAR (CAR (CDR (CDR L)))))
(986 (776 (774 (750 (748 5)))))
AT INSTRUCTION 41
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
```

```
                 LX002          2
                 TAGB           2
                 LX007          3
                 LX008          3
                 TAG7           3
                 LX010          4
***
***
(CONS (BP1 (CAR (CAR L)) (QUOTE PRIGHT&)) F)
(2046 (1854 (1844 (64 5)) (1846 0)) 0)
AT INSTRUCTION 94
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             TAG2              2
             TAGC              2
             LX017             3
             LX018             4
             LX019             3
             TAG12             3
             LX024             4
***
***
(CAR (CDR L))
(2414 (1152 5))
AT INSTRUCTION 78
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             TAG2              2
             TAGC              2
             LX017             3
             LX018             4
             LX019             3
             LX020             3
***
```

Analysis:

Three errors were detected during the proof procedure. The first error was detected at location 41 in the computation of:

```
(CDR (CAR (CAR (CDR (CDR L)))))
(986 (776 (774 (750 (748 5)))))
```

Using the path information we know that the error was detected when (CAR L) was NIL, (CDDR L) was not NIL, and RBP < (BP1 (CADDAR L) (QUOTE LEFT&)). Referring to our original function definition we see that the function computed at location 41 is being applied to the wrong argument. Recall from the last debugging session that the desired computation was:

```
(CDR (CDR (CAR (CDR (CDR L)))))
```

Therefore, the error is in the argument to the function being computed at location 41. The instruction at location 41 is (HRRZ 4 0 5). Therefore its argument is in accumulator 5 which is set at location 39 by a (HLRZ 5 0 3) instruction. We note that accumulator 5 is not referenced with this value except at location 41, and thus it is quite reasonable to believe that an error occurred at location 39. The instruction at location 39 has the effect of loading accumulator 5 with (CAR (CAR (CDR (CDR L)))) rather than the desired (CDR (CAR (CDR (CDR L)))). However, this is an easy change since we need merely replace the HLRZ operation at location 39 by a HRRZ operation. The cause of this error is confusion about the contents of accumulators or misspelling. However, we lean toward the latter since the nature of the remedy provides strong evidence. Recall that this was part of a compound error as discussed in the analysis of the previous set of bugs. We make the following modification:

```
location 39:    (HLRZ 5  0 3)      becomes    (HRRZ 5  0 3)
```

The second and third errors will be dealt with in one swoop. They were detected at locations 94 and 78 in the computation of:

```
(CONS (BP1 (CAR (CAR L))
           (QUOTE PRIGHT&)) F)
(2046 (1854 (1844 (64 5))
            (1846 0)) 0)

and

(CAR (CDR L))
(2414 (1152 5))
```

Both errors occured when (CAR L) and (CDDR L) were not NIL. The difference is that the first error occurs when (BP1 (CAAR L) (QUOTE PRIGHT&)) is greater than or equal to (BP1 (CAADDR L) (QUOTE LEFT&)) and the second error occurs when the latter condition is not true. If we were to proceed along lines proposed earlier, we would check if the functions computed at these locations are erroneous or if their arguments are not correct. Using this strategy, we would discover that we do not get a real idea as to the error. The problem is that we have branched on the wrong sense of the condition computed at location 73 and tested at location 74. Such errors are a possibility when there are two errors in the subtrees of the same condition. The error could be detected by the following scheme. Prior to the error analysis phase, if errors occur in all subtrees of a condition, then try to reverse the sense of the test. If all of the errors disappear, then our diagnosis is clearly correct. If some of the errors disappear, then our diagnosis is extremely likely to be valid. Otherwise there may be other errors. In the case of this example, we did indeed test the wrong sense of the condition. We were aware of this fact during the last debugging session. However, we did not discuss it because we feel that the present setting is more enlightening. Nevertheless, the problem should have been fixed at that time since the error did occur in the computation of a predicate. Such problems in the context of multiple errors are quite difficult and an adequate method to dispose of them is beyond the scope of this work. Therefore we change the sense of the test performed at location 74 by making the following modification:

```
location 74:    (JUMPE 1 TAG12)    becomes    (JUMPN 1 TAG12)
```

Once the previous errors have been fixed we use the following program as an input to the proof procedure and try to analyze the errors.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1           1             (PUSH 12 2)
     HIER1A          2             (HLRZ 5 0 1)
                     3             (JUMPN 5 TAG2)
```

```
LX001      4     (HRRZ 4 0 1)
           5     (HRRZ 3 0 4)
           6     (JUMPE 3 TAGA)
LX002      7     (JRST 0 TAGB)
TAG2       8     (HRRZ 4 0 1)
           9     (HRRZ 3 0 4)
          10     (JUMPN 3 TAGC)
LX003     11     (HLRZ 1 0 5)
          12     (HLRZ 2 0 4)
          13     (CALL 2 (E CONS))
LX004     14     (CALL 1 (E NCONS))
LX005     15     (HRRZ 2 0 5)
          16     (CALL 2 (E XCONS))
LX006     17     (JRST 0 HIERIA)
TAGB      18     (PUSH 12 1)
          19     (HLRZ 1 0 3)
          20     (HLRZ 1 0 1)
          21     (MOVEI 2 (QUOTE LEFT&))
          22     (CALL 2 (E BP1))
LX007     23     (MOVE 2 -1 12)
          24     (CALL 2 (E *GREAT))
LX008     25     (JUMPN 1 TAG7)
LX009     26     (POP 12 1)
          27     (JRST 0 TAGA)
TAG7      28     (PUSH 12 (C 0 0 TAGX))
          29     (HRRZ@ 1 -1 12)
          30     (HRRZ 1 0 1)
          31     (HLRZ 1 0 1)
          32     (HLRZ 1 0 1)
          33     (MOVEI 2 (QUOTE RIGHT&))
          34     (CALL 2 (E BP1))
LX010     35     (PUSH 12 1)
          36     (HRRZ@ 5 -2 12)
          37     (HRRZ 4 0 5)
          38     (HLRZ 3 0 4)
          39     (HRRZ 5 0 3)
          40     (HRRZ 1 0 4)
          41     (HRRZ 4 0 5)
          42     (HLRZ 2 0 4)
          43     (CALL 2 (E CONS))
LX011     44     (MOVE 2 4)
          45     (CALL 2 (E XCONS))
LX012     46     (JRST 0 HIERIA)
TAGX      47     (HRRZ 5 0 1)
          48     (HLRZ 2 0 1)
          49     (HRRZ@ 4 0 12)
          50     (HLRZ 1 0 4)
          51     (CALL 2 (E CONS))
LX013     52     (HRRZ 3 0 4)
          53     (HLRZ 2 0 3)
          54     (HLRZ 2 0 2)
          55     (CALL 2 (E XCONS))
LX014     56     (HRRZ 2 0 5)
          57     (CALL 2 (E CONS))
LX015     58     (MOVEI 2 (QUOTE NIL))
          59     (CALL 2 (E XCONS))
LX016     60     (SUB 12 (C 0 0 1 1))
          61     (JRST 0 HIERIA)
TAGC      62     (PUSH 12 1)
          63     (HLRZ 1 0 3)
          64     (HLRZ 1 0 1)
```

```
                     65          (MOVEI 2 (QUOTE LEFT&))
                     66          (CALL 2 (E BP1))
  LX017              67          (PUSH 12 1)
                     68          (HLRZ@ 1 -1 12)
                     69          (HLRZ 1 0 1)
                     70          (MOVEI 2 (QUOTE PRIGHT&))
                     71          (CALL 2 (E BP1))
  LX018              72          (POP 12 2)
                     73          (CALL 2 (E *LESS))
  LX019              74          (JUMPN 1 TAG12)
  LX020              75          (HLRZ@ 5 0 12)
                     76          (HLRZ 1 0 5)
                     77          (HRRZ@ 4 0 12)
                     78          (HLRZ 2 0 4)
                     79          (CALL 2 (E CONS))
  LX021              80          (HRRZ 2 0 4)
                     81          (CALL 2 (E CONS))
  LX022              82          (HRRZ 2 0 5)
                     83          (CALL 2 (E XCONS))
  LX023              84          (SUB 12 (C 0 0 1 1))
                     85          (JRST 0 HIER1A)
  TAG12              86          (PUSH 12 (C 0 0 TAGY))
                     87          (HLRZ@ 1 -1 12)
                     88          (HLRZ 1 0 1)
                     89          (MOVEI 2 (QUOTE PRIGHT&))
                     90          (CALL 2 (E BP1))
  LX024              91          (PUSH 12 1)
                     92          (HRRZ@ 2 -2 12)
                     93          (MOVEI 2 (QUOTE NIL))
                     94          (CALL 2 (E CONS))
  LX025              95          (JRST 0 HIER1A)
  TAGY               96          (HRRZ 5 0 1)
                     97          (HLRZ 1 0 5)
                     98          (HRRZ 2 0 5)
                     99          (CALL 2 (E CONS))
  LX026              100         (HLRZ@ 2 0 12)
                     101         (CALL 2 (E XCONS))
  LX027              102         (SUB 12 (C 0 0 1 1))
                     103         (JRST 0 HIER1A)
  TAGA               104         (SUB 12 (C 0 0 1 1))
                     105         (POPJ 12)

*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 64 AT INSTRUCTION 2
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
***
COMPUTATION NUMBER 748 AT INSTRUCTION 4
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
***
COMPUTATION NUMBER 750 AT INSTRUCTION 5
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
```

```
                 LX001             2
***
COMPUTATION NUMBER 774 AT INSTRUCTION 19
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
***
COMPUTATION NUMBER 982 AT INSTRUCTION 39
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
***
COMPUTATION NUMBER 984 AT INSTRUCTION 40
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
***
COMPUTATION NUMBER 986 AT INSTRUCTION 41
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
***
COMPUTATION NUMBER 988 AT INSTRUCTION 42
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
***
COMPUTATION NUMBER 996 AT INSTRUCTION 43
ALONG PATH:      LABEL         STACK DEPTH
                 HIER1             1
```

```
                  HIER1A             2
                  LX001              2
                  LX002              2
                  TAGB               2
                  LX007              3
                  LX008              3
                  TAG7               3
                  LX010              4
***
COMPUTATION NUMBER 1844 AT INSTRUCTION 69
ALONG PATH:  LABEL          STACK DEPTH
             HIER1                1
             HIER1A               2
             TAG2                 2
             TAGC                 2
             LX017                3
***
COMPUTATION NUMBER 1846 AT INSTRUCTION 70
ALONG PATH:  LABEL          STACK DEPTH
             HIER1                1
             HIER1A               2
             TAG2                 2
             TAGC                 2
             LX017                3
***
COMPUTATION NUMBER 1854 AT INSTRUCTION 71
ALONG PATH:  LABEL          STACK DEPTH
             HIER1                1
             HIER1A               2
             TAG2                 2
             TAGC                 2
             LX017                3
***
COMPUTATION NUMBER 2344 AT INSTRUCTION 94
ALONG PATH:  LABEL          STACK DEPTH
             HIER1                1
             HIER1A               2
             TAG2                 2
             TAGC                 2
             LX017                3
             LX018                4
             LX019                3
             TAG12                3
             LX024                4
***
*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE FILE CONTAINING THE REDERIVED OUTPUT FOR THE EXACT
INSTRUCTION AND PATH

*****
***
(CONS (CDR (CDR (CDR L)))
      (CAR (CDR (CDR (CAR (CDR (CDR L)))))))
(996 (984 (750 (748 5)))
     (988 (986 (982 (774 (750 (748 5)))))))
AT INSTRUCTION 43
ALONG PATH:  LABEL          STACK DEPTH
             HIER1                1
```

```
                 HIER1A            2
                 LX001             2
                 LX002             2
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
***
***
(CONS (BP1 (CAR (CAR L))
           (QUOTE PRIGHT&))
      F)
(2344 (1854 (1844 (64 5))
            (1846 0))
      0)
AT INSTRUCTION 94
ALONG PATH:  LABEL         STACK DEPTH
             HIER1             1
             HIER1A            2
             TAG2              2
             TAGC              2
             LX017             3
             LX018             4
             LX019             3
             TAG12             3
             LX024             4
***
```

Analysis:

Two errors were detected in the proof procedure. The first error was detected at location 43 in the computation of:

```
(CONS (CDR (CDR (CDR L)))
      (CAR (CDR (CDR (CAR (CDR (CDR L)))))))
(996 (984 (750 (748 5)))
     (988 (986 (982 (774 (750 (748 5)))))))
```

Using the path information we know that the error was detected when (CAR L) was NIL, (CDDR L) was not NIL, and RBP < (BP1 (CADDAR L) (QUOTE LEFT&)). Referring to our original function definition, we see that at this point we want the following:

```
(CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
      (CDR (CDR (CDR L))))
```

Clearly, the order of the arguments to the CONS operation has been reversed. Looking at the code we find that at location 43 we perform (CALL 2 (E CONS)) rather than the necessary (CALL 2 (E XCONS)). This conclusion is made on the basis of CONS being an antisymmetric function. An equivalent interpretation of the error is that the contents of accumulators 1 and 2 (which must contain the arguments to the function) have been permuted. Nevertheless, we opt for the first interpretation since less code need be changed. Clearly, the source of error here is a confusion about the contents of accumulators 1 and 2. We make the following modification:

```
location 43:    (CALL 2 (E CONS))    becomes    (CALL 2 (E XCONS))
```

The second error was detected at location 94 in the computation of:

```
(CONS (BP1 (CAR (CAR L))
           (QUOTE PRIGHT&))
      F)
(2344 (1854 (1844 (64 5))
            (1846 0))
      0)
```

Using the path information we know that the error was detected when both (CAR L) and (CDDR L) were not NIL and (BP1 (CAAR L) (QUOTE PRIGHT?)) was not greater than or equal to (BP1 (CAADDR L) (QUOTE LEFT?)). Referring to our original function definition we see that at this point we want the following:

```
(CONS F (CDR L))
```

Therefore the error is in the arguments to the function being computed at location 94. The desired arguments, F and (CDR L), have already been matched up and in fact are computed at locations 93 and 92 respectively. Thus the correction is to simply make sure that they reside in the proper accumulators for the CONS operation at location 94 to be right. This means that instead of loading accumulator 2 with NIL at location 93, we load accumulator 1 with this value. Notice that the error that was made was to load accumulator 2 with NIL at location 93 via (MOVEI 2 (QUOTE NIL)) thereby destroying the previous contents which was (CDR L). This error was detected and quite easily corrected because we always record all computations that have been performed whether or not they are referenced (recall the FN construct). This is useful because the matching process will make sure that the computation is performed. Thus when errors occur in arguments to functions we can easily make a fix since we know where and when the desired arguments were computed even though they may have been misused. The error in this case can be clearly attributed to an oversight by the programmer in typing a 2 instead of a 1. We make the following modification:

```
location 94:   (MOVEI 2 (QUOTE NIL))  becomes  (MOVEI 1 (QUOTE NIL))
```

Once the previous errors have been fixed we use the following program as an input to the proof procedure and try to analyze the errors.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1            1            (PUSH 12 2)
     HIER1A           2            (HLRZ 5 0 1)
                      3            (JUMPN 5 TAG2)
     LX001            4            (HRRZ 4 0 1)
                      5            (HRRZ 3 0 4)
                      6            (JUMPE 3 TAGA)
     LX002            7            (JRST 0 TAGB)
     TAG2             8            (HRRZ 4 0 1)
                      9            (HRRZ 3 0 4)
                      10           (JUMPN 3 TAGC)
     LX003            11           (HLRZ 1 0 5)
                      12           (HLRZ 2 0 4)
                      13           (CALL 2 (E CONS))
     LX004            14           (CALL 1 (E NCONS))
     LX005            15           (HRRZ 2 0 5)
                      16           (CALL 2 (E XCONS))
     LX006            17           (JRST 0 HIER1A)
     TAGB             18           (PUSH 12 1)
```

```
               (HLRZ 1 0 3)
               (HLRZ 1 0 1)
               (MOVEI 2 (QUOTE LEFT&))
               (CALL 2 (E BP1))
LX007          (MOVE 2 -1 12)
               (CALL 2 (E *GREAT))
LX008          (JUMPN 1 TAG7)
LX009          (POP 12 1)
               (JRST 0 TAGA)
TAG7           (PUSH 12 (C 0 0 TAGX))
               (HRRZ@ 1 -1 12)
               (HRRZ 1 0 1)
               (HLRZ 1 0 1)
               (HLRZ 1 0 1)
               (MOVEI 2 (QUOTE RIGHT&))
               (CALL 2 (E BP1))
LX010          (PUSH 12 1)
               (HRRZ@ 5 -2 12)
               (HRRZ 4 0 5)
               (HLRZ 3 0 4)
               (HRRZ 5 0 3)
               (HRRZ 1 0 4)
               (HRRZ 4 0 5)
               (HLRZ 2 0 4)
               (CALL 2 (E XCONS))
LX011          (MOVE 2 4)
               (CALL 2 (E XCONS))
LX012          (JRST 0 HIER1A)
TAGX           (HRRZ 5 0 1)
               (HLRZ 2 0 1)
               (HRRZ@ 4 0 12)
               (HLRZ 1 0 4)
               (CALL 2 (E CONS))
LX013          (HRRZ 3 0 4)
               (HLRZ 2 0 3)
               (HLRZ 2 0 2)
               (CALL 2 (E XCONS))
LX014          (HRRZ 2 0 5)
               (CALL 2 (E CONS))
LX015          (MOVEI 2 (QUOTE NIL))
               (CALL 2 (E XCONS))
LX016          (SUB 12 (C 0 0 1 1))
               (JRST 0 HIER1A)
TAGC           (PUSH 12 1)
               (HLRZ 1 0 3)
               (HLRZ 1 0 1)
               (MOVEI 2 (QUOTE LEFT&))
               (CALL 2 (E BP1))
LX017          (PUSH 12 1)
               (HLRZ@ 1 -1 12)
               (HLRZ 1 0 1)
               (MOVEI 2 (QUOTE PRIGHT&))
               (CALL 2 (E BP1))
LX018          (POP 12 2)
               (CALL 2 (E *LESS))
LX019          (JUMPN 1 TAG12)
LX020          (HLRZ@ 5 0 12)
               (HLRZ 1 0 5)
               (HRRZ@ 4 0 12)
               (HLRZ 2 0 4)
               (CALL 2 (E CONS))
```

```
    LX021          80        (HRRZ 2 0 4)
                   81        (CALL 2 (E CONS))
    LX022          82        (HRRZ 2 0 5)
                   83        (CALL 2 (E XCONS))
    LX023          84        (SUB 12 (C 0 0 1 1))
                   85        (JRST 0 HIER1A)
    TAG12          86        (PUSH 12 (C 0 0 TAGY))
                   87        (HLRZ@ 1 -1 12)
                   88        (HLRZ 1 0 1)
                   89        (MOVEI 2 (QUOTE PRIGHT&))
                   90        (CALL 2 (E BP1))
    LX024          91        (PUSH 12 1)
                   92        (HRRZ@ 2 -2 12)
                   93        (MOVEI 1 (QUOTE NIL))
                   94        (CALL 2 (E CONS))
    LX025          95        (JRST 0 HIER1A)
    TAGY           96        (HRRZ 5 0 1)
                   97        (HLRZ 1 0 5)
                   98        (HRRZ 2 0 5)
                   99        (CALL 2 (E CONS))
    LX026          100       (HLRZ@ 2 0 12)
                   101       (CALL 2 (E XCONS))
    LX027          102       (SUB 12 (C 0 0 1 1))
                   103       (JRST 0 HIER1A)
    TAGA           104       (SUB 12 (C 0 0 1 1))
                   105       (POPJ 12)
*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 64 AT INSTRUCTION 2
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
***
COMPUTATION NUMBER 748 AT INSTRUCTION 4
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             LX001             2
***
COMPUTATION NUMBER 750 AT INSTRUCTION 5
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             LX001             2
***
COMPUTATION NUMBER 774 AT INSTRUCTION 19
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             LX001             2
             LX002             2
             TAGB              2
***
COMPUTATION NUMBER 982 AT INSTRUCTION 39
ALONG PATH:  LABEL        STACK DEPTH
             HIER1             1
             HIER1A            2
             LX001             2
```

```
                  LX002              2
                  TAGB               2
                  LX007              3
                  LX008              3
                  TAG7               3
                  LX010              4
***
COMPUTATION NUMBER 984 AT INSTRUCTION 40
ALONG PATH:  LABEL           STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
             LX007               3
             LX008               3
             TAG7                3
             LX010               4
***
COMPUTATION NUMBER 986 AT INSTRUCTION 41
ALONG PATH:  LABEL           STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
             LX007               3
             LX008               3
             TAG7                3
             LX010               4
***
COMPUTATION NUMBER 988 AT INSTRUCTION 42
ALONG PATH:  LABEL           STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
             LX007               3
             LX008               3
             TAG7                3
             LX010               4
***
COMPUTATION NUMBER 996 AT INSTRUCTION 43
ALONG PATH:  LABEL           STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
             TAGB                2
             LX007               3
             LX008               3
             TAG7                3
             LX010               4
***
COMPUTATION NUMBER 1002 AT INSTRUCTION 45
ALONG PATH:  LABEL           STACK DEPTH
             HIER1               1
             HIER1A              2
             LX001               2
             LX002               2
```

```
                 TAGB              2
                 LX007             3
                 LX008             3
                 TAG7              3
                 LX010             4
                 LX011             5
***
*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE FILE CONTAINING THE REDERIVED OUTPUT FOR THE EXACT
INSTRUCTION AND PATH

*****
***
(CONS (CDR (CDR (CAR (CDR (CDR L)))))
      (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
            (CDR (CDR (CDR L)))))
(1002 (986 (982 (774 (750 (748 5)))))
      (996 (988 (986 (982 (774 (750 (748 5))))))
           (984 (750 (748 5)))))
AT INSTRUCTION 45
ALONG PATH:  LABEL         STACK DEPTH
             HIER1             1
             HIER1A            2
             LX001             2
             LX002             2
             TAGB              2
             LX007             3
             LX008             3
             TAG7              3
             LX010             4
             LX011             5
***
```

Analysis:

One error was detected during the proof procedure at location 45 in the computation of:

```
(CONS (CDR (CDR (CAR (CDR (CDR L)))))
      (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
            (CDR (CDR (CDR L)))))
(1002 (986 (982 (774 (750 (748 5)))))
      (996 (988 (986 (982 (774 (750 (748 5))))))
           (984 (750 (748 5)))))
```

Using the path information we know that the error was detected when (CAR L) was NIL, (CDDR L) was not NIL, and RBP < (BP1 (CADDAR L) (QUOTE LEFT&)). Referring to our original function definition we see that at this point we want the following:

```
(CONS (CAR (CDR (CAR (CDR (CDR L)))))
      (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
            (CDR (CDR (CDR L)))))
```

Therefore the error is in the arguments to the function being computed at location 45. The desired second argument is correct, but the first one is invalid. The instruction performed at location 45 is (CALL 2 (E XCONS)) and thus the argument in accumulator 2 is wrong. The desired contents of accumulator 2 is (CAR (CDR (CAR (CDR (CDR L))))). Looking at the code we see that accumulator 2 is last loaded at location 44 by the instruction (MOVE 2 4). However, this value is not necessary in the future and thus the instruction at this location may be removed. The validity of the previous removal is obvious when we recall the that the value in accumulator 2 is not referenced past location 45. An alternative reason is that the XCONS operation is assumed to destroy accumulators 1 and 2. In its place we need to compute (CAR (CDR (CAR (CDR (CDR L))))) since it has not yet been computed. This can be done quite easily since at this point accumulator 5 already contains (CDR (CAR (CDR (CDR L)))) and thus we need only obtain CAR of the contents of register 5. This is quite easily done by inserting (HLRZ 2 0 5) at location 44. The cause of this error is obviously confusion on the part of the programmer as to the contents of accumulator 2. We make the following modification:

```
location 44:    (MOVE 2  4)    becomes    (HLRZ 2  0 5)
```

Once the previous errors have been fixed we use the following program as an input to the proof procedure and try to analyze the errors.

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     HIER1            1             (PUSH 12 2)
     HIER1A           2             (HLRZ 5 0 1)
                      3             (JUMPN 5 TAG2)
     LX001            4             (HRRZ 4 0 1)
                      5             (HRRZ 3 0 4)
                      6             (JUMPE 3 TAGA)
     LX002            7             (JRST 0 TAGB)
     TAG2             8             (HRRZ 4 0 1)
                      9             (HRRZ 3 0 4)
                      10            (JUMPN 3 TAGC)
     LX003            11            (HLRZ 1 0 5)
                      12            (HLRZ 2 0 4)
                      13            (CALL 2 (E CONS))
     LX004            14            (CALL 1 (E NCONS))
     LX005            15            (HRRZ 2 0 5)
                      16            (CALL 2 (E XCONS))
     LX006            17            (JRST 0 HIER1A)
     TAGB             18            (PUSH 12 1)
                      19            (HLRZ 1 0 3)
                      20            (HLRZ 1 0 1)
                      21            (MOVEI 2 (QUOTE LEFT&))
                      22            (CALL 2 (E BP1))
     LX007            23            (MOVE 2 -1 12)
                      24            (CALL 2 (E *GREAT))
     LX008            25            (JUMPN 1 TAG7)
     LX009            26            (POP 12 1)
                      27            (JRST 0 TAGA)
     TAG7             28            (PUSH 12 (C 0 0 TAGX))
                      29            (HRRZ@ 1 -1 12)
                      30            (HRRZ 1 0 1)
                      31            (HLRZ 1 0 1)
                      32            (HLRZ 1 0 1)
                      33            (MOVEI 2 (QUOTE RIGHT&))
                      34            (CALL 2 (E BP1))
     LX010            35            (PUSH 12 1)
                      36            (HRRZ@ 5 -2 12)
                      37            (HRRZ 4 0 5)
```

```
                38      (HLRZ 3 0 4)
                39      (HRRZ 5 0 3)
                40      (HRRZ 1 0 4)
                41      (HRRZ 4 0 5)
                42      (HLRZ 2 0 4)
                43      (CALL 2 (E XCONS))
LX011           44      (HLRZ 2 0 5)
                45      (CALL 2 (E XCONS))
LX012           46      (JRST 0 HIER1A)
TAGX            47      (HRRZ 5 0 1)
                48      (HLRZ 2 0 1)
                49      (HRRZ@ 4 0 12)
                50      (HLRZ 1 0 4)
                51      (CALL 2 (E CONS))
LX013           52      (HRRZ 3 0 4)
                53      (HLRZ 2 0 3)
                54      (HLRZ 2 0 2)
                55      (CALL 2 (E XCONS))
LX014           56      (HRRZ 2 0 5)
                57      (CALL 2 (E CONS))
LX015           58      (MOVEI 2 (QUOTE NIL))
                59      (CALL 2 (E XCONS))
LX016           60      (SUB 12 (C 0 0 1 1))
                61      (JRST 0 HIER1A)
TAGC            62      (PUSH 12 1)
                63      (HLRZ 1 0 3)
                64      (HLRZ 1 0 1)
                65      (MOVEI 2 (QUOTE LEFT&))
                66      (CALL 2 (E BP1))
LX017           67      (PUSH 12 1)
                68      (HLRZ@ 1 -1 12)
                69      (HLRZ 1 0 1)
                70      (MOVEI 2 (QUOTE PRIGHT&))
                71      (CALL 2 (E BP1))
LX018           72      (POP 12 2)
                73      (CALL 2 (E *LESS))
LX019           74      (JUMPN 1 TAG12)
LX020           75      (HLRZ@ 5 0 12)
                76      (HLRZ 1 0 5)
                77      (HRRZ@ 4 0 12)
                78      (HLRZ 2 0 4)
                79      (CALL 2 (E CONS))
LX021           80      (HRRZ 2 0 4)
                81      (CALL 2 (E CONS))
LX022           82      (HRRZ 2 0 5)
                83      (CALL 2 (E XCONS))
LX023           84      (SUB 12 (C 0 0 1 1))
                85      (JRST 0 HIER1A)
TAG12           86      (PUSH 12 (C 0 0 TAGY))
                87      (HLRZ@ 1 -1 12)
                88      (HLRZ 1 0 1)
                89      (MOVEI 2 (QUOTE PRIGHT&))
                90      (CALL 2 (E BP1))
LX024           91      (PUSH 12 1)
                92      (HRRZ@ 2 -2 12)
                93      (MOVEI 1 (QUOTE NIL))
                94      (CALL 2 (E CONS))
LX025           95      (JRST 0 HIER1A)
TAGY            96      (HRRZ 5 0 1)
                97      (HLRZ 1 0 5)
                98      (HRRZ 2 0 5)
```

```
                         99          (CALL 2 (E CONS))
     LX026              100          (HLRZ@ 2 0 12)
                        101          (CALL 2 (E XCONS))
     LX027              102          (SUB 12 (C 0 0 1 1))
                        103          (JRST 0 HIER1A)
     TAGA               104          (SUB 12 (C 0 0 1 1))
                        105          (POPJ 12)

*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 748 AT INSTRUCTION 4
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
***
COMPUTATION NUMBER 750 AT INSTRUCTION 5
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
***
COMPUTATION NUMBER 774 AT INSTRUCTION 19
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
***
COMPUTATION NUMBER 776 AT INSTRUCTION 20
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
***
COMPUTATION NUMBER 914 AT INSTRUCTION 33
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
***
COMPUTATION NUMBER 922 AT INSTRUCTION 34
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
```

```
***
COMPUTATION NUMBER 982 AT INSTRUCTION 39
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 984 AT INSTRUCTION 40
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 986 AT INSTRUCTION 41
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 988 AT INSTRUCTION 42
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
COMPUTATION NUMBER 996 AT INSTRUCTION 43
ALONG PATH:  LABEL       STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
***
```

```
COMPUTATION NUMBER 1002 AT INSTRUCTION 44
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
             LX010              4
             LX011              5
***
COMPUTATION NUMBER 1004 AT INSTRUCTION 45
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
             LX010              4
             LX011              5
***
COMPUTATION NUMBER 1010 AT INSTRUCTION 46
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
             LX010              4
             LX011              5
             LX012              5
***
COMPUTATION NUMBER 1074 AT INSTRUCTION 48
ALONG PATH:  LABEL          STACK DEPTH
             HIER1              1
             HIER1A             2
             LX001              2
             LX002              2
             TAGB               2
             LX007              3
             LX008              3
             TAG7               3
             LX010              4
             LX011              5
             LX012              5
             TAGX               3
***
*****
THE FOLLOWING COMPUTATIONS MAY HAVE CAUSED A MISMATCH DUE TO NOT
BEING PRESENT IN ALL SUBPATHS OR A NON REDUNDANT CONDITION.
REFER TO THE FILE CONTAINING THE REDERIVED OUTPUT FOR THE EXACT
INSTRUCTION AND PATH
```

```
*****
***
(CAR (HIER1 (CONS (CAR (CDR (CAR (CDR (CDR L)))))
                  (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
                        (CDR (CDR (CDR L)))))
            (BP1 (CAR (CAR (CDR (CDR L))))
                 (QUOTE RIGHT&))))
(1074 (1010 (1004 (1002 (982 (774 (750 (748 5)))))
                  (996 (988 (986 (982 (774 (750 (748 5))))))
                        (984 (750 (748 5)))))
            (922 (776 (774 (750 (748 5))))
                 (914 0))))
AT INSTRUCTION 48
ALONG PATH:  LABEL        STACK DEPTH
             HIER1            1
             HIER1A           2
             LX001            2
             LX002            2
             TAGB             2
             LX007            3
             LX008            3
             TAG7             3
             LX010            4
             LX011            5
             LX012            5
             TAGX             3
***
```

Analysis:

One error was detected during the proof procedure at location 48 in the computation of:

```
(CAR (HIER1 (CONS (CAR (CDR (CAR (CDR (CDR L)))))
                  (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
                        (CDR (CDR (CDR L)))))
            (BP1 (CAR (CAR (CDR (CDR L))))
                 (QUOTE RIGHT&))))
(1074 (1010 (1004 (1002 (982 (774 (750 (748 5)))))
                  (996 (988 (986 (982 (774 (750 (748 5))))))
                        (984 (750 (748 5)))))
            (922 (776 (774 (750 (748 5))))
                 (914 0))))
```

Using the path information we know that the error was detected when (CAR L) was NIL, (CDDR L) was not NIL, and RBP < (BP1 (CADDAR L) (QUOTE LEFT&)). Referring to our original function definition we see that this computation is unnecessary. Moreover what is required at this point is the following:

```
(CDR (HIER1 (CONS (CAR (CDR (CAR (CDR (CDR L)))))
                  (CONS (CAR (CDR (CDR (CAR (CDR (CDR L))))))
                        (CDR (CDR (CDR L)))))
            (BP1 (CAR (CAR (CDR (CDR L))))
                 (QUOTE RIGHT&))))
```

Clearly, what happened here is that a CAR operation was computed rather than a CDR operation. In terms of machine instructions the previous is translated into the performance of a HLRZ rather than a HRRZ. By now we are rather adept at making such fixes and we simply replace the

(HLRZ 2 0 1) instruction at location 48 by (HRRZ 2 0 1). Note that we made use of the fact that results of the previous instruction at location 48 were never referenced in the future. Clearly, the cause of this error is mistyping of HLRZ for HRRZ. We make the following modification:

```
  location 48:    (HLRZ 2  0 1)   becomes   (HRRZ 2  0 1)
```

At this point the program is identical to the correct hand optimized program encountered earlier. Thus we have seen how the system can aid the user in debugging his program. The ultimate wish is to construct a system, employing similar reasoning as we have performed in this chapter, to debug and correct erroneous programs. Of course, not all errors could be caught by such a system. However, we feel that quite a reasonable number could be detected and corrected by such an automatic system. This should be quite obvious from the lengthy example presented here.

CHAPTER 7

EXAMPLES

7.A Introduction

In the previous three chapters we have presented a methodology for proving that LISP programs are correctly translated to assembly language programs. This presentation was coupled with illustrative examples. Often references were made to specific functions and how certain problems that occur in their handling were solved. In this chapter we intend to guide the reader through a pair of examples ranging from trivial to hard with a varying amount of detail.

The first example, REVERSE1, is designed to show how a simple function is proved equivalent. We next indicate the proof procedure for algorithm 2 of NEXT which has been encoded using the optimized LAP program as given in Figure 4.7. The examples use F instead of NIL so that no confusion will arise between the atom NIL and the value NIL. Recall that we use NIL to denote a result of a path in whose rederivation an error occurred. Hopefully, the reader will not find this distraction too painful. We also reiterate a previous comment about the computation numbers — i.e. the magnitude of the numbers is only used to denote a relative ordering between the computations.

For the functions REVERSE1 and NEXT we give the function definition in LISP and meta–LISP, and the LAP encoding. The proof of equivalence is preceded by a listing of the assembly language program with the internally assigned labels and program counter values. We also give the rederived form whose comprehension is hopefully facilitated by the inclusion of a dictionary whose entries are the computation numbers appearing in the rederived form. For each computation number, the corresponding entry indicates the program counter value at which it is performed and the path in terms of pairs of values. The first element of each pair is a label and the second denotes the stack depth at the time the label was encountered.

In the presentation of traces of the proof procedure we represent the canonical and rederived forms as trees. Actually, two trees correspond to each of the forms; one each for the symbolic and numeric representation of the function. In the trace the trees on the left and right denote the symbolic and numeric representations respectively. We indicate which of the two forms is being manipulated to match the other, and the various backward paths which are attempted when mismatches are due to recursive calls. For each node in the tree we point out how a matching instance is obtained for each computation performed in the node that has not been previously matched. This information consists of the computation numbers of the matching instances and if the match is of an explicit or implicit nature.

7.B REVERSE1

REVERSE1 is a familiar function in most correctness work. The function takes a list as its argument and returns the reverse of the top elements of its argument. The particular encoding of the function examined here uses an auxiliary function REVERESE1A which has an additional variable to temporarily store the new list. Thus we will be concerned with the function REVERSE1A rather than REVERSE1. For example, the function applied to (A B C) yields (C B A); similarly application to (A (B C D) E) would yield (E (B C D) A). The rederived form deviates from the canonical form only in the rearranging of the order of computing (CDR L) – i.e. in the original function definition it is computed prior to (CONS (CAR L) RL) while in our LAP program it is computed after (CONS (CAR L) RL).

LISP Encoding:

```
(DEFPROP REVERSE1 (LAMBDA (L) REVERSE1A L F)) EXPR)
(DEFPROP REVERSE1A (LAMBDA (L RL)
        (COND ((NULL L) RL)
              (T (REVERSE1A (CDR L) (CONS (CAR L) RL))))) EXPR)
```

MLISP Encoding:

```
REVERSE1(L) ≡ REVERSE1A(L,NIL)
REVERSE1A(L,RL) ≡ if NULL(L) then RL
                  else REVERSE1A(CDR(L),CONS(CAR(L),RL))
```

LAP Encoding:

```
(LAP REVERSE1 SUBR)
        (MOVEI 2 (QUOTE NIL))
        (JCALL 2 (E REVERSE1A))
        NIL

(LAP REVERSE1A SUBR)
        (PUSH P 1)
        (PUSH P 2)
        (JUMPN 1 TAG2)
        (MOVE 1 2)
        (JRST 0 TAG3)
TAG2    (MOVE 2 0 P)
        (HLRZ@ 1 -1 P)
        (CALL 2 (E CONS))
        (MOVE 2 1)
        (HRRZ@ 1 -1 P)
        (CALL 2 (E REVERSE1A))
TAG3    (SUB P (C 0 0 2 2))
        (POPJ P)
        NIL
```

```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     REVERSE1A         1           (PUSH 12 1)
                       2           (PUSH 12 2)
                       3           (JUMPN 1 TAG2)
     LX001             4           (MOVE 1 2)
                       5           (JRST 0 TAG3)
     TAG2              6           (MOVE 2 0 12)
                       7           (HLRZ@ 1 -1 12)
                       8           (CALL 2 (E CONS))
     LX002             9           (MOVE 2 1)
                       10          (HRRZ@ 1 -1 12)
                       11          (CALL 2 (E REVERSE1A))
     TAG3              12          (SUB 12 (C 0 0 2 2))
                       13          (POPJ 12)

*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 80 AT INSTRUCTION 7
ALONG PATH:  LABEL        STACK DEPTH
             REVERSE1A        1
             TAG2             3
***
COMPUTATION NUMBER 88 AT INSTRUCTION 8
ALONG PATH:  LABEL        STACK DEPTH
             REVERSE1A        1
             TAG2             3
***
COMPUTATION NUMBER 94 AT INSTRUCTION 10
ALONG PATH:  LABEL        STACK DEPTH
             REVERSE1A        1
             TAG2             3
             LX002            3
***
COMPUTATION NUMBER 96 AT INSTRUCTION 11
ALONG PATH:  LABEL        STACK DEPTH
             REVERSE1A        1
             TAG2             3
             LX002            3
***
COMPUTATION NUMBER 70 AT INSTRUCTION 3
ALONG PATH:  LABEL        STACK DEPTH
             REVERSE1A        1
***
```

```
*****
REDERIVED FORM
*****

  (EQ L F)                                     (70 5 0)
      /\                                          /\
     /  \                                        /  \
RL (REVERSE1A (CDR L) (CONS (CAR L) RL))  6 (96 (94 5) (88 (80 5) 6))
```

Trace of the Proof

```
*****
MANIPULATE CANONICAL FORM TO MATCH REDERIVED FORM
*****
***CANONICAL***

  (EQ L F)                                     (20 5 0)
      /\                                          /\
     /  \                                        /  \
RL (REVERSE1A (CDR L) (CONS (CAR L) RL))  6 (28 (22 5) (26 (24 5) 6))

***REDERIVED***

  (EQ L F)                                     (10 5 0)
      /\                                          /\
     /  \                                        /  \
RL (REVERSE1A (CDR L) (CONS (CAR L) RL))  6 (18 (16 5) (14 (12 5) 6))

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ L F)        (10 5 0)
***BY USING THE FORM***

  (EQ L F)                                     (20 5 0)
      /\                                          /\
     /  \                                        /  \
RL (REVERSE1A (CDR L) (CONS (CAR L) RL))  6 (28 (22 5) (26 (24 5) 6))

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
RL      6
***BY USING THE FORM***
RL      6
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(REVERSE1A (CDR L) (CONS (CAR L) RL))        (18 (16 5) (14 (12 5) 6))
***BY USING THE FORM***
(REVERSE1A (CDR L) (CONS (CAR L) RL))        (46 (40 5) (44 (42 5) 6))

***COMPUTATION NUMBER 12 IS MATCHED***
  EXPLICITLY BY COMPUTATION 42

***COMPUTATION NUMBER 14 IS MATCHED***
  EXPLICITLY BY COMPUTATION 44

***COMPUTATION NUMBER 16 IS MATCHED***
  EXPLICITLY BY COMPUTATION 40

***COMPUTATION NUMBER 18 IS MATCHED***
  EXPLICITLY BY COMPUTATION 46

*****
SUCCESSFUL MATCH
*****
```

7.C NEXT

NEXT is function which takes as its arguments a list L and an element X. It searches L for an occurrence of X. If such an occurrence is found, and if it is not the last element of the list, then the next element in the list is returned as the result of the function. Otherwise, NIL is returned. For example, application of the function to the list (A B C D E) in search of D would result in E, while a search for E or F would result in NIL.

The function is interesting because it exhibits what we have previously termed *loop shortcutting*. The actual proof demonstrates the examination of the set of backward paths in order to determine a mismatching recursive call. Hopefully, the proof will resolve any questions arising from our previous discussions. Furthermore, the LAP encoding is extremely compact. The inner loop is four instructions long. This is minimal when we consider that the inner loop consists of four operations – CAR, CDR, EQ test, and recursion. Finally, note that the order of computing the CDR operation is shown to be capable of being safely rearranged.

LISP Encoding:

```
(DEFPROP NEXT (LAMBDA (L X)
        (COND ((NULL L) F)
              ((EQ (CAR L) X)
                        (COND ((NULL (CDR L)) F)
                              (T (CADR L))))
              (T (NEXT (CDR L) X)))) EXPR)
```

MLISP Encoding:

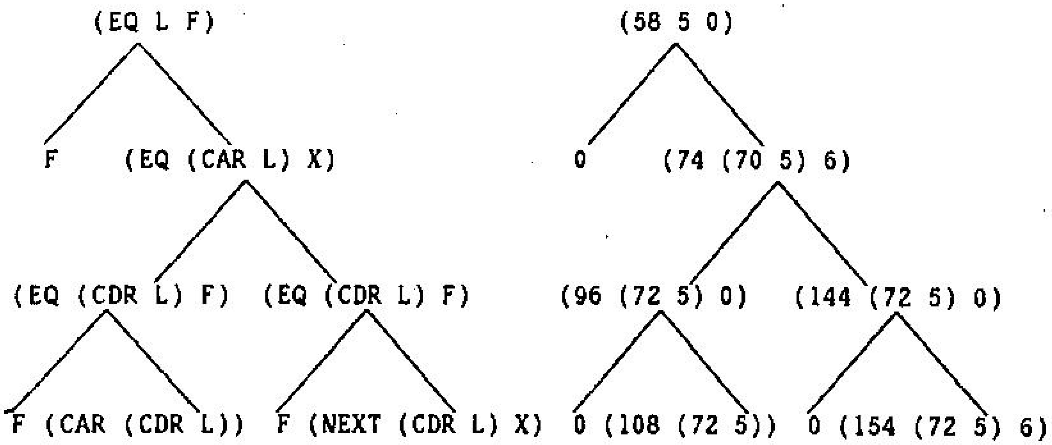
```
NEXT(L,X) = if NULL(X) then NIL
            else if CAR(L) EQ X then
                if NULL(CDR(L)) then NIL
                else CADR(L)
            else NEXT(CDR(L),X)
```

LAP Encoding:

```
(LAP NEXT SUBR)
NEXT     (JUMPE 1 DONE)
LOOP     (HLRZ 3 0 1)
         (HRRZ 1 0 1)
         (CAIE 3 0 2)
         (JUMPN 1 LOOP)
         (SKIPE 0 1)
         (HLRZ 1 0 1)
DONE     (POPJ P)
         NIL
```

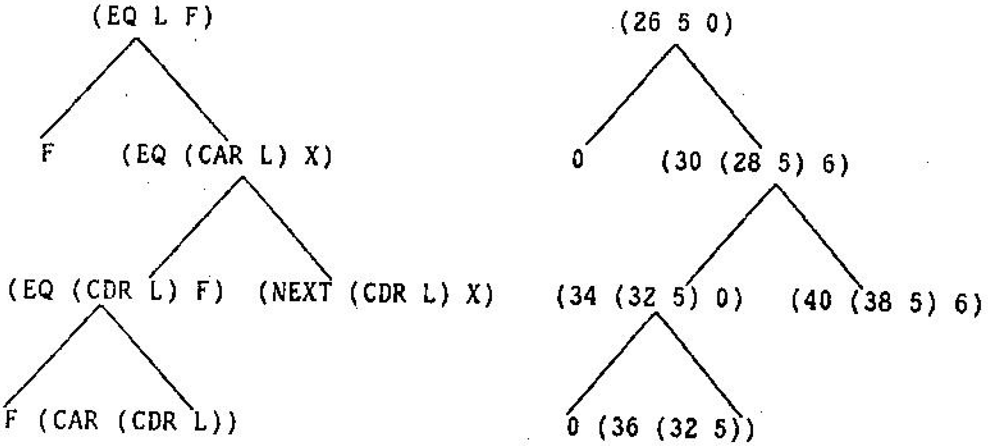
```
*****
INTERNAL REPRESENTATION OF THE LAP PROGRAM
*****
**** LABEL *** PROGRAM COUNTER *** INSTRUCTION *********
     NEXT             1            (JUMPE 1 DONE)
     LOOP             2            (HLRZ 3 0 1)
                      3            (HRRZ 1 0 1)
                      4            (CAIE 3 0 2)
     LX001            5            (JUMPN 1 LOOP)
     LX002            6            (SKIPE 0 1)
     LX003            7            (HLRZ 1 0 1)
     DONE             8            (POPJ 12)

*****
COMPUTATION NUMBER CORRESPONDENCE LIST
*****
COMPUTATION NUMBER 154 AT INSTRUCTION 5
ALONG PATH:  LABEL        STACK DEPTH
             NEXT              1
             LOOP              1
             LX001             1
***
COMPUTATION NUMBER 144 AT INSTRUCTION 5
ALONG PATH:  LABEL        STACK DEPTH
             NEXT              1
             LOOP              1
             LX001             1
***
COMPUTATION NUMBER 108 AT INSTRUCTION 7
ALONG PATH:  LABEL        STACK DEPTH
             NEXT              1
             LOOP              1
             LX002             1
             LX003             1
***
COMPUTATION NUMBER 72 AT INSTRUCTION 3
```

```
ALONG PATH:  LABEL          STACK DEPTH
             NEXT                1
             LOOP                1
***
COMPUTATION NUMBER 96 AT INSTRUCTION 6
ALONG PATH:  LABEL          STACK DEPTH
             NEXT                1
             LOOP                1
             LX002               1
***
COMPUTATION NUMBER 70 AT INSTRUCTION 2
ALONG PATH:  LABEL          STACK DEPTH
             NEXT                1
             LOOP                1
***
COMPUTATION NUMBER 74 AT INSTRUCTION 4
ALONG PATH:  LABEL          STACK DEPTH
             NEXT                1
             LOOP                1

***
COMPUTATION NUMBER 58 AT INSTRUCTION 1
ALONG PATH:  LABEL          STACK DEPTH
             NEXT                1
***

*****
REDERIVED FORM
*****
```

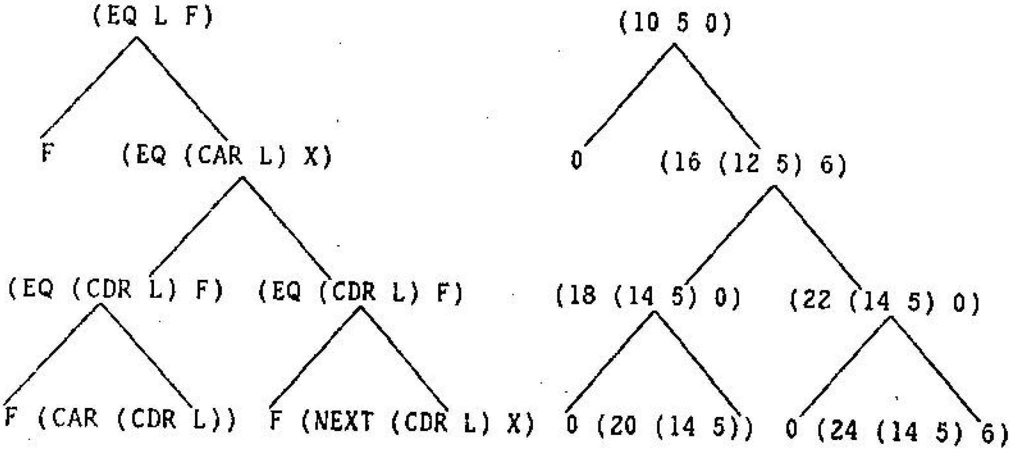


Trace of the Proof

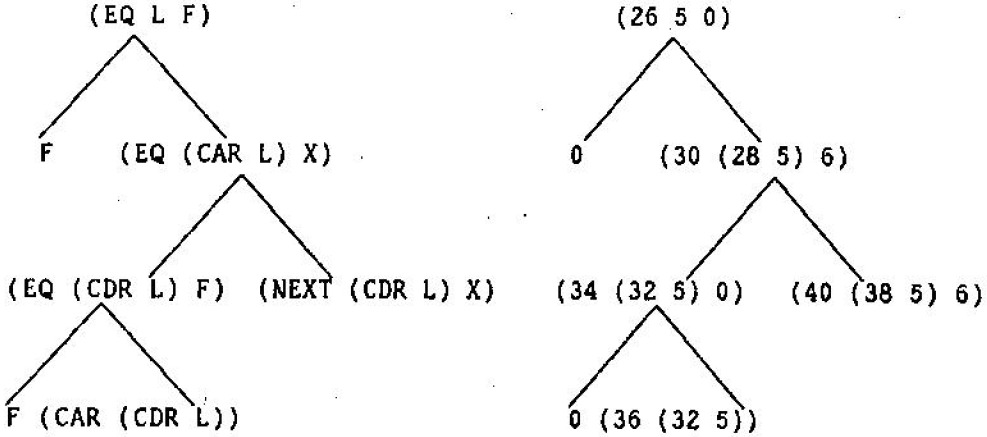
```
*****
MANIPULATE CANONICAL FORM TO MATCH REDERIVED FORM
*****
***CANONICAL***
```



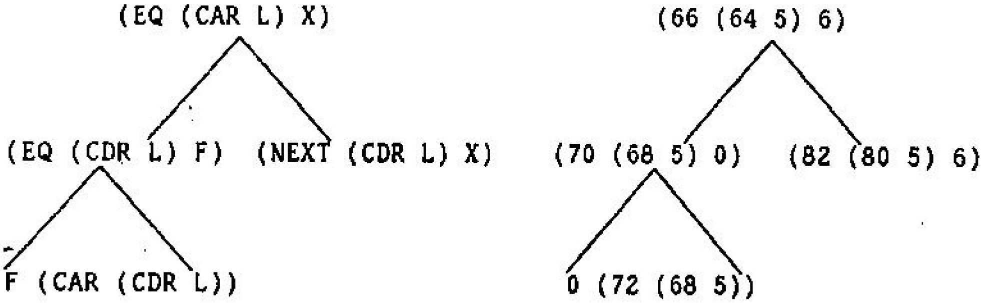
```
***REDERIVED***
```



```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ L F)          (10 5 0)
***BY USING THE FORM***
```

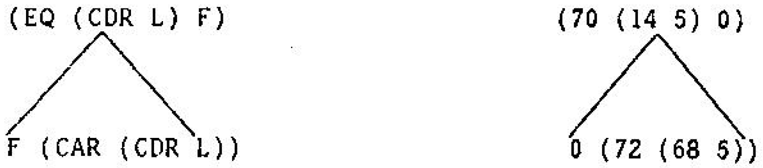


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F        0
***BY USING THE FORM***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CAR L) X)        (16 (12 5) 6)
***BY USING THE FORM***
```



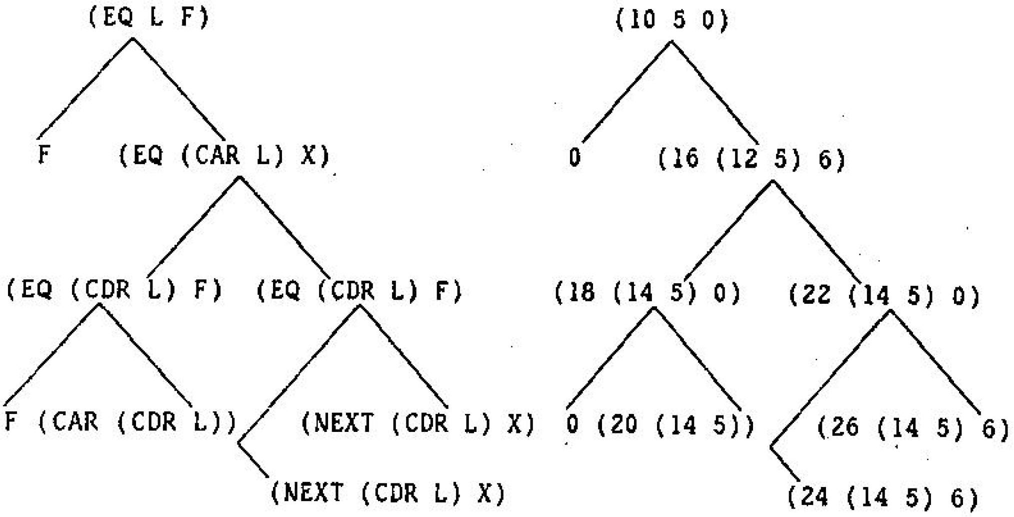
```
***COMPUTATION NUMBER 12 IS MATCHED***
  EXPLICITLY BY COMPUTATION 64

***COMPUTATION NUMBER 14 IS MATCHED***
  EXPLICITLY BY COMPUTATIONS 80 68
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)          (18 (14 5) 0)
***BY USING THE FORM***
```

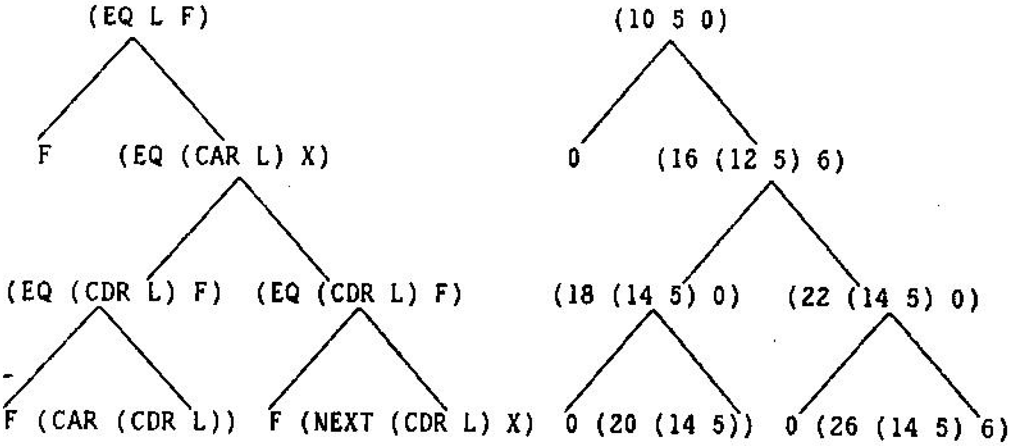


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F          0
***BY USING THE FORM***
F          0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(CAR (CDR L))          (20 (14 5))
***BY USING THE FORM***
(CAR (CDR L))          (126 (14 5))

***COMPUTATION NUMBER 20 IS MATCHED***
  EXPLICITLY BY COMPUTATION 126

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)          (22 (14 5) 0)
***BY USING THE FORM***
(NEXT (CDR L) X)        (160 (14 5) 6)

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F          0
***BY USING THE FORM***
(NEXT (CDR L) X)        (160 (14 5) 6)

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(NEXT (CDR L) X)        (24 (14 5) 6)
***BY USING THE FORM***
(NEXT (CDR L) X)        (298 (14 5) 6)

***COMPUTATION NUMBER 24 IS MATCHED***
  EXPLICITLY BY COMPUTATION 298
```

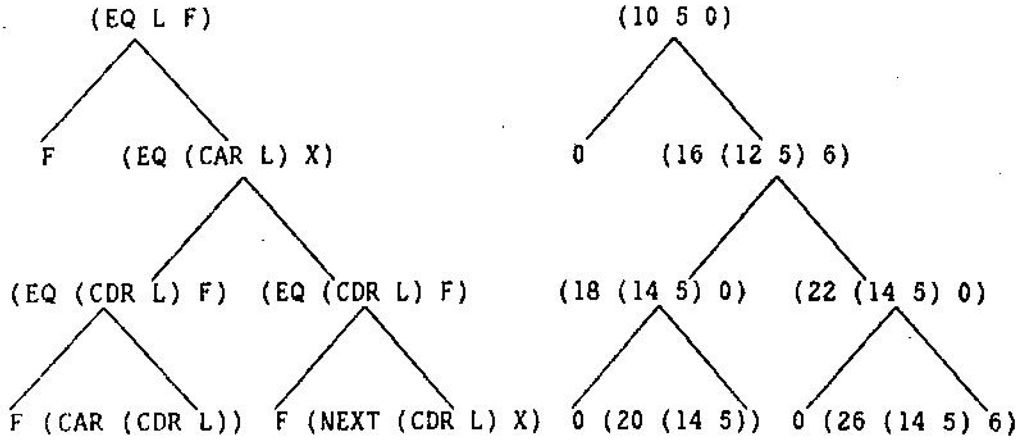
```
*****
DETERMINING IF MISMATCHES WERE DUE TO RECURSIVE CALLS
*****
*****
MANIPULATE REDERIVED FORM TO MATCH CANONICAL FORM TO GET RID OF
RECURSIVE MISMATCHES
*****
***CANONICAL***
```



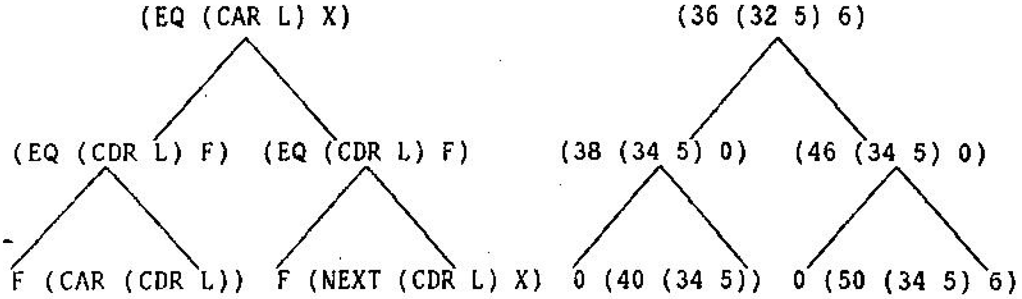
```
***REDERIVED***
```



```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ L F)        (10 5 0)
***BY USING THE FORM***
```

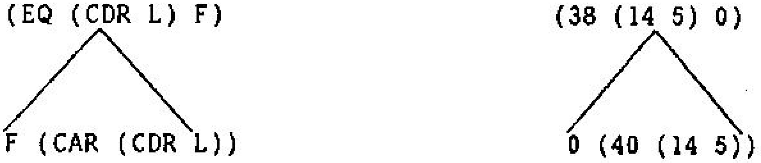


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F        0
***BY USING THE FORM***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CAR L) X)        (16 (12 5) 6)
***BY USING THE FORM***
```

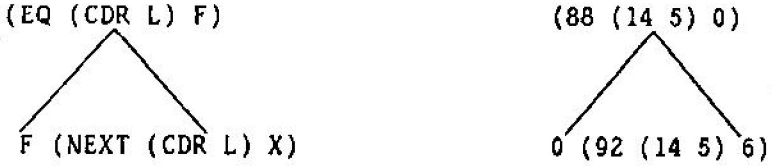


```
***COMPUTATION NUMBER 12 IS MATCHED***
  EXPLICITLY BY COMPUTATION 32

***COMPUTATION NUMBER 14 IS MATCHED***
  EXPLICITLY BY COMPUTATION 34
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)        (18 (14 5) 0)
***BY USING THE FORM***
```

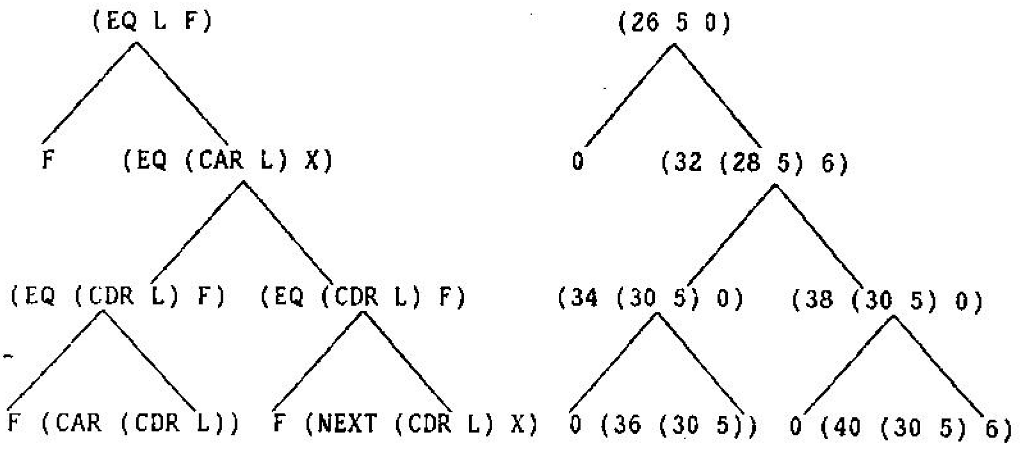


```
****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F        0
***BY USING THE FORM***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(CAR (CDR L))         (20 (14 5))
***BY USING THE FORM***
(CAR (CDR L))         (62 (14 5))

***COMPUTATION NUMBER 20 IS MATCHED***
  EXPLICITLY BY COMPUTATION 62

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)        (22 (14 5) 0)
***BY USING THE FORM***
```

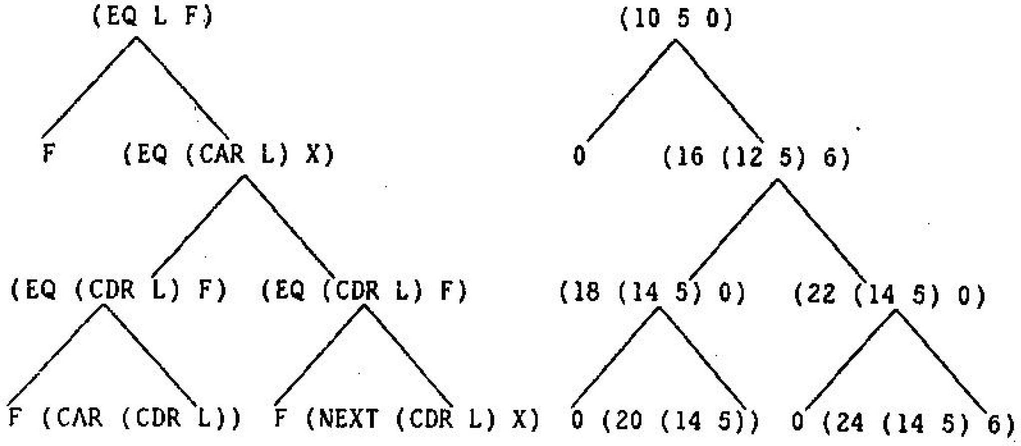


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(NEXT (CDR L) X)      (24 (14 5) 6)
***BY USING THE FORM***
F        0

*****
ATTEMPTING TO FIND A BACKWARD PATH MATCHING RECURSIVE MISMATCHES OF
COMPUTATION NUMBER 24
*****
```

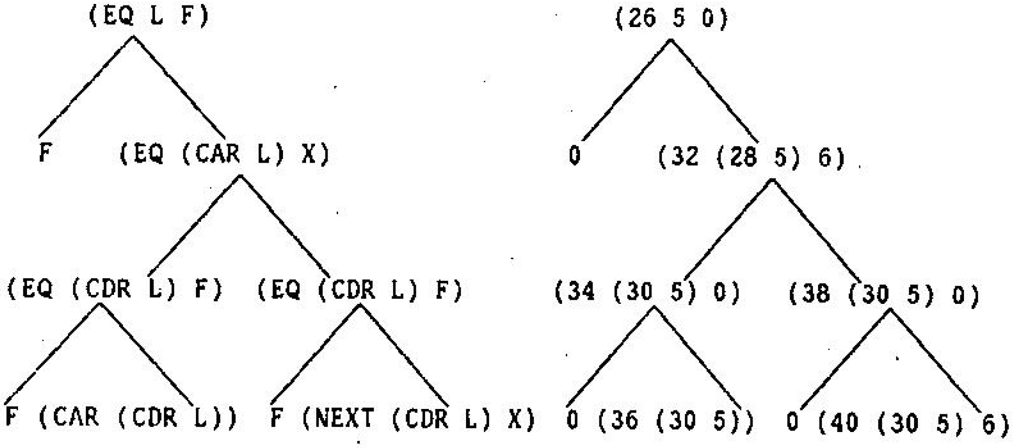
```
*****
MANIPULATE REDERIVED FORM TO MATCH BACKWARD PATH TO REMOVE RECURSIVE
CALL MISMATCH
*****
***REDERIVED FORM***
F        0

***BACKWARD PATH***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F        0
***BY USING THE FORM***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(NEXT (CDR L) X)       (26 (14 5) 6)
***BY USING THE FORM***
(NEXT (CDR L) X)       (162 (14 5) 6)

***COMPUTATION NUMBER 26 IS MATCHED***
  EXPLICITLY BY COMPUTATION 162

*****
MANIPULATE CANONICAL FORM TO MATCH REDERIVED FORM
*****
***CANONICAL***
```



```
***REDERIVED***
```

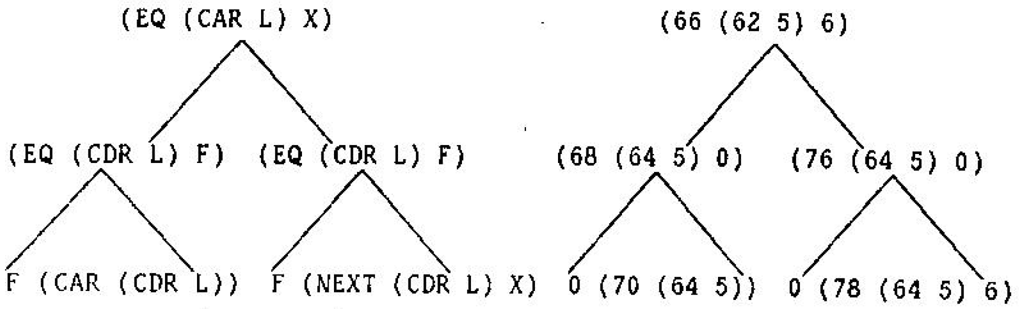


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ L F)        (10 5 0)
***BY USING THE FORM***
```

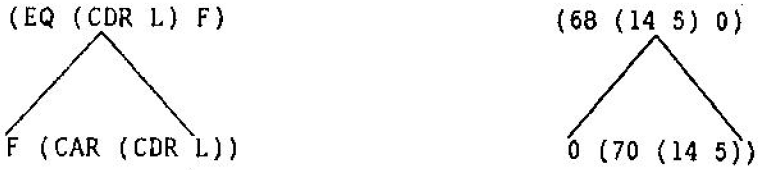


```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F         0
***BY USING THE FORM***
F         0
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CAR L) X)        (16 (12 5) 6)
***BY USING THE FORM***
```

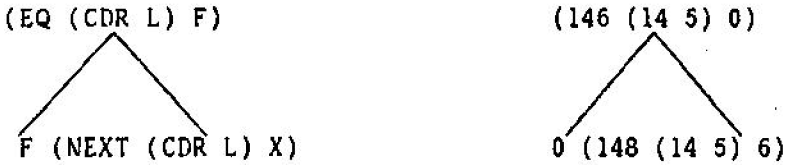


```
***COMPUTATION NUMBER 12 IS MATCHED***
  EXPLICITLY BY COMPUTATION 62

***COMPUTATION NUMBER 14 IS MATCHED***
  EXPLICITLY BY COMPUTATION 64

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)        (18 (14 5) 0)
***BY USING THE FORM***
```



```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F       0
***BY USING THE FORM***
F       0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(CAR (CDR L))         (20 (14 5))
***BY USING THE FORM***
(CAR (CDR L))         (122 (14 5))

***COMPUTATION NUMBER 20 IS MATCHED***
  EXPLICITLY BY COMPUTATION 122
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(EQ (CDR L) F)        (22 (14 5) 0)
***BY USING THE FORM***
```



```
(EQ (CDR L) F)                        (146 (14 5) 0)

 F (NEXT (CDR L) X)                   0 (148 (14 5) 6)
```

```
*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
F        0
***BY USING THE FORM***
F        0

*****
TRYING TO MATCH THE COMPUTATION NUMBERS APPEARING IN THE FUNCTION
*****
(NEXT (CDR L) X)      (24 (14 5) 6)
***BY USING THE FORM***
(NEXT (CDR L) X)      (274 (14 5) 6)

***COMPUTATION NUMBER 24 IS MATCHED***
  EXPLICITLY BY COMPUTATION 274

*****
SUCCESSFUL MATCH
*****
```

CHAPTER 8

CONCLUSIONS

The previous chapters have provided an indication as to how our original goal of proving the correctness of the translation process from a high level language to a low level language using LISP and LAP respectively has been accomplished. The reader hopefully will go away with more than the thought that a tool has been created. More importantly, we have gained a deeper understanding of how to deal automatically with translations of programs between high and extremely low level languages. The semantics of LISP have been scrutinized and reduced to a level compatible with our goals. This has resulted in the definition of the CMPLISP environment and the identification of protection as one of the primary weaknesses of the current implementation of LISP 1.6. In addition, the implementation of the proof system was done entirely in the subset of LISP defined by CMPLISP. This demonstrates that the concept of a free variable is not as critical as some may believe – the notion of a SPECIAL variable as defined in Chapter 2 proved to be more than adequate.

A procedure has also been presented for describing a computer instruction set. We have noted the capability of such a system in aiding the programmer in debugging an erroneous encoding. In the case of a compiler, the latter is quite useful as it provides a means of insuring that the object code will faithfully execute the source program without having to resort to a comparison of input output pairs. In fact, this is one of the main drawbacks of a decompilation system such as that proposed by [Housel73]. The latter uses sample executions to verify correctness. Clearly, such a method will only guarantee the correctness for the examples proved. An analogy can be drawn between the latter and our method. Namely, our system does for a compiler what decompilation does for a single program. Let us clarify this point. Our proof system cannot prove a compiler correct; however, for each program input to it, we may in fact give a statement as to whether or not the function has been correctly translated (or maybe no answer). Thus our compiler is only correct for the programs input to it. Note the duality between program and data – i.e. the data for a compiler is a program. Actually, there is no real distinction between program and data since a compiler is also a program. Nevertheless, it should be clear that compilers are generally a bit large for a decompilation system to handle.

At this point it is appropriate to summarize the ideas that enable our system to derive its power. First, throughout the work we have tried to shift the burden of work necessary for achieving tasks to the location where they would most easily be achieved. This has been seen by our use of an intermediate form for representing the function rather than the LISP function itself. This intermediate form simply recorded all of the computations performed in the program along with the conditions for their computation. The advantages were the decoupling of the notion of effect and value as seen in the handling of the SETQ, RPLACA, RPLACD, etc. constructs. The latter is also relevant to the use of the bindings of variables whenever these variables had known values (i.e. local variables, internal LAMBDA, and SPECIAL variables). Use of a normal form rather than a canonical form (using the strict definition of these terms) has enabled us to be flexible with respect to use of EQ and EQUAL and the capability to interchange them when necessary (i.e. we did not use a lexicographic minimum type of ordering as shown in the examples given in Figures 5.2 and 5.3). Similarly, for the use of equality information and redundant acts of assignment.

The actual proof procedure makes use of the distinction between matching and duplicate predicate detection. It also exploits the information that is returned by the rederivation process in proving the correctness of programs which implement recursion by bypassing the start of the program. We recall that generally speaking the canonical form was manipulated to match the rederived form. The reason for this decision is that we know from the canonical form what the order of the testing of conditions and computing of the arguments should be. This information is invaluable in the determination of when the rederived form corresponds to the original encoding of the algorithm. Notice that the proof procedure was completely independent of the description of the host computer instruction set. Equally important is the fact that nowhere in the machine description procedure, are any comments made as to how a sequence of instructions can be used to yield the encoding of a particular construct in the higher level language.

The remainder of this chapter consists of suggestions for future research as well as some observations for which we could not find a forum in any of the previous chapters.

8.A Suggestions for Future Research

During our previous discussions we have mentioned many ideas not all of which have been implemented. In this section we will recapitulate them as well as introduce some new ones. Of course, the main suggestion is the implementation of CMPLISP as an environment for LISP programs as well as an optimizing compiler. The optimizer is motivated by the examples that we have presented in the preceding chapters. Hopefully, we have demonstrated that the primary operations performed by a LISP program relate to data motion in the process of setting up correct linkages between the various functions. This type of operation cannot be optimized via the use of classical optimization techniques as espoused by various papers on optimization ([AllenF69],[Cocke70]). Our optimizations are to a large degree dependent on the machine on which the program is to be executed. Classical optimization generally deals with common subexpression elimination and strength reduction. Furthermore, implementation of an optimizer may result in certain optimizations which our system might prove to be incapable of handling. This would provide a useful indication of any deficiencies that went undetected.

We feel that optimization can take on a heuristic flavor. Such a procedure would attempt a variety of optimizations and determine if the resulting encoding preserves the intent of the original program. It is at this point that our verification system steps in and insures the correspondence. Another approach to optimization could be a process that examines the resulting code from a normal compiler, and tries to optimize it (e.g. conditional jumps to unconditional jumps and vice versa). This is a postoptimizing step and has in fact been suggested by [Nievergelt65]. Such a procedure can be complemented in part by the proof procedure which can mark all instructions which are never executed during the symbolic execution phase of the rederivation procedure. We have seen such examples in Chapter 1 (i.e. in the transition from Figures 1.6 and 1.19 to 1.7 and 1.20 respectively). An interesting approach to the implementation of an optimizer is to base it on a program understanding procedure using a set of heuristics closely mimicking the reasoning process that we used in optimizing the examples in Chapter 1. Recall that these optimizations depended heavily on an understanding of the computer executing these programs. Finally, the proof system as it stands can be used as a debugging system in the development of an optimizing compiler as well as in a computer-aided instruction method for assembly language programming.

As of the time of writing this thesis the SET, FEXPR, and PROG with a GO constructs have not yet been implemented. When work continues, they can be added to the system with ease. Some more difficult additions include error correction, loop economy, and topological sorting of the bypassed paths. Of these suggestions, we regard error recovery as being potentially the most interesting from a research point of view. We believe that such a procedure is a logical step in continuing the work from a program understanding point of view. Other fruitful lines of work include extending the concept of bypassing the start of a program to function calls other than recursion. This would yield a result similar to the concept of closure as discussed in [Wegbreit73]. Of course, the proof procedure would have to be slightly modified to include knowledge about valid entry points to other functions. In addition, we will have to prove that bypassing the start of the program is legal. Such a feature will make our system be capable of handling the concept of block–compiling which is a feature of BBN–LISP[Teitelman71]. Basically, in this mode, the compiler generates code for a set of functions which are declared to comprise a block and interfunction optimization is attempted.

Earlier we mentioned that the user indicates to the system certain information relevant to the proof procedure†. For example, the names of the commutative and antisymmetric functions must be declared. Other important information consists of the highest numbered accumulator destroyed by the function, and whether or not the function must be invoked via the CALL mechanism (useful for tracing). Some potentially useful information includes more detail as to the accumulators destroyed by functions, and which SPECIAL variables need not have their contents be identical upon function exit as upon function entry (this implies that they can be used as locations to store temporary results of computations). Note that the system is initialized to contain some information about the pre–declared functions (e.g. the antisymmetry of CONS and XCONS, etc.).

Another area where improvement could be made is in the equality determination algorithm. Currently, we can only handle equality in terms of instances of formulas. General relations must be repeated in the data base for each instance. For example, the identity CAR(CONS(A,B)) EQ A must be explicitly entered in the data base. This is done whenever it is determined that such a relation holds (in this example this occurs whenever a CONS is encountered). The problem with such an approach is that the data base becomes too large rather quickly. Such a system works adequately for the limited identities with which we are currently dealing. The clumsiness of this approach becomes quite obvious when a relationship such as associativity is considered. The capability for handling associativity is quite useful for operations such as *PLUS, *TIMES, and *APPEND. For example, if ⊗ is an associative operation, then just to handle A⊗B⊗C would require 4 entries in the data base. If ⊗ is also commutative, then we would need to double the requisite number of entries. Another suggestion along a similar vein is to incorporate identities pertaining to data of type other than lists (i.e. arithmetic as will be seen in one of the ensuing discussions). Finally, we would like to have a capability of drawing inferences from untested predicates during the rederivation process. This should be familiar from our discussion of the type of inferences that can be made from the two–valued nature of our predicates which nevertheless go undetected.

Some of the most interesting extensions to the present system include a shift to another computer and to another LISP implementation. Another computer would have a different architecture, and thus would serve as a test of the generality of our machine description process. In addition, we should be able to point out deficiencies in the machine description process as well as some possible rectification. A different LISP implementation would serve to indicate the adequacy of the data

† For a sample user session, see Appendix 6.

structures necessary for the computation model. For example, suppose a parameter stack is used for the arguments to a function (e.g. as in INTERLISP[Teitelman74]), then our system will still work. The only addition is a data structure, say PSTACK, which would contain the parameters. Other implementations may also have only one pointer per word (IBM 360) or even three pointers per word (CDC 6500-6600 series).

Earlier we discussed two different encodings for the NEXT function, Algorithms 1 and 2. At that point we mentioned that we could not directly prove their equivalence. Actually, this is not that difficult. The main problem was the fact that once L was determined not to be NIL, then in Algorithm 1 the nullness of CDR(L) was always tested, while in Algorithm 2 this was only done when CAR(L) was EQ to X. The canonical forms of the symbolic representation of the two Algorithms are given in Figure 8.1.

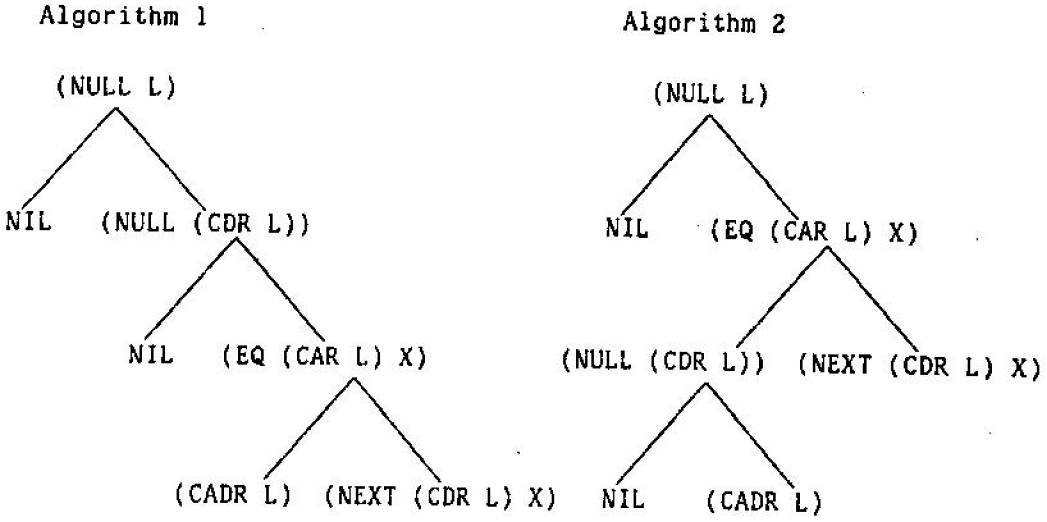


Figure 8.1 - Canonical Forms of Algorithms 1 and 2 for NEXT

Notice that Algorithm 1 could be converted to a form similar to Algorithm 2 by introducing the test EQ(CAR(L),X) in the true subcase of the test for the nullness of CDR(L). CAR(L) could be introduced because CDR(L) has been safely computed. This has been discussed in Chapter 5. Once this is done, we would have the form given in Figure 8.2.

The subsequent proof will have the effect of rearranging the conditions in Figure 8.2 to yield Figure 8.3. The next crucial step would be to expand the terminal node of Algorithm 2 corresponding to NEXT(CDR(L),X) to a test of EQ(CDR(L),NIL) with identical terminal nodes consisting of NEXT(CDR(L),X). A final step would have to recognize that under the subcase that CDR(L) is NIL, NEXT(CDR(L),X) is also NIL. This is an illustration of loop economy in addition to the use of certain implicit properties of the CAR and CDR operations. We feel that such proofs are more along the lines of a theorem prover for LISP functions.

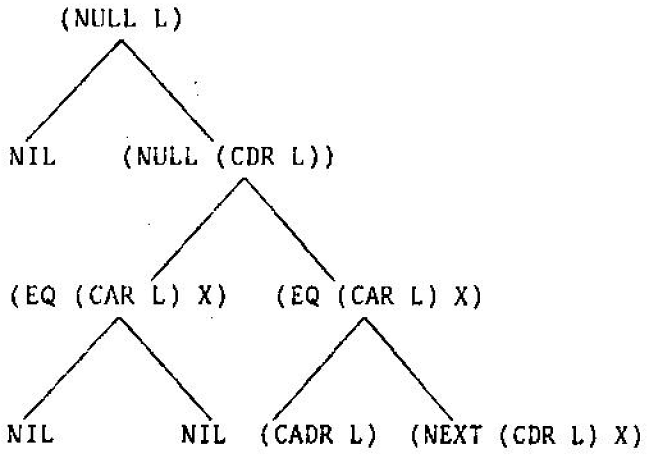


Figure 8.2

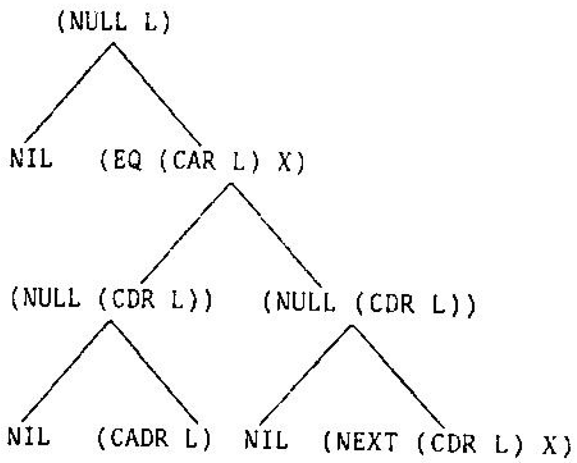


Figure 8.3

8.B Observations

During the process of examining optimizations we have encountered several problems which have caused us to make some observations as to the inefficiencies in LISP and in the instruction set of the PDP-10. The inefficiencies in LISP concern arithmetic functions, and the use of CONS and the subsequent need for garbage collection. Examination of the instruction set of the PDP-10 during the optimization simulation in Chapter 1 has shown that even greater increases in speed could be achieved if the PDP-10 had a greater variety of halfword operations which would more closely parallel the set available for full words.

8.B1 Arithmetic

In our discussions of optimizations, we did not show any optimizations that related to arithmetic. Most of our discussion of optimizations was relevant to data of type s-expression. This was not accidental. The absence of types in LISP makes arithmetic rather inefficient. This is because numbers are not represented by their values. Instead, they are generally represented as atoms with a property indicating that the value is to be interpreted as a number. Thus arithmetic operations such as addition, multiplication, comparison, etc. cannot be performed inline. This leads to gross inefficiencies. For example, we will briefly consider the variant of the definition of the Fibonacci function given in Figure 8.4[†].

```
f(x) ≡ if x=0 or x=1 then 1
            else f(x-1)+f(x-2)
```

Figure 8.4 - Fibonacci Function Definition

[†] We use the operand "=" in the definition which may be interpreted as either EQ or EQUAL. The important point to note is that since one of the operands is an atom (i.e. the number), there is no distinction between the EQ and EQUAL. For more details, see Sections 2.B4, 3.E, and 5.B.

The encoding generated by the LISP 1.6 compiler for this function is given in Figure 8.5. Notice the rather lengthy inner loop[†] (i.e. 17 instructions or 46.15μs). Optimizations can be performed to reduce the length of the inner loop. However, unlike previous examples, a considerable amount of time must be spent in the arithmetic operations thereby rendering insignificant much of the improvement in speed that is implied by optimizations such as those seen in Chapters 1 and 4.

```
(LAP FIB SUBR)
        (PUSH P 1)              push X on the stack
        (CAIN 1 (QUOTE 0))      skip if X is not 0
        (JRST 0 TAG3)           jump to TAG3
        (CAIE 1 (QUOTE 1))      skip if X is 1
        (JRST 0 TAG2)           jump to TAG2
TAG3    (MOVEI 1 (QUOTE 1))     load register 1 with 1
        (JRST 0 TAG1)           jump to TAG1
TAG2    (MOVEI 2 (QUOTE 1))     load register 2 with 1
        (MOVE 1 0 P)            load register 1 with X
        (CALL 2 (E *DIF))       compute X-1
        (CALL 1 (E FIB))        compute FIB(X-1)
        (MOVEI 2 (QUOTE 2))     load register 2 with 2
        (PUSH P 1)              push FIB(X-1) on the stack
        (MOVE 1 -1 P)           load register 1 with X
        (CALL 2 (E *DIF))       compute X-2
        (CALL 1 (E FIB))        compute FIB(X-2)
        (POP P 2)               restore FIB(X-1) from the stack
        (CALL 2 (E *PLUS))      compute FIB(X-2)+FIB(X-1)
TAG1    (SUB P (C 0 0 1 1))     undo the first push operation
        (POPJ P)                return
        NIL
```

Figure 8.5

One possible optimization of the function is the use of the transformations given in Figures 1.27 and 1.28. Specifically, we may apply the second transformation in Figure 1.27 to the function given in Figure 8.4. The derivation of the correct form takes two steps as shown in Figure 8.6. The resulting function is named h and is initially activated with h(x,0) since 0 is the identity element of the addition operation.

```
f(x) ≡ if x=0 or x=1 then 1  becomes  h(x,y) ≡ if x=0 or x=1 then 1+y
       else f(x-1)+f(x-2)                        else h(x-1,f(x-2)+y)
                                               ≡ if x=0 or x=1 then 1+y
                                                 else h(x-1,h(x-2,y))
```

Figure 8.6 - Fibonacci Function with Two Arguments

The compiler generated LAP encoding for the transformed function is given in Figure 8.7. Note

[†] As in Chapter 1 we also indicate, in parentheses, the execution time of the inner loop in microseconds (denoted by μs). Once again, the reader is cautioned that external function calls are only reflected by the amount of time necessary to perform the linking operation.

that there is little or no improvement over the previous form of the algorithm. The main benefit of the transformation is the possibility of implementing the final recursive call to FIB by iteration.

```
(LAP FIB SUBR)
        (PUSH P 1)               push X on the stack
        (PUSH P 2)               push Y on the stack
        (CAIN 1 (QUOTE 0))       skip if X is not 0
        (JRST 0 TAG3)            jump to TAG3
        (CAIE 1 (QUOTE 1))       skip if X is 1
        (JRST 0 TAG2)            jump to TAG2
TAG3    (MOVEI 1 (QUOTE 1))      load register 1 with 1
        (CALL 2 (E *PLUS))       compute 1+Y
        (JRST 0 TAG1)            jump to TAG1
TAG2    (MOVEI 2 (QUOTE 1))      load register 2 with 1
        (MOVE 1 -1 P)            load register 1 with X
        (CALL 2 (E *DIF))        compute X-1
        (MOVEI 2 (QUOTE 2))      load register 2 with 2
        (PUSH P 1)               push X-1 on the stack
        (MOVE 1 -2 P)            load register 1 with X
        (CALL 2 (E *DIF))        compute X-2
        (MOVE 2 -1 P)            load register 2 with Y
        (CALL 2 (E FIB))         compute FIB(X-2,Y)
        (MOVE 2 1)               load register 2 with FIB(X-2,Y)
        (POP P 1)                pop X-1 from the stack
        (CALL 2 (E FIB))         compute FIB(X-1,FIB(X-2,Y))
TAG1    (SUB P (C 0 0 2 2))      undo the first two push operations
        (POPJ P)                 return
        NIL
```

Figure 8.7

A better encoding is shown in Figure 8.8. Here we see a reduction in the total space occupied by the function from 23 to 15 instructions. Moreover, the length of the inner loop has been reduced from 19 (51.57μs) to 12 (33.60μs) instructions. Some of the optimizations performed are an efficient manner of encoding an OR operation, the realization that there is no need to return to FIB once the addition in the non–recursive case has been computed, and the removal of some stack operations. Unfortunately, the function as currently encoded cannot be recognized by the system. This is because we have bypassed the start of the program by using the information that if $x \neq 1$, then $x-1 \neq 0$. Recall the comment made in Chapter 4 with respect to our inability to prove identities involving arithmetic quantities. There we concluded that such features need a theorem prover for the said domain. We could easily conceive of similar relationships that should be exploited for other data types. Future work could incorporate a domain–dependent proof method.

```
(LAP FIB SUBR)
FIB     (CAIN 1 (QUOTE 0))          skip if X is not 0
        (SKIPA 1 (C 0 0             load register 1 with 1 and skip
                  (QUOTE 1)))
PC3     (CAIN 1 (QUOTE 1))          skip if X is not 1
        (JCALL 2 (E *PLUS))         return Y+1
        (PUSH P 1)                  push X on the stack
        (PUSH P 2)                  push Y on the stack
        (MOVEI 2 (QUOTE 2))         load register 2 with 2
        (CALL 2 (E *DIF))           compute X-2
        (POP P 2)                   pop Y from the stack
        (CALL 2 (E FIB))            compute FIB(X-2,Y)
        (EXCH 1 0 P)                exchange FIB(X-2,Y) with X
        (MOVEI 2 (QUOTE 1))         load register 2 with 1
        (CALL 2 (E *DIF))           compute X-1
        (POP P 2)                   pop FIB(X-2,Y) from the stack
        (JRST 0 PC3)                compute FIB(X-1,FIB(X-2,Y))
        NIL
```

Figure 8.8

The main problem with the previous two algorithms for computing the Fibonacci function is the recomputing of values. Our previous formulations can be characterized as employing *going down* induction while our next presentation will have the inductive variable increase (i.e. *going up* induction). This has the advantage that the act of computing f(a) is only done once. However, we cannot prove the equivalence of the two methods using our system for reasons indicated in Chapter 1. The algorithm is given in Figure 8.9. Note that we have once again used an auxiliary function, g, which is activated by f as shown.

```
f(x) ≡ if x=0 or x=1 then 1
           else g(x-1,1,1)
g(x,y,z) ≡ if x=0 then y
               else g(x-1,y+z,y)
```

Figure 8.9 - Fibonacci Function with Three Arguments

The LAP encoding generated by the LISP 1.6 compiler for g (renamed to FIB1) is given in Figure 8.10. Notice that in terms of space requirements, we need considerably more memory to store this algorithm than the previous formulation. This is in part due to the need for a more complex activation, i.e. f, for the auxiliary function. However, the execution time has been greatly reduced.

```
(LAP FIB1 SUBR)
        (PUSH P 1)                push X on the stack
        (PUSH P 2)                push Y on the stack
        (PUSH P 3)                push Z on the stack
        (CAIE 1 (QUOTE 0))        skip if X is 0
        (JRST 0 TAG2)             jump to TAG2
        (MOVE 1 2)                load register 1 with Y
        (JRST 0 TAG1)             jump to TAG1
TAG2    (MOVEI 2 (QUOTE 1))       load register 2 with 1
        (MOVE 1 -2 P)             load register 1 with X
        (CALL 2 (E *DIF))         compute X-1
        (MOVE 2 0 P)              load register 2 with Z
        (PUSH P 1)                push X-1 on the stack
        (MOVE 1 -2 P)             load register 1 with Y
        (CALL 2 (E *PLUS))        compute Y+Z
        (MOVE 3 -2 P)             load register 3 with Y
        (MOVE 2 1)                load register 2 with Y+Z
        (POP P 1)                 pop X-1 from the stack
        (CALL 3 (E FIB1))         compute FIB1(X-1,Y+Z,Y)
TAG1    (SUB P (C 0 0 3 3))       undo the first three push operations
        (POPJ P)                  return
        NIL
```

Figure 8.10

A better encoding is given in Figure 8.11. We have managed to reduce the amount of memory required by the function from 20 to 14 while the number of instructions comprising the inner loop has been decreased from 18 (51.64μs) to 11 (31.11μs). Some of the optimizations that were performed include using the stack only when necessary, rearranging the order of computing arguments, recycling a stack location when its contents are no longer needed, and conversion of recursion to iteration.

```
(LAP FIB SUBR)
        (CAIN 1 (QUOTE 0))        skip if X is not 0
        (JRST 0 END)              jump to END
        (PUSH P 2)                push Y on the stack
        (PUSH P 1)                push X on the stack
        (MOVE 1 3)                load register 1 with Z
        (CALL 2 (E *PLUS))        compute Z+Y
        (EXCH 1 0 P)              exchange Z+Y with X
        (MOVEI 2 (QUOTE 1))       load register 2 with 1
        (CALL 2 (E *DIF))         compute X-1
        (POP P 2)                 pop Z+Y from the stack
        (POP P 3)                 pop Y from the stack
        (JRST 0 FIB)              compute FIB(X-1,Y+Z,Y)
END     (MOVE 1 2)                load register 1 with Y
        (POPJ P)                  return
        NIL
```

Figure 8.11

Despite the seemingly good improvement in the size of the inner loop, the effect of the optimization is still quite minimal in terms of execution time. What is really needed is a shift in representation. We would like to have a function type of type numeric, say NEXPR. Such functions can be recognized by noticing if all of the operations are arithmetic in nature. An alternate solution is to have the programmer declare these functions to be numeric†. The encodings for such functions would be distinguished from typeless functions by having a prologue which checks at run time that all of the arguments are of the proper type and they are replaced by the corresponding numeric value. Similarly, there is an epilogue which converts the result of the function to its proper LISP representation. The function is then executed using the more efficient representation. The main objection to such a treatment is that if garbage collection were to occur while executing a procedure of type NEXPR, then some of the locations which are known to contain active pointers, will contain numbers rather than valid LISP pointers. The latter could lead to difficulties during the marking phase. However, we have stipulated that the function only performs arithmetic operations and thus no garbage collection is possible. For an example of this method see Figure 8.12 which yields the encoding, due to Steve Savitsky, of the algorithm presented in Figure 8.9.

```
(LAP FIB1 SUBR)
FIB1    (JUMPE 1 DONE)        jump to DONE if X is 0
LOOP    (ADD 3 2)             compute Y+Z and store in register 3
        (EXCH 2 3)            exchange Y+Z and Y
        (SOJG 1 LOOP)         subtract one from X and if nonzero
                              then compute FIB(X-1,Y+Z,Y)
DONE    (MOVE 1 2)            load register 1 with Y
        (POPJ P)              return
        NIL
```

Figure 8.12

This encoding is quite efficient as is evident by its length of 6 instructions and an inner loop of 3 instructions. As indicated earlier, we cannot prove arithmetic identities as is needed in this case – i.e. $X-1>0$ is equivalent to $X-1\neq 0$. We are also currently unable to handle this representation of numbers; although such an extension is a possible direction for future work. This would involve changing the the data type of the parameters to the function from LISP pointers to data pointers and the reverse for the results of the function.

8.B2 CONS Optimization

In our discussion of optimization we only briefly mentioned the CONS operation. This is not an oversight. Optimizations with respect to the CONS operation are rather subtle. The problem is that once we have exhausted the amount of free cells, we must go through the entire List Structure and find all of the inactive links. Such a process is rather time consuming. This is especially true if most of the nodes are active. In such a case, very little storage is reclaimed, and garbage collection will probably have to be reinvoked within a short period of time. This situation tends to negate any effect of optimizations with respect to execution time (savings in space are still valid).

† Similar features are available in MACLISP[Moon74].

Our studies of the effects of optimizations on execution time did not take garbage collection into consideration. One important direction for future work is to measure the effect on garbage collection of our optimizations. Earlier, in Chapter 1, we mentioned the possible importance of reducing the stack size with respect to decreasing the number of active links to be pursued during the marking phase of the garbage collection algorithm. We also stated that such a decrease would most probably be accompanied by an increase in the number of cells that are reclaimed.

In the previous paragraph we mentioned how we could possibly affect garbage collection by use of code optimization. We could also reduce the need for garbage collection to occur by observing the situations in which we find ourselves using the CONS operation when we do not need it. This is the case where we are trying to overcome a deficiency in the programming language or even in the implementation. One example of the latter, which was briefly mentioned in Chapter 2, is the hashed CONS. Such an implementation means that prior to the allocation of a cell from the List Structure, we determine if a cell has already been allocated having the same components. The tradeoff between constant allocation and this method is a more frequent need for garbage collection versus a constant need to check if a cell has already been allocated with the said components. Another example is the case when we have more arguments than our limit†. In this case we would use a CONS operation to form a list of arguments. Thus once the called function exits, the cells used for combining the arguments into a list are no longer necessary.

Another very serious deficiency of LISP is the inability to return more than one result from a function without making a list of the results. Such a feature is available in the POP2[Burstall71] language. It could be implemented for LISP by returning the first result in accumulator 1 and all the remaining results in consecutive locations in an area immediately above the top of the stack as indicated by the stack pointer. The only remaining task is to indicate how a multiple result is to be specified in LISP. We feel that the most natural way is to add the special form RLAMBDA which is identical to an internal LAMBDA of as many arguments as there are results being returned, and only one binding – i.e. the function having more than one result. For example, see function G in Figure 8.13. The only distinction with LAMBDA is that the values of all but the first argument are found on top of the stack. Thus if any other functions are to be called, then the stack must be adjusted to save these values below the stack pointer which points to the top of the stack. A typical solution is to store the value that was returned as the first result in the location pointed at by the stack pointer and then increase the stack pointer by a quantity equal to the number of results that were returned. We must also provide a syntactic entity for returning more than one result. The easiest way to achieve this is to have in addition to the property denoting the function type, a property denoting the number of results that the function returns. For example, the function H in Figure 8.13 is an EXPR which returns 3 results. The actual act of returning more than one result, say n results, is to simply return the last n items in the <function body sequence>‡. We also make the stipulation that a function returns the same number of results in all cases. A more cogent example is the function DIVISION which returns as its results the QUOTIENT and the REMAINDER when integer division is performed on its two arguments – i.e. the first argument is integer–divided by the second argument. For example, function SUMDIV, of two arguments, in Figure 8.13 will return the sum of the quotient and the remainder when DIVIDEND is integer–divided by DIVISOR. Note .the use of the function DIVISION which is known to return two values.

† LISP 1.6 provides another parameter passing mechanism in this case known as an LSUBR which uses the stack. We shall ignore this construct in this discussion.

‡ See the definition of a LISP function given in Chapter 2.

```
(DEFPROP G (LAMBDA (A)
                   ((RLAMBDA (B C D)
                             (M B C D))
                    (H A)))
 EXPR 1)

(DEFPROP H (LAMBDA (A)
                   (H1 A)
                   (H2 A)
                   (H3 A))
 EXPR 3)

(DEFPROP SUMDIV (LAMBDA (DIVIDEND DIVISOR)
                        ((RLAMBDA (QUOTIENT REMAINDER)
                                  (*PLUS QUOTIENT REMAINDER))
                         (DIVISION DIVIDEND DIVISOR)))
 EXPR 1)
```

Figure 8.13 - Examples of Functions that Return More than One Result

The above two examples lead us to a feeling that the programming language should provide a capability for control over the allocation, and, more importantly, the deallocation of cells. At the present, the programmer has no control over the latter. Thus allocation occurs until there are no more available cells at which time garbage collection will occur. This is similar to the heap in ALGOL 68[Van Wijngaarden69]. Other languages provide an implicit mechanism for the deallocation of storage – i.e. upon block exit in ALGOL 60[Naur60]. We have seen in the case of an insufficient number of argument slots a desire for deallocation by the called function, while the multiple results case shows a need for deallocation by the function to which control is returned. A third mechanism is deallocation at function exit which corresponds to deallocation upon block exit using ALGOL 60 terminology. These various deallocation schemes could be achieved by use of specialized CONS operations which leave messages around as to the lifetime of the cell that has been allocated. The next step is to decide upon some efficient means of deallocating the cells. In the case that the cells are to be deallocated at function exit, no problem exists. However, if cells are to be deallocated when the calling function exits or the called function exits, then more sophisticated mechanisms are needed.

It may be justly argued that the solutions proposed in the previous paragraph place too great of a burden on the programmer. This is quite justified. Instead, we could have a very comprehensive flow analysis package which can determine when deallocation may occur. Such a solution is quite applicable to the case when a function returns more than one result via CONS operations which are subsequently decomposed. Another disadvantage is the need to always check at function exit if any deallocation is to occur. Clearly, a better solution is to do the deallocation during the CONS routine. This has the advantage of having the beneficiary of the feature do the requisite work.

8.B3 New Instructions

One of the interesting results of the optimizations which were examined in Chapter 1 is a gaining of insight into some desirable machine instructions. These operations are basically relevant to halfword operations and tests involving stack locations. We would like to be able to accomplish more than one operation at a time. This is of benefit because the dominant component of the execution time of a LISP function is in establishing the links between functions. Thus any overlap that can be achieved implies greater efficiency.

A typical example is to be able to test halfwords in memory while simultaneously loading an accumulator. This is analogous to the instructions in the SKIP family when a non–zero AC field is specified. The benefit of such an operation is quite obvious when we realize that the operations given in the left half of Figure 8.14 can be accomplished with one instruction[†]. This could be accomplished by an instruction having as its mnemonic a SKIP concatenated to the desired halfword operation concatenated to the appropriate condition. For example see the right side of Figure 8.14.

```
if NULL(CDR(L)) then CDR(L)          (SKIPHRRZE@ reg address)
if NOT(NULL(CDR(L))) then CDR(L)     (SKIPHRRZN@ reg address)
if NULL(CAR(L)) then CAR(L)          (SKIPHLRZE@ reg address)
if NOT(NULL(CAR(L))) then CAR(L)     (SKIPHLRZN@ reg address)
```

Figure 8.14 - Proposed New PDP-10 Instructions

Recall that SKIP operations take an operand from memory, test it, and if the AC field specification is non–zero, then load the value into the designated accumulator. We would like to have the antisymmetric partner of this family of operators. Namely, we want to be able to test a value in an accumulator while simultaneously storing it in memory. In our case memory would be in the stack area. Similarly, we would want to see this extension applied to the added instructions in the previous paragraph. The usefulness of this operation is in the capability of storing values in memory for purposes of safety while performing computations relevant to the LISP function. By safety we refer to the need for saving contents of certain accumulators (i.e. results of computations) while function calls occur that may destroy the locations containing these values.

A machine feature that we found missing involves the effective address cycle. Recall the revised algorithm for the REVERSE function (using an auxiliary function and a reversed ordering of the arguments) whose encoding in Figure 1.36 was erroneous. The problem was misapplication of the HRRZS operation. In our case we wanted the indirect addressing only to happen on the memory fetch and not on the operand store. Such an instruction means that there is no need for the customary load store cycle.

The usefulness of instructions such as the previous can be questioned. The criticism is especially valid, when these operations are considered on a stand alone basis. However, when they are used together, along with such optimizations as bypassing the start of a function (via loop shortcutting), then much greater savings result. The main advantage is that we can eliminate the common trait of compiled code which is a multitude of branch instructions whose main purpose is to circumvent other branch instructions. In addition, when there are instructions whose effect it is to perform a test and a skip based on the result of the test, then the ability to perform more than one operation at a time takes on an added dimension of importance. We have seen this to be the case in the most

[†] We are assuming that NIL is implemented by a pointer having value zero.

optimal encodings of Algorithm 2 for NEXT given in Figure 1.21 and the algorithm for REVERSE using an auxiliary function given in Figure 1.37. Note that our extensions have dealt primarily with speeding up computations involving CAR and CDR operations. This is important since these are the primitive operations of LISP, and for a machine to optimally execute a language, we must exploit all possible efficiencies. Most computers do not have the flexibility derivable from halfword operations as exist on the PDP—10 (this includes indirect addressing coupled with the halfword operations). This results in pretty poor performance of LISP. However, we have been given half of the carrot (i.e. half word operations) by the PDP—10, and in this section we stated our desire to have the entire carrot (i.e. full range of halfword operations).

BIBLIOGRAPHY

[AllenF69] Allen, F., *Program Optimization*, Annual Review in Automatic Programming, Volume 5, 1969, pp. 239–307.

[AllenJ74] Allen, J., *Super LISP*, to be published by McGraw Hill.

[ASI66] American Standards Association, *American Standard FORTRAN*, New York, 1966.

[Bell71] Bell, G., and Newell, A., *Computer Structures: Readings and Examples*, McGraw Hill, New York, 1971.

[Bobrow72] Bobrow, R.J., Burton, R.R., and Lewis, D., *UCI LISP Manual*, University of California at Irvine Technical Report No. 21, University of California at Irvine, Irvine, California, October 1972.

[Boyer73] Boyer R.S., and Moore, J.S., *Proving Theorems about LISP Functions*, Proceedings of the Third IJCAI, 1973, pp. 486–493.

[Burks71] Burks, A.W., Goldstine, H.H., and von Neumann, J., *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument* in Computer Structures: Readings and Examples by Gordon Bell and Allen Newell, McGraw Hill, New York, 1971, pp. 92–119.

[Burstall71] Burstall, R.M., Collins, J.S., and Popplestone, R.J., *Programming in POP2*, University Press, Edinburgh, Scotland, 1971.

[Church41] Church, A., *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, New Jersey, 1941.

[Cocke70] Cocke, J., and Schwartz, J., *Programming Languages and their Compilers*, NYU Courant Institute, April 1970.

[Darlington73]Darlington, J., and Burstall, R.M., *A System which Automatically Improves Programs*. Proceedings of the Third IJCAI, 1973, pp. 479–485.

[DEC69] ,*PDP–10 System Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, 1969.

[Deutsch73] Deutsch, L.P., *An Interactive Program Verifier*, Ph.D Thesis, Department of Computer Science, University of California at Berkeley, May 1973.

[Floyd67] Floyd, R.W., *Assigning Meanings to Programs*, Proceeding of a Symposium in Applied Mathematics, Volume 19, Mathematical Aspects of Science, (Schwartz, J.T. ed.), American Math Society, 1967, pp. 19–32.

[Golomb65] Golomb, S.W. and Baumert, L.D., *Backtrack Programming*, Journal of the ACM, October 1965, pp. 516–524.

[Hollander73] Hollander, C.R., *Decompilation of Object Programs*, Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Digital Systems Laboratory Technical Report No. 54, Stanford Electronics Laboratories, Stanford, California, 1973.

[Housel73] Housel, B.C., *A Study of Decompiling Machine Languages into High–level Machine Independent Languages*, Ph.D. Thesis, Department of Computer Science, Purdue University, Lafayette, Indiana, 1973.

[King69] King, J., *A Program Verifier*, Ph.D. Thesis, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania, 1969.

[King74] King, J., *A New Approach to Program Testing*, IBM Research Report RC 5037, Yorktown Heights, New York, September 19,1974.

[Knuth68] Knuth, D., *Fundamental Algorithms: The Art of Computer Programming, Volume 1*, Addison–Wesley, Reading, Massachusetts, 1968.

[Knuth74] Knuth, D., *Structured Programming with GO TO Statements*, Stanford University Computer Science Department Report STAN–CS–74–416, May 1974.

[London71] London, R., *Correctness of Two Compilers for a LISP Subset*, Stanford Artificial Intelligence Project Memo AIM–151, Computer Science Department, Stanford University, October 1971.

[Lowry69] Lowry, E.S., and Medlock, C.W., *Object Code Optimization*, Communications of the ACM, January 1969, pp. 13–22.

[McCarthy60] McCarthy, J., *Recursive Functions of Symbolic Expressions and their Computation by Machine*, Communications of the ACM, April 1960, pp 184–195.

[McCarthy62] McCarthy, J., Abrahams, P., Edwards, D., Hart, T., and Levin, M., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[McCarthy63] McCarthy, J., *A Basis for a Mathematical Theory of Computation*, in Computer Programming and Formal Systems (Eds. Braffort and Hirshberg), North Holland, Amsterdam, 1963.

[Moon74] Moon, D.A., *MACLISP Reference Manual*, Project MAC – Massachusetts Institute of Technology, Cambridge Massachusetts, 1974.

[Naur60] Naur, P., (Ed.), *Revised Report on the Algorithmic Language ALGOL 60*, Communications of the ACM, May 1960 pp. 299–314.

[Nievergelt65] Nievergelt, J., *On the Automatic Simplification of Computer Programs*, Communications of the ACM, June 1965, pp.366–370.

[Newell73] Newell, A., *Artificial Intelligence and the Concept of Mind*, in Computer Models of Thought and Language (Eds. Schank and Colby), W.H. Freeman, San Francisco, 1973, pp. 1–60.

[Quam72] Quam, L.H., and Diffie W., *Stanford LISP 1.6 Manual*, Stanford Artificial Intelligence Project Operating Note 28.7, Computer Science Department, Stanford University, 1972.

[Samet73] Samet, H., *Measurement of Time and Space Requirements for LISP Functions*, unpublished, 1973.

[Samet74] Samet, H., *Proving Equality and Inequality of Instances of Formulas in Linear Time on the Average*, unpublished, 1974.

[Smith70] Smith, D.C., *MLISP*, Stanford Artificial Intelligence Project Memo AIM–135, Computer Science Department, Stanford University, October 1970.

[Teitelman71] Teitelman, W., Bobrow, D.G., Hartley, A.K., Murphy, D.L., *BBN–LISP TENEX Reference Manual*, Bolt Beranek and Newman, July 1971.

[Teitelman74] Teitelman, W., *INTERLISP Reference Manual*, XEROX Palo Alto Research Center, Palo Alto, California, 1974.

[Tesler73] Tesler, L.G., Enea, H.J., and Smith, D.C., *The LISP70 Pattern Matching System*, Proceedings of the Third IJCAI, 1973, pp. 671–676.

[VanLehn73] VanLehn, K.(Ed), *SAIL User Manual*, Stanford Artificial Intelligence Project Memo AIM–373, Stanford University, July 1973.

[Van Wijngaarden69] Van Wijngaarden, (Ed.), Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A., *Report on the Algorithmic Language ALGOL 68*, Numerische Mathematik, 14:79–218, 1969.

[Wegbreit73] Wegbreit, B., *Procedure Closure in EL1*, TR 13–73, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, May 1973.

[Weyhrauch74] Weyhrauch, R.W., and Thomas, A.J., *FOL: a Proof Checker for First–order Logic*, Stanford Artificial Intelligence Project Memo AIM–235, Computer Science Department, Stanford University, September 1974.

[Winograd71] Winograd, T., *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*, MAC TR–84, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1971.

APPENDIX 1

LAP

LAP is the name given to the code that is generated by a LISP compiler and is processed by the LISP assembler. It is distinguished from regular PDP-10 assembly language by its format which is in the form of a list. The main items to note are given below.

1. (CALL N (E FNAME)) indicates a PUSHJ to FNAME with N arguments in accumulators 1–N (0≤N≤5).

2. (JCALL N (E FNAME)) indicates a JRST to FNAME with N arguments in accumulators 1–N (0≤N≤5).

3. (C 0 0 M N) indicates a word containing the constant M (0≤M≤15) in the right half and N (0≤N≤15) in the left half.

4. (OPCODE AC ADDR INDEX) is the form of all other instructions where INDEX is optional. OPCODE is a PDP-10 instruction optionally suffixed by @ which designates indirect addressing. The AC and INDEX fields contain numbers from 0 to decimal 15 or P which designates accumulator 14. ADDR may be a number, a label, a list such as (QUOTE S-expression) to reference list structures, or (SPECIAL variable-name) to indicate a cell containing the value of the SPECIAL variable variable-name.

APPENDIX 2

MACHINE DESCRIPTION PRIMITIVES

The following primitives are used in the description of the instructions presented in this thesis.

ADDX(INSTRUCTION) performs the same function as ADD. It is used as the multi–purpose addition routine so other instructions that require addition need not have separate addition mechanisms. See for example the PUSHJ instruction.

ADDSUB(OPERATION,OPERAND1,OPERAND2) performs
OPERAND1←OPERAND1 OPERATION OPERAND2 where OPERATION is
either ADD or SUB.

ALLOCATESTACKENTRY(LOC) allocate an entry on top of the stack pointed at by the stack pointer in LOC.

CONTENTS(ADDRESS) returns the contents of ADDRESS.

DEALLOCATESTACKENTRY(LOC) deallocate an entry from the stack pointed at by the stack pointer in LOC.

EFFECTADDRESS(INSTRUCTION) returns the value of the address field modified by the contents of an index register if indexing is specified by INSTRUCTION.

EXTENDZERO(HALFWORD) returns a word containing a data pointer of value 0 in the left half and HALFWORD in the right half.

FLAGSPOINTER() forms a data pointer containing the current value of the flags.

FORMRETURNADDRESS(ADDRESS) forms a data pointer whose contents is ADDRESS.

INDIRECT(ADDRESS) repeatedly performs indirect addressing on ADDRESS until no more is required.

LEFTCONTENTS(ADDRESS) returns the left half of the contents of ADDRESS.

LOADSTORE(ADDRESS,WORD) stores the value WORD in location ADDRESS.

LOADSTORELEFT(ADDRESS,HALFWORD) stores the value of halfword HALFWORD in the left half of location ADDRESS.

LOADSTORERIGHT(ADDRESS,HALFWORD) stores the value of halfword HALFWORD in the right half of location ADDRESS.

MAKEMASK(CONTENTS) forms a bit mask with zeros corresponding to all bits that are 1 in CONTENTS and NIL (denoting don't care) in all other positions.

MAKEPOINTER(LEFT,RIGHT) forms a word containing LEFT and RIGHT in its left and right halves respectively.

RIGHTCONTENTS(ADDRESS) returns the right half of the contents of ADDRESS.

SETMASKEDBITS(DESTINATION,MASK,NEWVALUE) uses MASK (i.e. the bits that are 1) to set the corresponding bits in DESTINATION to NEWVALUE.

SUBX(INSTRUCTION) performs the same function as SUB. It is used as the multi--purpose subtraction routine so other instructions that require addition need not have separate subtraction mechanisms. See for example the POPJ instruction.

APPENDIX 3

PDP–10 OPERATIONS

ADD The contents of the effective address is added to the contents of AC and the result is left in AC.

```
FEXPR ADD(ARGS);
ADDX(ARGS);

EXPR ADDX(ARGS);
LOADSTORE(ACFIELD(ARGS),
          ADDSUB(QUOTE(ADD),
                 CONTENTS(ACFIELD(ARGS)),
                 CONTENTS(EFFECTADDRESS(ARGS))));
```

CAIE The contents of AC is compared with the effective address and the next instruction is skipped if equality holds.

```
FEXPR CAIE(ARGS);
BEGIN
        NEW ACG,MEMG,TST;
        MEMG←EXTENDZERO(EFFECTADDRESS(ARGS));
        ACG←CONTENTS(ACFIELD(ARGS));
        TST←CHECKTEST(ACG,MEMG);
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                           ELSE NEXTINSTRUCTION());
        TRUEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION CAIETRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION CAIEFALSE);
        END;

FEXPR CAIETRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR CAIEFALSE(ARGS);
NEXTINSTRUCTION();
```

CAIN The contents of AC is compared with the effective address and the next instruction is skipped if equality does not hold.

```
FEXPR CAIN(ARGS);
BEGIN
        NEW ACG,MEMG,TST;
        MEMG←EXTENDZERO(EFFECTADDRESS(ARGS));
        ACG←CONTENTS(ACFIELD(ARGS));
        TST←CHECKTEST(ACG,MEMG);
        IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                           ELSE UNCONDITIONALSKIP());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION CAINTRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION CAINFALSE);
        END;

FEXPR CAINTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR CAINFALSE(ARGS);
NEXTINSTRUCTION();
```

CALL A special LAP instruction which is analogous to a PUSHJ. The difference is that it is used to invoke LISP functions via the property list. This is useful when a trace of the arguments to a function is desired or when the actual binding of a function changes.

```
FEXPR CALL(ARGS);
BEGIN
        ALLOCATESTACKENTRY(REGSTKPTR);
        ADDX(<REGSTKPTR,X11>);
        LOADSTORERIGHT(RIGHTCONTENTS(REGSTKPTR),FORMRETURNADDRESS(PCG));
        LOADSTORELEFT(RIGHTCONTENTS(REGSTKPTR),FLAGSPOINTER());
        UNCONDITIONALJUMP(CALLADDRESSFIELD(ARGS));
        END;
```

CAME The contents of AC is compared with the contents of the effective address and the next instruction is skipped if equality holds.

```
FEXPR CAME(ARGS);
BEGIN
        NEW ACG,ADDRESSG,MEMG,TST;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        ACG←CONTENTS(ACFIELD(ARGS));
        TST←CHECKTEST(ACG,MEMG);
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                           ELSE NEXTINSTRUCTION());
        TRUEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION CAMETRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION CAMEFALSE);
        END;

FEXPR CAMETRUE(ARGS);
UNCONDITIONALSKIP();
```

```
FEXPR CAMEFALSE(ARGS);
NEXTINSTRUCTION();
```

CAMN The contents of AC is compared with the contents of the effective address and the next instruction is skipped if equality does not hold.

```
FEXPR CAMN(ARGS);
BEGIN
        NEW ACG,ADDRESSG,MEMG,TST;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        ACG←CONTENTS(ACFIELD(ARGS));
        TST←CHECKTEST(ACG,MEMG);
        IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                              ELSE UNCONDITIONALSKIP());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION CAMNTRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION CAMNFALSE);
        END;

FEXPR CAMNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR CAMNFALSE(ARGS);
NEXTINSTRUCTION();
```

EXCH Exchange the contents of the effective address with the contents of AC.

```
FEXPR EXCH(ARGS);
BEGIN
        NEW ACG,ADDRESSG;
        ADDRESSG←EFFECTADDRESS(ARGS);
        ACG←CONTENTS(ACFIELD(ARGS));
        LOADSTORE(ACFIELD(ARGS),CONTENTS(ADDRESSG));
        LOADSTORE(ADDRESSG,ACG);
        END;
```

HLRZ Load the right half of AC with the left half of the contents of the effective address and clear the left half of AC.

```
FEXPR HLRZ(ARGS);
LOADSTORE(ACFIELD(ARGS),EXTENDZERO(LEFTCONTENTS(EFFECTADDRESS(ARGS))));
```

HLRZ@ Same as HLRZ with indirect addressing.

```
FEXPR HLRZ@(ARGS);
LOADSTORE(ACFIELD(ARGS),
          EXTENDZERO(LEFTCONTENTS(INDIRECT(CONTENTS(EFFECTADDRESS(ARGS))))));
```

HLRZS Move the left half of the contents of the effective address to the right half of the effective address and clear the left half of the effective address. If the AC field specification is non–zero, then AC is also loaded with the same value as was placed in the effective address.

```
FEXPR HLRZS(ARGS);
BEGIN
        NEW ADDRESS,CONTNTS;
        ADDRESS←EFFECTADDRESS(ARGS);
        CONTNTS←EXTENDZERO(LEFTCONTENTS(ADDRESS));
        LOADSTORE(ACFIELD(ARGS),CONTNTS);
        LOADSTORE(ADDRESS,CONTNTS);
        END;
```

HLRZS@ Same as HLRZS with indirect addressing.

```
FEXPR HLRZS@(ARGS);
BEGIN
        NEW ADDRESS,CONTNTS;
        ADDRESS←INDIRECT(CONTENTS(EFFECTADDRESS(ARGS)));
        CONTNTS←EXTENDZERO(LEFTCONTENTS(ADDRESS));
        LOADSTORE(ACFIELD(ARGS),CONTNTS);
        LOADSTORE(ADDRESS,CONTNTS);
        END;
```

HRLM Store the right half of AC in the left half of the effective address.

```
FEXPR HRLM(ARGS);
LOADSTORELEFT(EFFECTADDRESS(ARGS),RIGHTCONTENTS(ACFIELD(ARGS)));
```

HRLM@ Same as HRLM with indirect addressing.

```
FEXPR HRLM@(ARGS);
LOADSTORELEFT(INDIRECT(CONTENTS(EFFECTADDRESS(ARGS))),
              RIGHTCONTENTS(ACFIELD(ARGS)));
```

HRRM Store the right half of AC in the left half of the effective address.

```
FEXPR HRRM(ARGS);
LOADSTORERIGHT(EFFECTADDRESS(ARGS),RIGHTCONTENTS(ACFIELD(ARGS)));
```

HRRM@ Same as HRRM with indirect addressing.

```
FEXPR HRRM@(ARGS);
LOADSTORERIGHT(INDIRECT(CONTENTS(EFFECTADDRESS(ARGS))),
               RIGHTCONTENTS(ACFIELD(ARGS)));
```

HRRZ Load the right half of AC with the right half of the contents of the effective address and clear the left half of AC.

```
FEXPR HRRZ(ARGS);
LOADSTORE(ACFIELD(ARGS),EXTENDZERO(RIGHTCONTENTS(EFFECTADDRESS(ARGS))));
```

HRRZ@ Same as HRRZ with indirect addressing.

```
FEXPR HRRZ@(ARGS);
LOADSTORE(ACFIELD(ARGS),
          EXTENDZERO(RIGHTCONTENTS(INDIRECT(CONTENTS(EFFECTADDRESS(ARGS))))));
```

HRRZS Move the right half of the contents of the effective address to the right half of the effective address and clear the left half of the effective address. If the AC field specification is non–zero, then AC is also loaded with the same value as was placed in the effective address.

```
FEXPR HRRZS(ARGS);
BEGIN
        NEW ADDRESS,CONTNTS;
        ADDRESS←EFFECTADDRESS(ARGS);
        CONTNTS←EXTENDZERO(RIGHTCONTENTS(ADDRESS));
        LOADSTORE(ACFIELD(ARGS),CONTNTS);
        LOADSTORE(ADDRESS,CONTNTS);
        END;
```

HRRZS@ Same as HRRZS with indirect addressing.

```
FEXPR HRRZS@(ARGS);
BEGIN
        NEW ADDRESS,CONTNTS;
        ADDRESS←INDIRECT(CONTENTS(EFFECTADDRESS(ARGS)));
        CONTNTS←EXTENDZERO(RIGHTCONTENTS(ADDRESS));
        LOADSTORE(ACFIELD(ARGS),CONTNTS);
        LOADSTORE(ADDRESS,CONTNTS);
        END;
```

JCALL A special LAP instruction which is analogous to a JRST. The difference is that it is used to invoke LISP functions via the property list. This is useful when a trace of the arguments to a function is desired or when the actual binding of a function changes.

```
FEXPR JCALL(ARGS);
UNCONDITIONALJUMP(CALLADDRESSFIELD(ARGS));
```

JRST Unconditional jump to the effective address.

```
FEXPR JRST(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));
```

JUMPE Jump to the effective address if the contents of AC is zero; otherwise continue execution at the next instruction.

```
FEXPR JUMPE(ARGS);
BEGIN
        NEW TST;
        TST←CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALJUMP(EFFECTADDRESS(ARGS))
                           ELSE NEXTINSTRUCTION());
        TRUEPREDICATE();
        CONDITIONALJUMP(ARGS,FUNCTION JUMPETRUE);
        JUMPALTERNATIVE(ARGS,FUNCTION JUMPEFALSE);
        END;

FEXPR JUMPETRUE(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

FEXPR JUMPEFALSE(ARGS);
NEXTINSTRUCTION();
```

JUMPN Jump to the effective address if the contents of AC is unequal to zero; otherwise continue execution at the next instruction.

```
FEXPR JUMPN(ARGS);
BEGIN
        NEW TST;
        TST←CHECKTEST(CONTENTS(ACFIELD(ARGS)),ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                           ELSE UNCONDITIONALJUMP(EFFECTADDRESS(ARGS)));
        FALSEPREDICATE();
        CONDITIONALJUMP(ARGS,FUNCTION JUMPNTRUE);
        JUMPALTERNATIVE(ARGS,FUNCTION JUMPNFALSE);
        END;

FEXPR JUMPNTRUE(ARGS);
UNCONDITIONALJUMP(EFFECTADDRESS(ARGS));

FEXPR JUMPNFALSE(ARGS);
NEXTINSTRUCTION();
```

MOVE Load AC with the contents of the effective address.

```
FEXPR MOVE(ARGS);
LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));
```

MOVEI Load the right half of AC with the effective address, and clear the left half.

```
FEXPR MOVEI(ARGS);
LOADSTORE(ACFIELD(ARGS),EXTENDZERO(EFFECTADDRESS(ARGS)));
```

MOVEMStore the contents of AC into the location indicated by the effective address.

```
FEXPR MOVEM(ARGS);
LOADSTORE(EFFECTADDRESS(ARGS),CONTENTS(ACFIELD(ARGS)));
```

MOVS Load the left half of AC with the contents of the right half of the effective address and load the right half of AC with the contents of the left half of the effective address.

```
FEXPR MOVS(ARGS);
LAMBDA(CONTNTS);
LOADSTORE(ACFIELD(ARGS),MAKEPOINTER(RIGHTHALF(CONTNTS),LEFTHALF(CONTNTS)));
(CONTENTS(EFFECTADDRESS(ARGS)));
```

POP Move the contents of the location addressed by the right half of AC to the effective address and then subtract octal 1 000 001 from AC to decrement both halves by one. If the subtraction causes the count in the left half of AC to reach −1, then the Pushdown Overflow flag is set.

```
FEXPR POP(ARGS);
BEGIN
        LOADSTORE(EFFECTADDRESS(ARGS),CONTENTS(RIGHTCONTENTS(ACFIELD(ARGS))));
        DEALLOCATESTACKENTRY(ACFIELD(ARGS));
        SUBX(<ACFIELD(ARGS),X11>);
        END;
```

POPJ Subtract octal 1 000 001 from AC to decrement both halves by one and place the result back in AC. If subtraction causes the count in the left half of AC to reach −1, then set the Pushdown Overflow flag. The next instruction is taken from the location addressed by the right half of the location that was addressed by AC right prior to decrementing.

```
FEXPR POPJ(ARGS);
BEGIN
        NEW LAB;
        LAB←RIGHTCONTENTS(RIGHTCONTENTS(ACFIELD(ARGS)));
        DEALLOCATESTACKENTRY(ACFIELD(ARGS));
        SUBX(<ACFIELD(ARGS),X11>);
        UNCONDITIONALJUMP(LAB);
        END;
```

PUSH Add octal 1 000 001 to AC to increment both halves by one and then move the contents of the effective address to the location now addressed by the right half of AC. If the addition causes the count in the left half of AC to reach zero, then the Pushdown Overflow flag is set.

```
FEXPR PUSH(ARGS);
BEGIN
        NEW MEM;
        MEM←CONTENTS(EFFECTADDRESS(ARGS));
        ALLOCATESTACKENTRY(ACFIELD(ARGS));
        ADDX(<ACFIELD(ARGS),X11>);
        LOADSTORE(RIGHTCONTENTS(ACFIELD(ARGS)),MEM);
        END;
```

PUSHJ Add octal 1 000 001 to AC to increment both halves by one and place the result back in AC. If addition causes the count in the left half of AC to reach zero, then set the Pushdown Overflow flag. Store the contents of the PC (Program Counter) in the location now addressed by the right half of AC and continue execution at the effective address.

```
FEXPR PUSHJ(ARGS);
BEGIN
        NEW ADDRESS;
        ADDRESS←EFFECTADDRESS(ARGS);
        ALLOCATESTACKENTRY(ACFIELD(ARGS));
        ADDX(<ACFIELD(ARGS),X11>);
        LOADSTORERIGHT(RIGHTCONTENTS(ACFIELD(ARGS)),FORMRETURNADDRESS(PCG));
        LOADSTORELEFT(RIGHTCONTENTS(ACFIELD(ARGS)),FLAGSPOINTER());
        UNCONDITIONALJUMP(ADDRESS);
        END;
```

SKIPA Skip the next instruction. If the AC field specification is non–zero, then load AC with the contents of the effective address.

```
FEXPR SKIPA(ARGS);
BEGIN
        IF ACFIELD(ARGS) NEQ 0 THEN
            LOADSTORE(ACFIELD(ARGS),CONTENTS(EFFECTADDRESS(ARGS)));
        UNCONDITIONALSKIP();
        END;
```

SKIPE Skip the next instruction if the contents of the effective address is equal to zero. If the AC field specification is non–zero, then load AC with the contents of the effective address.

```
FEXPR SKIPE(ARGS);
BEGIN
        NEW ADDRESSG,MEMG,TST;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        IF ACFIELD(ARGS) NEQ 0 THEN LOADSTORE(ACFIELD(ARGS),MEMG);
        TST←CHECKTEST(MEMG,ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                           ELSE NEXTINSTRUCTION());
        TRUEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION SKIPETRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION SKIPEFALSE);
        END;

FEXPR SKIPETRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR SKIPEFALSE(ARGS);
NEXTINSTRUCTION();
```

SKIPN Skip the next instruction if the contents of the effective address is not equal to zero. If the AC field specification is non–zero, then load AC with the contents of the effective address.

```
FEXPR SKIPN(ARGS);
BEGIN
        NEW ADDRESSG,MEMG,TST;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        IF ACFIELD(ARGS) NEQ 0 THEN LOADSTORE(ACFIELD(ARGS),MEMG);
        TST←CHECKTEST(MEMG,ZEROCNST);
        IF TST THEN RETURN(IF CDR TST THEN NEXTINSTRUCTION()
                           ELSE UNCONDITIONALSKIP());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,FUNCTION SKIPNTRUE);
        SKIPALTERNATIVE(ARGS,FUNCTION SKIPNFALSE);
        END;

FEXPR SKIPNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR SKIPNFALSE(ARGS);
NEXTINSTRUCTION();
```

SUB The contents of the effective address is subtracted from the contents of AC and the result is left in AC.

```
FEXPR SUB(ARGS);
SUBX(ARGS);
```

```
EXPR SUBX(ARGS);
LOADSTORE(ACFIELD(ARGS),
          ADDSUB(QUOTE(SUB),
                 CONTENTS(ACFIELD(ARGS)),
                 CONTENTS(EFFECTADDRESS(ARGS))));
```

TDZA Zero the bits in AC corresponding to the bits that are 1 in the contents of the effective address and skip the next instruction.

```
FEXPR TDZA(ARGS);
BEGIN
        NEW ACG,ADDRESSG,MEMG;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        ACG←CONTENTS(ACFIELD(ARGS));
        LOADSTORE(ACFIELD(ARGS),SETMASKEDBITS(ACG,MEMG,0));
        UNCONDITIONALSKIP();
        END;
```

TDZN Zero the bits in AC corresponding to the bits that are 1 in the contents of the effective address and skip the next instruction if any of the bits were one. Regardless of the outcome of the test, AC is set to zero. However, the next instruction is skipped only if the AC was originally non–zero.

```
FEXPR TDZN(ARGS);
BEGIN
        NEW ACG,ADDRESSG,MEMG,TST;
        ADDRESSG←EFFECTADDRESS(ARGS);
        MEMG←CONTENTS(ADDRESSG);
        ACG←CONTENTS(ACFIELD(ARGS));
        LOADSTORE(ACFIELD(ARGS),SETMASKEDBITS(ACG,MEMG,0));
        TST←CHECKTEST(ACG,MAKEMASK(MEMG));
        IF TST THEN RETURN(IF CDR TST THEN UNCONDITIONALSKIP()
                                  ELSE NEXTINSTRUCTION());
        FALSEPREDICATE();
        CONDITIONALSKIP(ARGS,TDZNTRUE);
        SKIPALTERNATIVE(ARGS,TDZNFALSE);
        END;

FEXPR TDZNTRUE(ARGS);
UNCONDITIONALSKIP();

FEXPR TDZNFALSE(ARGS);
NEXTINSTRUCTION();
```

APPENDIX 4

DETECTABLE ERRORS

Presently the following errors are detected during the rederivation process. These errors generally pertain to well-formedness restrictions. In addition to the error message, if the error involves a location, then the user is given the storage history corresponding to the location.

In the listing of the error messages we use angled brackets to denote parameters to the message.

1. ILLEGAL STACK POINTER FORMAT
2. <LABEL> IS AN ILLEGAL BACKWARD LABEL SINCE ALL ENTRY PATHS MUST BE OF A UNIFORM STACK DEPTH
3. JUMPING TO <LABEL> WHICH IS OUTSIDE OF THE CURRENT FUNCTION AND NOT DECLARED TO BE ENTERABLE IN SUCH A MANNER HAS NOT BEEN IMPLEMENTED YET
4. OFFLINE SETQ ARGUMENT MUST BE A SPECIAL VARIABLE
5. SPECIAL VARIABLE <SPECIAL VARIABLE NAME> HAS A NON ZERO LEFT HALF
6. ACCUMULATOR <ACCUMULATOR NUMBER> HAS A NON ZERO LEFT HALF
7. PARAMETER <PARAMETER NAME> MUST BE A LISP POINTER
8. LEFT HALF OF ACCUMULATOR CONTAINING PARAMETER <PARAMETER NAME> MUST BE 0
9. ATTEMPTING TO RETRIEVE A RETURN ADDRESS FROM BELOW THE BOTTOM OF THE STACK
10. LEFT HALF OF THE RETURN ADDRESS ON THE STACK MUST BE ZERO OR CONTAIN THE APPROPRIATE FLAGS
11. RETURN ADDRESS ON THE STACK MUST BE A LABEL
12. SPECIAL VARIABLE <SPECIAL VARIABLE NAME> IS NOT A LISP POINTER
13. PARTFLAG IN TROUBLE NULL PAIR PASS 2 ERROR SEE HJS
14. ALL LABELS SHOULD HAVE BEEN DETERMINED DURING PASS 1
15. SETQ TO AN UNDECLARED SPECIAL VARIABLE
16. CANNOT STORE INTO A LISP POINTER IN LINE WITHOUT MAKING SURE THAT IT IS NOT AN ATOM
17. VALUES STORED INTO LISP POINTERS MUST ALSO BE LISP POINTERS
18. ILLEGAL DESTINATION FOR STORAGE OPERATION
19. NO REDUNDANT PATH CAN BE FOUND FOR THE BACKWARD JUMP AT THIS INSTRUCTION

20. THE LEFT HALF OF ACCUMULATOR 1 CONTAINING THE RESULT OF THE FUNCTION MUST BE 0

21. <HALF NAME> HALF OF STACK POINTER NOT THE SAME AS UPON FUNCTION ENTRY

22. SPECIAL VARIABLE <SPECIAL VARIABLE NAME> DOES NOT HAVE THE SAME VALUE UPON FUNCTION EXIT AS IT HAD UPON FUNCTION ENTRY

23. CAN NOT SELECTIVELY TEST BITS OF A LISP POINTER

24. BIT COMPARISON NOT AVAILABLE FOR <TYPE NAME>

25. MISMATCH OF TYPES <TYPE NAME> AND <TYPE NAME> FOR CHECKTEST

26. ONLY BIT VALUES OF 0 OR 1 MAY BE USED IN A MASK

27. SETTING BITS BETWEEN TYPES <TYPE NAME> AND <TYPE NAME> HAS NOT BEEN IMPLEMENTED YET

28. EQUALITY OF DIFFERENT TYPES IS UNKNOWN

29. STACKPOINTER CAN NOT REFER TO LOCATIONS ABOVE THE HIGHEST POINT THAT HAS BEEN ALLOCATED BY A PUSH OR PUSHJ INSTRUCTION

30. CAN NOT DEALLOCATE THE STACK BELOW ITS BOTTOM

31. INVALID <HALF NAME> HALF OF THE STACK POINTER

32. RESULT OF THE FUNCTION MUST BE A LISP POINTER

APPENDIX 5

RELOCATION ARITHMETIC

In Chapter 4 we mentioned that limited arithmetic (addition and subtraction) between different data types can be recognized. The primary purpose of this feature is to enable the handling of operations necessary for computing addresses. In the discussion the term constant is used to denote an integer number.

When arithmetic is performed on full words, the operation is first performed on the right halves of the corresponding words. If any carry or borrow terms occur, then a flag is set. The operation is next performed on the left halves of the words and the presence of a carry or borrow term is taken into account. As noted earlier, the operations are performed using two's complement arithmetic.

Zero may be added to, or subtracted from, any data type. The result is the nonzero operand. The zero may be in the form of a LISP pointer of value NIL.

Addition is only allowed when one of the operands is a number, and the remaining operand is either a stack pointer, address, stack size pointer, or a number. The previous also hold for subtraction. In addition, subtraction is permitted when the operands are both of the same data type where the data type is either a stack pointer, address, stack size pointer, or a number.

If both operands to a subtraction operation are LISP pointers then special treatment is made to allow for the possibility that the result of the subtraction could possibly be used in a test against zero. Recall from Chapter 4 that whenever a subtraction operation involves two LISP pointers, an EQSUBI construct is used. Moreover, if the two LISP pointers reside in the right half of the word, and the left halves of the words originally contained zero, then in addition to making the result of the operation in the right half EQSUBI, we also make the result of the operation in the left half EQSUBD. This is done in order to indicate the dependence of the contents of the left half on the contents of the right half in case a borrow term was necessary to carry out the subtraction procedure. The significance lies in the situation when the contents of the right half is known to be zero. In this case the left half would also be zero. Recall [illegible] primary reason for the EQSUBD construct is to indicate that a borrow might have been necessary during the subtraction process.

APPENDIX 6

SYSTEM USER MANUAL

The system has been implemented and is up and running on the PDP–10 at the Stanford Artificial Intelligence Project. Another version has been constructed to run using the TENEX operating system with ILISP[Bobrow72] at the Institute for Mathematical Studies in the Social Sciences (IMSSS) at Stanford. The adaptation was made by Tom Wolpert. The following manual pertains to the implementation at the Stanford Artificial Intelligence Project.

A6.A System Overview

The input to the system is a pair of files. The first contains a set of LISP functions, while the second contains a LAP program (or programs). The LISP file must contain the encoding of the function to be verified and all functions called by this function which are not predefined in LISP. The LAP file need only contain the encoding of the function to be verified.

The LISP file must also contain at its beginning declarations for all the SPECIAL (i.e. free or global) variables read or modified by the function to be verified or by functions invoked directly or indirectly by this function. For example to declare variable A to be SPECIAL the following statement is used:

```
(DEFPROP A T SPECIAL)
```

If a function is used about which no information is known to the system, then a set of questions is posed to the user to which he is asked to respond. These questions are of the nature of flowanalysis, global variables read or modified by the function, list modification, etc. If trouble persists or you feel that the system is unjustly unaware of any specific functions then complain to the author.

PROGS are not currently allowed. The main stumbling point is that the GO construct has not been implemented. However, it is clear that any PROG without a GO can be rewritten as an EXPR and in fact this is what should be done in order to prepare a PROG to be input to the system. Note that no changes need be made to the LAP file.

There are several restrictions made on the object and source programs:

1. Only EXPRS are handled (i.e. no FEXPRS).
2. A function may have at most 5 arguments.
3. The arguments appear in accumulators 1–5.
4. A function returns its result in accumulator 1.
5. The only memory locations which may be used are the accumulators, the push down stack, SPECIAL variable VALUE cells, and locations within the program.
6. The constructs SETQ, RPLACA, RPLACD, and internal LAMBDAs (i.e. LAMBDA expressions used for temporary values – e.g. common subexpressions) are implemented.
7. The SET command as well as the LAMBDA feature used in the context of a function definition within a function is not allowed.
8. Use F instead of NIL in the input LISP program.

9. Make sure that all numbers are quoted – i.e. (QUOTE 1) instead of 1.
10. Do not use the character "←" in the input LISP program.
11. Special (i.e. free) variables may be used. However, they must be global to the entire set of functions being defined.

Not all of the instructions of the PDP–10 have been implemented in the system. Thus if you get a message such as "OPCODE UNKNOWN – COMPLAIN TO HJS", then do so since missing opcodes can generally be added with little difficulty. Currently the following instructions can be used. An "@" symbol following an opcode designates that indirect addressing may be used with the opcode.

OPCODES: ADD, CAIE, CAIN, CALL, CAME, CAMN, EXCH, HLRZ, HLRZ@, HLRZS, HLRZS@, HRLM, HRLM@, HRRM, HRRM@, HRRZ, HRRZ@, HRRZS, HRRZS@, JCALL, JRST, JUMPE, JUMPN, MOVE, MOVEI, MOVEM, MOVS, POP, POPJ, PUSH, PUSHJ, SKIPA, SKIPE, SKIPN, SUB, TDZA, TDZN.

A6.B Using the System

The system is run with the command DO PROOF[L,HJS].

Once this command is given you simply respond to questions posed by the system. A response is called for whenever the system outputs two stars – i.e. ** . Note that while processing a proof the system creates a file named COMMON.XYZ to pass information between its two phases (see below). Therefore, the user should make sure that he has no such file on his disk area or he will lose the previous file since the system makes no explicit check.

The following point is only of importance if a system crash occurs and you do not wish to start over. The basic proof system is broken up into two parts. The first part analyzes the LAP program and produces a representation for it in an intermediate form. The actual proof is performed in the second part of the system which uses the results of the first part by reading the file COMMON.XYZ . If you have finished the first part and do not wish to repeat it, then simply start up the system with the command DO PROOF(2)[L,HJS] . As an aid in detecting the feasibility of such a step the system will type out on your terminal the string:

PART 2 OF PROOF STARTED

A typical terminal session proceeds as follows:

"." in the extreme left indicates monitor level command.
"**" in the extreme left indicates that the proof system is waiting for a response to one of its questions.
"*" in the extreme left indicates that you are talking to LISP.
"%" indicates the start and end of a comment further explaining a command.
text not preceded by any of the above indicates system typeout such as a description of the input requested.

File names are specified in the following ways:

Talking to the monitor	Talking to PROOF
FOO	FOO
FOO.BAZ	FOO.BAZ)
FOO.BAZ[LSP,YOU]	((LSP,YOU)(FOO.BAZ))

```
.DO PROOF[L,HJS]

↑C                                                % SYSTEM OUTPUT %
.
8                                                 % SYSTEM OUTPUT %

(ENDPROG 7236)                                    % SYSTEM OUTPUT %
(NORELEASEEXIT 7238)                              % SYSTEM OUTPUT %
FINISHED-LOADING                                  % SYSTEM OUTPUT %

ENTER FILENAME CONTAINING LAP PROGRAM TO BE REDERIVED
**(REVPRE.LAP)

IF ACCUMULATOR P IS NOT OCTAL 14 THEN ENTER ITS OCTAL EQUIVALENT
OTHERWISE ENTER NIL
**NIL

ENTER NAME OF FUNCTION TO BE REDERIVED
**REVERSE1A


***

WARNING:  DO NOT USE THE ← CHARACTER IN THE INPUT PROGRAM

WARNING:  USE F INSTEAD OF NIL IN THE INPUT PROGRAM

WARNING:  THE SYSTEM WILL CREATE A FILE NAMED COMMON.XYZ ON YOUR DISK
 AREA UNLESS YOU STOP THE PROGRAM NOW

WARNING:  ALL NUMBERS MUST BE QUOTED - I.E. (QUOTE 1) INSTEAD OF 1

***


ENTER FILENAME CONTAINING LISP ENCODING OF THE FUNCTION TO BE
REDERIVED AND THE FUNCTIONS INVOKED BY IT
**REVPRE

THE ONLY KNOWN ANTISYMMETRIC FUNCTIONS ARE THE PAIRS CONS XCONS
AND *LESS *GREAT.  ENTER A LIST OF ANY OTHER PAIRS OR NIL
**NIL

THE ONLY KNOWN COMMUTATIVE FUNCTIONS ARE EQ EQUAL *PLUS AND
*TIMES.  ENTER A LIST OF OTHERS OR NIL
```

```
**NIL

ONLY FUNCTIONS KNOWN NOT TO DESTROY ALL ACCUMULATORS ARE CONS
XCONS NCONS AND ATOM.  ENTER ANY OTHERS AND THE NUMBER OF THE
HIGHEST ACCUMULATOR THAT THEY DESTROY IN A DOTTED PAIR FORMAT.
OTHERWISE ENTER NIL
**NIL

ENTER A LIST OF FUNCTIONS WHICH CAN BE ACTIVATED WITHOUT THE
CALL MECHANISM.  BE SURE TO ENTER THE NAME OF THE FUNCTION BEING
REDERIVED IF APPLICABLE.  OTHERWISE ENTER NIL
**NIL

ENTER A LIST OF FUNCTIONS WHICH MUST BE ACTIVATED BY THE CALL
MECHANISM.  DON'T INCLUDE CONS, XCONS, NCONS, ATOM, EQ, NOT,
NULL, CAR, AND CDR WHICH ARE ALREADY KNOWN.
OTHERWISE ENTER NIL
**NIL

ENTER FILENAME FOR DEBUGGING AND PROOF INFORMATION
**(REVPRE.PRF)                                % ERRORS AND DEBUGGING      %
                                              % INFORMATION CAN BE        %
                                              % FOUND HERE                %
TRACE OF THE PROOF DESIRED?   TYPE Y OR N     % ONLY FOR HARDY SOULS      %
**N                                           % FOR AN INTERPRETATION     %
                                              % CONTACT                   %
                                              % HANAN SAMET @ EXT 7-4971  %
DO PROOF(2)[L,HJS]                                     % SYSTEM OUTPUT %


EXIT                                                   % SYSTEM OUTPUT %
↑C                                                     % SYSTEM OUTPUT %
..                                                     % SYSTEM OUTPUT %
↑C                                                     % SYSTEM OUTPUT %
..                                                     % SYSTEM OUTPUT %

8                                                      % SYSTEM OUTPUT %

(ENDPROG 7247)                                         % SYSTEM OUTPUT %
(NORELEASEEXIT 7249)                                   % SYSTEM OUTPUT %
FINISHED-LOADING                                       % SYSTEM OUTPUT %

NIL                                                    % SYSTEM OUTPUT %

PART 2 OF PROOF STARTED
SUCCESSFUL MATCH                                   % EQUIVALENCE HOLDS %
↑C                                                     % SYSTEM OUTPUT %
```

APPENDIX 7

INSTRUCTION EXECUTION TIMES

Instruction	Basic Speed	Effective Address is an Accumulator
ADD	2.75	2.41
CAIE	1.79	
CAIN	1.79	
CALL	3.11	
CAME	2.75	2.41
CAMN	2.75	2.41
EXCH	3.01	2.32
HLRZ	2.43	2.09
HLRZ@	3.55	3.21
HLRZS	2.87	2.44
HLRZS@	3.99	3.56
HRLM	3.01	2.58
HRLM@	4.13	3.70
HRRM	3.01	2.58
HRRM@	4.13	3.70
HRRZ	2.43	2.09
HRRZ@	3.55	3.21
HRRZS	2.87	2.44
HRRZS@	3.99	3.56
JCALL	1.47	
JRST	1.47	
JUMPE	1.79	
JUMPN	1.79	
MOVE	2.43	2.09
MOVEI	1.47	
MOVEM	2.58	2.23
MOVS	2.43	2.09
POP	4.15	3.80
POPJ	3.18	
PUSH	4.07	3.73
PUSHJ	3.11	
SKIPA	2.61	2.27
SKIPE	2.61	2.27
SKIPN	2.61	2.27
SUB	2.75	2.41
TDZA	2.92	2.58
TDZN	2.92	2.58

APPENDIX 8

DEPTH FIRST NUMBERING ALGORITHM

The algorithm for converting a breadth first numbering representation to a depth first representation which was alluded to in Section 3.E is given below. In the algorithm, we assume that functions are represented in infix notation in the form of lists. LIST[i] denotes the i'th element of LIST. <A,B,C> denotes the list consisting of elements A, B, and C. The algorithm is encoded using the procedure renumber which has as its parameters the FORM being renumbered, a number MAXSEEN denoting the highest computation number that has been encountered, and a list of pairs of numbers REASSIGN. The first element of each pair in REASSIGN corresponds to the highest computation number that has been encountered, and the second element of each pair in REASSIGN is its corresponding reassigned number. The list REASSIGN is sorted in decreasing order and is updated as each condition is processed. Furthermore, there are two global variables MAXSEENTEMP and MAXASSIGN. MAXSEENTEMP and MAXASSIGN are used by procedure renumber function which performs the actual renumbering. MAXSEENTEMP keeps track of the highest number encountered while processing the form serving as the argument to renumber function, and MAXASSIGN keeps track of the highest computation number that has been used in the process of renumbering the input form. Note the use of the operators PREDICATE, CONCLUSION, and ALTERNATIVE to obtain the relevant components of a conditional form. Procedure renumber is initially activated with the call renumber(form,0,<(0,0)>), and both global variables MAXSEENTEMP and MAXASSIGN are initially 0.

```
list procedure renumber(list FORM; integer MAXSEEN; list REASSIGN);
begin
  if conditional_form(FORM) then
     make_conditional_form(renumber_function(PREDICATE(FORM),
                                             REASSIGN),
                           renumber(CONCLUSION(FORM),
                                    MAXSEEN←MAXSEENTEMP,
                                    add_to_list((MAXSEEN,MAXASSIGN),
                                                REASSIGN)),
                           renumber(ALTERNATIVE(FORM),
                                    MAXSEENTEMP←MAXSEEN,
                                    add_to_list((MAXSEEN,MAXASSIGN),
                                                REASSIGN))))
  else renumber_function(FORM,REASSIGN);
  end;
```

Procedure make conditional form creates a conditional form given a condition, conclusion, and an alternative clause. Procedure renumber function is best described verbally as follows:

(1) For each number, say NUM, appearing in the form FORM search REASSIGN for the first pair having its first entry less than or equal to NUM, say (MAXS,MAXA). Replace NUM by MAXA+NUM−MAXS. In addition, if NUM is greater than MAXSEENTEMP, then update MAXSEENTEMP (e.g. MAXSEENTEMP←NUM).

(2) Update MAXASSIGN by using the first pair in REASSIGN, say (MAXS,MAXA) to yield MAXASSIGN←MAXA+MAXSEENTEMP−MAXS

Using the most recent value of MAXASSIGN to form REASSIGN in the second recursive activation of procedure renumber insures that computations performed in the alternative clause of the conditional form will have higher computation numbers associated with them than with those performed in the conclusion. Furthermore step (1) in procedure renumber_function makes sure that the renumbering in the alternative clause will only occur for the computations that are solely performed in the alternative clause. Note that when a condition or conclusion clause are being processed, the pairs in REASSIGN have identical components.