# PROVING THEOREMS ABOUT LISP FUNCTIONS

by

Robert S. Boyer and J Strother Moore

Department of Computational Logic,
University of Edinburgh, Scotland.

## ABSTRACT

We describe some simple heuristics combining evaluation and mathematical induction which we have implemented in a program that automatically proves a wide variety of theorems about recursive LISP functions. The method the program uses to generate induction formulas is described at length. The theorems proved by the program include that REVERSE is its own inverse and that a particular SORT program is correct. Appendix B contains a list of the theorems proved by the program.

## KEY WORDS

LISP, automatic theorem proving, structural induction, proving programs correct.

## INTRODUCTION

We are concerned with proving theorems in a first-order theory of lists, akin to the elementary theory of numbers. We use a subset of LISP as our language because recursive list processing functions are easy to write in LISP and because theorems can be naturally stated in LISP; furthermore, LISP has a simple syntax and is universal in Artificial Intelligence. We employ a LISP interpreter to 'run' our theorems and a heuristic which produces induction formulas from information about how the interpreter fails. We combine with the induction heuristic a set of simple rewrite rules of LISP and a heuristic for generalizing the theorem being proved.

Our program accepts as input a LJSP expression, e.g.,

(EQUAL (REVERSE (REVERSE A)) A),

possibly involving skolem constants (e.g., A, B and C throughout this paper) which stand for universally quantified variables ranging over all lists. The program attempts to show that the value of the input expression is always equal to T {whenever the skolem constants are replaced by arbitrary lists). Theorems we have proved automatically include:

```
(EQUAL (REVERSE (REVERSE A)) A)
(IMPLIES (OR (MEMBER A B) (MEMBER A C))
   (MEMBER A (UNION B c)))
```

and

```
[ORDERED (SORT A))
```

where EQUAL is a primitive function (i.e. built into the theorem prover) but REVERSE, IMPLIES, OR, MEMBER, UNION, ORDERED, and SORT are defined by the user of the program. The program uses only its knowledge of the LISP primitives and the LISP definitions supplied by the uBer. No further information is required of the user.

This paper describes many aspects of the program in brevity. A thorough presentation is forthcoming in part II of Moore*'a: thesis.

## OUR LISP SUBSET

We use a subset of pure LISP which has as primitives NIL, CONS, CAR, CDR, COND, and EQUAL. We do no.t prove theorems about functions that involve side effects, RPLACA, QUOTE, or LABEL. We use lists of NIL to represent natural numbers: 0 is NIL, 1 *is (CONS* Nil NIL), and ADP1 is defined as;

```
(LAMBDA (X) (CONS NIL X)) .
```

(Our arithmetic is thus a version of Peano successor arithmetic.)

Our equality privitive is EQUAL rather than EQ. Our COND primitive takes three arguments (for simplicity without loss of power). (COND A B C) in our system is (COND (A B) (T C)) in more traditional LISP systems.

The userof the theorem prover supplies function definitions almost exactly as in LISP, with the function DEFINE. For example,

```
DEFINE ( (APPEND {LAMBDA (X Y)
                         (COND X
                               (CONS (CAR X)
                                 (APPEND (CDR X) Y) )J
                               Y ) »  ).
```

## EVAL

Our LISP interpreter, EVAL, is similar in many ways to a normal LISP interpreter; EVAL applies function definitions and handles primitives like COND. EVAL is recursive; it evaluates arguments before applying and evaluating function definitions. Our EVAL has special provisions for handling skolem constants and terms in which they appear. The following examples illustrate the behaviour of our EVAL:

```
EVAL( NIL ) - NIL
EVAL { A ) = A
EVAL( (CONS A B) ) = (CONS A B)
EVAL( (CAR (CONS A B)) ) = A
EVAL  (CDR (CAR (CONS A B)J) ) = (CDR A)
EVAL( (COND (CONS A B) C D) ) = C
EVAL( (APPEND (CONS A B) C) ) = (CONS A
                                   (APPEND B C))
```

The last example is justified because regardless of the values of A, B and C, the first argument to APPEND is not NIL, so that the COND in the definition of APPEND can be evaluated.

EVAL tries to evaluate (APPEND B C) further but fails because it 'recurses into[1] the skolem constant B. (See the definition of APPEND above.)

When evaluating the form (foo ti ... $t_n$) where foo is a defined function, we recursively evaluate the arguments first. Call the value of $t_{i(}$ tj. EVAL binds the formal variables of foo to their values, tj_, and then evaluates the definition of foo. If a recursive call of foo is encountered in this function body the arguments are evaluated aa usual. Then, if one of the evaluated arguments i8 a CAR or CDR expression, it is

added to a list called the BOMBLIST. In this case the definition of foo is not re-applied for this recursive call. The current evaluation of the function body is continued in the hopes of adding more terms to the BOHBLIST. Finally, EVAL returns (foo $t_1$ ... tl).

Thus, in evaluating (APPEND B C) the recursive call (APPEND (CDR B) C) is encountered in the definition of APPEND. (CDR B) is added to the BOHBLIST indicating recursion on B. Finally, (APPEND B C) is returned as the value of (APPEND B C).

## EVALUATION AND INDUCTION

Partial evaluation is sufficient to prove a few trivial theorems, for example:

(EQUAL (APPEND NIL B) B)

EVALs to T since partial evaluation of the APPEND yields B, even though the structure of B is not known. However, induction is usually needed to prove even simple theorems about recursive LISP functions.

It is intuitively clear that evaluation arid induction are complements. The paradigm for evaluating a simple recursive function FOO is: evaluate (FOO (CONS A B)) in terms of (FOO B) and handle the NIL case separately. The paradigm for a simple inductive proof that (FOO X) is T for any argument X is: show that *(FOO* ML) *is* T, and then assuming that *(?00* P) *is* T, show that (FOO (CONS A B)) is T.

In particular, recursion starts with some structure and decomposes it while induction starts with NIL and builds up. This duality *cim be* used to great advantage: Evaluation can be used to reduce the induction conclusion (FOO (CONS A B)) to a statement involving the induction hypothesis. (FQO B)T provided that the (CONS A B) is one cf the structures that FOO decomposes in its recursion.
Suppose that we wish to prove by induction that (FOO X) is T for all X. To show that (FOO NIL) is T, the obvious thing to do is call EVAL and let evaluation solve the problem (for example, EVAI( (APPEND NIL B) ) is B). We then assume that (FOO B) is T, and try to show that (FOO (CONS A B)) is T, for a new skolem constant A. The obvious thing to do now is to call EVAL again and let the recursion in FOO decompose the (CONS A B). The result will (hopefully) be some simple expression E, involving (FOO B); we then use the hypothesis that (FOO B) is T to show that E is T. This process is illustrated by the examples in the next two sections.

Of course, if FOO has more than one argument, one must choose which one(s) to induct upon. But the link between evaluation and induction makes the choice obvious: induct on the structures that FOO recurses on, that is, on the structures that are being recursively decomposed by FOO, By choosing those structures we insure that when EVAL is called on the induction conclusion, (FOO (CONS A B)), FOO will be able to recurse at least one step and the problem will be reduced by EVAL to one involving the induction hypothesis, (FOO B).

However, the terms that FOO is trying to recurse on are just those that generate the 'errors' noted earlier. To determine what to induct upon we first EVAL the expression (expecting to fail) and then induct upon some term on the BOHBLIST, that is, some term which EVAL failed to evaluate.

## A SIMPLE EXAMPLE OF EVALUATION AND INDUCTION.

Suppose we wish to show that:

(1) (EQUAL (APPEND A (APPEND B C))
(APPEND (APPEND A B) C))

always evaluates to T, for any A, B and C.

The obvious way to proceed is to EVAL the expression and see if it is T. EVAL is unable to make any headway in evaluating (1) and simply returns (1) as its answer. However, *in attempting to evaluate* (1), EVAL placed four terms on the BOMBLIST. Recall that in the definition of APPEND the first argument is CDRed in the recursion but the second argument is not changed. In formula (1) there are four calls to APPEND. Two recurse upon A, one upon B, and one upon (APPEND A B).

Resorting to induction, we choose to induct upon A (we might have chosen B, but we choose A by 'popularity'.) First we try the 'NIL case' (i.e., (1) with A replaced by NIL):

(2) (EQUAL (APPEND NIL (APPEND B C))
(APPEND (APPEND NIL B) C)).

When we EVAL this, partial evaluation of the APPEND's makes (2) equivalent to:

(3) (EQUAL (APPEND B C) (APPEND B C))

since APPEND returns its second argument if the first is NIL. However, (3) is just a partial result, and now that the arguments have been EVALed, the EQUAL is evaluated to T, since in our LISP two identical expressions return EQUAL results. So the 'NIL case' has been shown to be T by evaluation.

*Next we must show that:*

(4) (EQUAL (APPEND (CONS A1 A) (APPEND B C))
(APPEND (APPEND (CONS A1 A) B) C))

is always T, if we assume that:

(5) (EQUAL (APPEND A (APPEND B C))
(APPEND (APPEND A B) C)).

is T.

But EVAL transforms (4) into:

(6) (EQUAL (CONS A1 (APPEND A (APPEND B C)))
(CONS A1 (APPEND (APPEND A B) C)))

and then (from its knowledge of EQUAL) transforms (6) into:

(7) (EQUAL (APPEND A (APPEND B C))
(APPEND (APPEND A B) C)).

But (7) is exactly the same as (5) which we are assuming (inductively) is always T. Hence, by evaluation and the induction hypothesis we have shown that (4), the induction conclusion, is always T. So the associativity of APPEND has been proved. Observe that EVAL was responsible for converting the induction conclusion (4) into an expression involving the induction hypothesis (5).

Our program produces precisely this proof. Its only knowledge about APPEND is its LISP definition.

USING THE INDUCTION HYPOTHESIS AND GENERALIZATION.

Using the induction hypothesis is not always as easy as it was above. A good example occurs in our program's proof of:

(8) (EQUAL (REVERSE (REVERSE A)) A),

where the definition of REVERSE is:

```
(LAMBDA (x)
        (COND X
              (APPEND (REVERSE (CDR X))
                      (CONS (CAB X) NIL))
              NIL).
```

If we induct on A in (8) we find that the NIL case evaluates to T. We therefore assume (8) as our induction hypothesis and try to prove:

(9) (EQUAL (REVERSE (REVERSE (CONS A1 A))) (CONS Al A))

This evaluates to:

(10) (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL)))
(CONS Al A))

We now wish to use the induction hypothesis, (8). Since it is an equality our heuristic is to 'cross-fertilize' (10) with it, by replacing the A in the right-hand side of (10) by the left-hand side of (8), giving:

(11) (EQUAL (REVERSE (APPEND (REVERSE A) (CONS A1 NIL)))
(CONS Al (REVERSE (REVERSE A)))).

We then consider (8) to be 'used' and throw it away. We must now prove (11).

At this point we note that (REVERSE A) is a subformula which appears on both sides of an EQUAL. Furthermore, from the definition of REVERSE the program can determine that the output of (REVERSE A) can be any list at all. On these grounds we choose to generalize the theorem to be proved by replacing (REVERSE A) in (11) by a new skolem constant, B, and set out to prove:
(12) (EQUAL (REVERSE (APPEND B (CONS A1 NIL)))
(CONS Al (REVERSE B))).
But (12) is easy to prove. EVAL tells us to induct on B. The NIL case EVALs to T. Assuming (12) as the induction hypothesis, we EVAL the 'CONS case':
(EQUAL (REVERSE (APPEND (CONS B1 B)(C0NSA1 NIL)))
(13) (CONS Al (REVERSE (CONS B1 B)))) and get
(14) (EQUAL
(APPEND (REVERSE (APPEND B (CONS A1 NIL)))
(CONS B1 NIL))
(CONS A1 (APPEND {REVERSE B) (CONS B1 NIL)))).

We now use our hypothesis, (12), by cross-fertilizing (14) with it, replacing (REVERSE (APPEND B (CONS A1 NIL))) in the left-hand side of (14) by the right-hand side of (12), yielding:

(15) (EQUAL
(APPEND (CONS A1 (REVERSE B)) (COHS B1 MIL))
(CONS A1 (APPEND (REVERSE B) (CONS BI NIL))))

Finally, (15) EVALs to T because the left-hand side APPEND evaluates to:

(CONS A1 (APPEND (REVERSE B) (CCNS B1 NIL))),

which is the right-hand side, so the EQUAL returns T. The theorem is therefore proved.

Our theorem prover takes 8 seconds to produce this proof. If the reader thinks that this theorem is utterly trivial, he is invited to try to prove the similar theorem:

(EQUAL (REVERSE (APPEND A B))
(APPEND (REVERSE B) (REVERSE A))),

which is also proved by the program.

A DESCRIPTION OF THE PROGRAM.

Besides EVAL there are five basic subroutines in our system: NORMALIZE, REDUCE, FERTILIZE, GENERALIZE, and INDUCT. Below are brief descriptions of these routines.

NORMALIZE applies about ten rewrite rules to LISP expressions. For example:

(COND (COND A B C) D E) becomes (CCND A (COND B D E) (COND C D E)), and (COND A A NIL) becomes A.

APPENDIX C lists the rewrite rules.

REDUCE attemptsto propagate the results of the tests in COND statements down the branches of the COND tree. Thus,

(COND A (COND A B C) (P A)) becomes
(COND A B (P NIL)).

FERTILIZE is responsible for 'using' the hypothesis of an implication when it is an equality. A theorem of the form:

x = y - p(y)

is rewritten to

p(x) v x / y.

We make fertilizations of the form:

x =■ y - f(z) = g(y)=> f(z) = g(x) v x / y

before any other kind. We call such substitutions 'cross-fertilizations'; we prefer cross-fertilizations because they frequently allow the proofs we want. After fertilizing we never again look at the equality hypothesis although we retain it for soundness.

GENERALIZE is responsible for generalizing the theorem to be proved. This is done by replacing some common subformulas in the theorem by new skolem constants. To prove something of the form:

p(f(A)) = q(f(A))

we try proving

p(B) = q(B),

and to prove

plflA)) -q(f(A))

we try

p(B) - q(B),

where B is a new skolem constant. However, if the subformula f(A) is of a highly constrained type, for instance, it is always a number, an additional condition is imposed on the new skolem constant.

If the theorem to be generalized is:

(EQUA1 (ADD (LENGTH A) B) (ADD B (LENGTH A))),

GENERALIZE produces as output:

(COHD (LENGTYPE C) (EQUAL (ADD C B) (ADD B C)) T)

where LENGTYPE is a LISP function written by GENERALIZE from the LISP definition of LENGTH. In this particular case, the function written "by GENERALIZE has precisely the definition of NUMBERP, namely:

```
(LAMBDA (X)
        (COND X
              (COND (CAR X) NIL (NUMBERP (CDR X)))
              T)).
```

To perform the generalization described in the previous section, GENERALIZE wrote the 'type function' for REVERSE:

(LAMBDA (X) T),

which was recognized as being no restriction at all and then ignored. The problem of recognizing the output of a recursive function is clearly undecidable and very difficult. To write these type functions, GENERALIZE uses some heuristics which arc often useful.

INDUCT is the program that embodies our indu'tion heuristic. We now describe the form in which it presents the new induction formula to the other routines and how the induction hypothesis is saved for use.

If the theorem to be proved by induction is (F00 A) and EVAL indicates that F00 recursen on the CDR of A, the output of INDUCT is:

```
(COND (FOO NIL)
      (COND (F00 A) (FOO (CONS A1 A)) T)
      NIL).
```

which becomes the theorem to be proved. This is ,just the LISP expression for:

(FOO NIL) & ((F00 A) -> (FOO (CONS A1 A))).

The definitions of AND and IMPLIES are in APPENDIX A.

The precise form of the induction formula output by INDUCT is dictated by the types of 'errors' encountered *by* EVAL. For example, if both the CAR and the CDR of A occur on the BOMBLIST, then the induction formula is:

(FOO NIL) & (((FOO A1) & (FOO A)) - (FOO (CONS A1 A))).

For simultaneous recursion on two variables (e.g. LTE in APPENDIX A) or CDRing twice in a function (e.g., ORDERED), INDUCT produces appropriate induction formulas. All of this information is collected from the BOMBLIST produced by EVAL.

## CONTROL STRUCTURE OF THE PROGRAM.

The control structure of our system is very simple. To prove that some LISP expression, THM, always evaluates to T, we execute the following loop:

```
loop: set OLDTHM to THM;
      set THM to REDUE(N0RMALIZE(EVAL(THM)));
      if THM = T, then return;
      if THM is not EQUAL to OLDTHM, then goto loop;
      if fertilization applies, then set THM to
         FERTILIZE(THM)
      otherwise, if THM is of the form (COND p q NIL)
       then set THM to (COND INDUCT{GENERALIZE(P))
                              q NIL)
      otherwise, set THM to INDUCT(GENERALIZE(THM));
      goto loop;
```

It should be noted that all of the important control structure is embedded in the LISP expression THM. For example, when INDUCT needs to prove the conjunction of the NIL case and the induction step, it is actually done by replacing the expression THM by a LISF expression which has value T if and only if that conjunction is true. If the NIL case evaluates to T, then EVAL returns the second conjunct, which becomes the theorem to be proved.

## CONCLUSION.

We find it natural to use the routines EVAL, NORMALIZE, and REDUCE both to rewrite LISP expresssions; and prove theorems. Our experience confirms, and was motivated by, a conviction that proofs and computations are essentially similar. This conviction was inspired by Bob Kowalski and Pat Hayes, and the beauty of LISP. Cur program is in the style of theorem proving programs written by Woody Bledsoe.

We would like to note that our program uses no search and applies no lemmas. Consequently our theorem prover frequently reproves simple facte like the r.r~ociativity of APPEND. The philosophy of our program is to make the correct guess the first time and to pursue one goal with power and perseverance.

Our program uses structural induction, which was introduced into the literature by Burs'all (1969), although it was used earlier by McCarthy and Painter (1967) in a compiler correctness proof. Common alternative inductive methods for recursive languages are computational induction (Park, 1969, and deBakker and Scott, 1969) and recursion induction (McCarthy, 1963). Both are essentially induction on the depth of function calls. Milner (1972) and Milner and Weyhrnueh (1972) describe a proof checker for Scott's Logic for Computable Functions (Scott, 1970) which uses computational induction. The most commonly used method is for flow-diagram languages and was suggerted by Naur (I966) and Floyd (1967). "n this approach, inductive assertions are attached to points in a program and are used to generate 'verification conditions', which are theorems which roust be proved to establish the correctness of the program. King (1969), Good (1970), Cooper (1971) and Gerhart (1972) have implemented systems which use this method for languag s which include assignments (possibly to arrays) find jumps or loops, but without defined procedure calls. The user supplies the inductive assertions and the systems generate the verification conditions. King and Cooper have incorporated automatic theorem provers which attempt to prove the theorems generated. Topor and Burstall (1973) use a Floyd-liku method on a language with procedure calls. They require user supplied inductive assertions but a symbolic interpreter (like Our EVAl.) generates the verification *conditions, Deutsch* (1973) also uses symbolic evaluation. Wegbreit (1973) and Katz and Manna (1973) present heuristics for generating inductive assertions automatically. Brotz and Floyd (1973) have implemented an arithmetic theorem prover *very* similar to ours. Their system generates its own induction formulas and uses the generalization heuristic we use (without 'type functions'). They induct upon the right-most skolem constant appearing in the statement of the theorem rather than using EVAL and the BOMBLIST as we do. Their heuristic will always choose a term recursed upon (due to restrictions on the forms of recursive equations allowed) but it will not always choose the one we choose. Darlington and Burstall (1973) describe a system which will take functions such as the ones in our LISP subset and write equivalent programs which are more efficient. This system will replace recursion by iteration, rnerge loops, and use data structures (destructively) when permitted.

APPENDIX A contains the definitions of the LISP functions we use in the proofs of the theorems in APPENDIX B. The program automatically proves all of the theorems in APPENDIX B. The average time it takes to prove each theorem is 8 seconds on an ICL 4130 using POP-2. The time is almost completely spent in POP-2 list processing, where the time for a CONS is 400 microseconds, and for CAR and CDR it is 50 microseconds.

Our work is supported by a British Science Research Council Grant. Our thanks to Professor Bernard Meltzer.

REFERENCES.

Burstall, R.M., (1969). 'Proving Properties of Programs by Structural Induction'. Computer Journal, Vol. 12, pp. 41-8.

McCarthy, J. and Painter, J.A., (1967). 'Correctness of a Compiler for Arithmetic Expressions'. In Proceedings of a Symposium in Applied Mathematics. Vol. 19. Mathematical Aspects of Computer Science, pp. 33-41 • (ed. Schwartz, J.T.). Providence, Rhode Island, American Mathematical Society.

Park, D., (1969). 'Fixpoint Induction and Proofs of Program Properties'. In Machine Intelligence *5* (eds Meltzer, B. and Michie, D), Edinburgh University Press, pp. 59-78.

deBakker, J.W. and Scott, D., (1969). 'A Theory of Programs'. Unpublished memo., Vienna.

McCarthy, J., (1963). 'A Basis for a Mathematical Theory of Computation'. In Computer Programming and Formal Systems, pp. 33-70. (eds Braffort, P. and Hirschberg, D,). Amsterdam, North Holland.

Milner, R., (1972). 'Implementation and Application of Scott's Logic for Computable Functions'. In Proceedings of an ACM Conference on Proving Assertions about Programs. SIGPLAN Notices, Vol. 7, No, 1 (January 1972), pp. 1-6.

Milner, R., and Weyhrauch, R., (1972). 'Proving Compiler Correctness in a Mechanized Logic'. In Machine Intelligence 7. pp. 5[1]-70 (eds Meltzer, B. and Michie, D.) Edinburgh University Press.

Scott, D., (1970). 'Outline of a Mathematical Theory of Computation[1]. Oxford University Computing Laboratory, Programming Research Croup, Technical Monograph PRG-2. November 1970.

Naur, P., (1966), 'Proof of Algorithms by General Snapshots'. BIT, Vol. 6, pp. 310-316.

Floyd, R.W., {1967). 'Assigning Meaning to Programs'. In Proceedings of a Symposium in Applied Mathematics. Vol. 19. Mathematical Aspects of Computer Science, pp. 19-32. (ed. Schwartz, J.T.). Providence, Rhode Island, American Mathematical Society.

King, J., (1969). 'A Program Verifier'. Ph.D. Thesis, Carnegie-Mellon University, U.S.A.

Good, D., (1970). 'Toward a Man-Machine System for Proving Program Correctness'. Ph.D. Thesis, University of Wisconsin, U.S.A.

Cooper, D. (1971). 'Programs for Mechanical Program Verification'. In Machine Intelligence 6. pp. 43-59, (eds Meltzer, B. and Michie, D). Edinburgh University Press.

Gerhart, S., (1972). 'Verification of APL Programs'. Ph.D. Thesis, Carnegie-Mellon University, U.S.A.

Topor, R., and Burstall, R.M. (1973) - Private Communication.

Deutsch, P., (1973) Forthcoming Ph.D. Thesis.

Brotz, D. and Floyd, R.W. (1973). 'Proving Theorems by Mathematical Induction'. Stanford Computer Science Department Report.

Wegbreit, B., (1973)- 'Heuristic Methods for Mechanically Deriving Inductive Assertions'. In Proceedings of IJCAI, 1973 (to appear).

Katz, S.M., and Manna, Z., (1973). 'A Heuristic Approach to Program Verification'. In Proceedings of IJCAI 1973 (to appear).

Darlington, J. and Burstall, R.M., (1973}- 'A System which automatically improves Programs'. In Proceedings of IJCAI 1973 {to appear).

APPENDIX *&*. FUNCTION DEFINITIONS.

```
DEFINE((ADD (LAMBDA {X Y)
  (COND X (CONS NIL (ADD (CDR X) Y)) (LENGTH Y)))))
[NOTE: CONSING NIL ONTO A NUMBER IS JUST
ADDING 1 TO IT.  'LENGTH' IS USED TO INSURE
THAT THE OUTPUT IS ALWAYS A NUMBER.]
DEFINE((ADDTOLIS (LAMBDA (X Y)
  (COND Y
        (COND (LTE X (CAR Y))
              (CONS X Y)
              (CONS (CAR Y) (ADDTOLIS X (CDR Y))))
        (CONS X NIL)))))
DEFINE((AND (LAMBDA (X Y)
  (COND X (COND Y T NIL) NIL))))
[NOTEt  'AND' IS DEFINED SO THAT IT IS ALWAYS
BOOLEAN, EVEN IF X AND Y ARE NOT.  THE SAME
HOLDS FOR 'OR', 'NOT', AND 'IMPLIES'.]
DEFINE((APPEND (LAMBDA (X Y)
  (COND X (CONS (CAR X) (APPEND (CDR X) Y)) Y))))
DEFINE((ASSOC (LAMBDA (X Y)
  (COND Y
        (COND (CAR Y)
              (COND (EQUAL X (CAR (CAR Y)))
                    (CAR Y)
                    (ASSOC X (CDR Y)))
              (ASSOC X (CDR Y)))
        NIL))))
DEFINE((BOOLEAN (LAMBDA (x)
  (COND X (EQUAL X T) T))))
DEFINE((CDRN (LAMBDA (X Y)
  (COND Y ( COND X (CDRN (CDR X) (CDR T)) Y) NIL))))
[NOTE: 'CDRN' RETURNS THE XTH CDR OF Y.]
DEFINE((CONSNODE (LAMBDA (X Y)
  (CONS NIL (CONS X Y)))))
DEFINE((COPY (LAMBDA (x)
```

```
 (COND X
       (CONS (COPY (CAR X)) (COPY (CDR X)))
       NIL))))
DEFINE((COUNT (LAMBDA (x Y)
 (CONE Y
       (COND (EQUAL X (CAR Y))
             (CONS NIL (COUNT X (CDR Y)))
             (COUNT X (CDR Y)))
       NIL))))
[NOTE: 'COUNT' RETURNS THE NUMBER OF TIMES
X OCCURS AS AN ELEMENT OF Y.]
DEFINE((DOUBLE (LAMBDA (x)
 (COND X
       (CONS NIL (CONS NIL (DOUBLE (CDR X))))
       NIL))))
DEFINE((ELEMENT (LAMBDA (X Y)
 (COND Y
       (COND X (ELEMENT (CDR X) (CDR Y)) (CAR Y))
       NIL))))
DEFINE(EQUALP (LAMBDA (X Y)
 (COND X
       (COND Y
             (COND (EQUALP (CAR X) (CAR Y))
                   (EQUALP (CDR X) (CDR Y))
                   NIL)
             NIL)
       (COND Y NIL T)))))
DEFINE((EVEN1 (LAMBDA (X)
 (COND X (NOT (EVEN1 (CDR X))) T))))
DEFINE((EVEN2 (LAMBDA (x)
 (COND X
       (COND (CDR X) (EVEN2 (CDR (CDR X))) NIL)
       T)))>
DEFINE (FLATTEN (LAMBDA (X)
 (COND (NODE X)
       (APPEND (FLATTEN (CAR (CDR x)))
               (FLATTEN (CDR (CDR X))))
       (CONS X NIL)))))
[NOTE: 'FLATTEN' RETURNS A LIST OF TIPS IN
A BINARY TREE OF 'NODES'.   SEE 'CONSNODE'
FOR THE DEFINITION OF HOW TO BUILD A NODE.]
DEFINE((GT (LAMBDA (X Y)
 (COND X (COND Y (GT (CDR X) (CDR Y)) T) NIL))))
DEFINE((HALF (LAMBDA (X)
 (COND
  X
  (COND (CDR X) (CONS NIL (HALF (CDR (CDR X)))) NIL)
  NIL))))
DEFINE((IMPLIES (LAMBDA (X Y)
 (COND X (COND Y T NIL) T))))
DEFINE((INTERSEC (LAMBDA (X Y)
 (COND X
       (COND (MEMBER (CAR X) Y)
             (CONS (CAR X) (INTERSEC (CDR X) Y))
             (INTERSEC (CDR X) Y))
       NIL))))
DEFINE((LAST (LAMBDA (X)
 (COND X
       (COND (CDR X) (LAST (CDR X)) (CAR X))
       NIL))))
DEFINE((LENGTH (LAMBDA (X)
 (COND X (CONS NIL (LENGTH (CDR X))) NIL))))
DEFIHE((LIT (LAMBDA (X Y Z)
 (COND X (APPLY Z (CAR X) (LIT (CDR X) Y Z)) Y))))
[NOTE: 'LIT'' IS A GENERAL PURPOSE FUNCTION,
FOR EXAMPLE, (APPEND X T) = (LIT X Y CONS).
OUR PROGRAM CANNOT HANDLE FUNCTIONS AS ARGS,
BUT SOME FACTS ABOUT 'LIT' CAN BE VERIFIED
WITHOUT KNOWING WHAT 'APPLY' DOES.]
DEFINE((LTE (LAMBDA (X I)
 (COND X (COND Y (LTE (CDR X) (CDR Y)) NIL) T))))
DEFINE((MAPLIST (LAMBDA (X Y)
 (COND X
       (CONS (APPLY Y (CAR X)) (MAPLIST (CDR X) Y))
       NIL))))
DEFIHE((MEKBER (LAMBDA (x Y)
 (COND Y
       (COND (EQUAL X (CAR Y)) T (MEMBER X (CDR Y)))
       NIL))))
DEFIRE((MONOT1 (LAMBDA (X)
 (COND X
       (COND (CDR X)
             (COND (EQUAL (CAR X) (CAR (CDR X)))
                   (MONOT1 (CDR X)) NIL)
             T)
       T))))
DEFINE((MONOT2 (LAMBDA (X Y)
 (COND
     Y
     (COND (EQUAL X (CAR Y)) (M0N0T2 X (CDR Y)) NIL)
     T))))
DEFINE((MONOT2P (LAMBDA (X)
 (COND X (MONOT2 (CAR X) (CDR X)) T))))
[NOTE: A LIST IS 'MONOTONOUS' IF ALL THE
ELEMENTS ARE THE SAME.   'MONOTI' AND 'MONOT2P'
ARE TWO DIFFERENT WAYS TO DETECT THIS.]
DEFINE((MULT (LAMBDA (X Y)
 (COND X (ADD Y (MULT (CDR X) Y)) NIL))))
DEFINE((NODE (LAMBDA (X)
 (COND X
       (COND (CAR X) NIL (COND (CDR X) T NIL))
       NIL))))
DEFINE((NOT (LAMBDA (X)
 (COND X NIL T))))
DEFINE((NUMBERP (LAMBDA (X)
 (COND X (COND (CAR X) NIL (NUMBERP (CDR X))) T))))
DEFINE((OCCUR (LAMBDA (X Y)
 (COND
  (EQUAL X Y)
  T
  (COND Y
        (COND (OCCUR X (CAR Y)) T (OCCUR X (CDR Y)))
        NIL)))))
DEFINE((OR (LAMBDA (X Y)
 (COND X T (COND Y T NIL)))))
DEFINE((ORDERED (LAMBDA (x)
 (COND X
       (COND (CDR X)
             (COND (LTE (CAR X) (CAR (CDR X)))
                   (ORDERED (CDR x))
                   NIL)
             T)
       T))))
DEFINE((PAIRLIST (LAMBDA (X Y)
 (COND
  X
  (COND
   Y
   (CONS (CONS (CAR X) (CAR Y))
         (PAIRLIST (CDR X) (CDR Y)))
   (CONS (CONS (CAR X) NIL) (PAIRLIST (CDR X) NIL)))
  NIL))))
DEFINE((REVERSE (LAMBDA (x)
 (COND X
       (APPEND (REVERSE (CDR X)) (CONS (CAR X) NIL))
       NIL))))
DEFINE((SORT (LAMBDA (x)
 (COND X (ADDTOLIS (CAR X) (SORT (CDR X))) NIL))))
DEFINE((SUBSET (LAMBDA (X Y)
 (COND
    X
    (COND (MEMBER (CAR X) Y) (SUBSET (CDR X) Y) NIL)
    T))))
DEFINE((SUBST (LAMBDA (X Y Z)
 (COND
  (EQUAL Y Z)
  X
  (COND
      Z
      (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR 2)))
      NIL)))))
DEFINE(SWAPTREE (LAMBDA (X)
 (COND (NODE X)
       (CONSNODE (SWAPTREE (CDR (CDR X)))
```

```
                (SWAPTREE (CAR (CDR X))))
        X))))
DEFINE((TIPCOUNT (LAMBDA (X)
 (COND (NODE X)
        (ADD (TIPCOUNT (CAR (CDR X)))
             (TIPCOUNT (CDR {CDR x))))
        1))))
DEFINE((UNION (LAMBDA (X T)
 (COND x
        (COND (MEMBER (CAM X) Y)
              UNION (CDR X) Y)
             (CONS (CAR X) (UNION (CDR X) Y)))
```

APPENDIX B. THEOREMS PROVED AUTOMATICALLY.

APPEND, LENGTH AND REVERSE THEOREMS

```
(EQUAL (APPEND A (APPEND B C))
       (APPEND (APPEND A B) C))

(IMPLIES (EQUAL (APPEND A B) (APPEND A C))
         (EQUAL E C))

(EQUAL (LENGTH (APPEND A B)) (LENGTH (APPEND B A)))

(EQUAL (REVERSE (APPEND A B))

       (APPEND (REVERSE B) (REVERSE A)))

(EQUAL (LENGTH (REVERSE D)) (LENGTH D))

(EQUAL (REVERSE (REVERSE A)) A)

(IMPLIES A (EQUAL (LAST (REVERSE A)) (CAR A)))
```

MEMBER, UNION, ETC THEOREMS

```
(IMPLIES (MEMBER A B) (MEMBER A (APPEND B C)))

(IMPLIES (MEMBER A B) (MEMBER A (APPEND C B)))

(IMPLIES (AND (NOT (EQUAL A (CAR B))) (MEMBER A B))
         (MEMBER A (CDR B)))

(IMPLIES (OR (MEMBER A B) (MEMBER A C))
         (MEMBER A (APPEND B C)))

(IMPLIES (AND (MEMBER A B) (MEMBER A C))
         (MEMBER A (INTERSEC B C)))

(IMPLIES (OR (MEMBER A B) (MEMBER A C))
         (MEMBER A (UNION B c)))

(IMPLIES (SUBSET A B) (EQUAL (UNION A B) B))

(IMPLIES (SUBSET A B) (EQUAL (INTERSEC A B) A))

(EQUAL (MEMBER A B)
       (NOT (EQUAL {ASSOC A (PAIRLIST B C)) NIL)))
```

HAPLIST THEOREMS

```
(EQUAL (MAPLIST (APPEND A B) C)
       (APPEND (MAPLIST A C) (MAPLIST B C)))

(EQUAL (LENGTH (MAPLIST A B)) (LENGTH A))

(EQUAL (REVERSE (MAPLIST A B))
       (MAPLIST (REVERSE A) B))
```

MISCELLANEOUS THEOREMS

```
(EQUAL (LIT (APPEND A B) CD) (LIT A (LIT B C D) D))

(IMPLIES (AND (BOOLEAN A) (BOOLEAN B))
         (EQUAL (AND (IMPLIES A E) (IMPLIES B A))
                (EQUAL A B)))

(EQUAL (ELEMENT B A)

       (ELEMENT (APPEND C B) (APPEND C A)))

(IMPLIES (ELEMENT B A) (MEMBER (ELEMENT B A) A))

(EQUAL (CDRN C (APPEND A B))

       (APPEND (CDRN C A) (CDRN (CDRN A C) B)))

(EQUAL (CDRN (APPEND B C) A) (CDRN C (CDRN B A)))

(EQUAL (EQUAL A B) (EQUAL B A))

(IMPLIES (AND (EQUAL A B) (EQUAL B c)) (EQUAL A C))
(IMPLIES
     (AND (BOOLEAN A) (AND(BOOLEAN B) (BOOLEAN C)))
     {EQUAL (EQUAL (EQUAL A B) C)
            (EQUAL A (EQUAL B C))))
```

ARITHMETIC THEOREMS

```
(EQUAL (ADD A B) (ADD B A))

(EQUAL (ADD A (ABD B C)) (ADD (ADD A B) C))

(EQUAL (MULT A B) (MULT B A))

(EQUAL (MULT A (ADD B C))

       (ADD (MULT A B) (MULT A C)))

(EQUAL (MULT A (MULT B C)) (MULT (MULT A B) C))

(EVENt (DOUBLE A))

(IMPLIES (NUMBER? A) (EQUAL (HALF (DOUBLE A)) A))
(IMPLIES (AND (NUMBERP A) (EVENI A))
         (EQUAL (DOUBLE (HALF A)) A))

(EQUAL (DOUBLE A) (MULT 2 A))

(EQUAL (DOUBLE A) (MULT A 2))

(EQUAL (EVEK1 A) (EVEN2 A))
```

GT, LTE, ORDERED AND SORT THEOREMS

```
(QT (LENGTH (CONS A B)) (LENGTH B))

(IMPLIES (AND (GT A B) (GT B C)) (GT A C))

(IMPLIES (GT A B) ( NOT (GT B A)))

(LTE A (APPEND B A))

(OR (LTE A B) (LTE B A))

(OR (GT A B)
    (OR (GT B A) (EQUAL (LENGTH A) (LENGTH Bj)))

(EQUAL (M0N0T2P A) (MONOT1 A))

(ORDERED (SORT A))

(IMPLIES (AND (MONOT1 A) (MEMBER B A))
         (EQUAL (CAR A) B))

(LTE (CDRN A B) B)
```

(EQUAL (MEMBER A (SORT B)) (MEMBER A B))

(EQUAL (LENGTH A) (LENGTH (SORT A)))

(EQUAL (COUNT A B) (COUHT A (SORT B)))

(IMPLIES (ORDERED A) (EQUAL A (SORT A)))

(IMPLIES (ORDERED (APPEND A B)) (ORDERED A))

(IMPLIES (ORDERED (APPEND A B)) (ORDERED B))

(EQUAL (EQUAL (SORT A) A) (ORDERED A))

(LTE (HALF A) A)

THEOREMS ABOUT TREES

(EQUAL (COPY A) A)

(EQUAL (EQUAL P A B) (EQUAL A B))

(EQUAL (SUBST A A B) B)

(IMPLIES (MEMBER A B) (OCCUR A B))

(IMPLIES (NOT (OCCUR A B)) (EQUAL (SUBST C A B) B))

(EQUAL (EQUALP A B) (EQUALP B A))

(IMTLIES (AND (EQUALP A B) (EQUALP B C))
(EQUALP A C))

(EQUAL (SWAPTREE (SWAPTREE A)) A)

(EQUAL (FLATTEN (SWAPTREE A)) (REVERSE (FLATTEN A)))

(EQUAL (LENGTH (FLATTEN A)) (TIPCOUKT A))

## APPENDIX C. REWRITE RULES APPLIED BY NORMALIZE.

In the rules below, lower oase letters represent arbitrary forms. Forms matching those on the loft-hand side of the arrows are replaced by the appro-priate instances of the forms on the right. IDENT is a routine which takes two terms as arguments and returns 'equal' if they are syntactically identical (such as (CONS A B) and (CONS A B), or (CONS NIL NIL) and 1), 'unequal' if they are obviously unequal (such as (CONS A B) and NIL, or (CONS A B) and A), or 'unknown'. BOOLEAN is a routine which returns true or false depending upon whether its argument is boolean by inspecting its definition with an inductive assumption that any recursive calls are to be considered boolean. NORMALIZE rewrites the arguments to the term it is given before rewriting the top-level expression. Finally, any rule involving EQUAL has a symmetric version not presented in which the arguments to the EQUAL have been interchanged.

(EQUAL x y) =>T, if IDENT(x,y) = 'equal'
(EQUAL x y) => NIL, if IDENT(x.y) = 'unequal'
(EQUAL x T) => x, if BOOLEAN(x)
(EQUAL (EQUAL x y) *z)* => (COND (EQUAL x y)
(EQUAL z T)
(COND 2 NIL T))
(COND (CONS U V) x y) => x
(COND NIL x y) =c y
(COND *x* T NIL) ^> x, if BOOLEAN(x)
COND *x* y y) => $_{y}$
(COND x x NIL) => *x*
(f $X_1$...(COHD y u v) . . . x ) => (COND y
(f *x* . . . u . . . x )