

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM 151

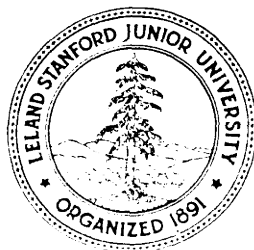
COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS 240

CORRECTNESS OF TWO COMPILERS FOR A LISP SUBSET

BY
RALPH L. LONDON

OCTOBER 1971

COMPUTER SCIENCE DEPARTMENT
STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-151

OCTOBER 1971

COMPUTER SCIENCE DEPARTMENT
REPORT CS-240

CORRECTNESS OF TWO COMPILERS FOR A LISP SUBSET

by

Ralph L. London

ABSTRACT: Using mainly structural induction, proofs of correctness of each of two running Lisp compilers for the PDP-10 computer are given. Included are the rationale for presenting these proofs, a discussion of the proofs, and the changes needed to the second compiler to complete its Proof.

To be Presented at the Conference on Proving Assertions about Programs, New Mexico State University, January 1972,

This research was supported in part by the Advanced Research Projects Agency - of the Office of the Secretary of Defense under Contract SD-183 and in part by the National Aeronautics and Space Administration under Contract NSR 05-020-500.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, the National Aeronautics and Space Administration, or the U. S. Government,

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information (or its successors), Springfield, Virginia 22151. Price: Full size CODY \$3.00; microfiche copy \$0.95.

CORRECTNESS OF TWO COMPILERS FOR A LISP SUBSET

by

Ralph L. London

INTRODUCTION AND JUSTIFICATION

This paper contains proofs of correctness of each of two useful, running compilers, named CO and C4. The source language for both compilers is the same subset of pure (basic) Lisp, which subset excludes special or global variables, function names as arguments, and the form label: the object language is essentially assembly code for the PDP-10 computer; and the compilers themselves are written recursively in RLISP (Hearn 1970), a version of Lisp with Algol-like syntax. The compilers were written by John McCarthy as part of a series of progressively more optimizing compilers for use in a course at Stanford entitled "Computing with Symbolic Expressions." Only later have these compilers been considered for proving correctness. A listing of the compilers and sample output are in the Appendices.

The proof P4 of correctness of the compiler C4 is a modification and extension of the proof P0 for C0. The organization of this paper is first to prove C0 correct exclusively. A brief discussion of the proof appears just after the proof. Then using the same machinery that is defined, and using much of the proof P0, the compiler C4 is proved correct. This serial organization, reflecting the essential chronology of the work, seems preferable to proving the two compilers in parallel. The reader should now ignore C4 (and P4) until the start of P4, except to note that the input and overall statement of correctness are the same as for C0.

To prove the correctness of a compiler is a frequently heard challenge. The present proof partly responds to the challenge: The compiler is sufficiently lengthy and complex not to be viewed as merely another cooked-up research example. As evidence of this, Whitfield Diffie has shown the compiler capable of compiling itself successfully. Yet the compiler has certain toy-problem aspects, for example accepting a subset of full Lisp, the inefficiency of the resulting object code, and the simple parser. It is certainly not a production compiler. Nevertheless, exhibiting yet another proof seems justified since (i) a compiler is somewhat different from other algorithms that have been proved (there are at least two programs being executed, the compiler and the object program, and, to a lesser extent, the source program); (ii) there has been little progress in proving compilers correct beyond the work of McCarthy & Painter (1967), Painter (1967), Kaplan (1967), Burstall (1969), and Burstall & Landin (1969), although the work of McGowan (1971) should be mentioned; (iii) there remains the worthwhile goal of being able to prove compilers correct; (iv) this proof has been made to serve as the nucleus of a proof of correctness of a more optimizing compiler in the existing series; (v) the informal proof serves as the basis of more formalized proofs, the latter being necessary if a proof of

correctness is to be checked by a proof checker (Milner 1972); and (vi) the correctness of the compiler is not immediately obvious.

THE PROBLEM STATEMENT, NOTATION, AND PLAN OF ATTACK

The reader is assumed to have a basic knowledge of Lisp, say from Weissman's (1967) primer. The input to the compiler is (DE NAME (args) body), DE is for Define Expression and NAME is the name of the function being compiled. The quantity (args) is the list of arguments (formal parameters) for the function NAME and body is the body of the function. The calling convention is that a defined function f of $N \geq 0$ arguments, say $arg1, arg2, \dots, argN$, will find run-time values of those arguments in successive accumulators starting in $ac1$, which holds $arg1$, and the result $f(arg1, arg2, \dots, argN)$ will be returned in $ac1$. This convention applies also to any function call compiled by the compiler in response to a call in the source code, e.g. the call to CAR in WE SIMPLE (X) (CAR X)). In particular the call may be a recursive call, e.g.

```
(DE COMPLEX (X Y) (COND ((NULL X) (CONS Y X))
                        (T (COMPLEX (CDR X) Y)) )).
```

We now give a more detailed and more precise description of the allowable syntax and its intended meaning. The list (args) is a list of atoms excluding NIL, T, and numeric-atoms; body is an expression where expression is defined recursively below ($N \geq 0$ in all relevant cases). The value of an expression EXP, denoted $V \text{ EXP}$, is recursively defined at the same time (as an "informalization" of the Lisp EVAL function),

- (i) atom, in particular NIL, T, or a numeric-atom, $V \text{ atom}$:
 $V \text{ NIL} = (\text{QUOTE NIL})$ [0 in this compiler],
 $V \text{ T} = (\text{QUOTE T})$, where a non-NIL value is considered equal to $V \text{ T}$,
 $V \text{ numeric-atom} = (\text{QUOTE numeric-atom})$, and
 $V \text{ other atom} = \text{its binding, i.e., run-time value which may not be a function name,}$
- (ii) (AND EXP1 EXP2 ... EXPN), $V \text{ AND-expression} = T$ if all $V \text{ EXP}_i$ are non-NIL otherwise NIL. $V \text{ (AND)}$ = T. AND evaluates its arguments from left to right until either NIL is found in which case the remaining arguments are not evaluated, or until the last argument is evaluated,
- (iii) (OR EXP1 EXP2 ... EXPN), $V \text{ OR-expression} = T$ if any $V \text{ EXP}_i$ is non-NIL otherwise NIL. $V \text{ (OR)}$ = NIL. OR evaluates its arguments from left to right until either a non-NIL is found in which case the remaining arguments are not evaluated, or until the last argument is evaluated,
- (iv) (NOT EXP), $V \text{ NOT-expression} = T$ if $V \text{ EXP}$ is NIL otherwise NIL.

- (v) (COND (EXP1 EXP2) (EXP3 EXP4) ..., (EXP[2N-1] EXP[2N])),
V COND-expression is determined as follows. The expressions EXP1, EXP3, ..., EXP[2N-1] are evaluated starting with EXP1 until the first EXP[2i-1] is found whose value is non-NIL. V COND-expression is then V EXP[2i]. If no EXP[2i-1] exists with non-NIL value, then V COND-expression is undefined.
- (vi) (QUOTE EXP), V QUOTE-expression = EXP, i.e. EXP unevaluated,
- (vii) (fname EXP1 EXP2 ..., EXPN) where fname \neq AND, OR, NOT, COND, QUOTE, V function-expression = fname(V EXP1, V EXP2, ..., V EXPN), i.e. the value of the function fname applied to its evaluated arguments V EXP1, V EXP2, ..., V EXPN. The arguments are evaluated once before the function is called,
- (viii) ((LAMBDA (atom1 atom2 ..., atomN) EXP) EXP1 EXP2 ..., EXPN)
where atomi \neq NIL, T, numeric-atom, V LAMBDA-expression is determined as follows. A LAMBDA-expression defines a function which has no explicit (atomic) name. V LAMBDA-expression is the value of this function applied to its evaluated arguments V EXP1, V EXP2, ..., V EXPN. In other words, V LAMBDA-expression = V EXP where V EXP is computed after the substitutions atom1 \leftarrow V EXP1, atom2 \leftarrow V EXP2, ..., atomN \leftarrow V EXPN have been made in EXP. If there is a clash of bound variables, the convention is that the innermost binding governs,

Since function names are forbidden as arguments, the expression ((LAMBDA (X) (X)) Y) means a call to the function X of no arguments rather than a call to the function argument Y. The above syntax forbids ((X)), ((X)), etc. as expressions,

The compiler is proved correct under the assumption that its input is syntactically correct. Since no error checking is done by the compiler, nothing is claimed for the results, if any, of incorrect input. Correct input also means, for example, that a list of formal parameters consists of distinct atoms and that the number of formal parameters is always equal to the number of actual parameters. There are presumably many other such conditions, violations of some of which may have reasonable interpretations,

The statement of correctness of the compiler is that the compiler-produced object code, when executed, leaves a result in acl equal to the value of the source language function applied to the same arguments. The object code takes its N arguments from the accumulators ac1, ..., acN. If A = a1 a2 ..., aN represents the arguments, then the correctness statement may be restated as requiring that the equation

$$V ((\text{DE NAME (args) body}) A) = \text{contents of } ac1$$

holds after executing the list of compiler-produced instructions

COMP(NAME, (args), body)

starting with aci holding ai for $1 \leq i \leq N$.

The following facts about the PDP-10 computer are from a writeup by McCarthy: The PDP-10 has a 36 bit word and an 18 bit address. In instructions and in accumulators used as index registers this is the right part of the word where the least significant bits in arithmetic reside.

There are 16 general registers which serve simultaneously as accumulators (receiving the results of arithmetic operations), index registers (modifying the nominal addresses of instructions to form effective addresses), and as the first 16 registers of memory (if the effective address of an instruction is less than 16, then the instruction uses the corresponding general register as its operand).

All instructions have the same format and are written for the LAP assembly program in the form

(<op name> <accumulator> <address> <index register>),

Thus (MOVE 1 3 P) causes accumulator 1 to receive the contents of a memory register whose address is $3+c(P)$, i.e. $3+\langle$ the contents of general register P \rangle . In the following description of instructions, <ef> denotes the effective address of an instruction.

MOVE	$c(ac) \leftarrow c(\langle ef \rangle)$
MOVEI	$c(ac) \leftarrow \langle ef \rangle$
HLR2 (used in C4 only)	$c(\text{left half } ac) \leftarrow \text{right half of } c(\langle ef \rangle)$
HRR2 (used in C4 only)	$c(\text{right half } ac) \leftarrow c(\text{right half of } c(\langle ef \rangle))$
SUB	$c(ac) \leftarrow c(ac) - c(\langle ef \rangle)$
JRS1	go to <ef>
JUMPE	if $c(ac) = 0$ then go to <ef>
JUMPN	if $c(ac) \neq 0$ then go to <ef>
CAME (used in C4 only)	If $c(ac) = c(\langle ef \rangle)$ then skip next instruction
CAMN (used in C4 only)	If $c(ac) \neq c(\langle ef \rangle)$ then skip next instruction
PUSH	$c(\text{right half of } ac) \leftarrow c(\langle ef \rangle)$; the contents of each half of ac is increased by one
POPJ	(POPJ P) is used to return from a subroutine

These instructions are adequate for compiling basic Lisp code with the addition of the subroutine calling pseudo-instruction. (CALL n (E <subr>)) is used for calling the Lisp subroutine <subr> with n arguments. The convention is that the arguments will be stored in successive accumulators beginning with accumulator 1, and the result will be returned in accumulator 1. In particular the functions ATOM and CONS are called with (CALL 1 (E ATOM)) and (CALL 2 (E CONS)) respectively. Note that the instruction (SUB P (C 0 0 3 3)) just deletes the top three elements of the stack P. (PUSH P ac) is used

to put $c(ac)$ on the stack P . This ends the facts about the PDP-10 computer.

To show the result and effect of executing a section of assembly code, notation of hand-simulation, desk-checking, or tracing of code is used. It is best explained by example. Starting with N accumulators each holding a value and an empty stack P , namely

```
ac1|α1
ac2|α2
...
acN|αN
P|
```

the list of instructions

```
((Instructions to leave α1 in ac1)
(PUSH P1)
...
(Instructions to leave αN in ac1)
(PUSH P 1)
(MOVE 1 1-N P)
(MOVE 2 2-h) P)
...
(MOVE N 0 P)
(SUB P (C 0 0 NN))
(CALL N (E name)))
```

gives the trace

```
ac1|α1* α1* α2* ... αN* α1* name(α1 α2 ... αN)
ac2|α2* α2* undsf
...
acN|αN* αN* undef
P|α1* α2* ... αN* ,
```

Thus the value $name(α1 α2 ... αN)$ is in $ac1$, undef (an undefined quantity) is in $ac1$ for $2 \leq i \leq N$ since these accumulators are unsafe over name, and the stack P is unaltered from the start. The trace shows the final result of tracing; the intermediate results are recorded but marked by an asterisk (*) as being no longer present.

The plan of attack is as follows:

- (i) Prove correct 3 auxiliary Procedures [MKPUSH(N, M), PRUP(VARS, N), and LOADAC(N, K)] which are not part of the main recursiveness of the compiler (lemmas 1-3).
- (ii) Under the assumption of no conditional expressions or Boolean expressions (i.e. no COND, AND, OR, NOT), prove the compiler correct (theorems 1-3 and termination), and
- (iii) Prove the compiler correct without the restrictive assumption

of (ii) (theorems 4-7).

The proof techniques to be used are mainly those shown in London(1970). The factorization into (ii) and (iii), convenient for constructing, for presenting, and for reading the proof, shows how one can prove an algorithm in suitable segments rather than having to do it all at once. If the reader omits theorems 4-7 of (iii), the proof of correctness of an interesting subcompiler results. In this part recursion is still allowed in the sense that the compiler will correctly compile a recursive function. But the object code may not terminate if such a recursive function is called since there is no branching to "stop" the recursion?

The numbering of the lemmas and theorems reflects the order of their discovery and proof. The order could be altered by merging theorems 1 and 7 and by placing theorem 3 as the last theorem if the sole interest were to prove the entire compiler.

PROOF OF AUXILIARY FUNCTIONS FOR C0

The Lisp operation CONS is denoted in RLISP by an infix dot(.): $A.3 = (CONS A B)$. By inspection of the whole compiler, it follows that all numerically-valued quantities are integers. • is used as an end-of-proof marker.

Lemma 1. If $N > 0$ and $M > 0$, then $MKPUSH(N,M) =$

((PUSH P M)
(PUSH P M+1)
...
(PUSH P N)),

If $M > 0$, then $MKPUSH(0,M) = NIL$.

Proof. Backwards induction on M . If $M > N$, $MKPUSH(N,M) = NIL$. If $M = N$, we have $(PUSH P M).NIL = ((PUSH P N))$. Assume the lemma for $M \leq N$ and consider $M+1 > 0$.

$MKPUSH(N,M-1) = (PUSH P M-1).MKPUSH(N,M)$ since $N > M-1$

$= (PUSH P M-1).$
 $((PUSH P M)$
 $(PUSH P M+1)$
 $...$
 $(PUSH P N))$ by induction hypothesis for M

$= ((PUSH P M-1)$
 $(PUSH P M)$
 $(PUSH P M+1)$
 $...$
 $(PUSH P N))$ by definition of CONS, •

Alternative notation may be used to avoid the three dots (...) in the lemma and in the proof. Analogously to the sigma notation for indicating s-urns (e.g. $\sigma(i=1, N, A[i])$), define a list functional L:

$$L(i=M, N, (\text{PUSH } P \ i)) = \text{NIL} \text{ if } N < M$$

$$L(i=M, N, (\text{PUSH } P \ i)) = (\text{PUSH } P \ M), L(i=M+1, N, (\text{PUSH } P \ i)) \\ \text{if } N \geq M$$

Whereas sigma denotes iterated addition, L denotes iterated CONSing.

The lemma is restated as $\text{MKPUSH}(N, M) = L(i=M, N, (\text{PUSH } P \ i))$. The proof of the induction step becomes

$$\begin{aligned} \text{MKPUSH}(N, M-1) &= (\text{PUSH } P \ M-1), \text{MKPUSH}(N, M) \\ &= (\text{PUSH } P \ M-1), L(i=M, N, (\text{PUSH } P \ i)) \\ &= L(i=M-1, N, (\text{PUSH } P \ i)), \end{aligned}$$

Similar notation may be used for lemmas 2 and 3 below.

Lemma 2. Let $\text{VARS} = (x_1 \ x_2 \ \dots \ x_M)$. Then $\text{PRUP}(\text{VARS}, N) = ((x_1, N) \ (x_2, N+1) \ \dots \ (x_M, N+M-1))$. This list of pairs is called the PRUP list, short for "pair-up."

Proof. Induction on M. If $M = 0$, then $\text{PRUP}(\text{VARS}, N) = \text{NIL}$ since $\text{NULL } \text{VARS}$. Assume for $M \geq 0$ and consider $M+1$.

$$\begin{aligned} \text{PRUP}(\text{VARS}, N) &= (\text{CAR } \text{VARS}, N), \text{PRUP}(\text{CDR } \text{VARS}, N+1) \quad \text{since } M+1 > 0 \text{ implies} \\ &\quad \text{not NULL } \text{VARS} \\ &= (x_1, N), ((x_2, N+1) \ \dots \ (x_{M+1}, N+M)) \quad \text{by the induction} \\ &\quad \text{hypothesis for CDR } \text{VARS} \\ &= ((x_1, N) \ (x_2, N+1) \ \dots \ (x_{M+1}, N+M)) \text{ by use of } \dots \end{aligned}$$

$$\begin{aligned} \text{Lemma 3. } \text{LOADAC}(N, K) &= ((\text{MOVE } K \ N \ P) \\ &\quad (\text{MOVE } K+1 \ N+1 \ P) \\ &\quad \dots \\ &\quad (\text{MOVE } K-N \ 0 \ P)), \end{aligned}$$

Proof. Backwards induction on N. If $N > 0$, the result is NIL. If $N = 0$, we have $(\text{MOVE } K \ 0 \ P), \text{NIL} = ((\text{MOVE } K-0 \ 0 \ P))$. Assume the lemma for $N \leq 0$ and consider $N-1$.

$$\begin{aligned} \text{LOADAC}(N-1, K) &= (\text{MOVE } K \ N-1 \ P), \text{LOADAC}(N, K+1) \text{ since } N-1 < 0 \\ &= (\text{MOVE } K \ N-1 \ P), ((\text{MOVE } K+1 \ N \ P) \ \dots \ (\text{MOVE } K+1-N \ 0 \ P)) \\ &\quad \text{by induction hypothesis for } N \end{aligned}$$

= ((MOVE K N-1 P) (MOVE K+1 N P) ..., (MOVE K-(N-1) 0 P))
by use of . and arithmetic. •

THE RUN-TIME STACK

The object code uses a run-time stack in a rather standard way for holding the actual Parameters of both function calls and LAMBDA expression evaluations. As each actual parameter (binding) is evaluated, it is pushed onto the stack. This suffices for a LAMBDA expression but not for a function. After all of the latter's actual parameters are evaluated and pushed onto the stack, all are moved to the accumulators and popped from the stack in order to satisfy the conventions for calling a function. The first task of the compiled function definition is to push the actual parameters back to the stack from the accumulators. Thus for both a function and a LAMBDA expression, the respective code body accesses or obtains the actual parameter from the stack.

We forgo stating the various possible stack configurations in full generality to avoid (presumably) less than transparent notation. What is in principle required can be seen by an example:

```
(DEF (A B) (G A ((LAMBDA (A) (CAR A)) B) A B))
```

This must be compiled identically to

```
(DEF (A B) (G A ((LAMBDA (A1) (CAR A1)) B) A B))
```

where the bound A of the LAMBDA expression has been renamed A1. The accessible variables of F are A and B; those of the LAMBDA expression are A1 and B. At the point of compiling the argument A of CAR A, the stack P (at run-time) will be

P	A	B	A	B
-----	-----	-----	-----	-----
actual	the first	actual	parameter	
parameters	actual parameter	corresponding		
to the call	to the call of G	to A1		
of F				

The compile-time PRUP list will be ((A,4) (A,1) (B,2)) or, using A1, ((A1,4) (A,1) (B,2)). Note the absence of a 3 since that spot holds a temporary value and not the value of an actual parameter usable in the body of the LAMBDA expression (in this example either A1 or B but not A).

Thus the compilation of the argument A of CAR (at case 3 of COMPEXP with M = -4 as it would be) produces a MOVE involving the top of the stack, namely (MOVE 1 M+4 P) = (MOVE 1 0 P), and not (MOVE 1 M+1 P) = (MOVE 1 -3 P). A compilation of B at this point would produce (MOVE 1 M+2 P) = (MOVE 1 -2 P).

After compiling the fourth, and last, actual Parameter of G, the stack will be

P A B	A CAR B A B	,
-----	-----	
actual parameters	actual parameters	
to the call of F	to the call of G	

We shall need to show that the proper run-time stack configuration is set up and maintained, and that the quantity M and the Integers in the PRUP list together produce the correct accessing from the stack P. The quantity -M gives the number of stack locations currently accessible by the function being compiled. Let us define the predicate STACKOK(M, PRUP) to mean (i) -M is the correct number of stack locations, and (ii) M and the Integers in the PRUP list at compile-time together produce the correct accessing of the stack at run-time. The definition of STACKOK includes the representation of "what the compiler knows so far" concerning the location in the stack of variables and temporary values. As part of no error checking the compiler assumes an infinite run-time stack with no tests for stack overflow. The proof accordingly makes the same assumption.

PROOF OF THE MAIN THEOREMS FOR C0

The main proof technique used for theorems 1, 2, and 4-7 is structural induction on expressions. Each theorem states what a procedure of the compiler does: theorems 1 and 7 for COMPEXP, 2 for COMPLIS, 4 for COMPANDOR, 5 for COMBOOL, and 6 for COMCONO. Each of these procedures is recursive and also can call many of the other procedures. To prove these theorems for an arbitrary expression EXP, the following induction hypothesis is used for each theorem: Theorems 1, 2, and 4-7 have all been proved for all subexpressions of EXP. To invoke one of these theorems inductively on a subexpression, it is necessary to verify that all hypotheses of that theorem are satisfied.

The length of the list X will be denoted by L X. All procedures of the compiler except for PRUP produce as values a list of compiled instructions, as may be verified by inspection (in particular noting each one-line code generation is a one-element list and otherwise the APPEND function is used). The quantities VPR and M, which appear as actual parameters to the procedures in theorems 1, 2, and 4-7, are unchanged by these procedures in view of the definition of functional evaluation.

Theorem 1 [Definition of COMPEXP(EXP, M, VPR)]. Assume the following conditions hold at the call of COMPEXP(EXP, M, VPR):

C1: EXP is an expression.

C2: $M \leq 0$ and -M is the number of stack locations currently accessible by the function being compiled.

- c3: Variables currently accessible to EXP are x_1, x_2, \dots, x_K with $K \leq -M$.
- c4: VPR is a PRUP list of K pairs (x_i, j) , $1 \leq j \leq -M$, of the currently accessible variables where the innermost occurrence (Of a formal parameter) of a duplicated variable name appears first on VPR, e.g., $((E, 7)) (B, 8) (D, 6) (A, 1) (B, 2) (C, 3))$.
- c5: At run-time the stack P contains the values of the variables and temporary values as
 $P | x_1 x_2 \dots x_{[-M]}$
 where $x_{[-M]}$ is at the top of the stack.
- c6: STACKOK(M, VPR).
- c7: EXP is an atom ($\neq \text{NIL}$, $\neq T$, $\neq \text{numeric-atom}$) \Rightarrow EXP is a variable x_i , $1 \leq i \leq K$, on the VPR list.

Result. After execution of the list, I, of instructions produced by COMPEXP, the accumulator ac1 contains V EXP. P is safe over the execution of I. Note that the accumulators are Unsafe over the execution of I.

Proof of definition of COMPEXP (under the assumption of no conditional or Boolean expressions; theorem 7 proves COMPEXP with such expressions). Structural Induction on EXP. **Basis step:** EXP is an atom, either NIL, T, a numeric-atom, or other atom. If EXP is NIL, then case 1 of COMPEXP produces $((\text{MOVEI } 1 \ 0))$ so ac1 holds $0 = V \text{ NIL}$. If EXP is T, then case 2 produces $((\text{MOVEI } 1 \ (\text{QUOTE } T)))$ so ac1 holds $(\text{QUOTE } T) = VT$. If EXP is a numeric-atom, then case 2 produces $((\text{MOVEI } 1 \ (\text{QUOTE } \text{numeric-atom})))$ so ac1 holds $(\text{QUOTE } \text{numeric-atom})$, the correct value. If EXP is an other atom, then case 3 produces $((\text{MOVE } 1 \ M + \text{CDR ASSOC}(\text{EXP}, \text{VPR}) \ P))$. By C7 let $\text{EXP} = x_i$ appear first on VPR in the pair (x_i, j) . By C4 $\text{CDR ASSOC}(\text{EXP}, \text{VPR}) = \text{CDR } (x_i, j) = j$. By C5 and C6 the instruction $(\text{MOVE } 1 \ M + j \ P)$ loads ac1 with $V x_i$. Note $1 \leq j \leq -M \Rightarrow M + 1 \leq M + j \leq 0$, i.e. a valid stack access.

Induction step: CAR EXP and CDR EXP are always defined at cases 4-7 (a total of 10 occurrences) since NOT ATOM EXP because case 3 failed. If $\text{EXP} = (\text{QUOTE } \alpha)$, then case 6 is the first to hold producing $((\text{MOVEI } 1 \ (\text{QUOTE } \alpha)))$ as required.

If $\text{EXP} = (\text{fname } \alpha)$ with fname not one of AND, OR, NOT, COND, QUOTE, then case 7 is the first to hold. EXP thus is a (non-special) function to be evaluated using arguments of the list $\alpha = (\alpha_1 \alpha_2 \dots \alpha_N)$ where $N = L \ \alpha \geq 0$. The list of instructions produced is

```
((COMPLIS(( $\alpha$ ), M, VPR))
 (LOADAC(1-N, 1))
 (SUB P CC 0 0 N N))
 (CALL N (E fname)))
```

Conditions D1-D7 (see theorem 2) for inductively invoking COMPLIS hold as follows:

D1: Definition of (a),

D2: C2,

33: C3 on U, a subpart of EXP,

D4,D5,D6: C4,C5,C6, respectively,

37: Assumption of syntactically correct input,

Using the definitions of COMPLIS and LOADAC, we obtain

```
---      ((Instructions to leave V α1 in ac1)
          (PUSH P 1)
COMPLIS      ...
          (Instructions to leave V αN in ac1)
---      (PUSH P 1)
          (MOVE 1 1-N P)
LOADAC      (MOVE 2 2-N P)
          ...
---      (MOVE N 0 P)
          (SUB P (C 0 0 N N))
          (CALL N (E fname))) ,
```

Tracing these instructions, namely

```
ac1|α1* α1* α2* ... αN* α1* fname(V α1,V α2,...,V αN)
ac2|α2* α2* undef
...
acN|αN* αN* undef
P|α1* α2* ... αN*
```

gives the desired result (including the case $N=0$) since $V \text{ EXP} = \text{fname}(V \alpha_1, V \alpha_2, \dots, V \alpha_N)$. Note that the instruction $(\text{CALL } N \text{ (E fname)})$ may be a recursive call since the standard conventions of arguments and returned value are obeyed, and the arguments are stacked (saved) by the called function. Recall that function names are forbidden as arguments SO a formal parameter name maybe called by a CALL Instruction,

Finally If $\text{EXP} = ((\text{LAMBDA } (\alpha) \beta) \epsilon)$, then only case 8 holds, Since case 7 fails, NOT ATOM CAR EXP. Let $N = L \epsilon = L \alpha$ by correct input. The list of instructions produced is

```
((COMPLIS((ε),M,VPR))
 (COMPEXP(β,M-N,APPEND(PRUP((α),1-M),VPR)))
 (SUB P (C 0 2 N N))) ,
```

Conditions D1-D7 for inductively invoking COMPLIS hold as follows:

D1: Definition of (ε), D2: C2,D3:C3 on (ε), a subpart of EXP,

D4,D5,D6: C4,C5,C6, respectively. D7: Syntactically correct input.

Conditions C1-C7 for inductively invoking COMPEXP hold as follows:

C1: β is an expression by the syntax definition involving LAMBDA.
 C2: $M-N \leq 0$ since $M \leq 0$ and $N \geq 0$. There are now $-(M-N) = -M+N$ stack locations currently accessible.
 C3: Variables currently accessible to β are x_1, x_2, \dots, x_{K+N} , i.e. there are now $K+N$ variables allowed in β , $K+N \leq -M+N$ since $K \leq -M$.
 C4: Definition of PRUP and C4, C5, and C6 applied to VPR. The new pairs are put first. The new indices are $1-M = -M+1$ through $-M+N$.
 C5: C5 for x_1, \dots, x_{-M} together with $\text{COMPLIS}((\epsilon), M, \text{VPR})$ for $x_{-M+1}, \dots, x_{-M+N}$.
 C6: C6, C4 just above, and C5 just above.
 C7: Syntactically correct input and the augmented PRUP list.

Hence tracing these instructions, namely

```

ac1 | x[-M+1]* ... x[-M+N]* V EXP
    P | x1 x2 ... x[-M] x[-M+1]* ... x[-M+N]*
  
```

gives the desired result (including the case $N = 0$), since COMPLIS essentially makes the substitutions at $v \in l$ and then COMPEXP computes V_p which is now $V \cdot \text{EXP}$.

In all cases the stack P is safe over the execution of l . Note that VPR remains unaltered even in the LAMBDA case because here the augmented PRUP list in the call to COMPEXP is a copy only for that recursive call when that call finishes the outer VPR list is intact. *

Theorem 3 [Definition of $\text{COMPLIS}(U, M, \text{VPR})$]. Assume the following conditions hold at the call of $\text{COMPLIS}(U, M, \text{VPR})$:

D1: $U = (u_1, u_2, \dots, u_N)$ is a list of arguments,
 D2: COMPEXP 's C_2 ,
 D3: Variables currently accessible to the members of U are x_1, x_2, \dots, x_K with $K \leq -M$,
 D4, D5, D6: COMPEXP 's C_4, C_5, C_6 , respectively,
 D7: COMPEXP 's C_7 with EXP replaced by U .

Result, $\text{COMPLIS} = ((\text{instructions to leave } V u_1 \text{ in ac1})$
 (PUSH P 1)
 \vdots
 (instructions to leave $V u_N$ in ac1)
 (PUSH P 1)).

Proof of definition of COMPLIS . Structural Induction on U .
 Basis step: U is NULL whence $\text{COMPLIS} = \text{NIL}$. Induction step: Since $U \neq \text{NIL}$, $\text{COMPLIS}(U, M, \text{VPR})$

```

= ((COMPEXP(u1, M, VPR))
   (PUSH P 1)
   (COMPLIS((u2 ... uN), M-1, VPR))) .
  
```

Conditions C1-C7 for inductively Invoking COMEXP hold by D1-D7, respectively. Hence invoking COMEXP shows

(COMEXP(u1,M,VPR)) = (Instructions to leave V u1 in ac1)

with the stack P safe, (PUSH P 1) stacks V u1 on the top of P, Conditions D1-D7 for invoking the induction hypothesis for COMPLIS hold as follows:

D1: By D1 for U.

D2: By D2 and (PUSH P 1) which means there are now $-(M-1) = -M+1$ stack locations, the top one being a temporary value,

D3: By D3 ($K \leq -M > K \leq -M+1$),

D4: By D4.

D5: By D5 and (PUSH P 1), P is P[X1 X2 ... X[-M] V u1 ,

D6: By D6 and D5 just above,

D7: By D7.

Hence the induction hypothesis shows $\text{COMPLIS}((u2 \dots uN), M-1, VPR) =$

((Instructions to leave V u2 In ac1)
(PUSH P 1)
...
(Instructions to leave V uN In ac2)
(PUSH P 1)),

Hence $\text{COMPLIS}(U, M, VPR) =$

((instructions to leave V u1 In ac1)
(PUSH P 1)
...
(Instructions to leave V uN in ac1)
(PUSH P 1)), •

Theorem 3 [Correctness of the compiler]. Let $A = a1 a2 \dots aN$ be an arbitrary list of actual parameters. Starting with ac holding a_i , $1 \leq i \leq N$, and after execution of the list, I, of instructions produced by $\text{COMP}(\text{NAME}, (\text{args}), \text{body})$ we have

$V((\text{DE NAME}(\text{args}) \text{ body}) A) = \text{contents of ac1}$

and the stack P is Safe over the execution of I.

Proof. Let $N = L(\text{args})$. $\text{COMP}(\text{NAME}, (\text{args}), \text{body})$

= ((LAP NAME SUBR)
(MKPUSH(N,1))
(COMEXP(body, -N, PRUP((args), 1)))
(SUB P (C 0 0 N N))
(POPJ P)
NIL)

```

      = ((LAP NAME SUBR)
---      (PUSH P 1)
MKPUSH      (PUSH P 2)
      ...
---      (PUSH P N)
COMPEXP      (Instructions to leave V body in ac1)
---      (SUB P (C 0 0 N N))
      (POPJ P)
      NIL )

```

by using the definitions of MKPUSH and COMPEXP although it remains to show that MKPUSH and COMPEXP may be invoked. Since $N \geq 0$ we may invoke MKPUSH. The conditions C1-C7 for COMPEXP hold as follows:

- C1: body is an expression by the assumption of syntactically correct input.
- C2: $-N = -\text{LENGTH}(\text{args}) \leq 0$. $-N = N$ is the correct number of stack locations since precisely $L(\text{args})$ locations are accessible.
- C3: the accessible variables are a_1, a_2, \dots, a_N .
- C4: By definition of $\text{PRUP}(\text{args}, 1)$.
- C5: By the number N of $(\text{PUSH P } i)$ instructions.
- C6: $\text{STACKOK}(-N, \text{PRUP})$ holds by the definition of PRUP and the order of the PUSH instructions.
- C7: By syntactically correct input and the definition of $\text{PRUP}(\text{VARS}, 1)$.

Thus starting with acf holding a_i for $1 \leq i \leq N$, we have the trace

```

ac1|a1* V body
ac2|a2* undef
...
acN|aN* undef
P|a1* a2* ... aN* ,

```

Since $V \text{ body} = ((\text{DE NAME}(\text{args}) \text{ body}) A)$ and since the stack P is safe, the result is proved. (If conditional and Boolean expressions are allowed, then theorem 7 is needed.) •

Theorem 4 [Definition of $\text{COMPANDOR}(U, M, L, \text{FLG}, \text{VPR})$]. Assume the following conditions hold at the call of $\text{COMPANDOR}(U, M, L, \text{FLG}, \text{VPR})$:

- E1: $U = (u_1 u_2 \dots u_N)$ is a list of Boolean expressions,
- E2: COMPEXP's C2,
- E3: COMPLIS's D3,
- E4, E5, E6: COMPEXP's C4, C5, C6, respectively,
- E7: COMPLIS's D7,
- E8: L is a label.
- E9: FLG is T or NIL,

Result, COMPANDOR produces a list, I, of instructions given by

```

FLG | Algol equivalent of I
-----|-----
NIL | if NOT u1 then go to L;
    | if NOT u2 then go to L;
    | ...
    | if NOT uN then go to L;
at-a-|-----
    | if u1 then go to L;
    | if u2 then go to L;
    | ...
    | if uN then go to L;

```

with the statement labeled L not in I, P is safe over the execution of I,

Proof of definition of COMPANDOR, Structural Induction on U,
 Basis step: U is NULL whence COMPANDOR = NIL, Induction step:
 Assume FLG = T, COMPANDOR(U,M,L,FLG,VPR)

```

= ((COMBOOL(u1,M,L,FLG,VPR))
   (COMPANDOR((u2 ... uN),M,L,FLG,VPR))) by definition of
      COMPANDOR since U ≠ NULL

= ((if u1 then go to L;)
   (COMPANDOR((u2 ... uN),M,L,FLG,VPR))) by Inductively
      Invoking COMBOOL on the Boolean expression u1

= ((if u1 then go to L;)
   (if u2 then go to L;)
   ...
   (if uN then go to L;)) by Inductively Invoking COMPANDOR
      on the list (u2 ... uN); E2-E7 hold prior to
      invoking COMPANDOR since P is safe over "if u1
      then go to L;" and both M and VPR are unaltered
      by COMBOOL,

```

L is in neither the first instruction nor in instructions 2 through N whence L is outside I. Similarly the stack P is safe. The case FLG = NIL is proved similarly. •

Theorem 5 [Definition of COMBOOL(P,M,L,FLG,VPR)], Assume the following conditions hold at the call of COMBOOL(P,M,L,FLG,VPR):

F1: P is a Boolean expression,
 F2-F7: COMEXP's C2-C7, respectively, with EXP replaced by p,
 F8: L is a label,
 F9: FLG is T or NIL.

Result, COMBOOL produces a list, I, of instructions given by

```
FLG | Algol equivalent of I
-*a- | -----
NIL I if NOT P then go to L;
T | If P then go to L;
```

4th the statement labeled L not in I, P is safe over the execution of I,

Proof of definition of COMBOOL, Structural Induction on P.
Assume FLG = T, Basis step: P is an atom, COMBOOL(P,M,L,FLG,VPR)

```
= ((COMPEXP(P,M,VPR))
  (JUMPN 1 L)) by case 1 of COMBOOL
```

```
≠ ((Instructions to leave V P in ac1)
  (JUMPN 1 L)) by "Inductively" Invoking COMPEXP (more
  precisely, by repeating on the atom P the basis
  step of the proof of COMPEXP; induction is
  Invalid since the P in COMPEXP is not a sub-
  structure of P in COMBOOL)
```

```
= ((if P then go to L)) by checking 2 cases,
```

Induction step: CAR P and CDR P are always defined at cases 2-5
since NOT ATOM P because case 1 failed, Also CADR P is defined at
case 4 since the NOT operator must have an argument,

If P = (AND α), then from case 2b (with FLG = T) COMBOOL

```
= ((COMPANDOR((α),M,L1,NIL,VPR))
  (JRST 0 L) [the 0 is redundant]
  L1) by letting GENSYM() be the label L1 ≠ L
  since each call to GENSYM gives a unique
  value
```

```
= ((if NOT α1 then go to L1)
  (if NOT α2 then go to L1)
  ...
  (if NOT αN then go to L1)
  (JRST 0 L)
  L1) by inductively invoking COMPANDOR on (α),
  a Boolean list
```

```
= ((if P then go to L; L1:) by checking cases that define
  AND (including evaluation only until the
  first NIL α1 and the case (AND) with NULL
  α),
```

If P = (OR α), then from case 3a (with FLG = T) COMBOOL

= (COMPANDOR((α),M,L,T,VPR))

= ((if α_1 then go to L)
(if α_2 then go to L))

((if α_n then go to L))) by inductively invoking COMPANDOR
on (α), a Boolean list

= (if P then go to L) by checking cases that define OR
(including evaluation only until the first
non-NIL α_i and the case(OR) with NULL α).

If P = (NOT α_1), then from case 4 COMBOOL

= (COMBOOL((α_1),M,L,NOT FLG,VPR))

= (if NOT α_1 then go to L) by inductively invoking COMBOOL
on (α_1), a one-element Boolean list

= (if P then go to L) by definition of P.

If P is any other Boolean expression, then case 5 yields

((COMPEXP(P,M,VPR))
(JUMPN 1 L)).

Immediate inductive invoking of COMPEXP is invalid because the P in COMPEXP is *not* a substructure of P in COMBOOL. But control's reaching case 5 of COMBOOL means P is not an atom (case 1) and means CAR P is neither AND, OR, NOT (cases 2-4). Thus COMPEXP(P,M,VPR) will be computed by one of its cases 5-8 all of whose procedures are called with substructures of P. (It is crucial to avoid case 4 of COMPEXP to avoid the cycle COMBOOL(P,...) \rightarrow COMPEXP(P,...) \rightarrow COMBOOL(P,...).) COMPEXP(P,M,VPR) may be calculated by repeating the proof of cases 5-8 on P (see theorems 7 and 1); this yields the same calculation as the basis step for COMBOOL. Since the definition of GENSYM guarantees unique labels being generated, the label L is not in the "instructions to leave V P in act."

The case FLG = NIL is proved similarly. •

Theorem 6 [Definition of COMCOND(U,M,L,VPR)]. Assume the following conditions hold at the call of COMCOND(U,M,L,VPR):

- G1: U = ((u1 u2) (u3 u4) ... (u[2N-1] u[2N])) is a list of pairs of expressions, the first of each pair being a Boolean expression,
- G2-G7: COMPEXP's C2-C7, respectively, with EXP replaced with u,
- G8: L is a label,

Result. COMCOND gives a list, I, of instructions equivalent to the Algo!

$ac1 := \text{if } u1 \text{ then } u2 \text{ else if } u3 \text{ then } u4 \dots \text{ else}$
 $\text{if } u[2N-1] \text{ then } u[2N]; L;$

P is safe over the execution of I. If no $u[2]-1$ is non-NIL, the value in $ac1$ is undefined. In other words $ac1 := V \text{ COND-expression}$.

Proof of definition of COMCOND. Structural induction on U.
 Basis step: U is NULL whence COMCOND produces, as required, just the label L. Induction step: NOT NULL U and correct syntax imply CAAR U, CADAR U, and CDR U are always defined. COMCOND(U, M, L, VPR)

= ((COMBOOL(u1, M, L1, NIL, VPR))
 (COMPEXP(u2, M, VPR))
 (JRST L)
 L1
 (COMCOND(((u3 u4) ... (u[2N-1] u[2N])), M, L, VPR)))
 by letting GENSYM() be the label L1 \neq L

= ((if NOT u1 then go to L1))
 (Instructions to leave V u2 in ac1)
 (JRST L)
 L1
 (ac1 := if u3 then u4 ... else if u[2N-1] then u[2N]; L))
 by inductively invoking COMBOOL, COMPEXP, and COMCOND

= (ac1 := if u1 then u2 ... else if u[2N-1] then u[2N]; L)
 by checking cases involving V u1.

P is safe as required. The case of no $u[2]-1$ being non-NIL gives an undefined result as required (in particular for $N = 0$), •

Theorem 7. COMPEXP(EXP, M, VPR) as defined in theorem 1 also holds for conditional and Boolean expressions.

Proof. (An addition to the proof of theorem 1.) Basis step: Vacuous. Induction step: If EXP = (Boolean α) with Boolean one of AND, OR, NOT, then case 4 is the first to hold. COMPEXP(EXP, M, VPR)

= ((COMBOOL(EXP, M, L1, NIL, VPR))
 (MOVEI 1 (QUOTE T))
 (JRST 0 L2)
 L1
 (MOVEI 1 0)
 L2) where L1 \neq L2 are the two GENSYM() labels

= ((if NOT EXP then go to L1))
 (MOVEI 1 (QUOTE T))
 (JRST 0 L2)
 L1
 (MOVEI 1 0)
 L2) by repeating the proof of cases 2-4, all

involving substructures, of COMBOOL(EXP,..)
since case 4 of COMPEXP means CAR EXP is
either AND, OR, NOT,

If V EXP = T, then ac1 holds (QUOTE T) as required since the (MOVEI 1
(QUOTE T)) and the (JRST 0 L2) instructions are executed. If V EXP =
NIL, then ac1 holds 0 as resulted since control goes to L1 and the
(MOVEI 1 0) is executed,

If EXP = (COND α), then case 5 is the first to hold, COMPEXP =
COMCOND(α), M, L, VPR) using the label L for GENSYM(), Invoking
COMCOND inductively shows the required value, according to the
definition of COND, is in ac1,•

TERMINATION OF THE COMPILER C0

Except to COMP in theorem 3, add the statement "and the
procedure terminates" to the result of each procedure definition of
the compiler. The induction hypothesis will show termination of each
procedure call on a substructure. The induction step is now reduced
to essentially "straight-line code" which terminates, COMP terminates
since MKPUSH and COMPEXP do,

To show that COMBOOL and COMPEXP terminate when one is called
from the other on the original structure, We can repeat a proof Part
as was done in the proofs of theorems 5 and 7,

DISCUSSION OF THE PROOF P0

The process of constructing this proof may be viewed as
discovering enough of the assumptions about the input and the
programming conventions used in writing the compiler, as stating
them, and as proving them to be preserved or consistently followed
over all the procedures of the compiler. The successful
factorization involving conditional and Boolean expressions was
useful in doing this. The recursion of the compiler has been handled
by the statements of the theorems, including three dots (...), as
needed, and by the use of structural induction. In addition, some
lessons of top-down programming (Dijkstra 1970), stepwise refinement
(Wirth 1971), and Hoare's (1971) approach were applied in the proof
process although informally,

It is noteworthy that the proof process uncovered no errors in
the compiler. A previous version of this paper omitted completely
numeric-atoms although condition C7 (then written without the clause
"# numeric-atom") unintentionally excluded them. Diffie noticed
their omission when the compiler aborted while compiling a factorial
function. Since numeric-atoms are needed for self-compilation, case
2 of COMPEXP was changed to include numeric-atoms. No other changes
were made to the compiler. The previous version of this paper did
not exclude the use of NIL, T, and numeric-atoms as formal parameters
nor the use of function names as arguments. They must be excluded

since the compiler fails on these inputs.

Despite the compiler's being written purely functionally, this proof may be usefully viewed as employing inductive assertions. When applied to recursive procedures of the kind in the compiler, the method verifies the conditions necessary for calling a procedure (including a recursive call). The result of the procedure is then used to show what is true after the call (even if the procedures are called merely as arguments to the APPEND function). This is the same way a standard iterative program is proved.

Unexplored so far are the implications for automatic proof checking, of the length of this informal, but hopefully rigorous proof. Next is the Proof P4.

THE COMPILER C4 AND PROOF of CORRECTNESS P4

The input to the compiler C4 and the overall statement of correctness are the same as for C0. The compiler C4 is similar in structure to C0, has twice as many lines of code as C0, and produces about half as many instructions for a given function as C0. In response the proof P4 contains eleven new theorems and lemmas (Theorems 8-12 and Lemmas 4-9) corresponding to the eleven new functions in C4. Also P4 contains modifications to the proofs (mainly additional cases) of theorems 1, 3, and 5-7 reflecting the changes in C4 to the functions of C0. The similar structure allows much of the proof P0, without change, to become a part of P4. In particular, the statements of lemmas 1 and 2 and theorems 1-7 are unchanged (LOADAC, the subject of lemma 3, is a completely new function) because the generally more efficient compiled code of C4 accomplishes the same overall effect as does the code of C0. The proofs of the new theorems and the proofs of modifications in P4 are the "same kind" of proofs as in P0. (Diffie has self-compiled C4 successfully also.)

McCarthy described the three main differences between C0 and C4 in a writeup. The second difference is the main source of improvement in the compiled code as well as the main reason for the length of P4.

(i) When the argument of CAR or CDR is a variable, C4 compiles a (HLRZ@ 1 | P) or (HRRZ@ 1 | P) which gets the result through the stack without first compiling the argument into an accumulator,

(ii) When C4 has to set up the arguments of a function in the accumulators. On general, C4 must compute the arguments one at a time and save them on the stack, and then load the accumulators from the stack, however, if one of the arguments is a variable, is a quoted expression, or can be obtained from a variable by a chain of CARS and CDRs, then it need not be computed until the time of loading accumulators since it can be computed using only the accumulator in which it is wanted,

(iii) C0 computes Boolean expressions badly and generates many unnecessary labels and JRSTs. C4 is more sophisticated about this.

c4 uses four additional PDP-10 instructions: HLRZ0, HRRZ0, CAME, and CAMN. The first two are used, with the e-sign denoting indirect reference, to obtain CAR and CDR, respectively. An assumption of P4 is that the instruction HLRZ0 means $c(ac) \leftarrow CAR(c(\langle ef \rangle))$ and that HRRZ0 means $c(ac) \leftarrow CDR(c(\langle ef \rangle))$. Because CAR and CDR are compiled open rather than closed, as would be the case for an arbitrary function call, it must be explicitly emphasized that CAR and CDR of T, NIL, or numeric-atom are considered incorrect input. Since NULL and EQ are compiled open, the values of both must be explicitly defined for P4:

$V \text{ (NULL EXP)} = T \text{ iff } V \text{ EXP} = \text{NIL}$

$V \text{ (EQ EXP1 EXP2)} = T \text{ iff } V \text{ EXP1} = V \text{ EXP2}$

with these definitions and motivation, the proof P4, organized in bottom-up style, follows.

The listings of the two compilers were checked by hand to discover the differences. The same set of differences was obtained when the listings were computer-compared by a file comparison utility program. These differences showed where new theorems were needed and where old proofs needed modification.

Lemma 4 [Definition of CCCHAIN(EXP)]. Assume EXP is a non-atomic expression, $\text{CCCHAIN(EXP)} = T$ if and only if EXP is of the form

$(\text{CPR} (\text{CPR} (...(\text{CPR } \alpha))))$

with at least one β . Each β is either A or D (thus producing CAR or CDR) and α is an atom. In other words, $\text{CCCHAIN(EXP)} = T$ iff EXP is a car-cdr chain.

Proof. Induction on the number N of leading β 's in EXP. Basis steps: If $N = 0$ then CCCHAIN gives NIL because CAR EXP is neither CAR nor CDR. If $N = 1$ then $\text{EXP} = (\text{CPR } \alpha)$. The result is T because CPR is CAR or CDR and α is an atom. CCCHAIN α is not called.

Induction step: If $\text{EXP} = (\text{CPR1R} (\text{CPR2R} (...(\text{CPRNR } \alpha))))$ with $N \geq 2$, then CPR1R is CAR or CDR so the left part of the AND is true. Since $N \geq 2$, $(\text{CPR2R} (...(\text{CPRNR } \alpha)))$ is not an atom. CCCHAIN may be invoked inductively, yielding T and hence CCCHAIN EXP gives T. •

Lemma 5 [Definition of CLASS1(U, V)]. Input assumptions:

U is a list of expressions (u1 u2 ... uN),
V is an S-expression.

Result. Let c_i be the classifying integer of u_i , namely

u_i	i	c_i
T, NIL, numeric-atom		0
other atom	1	1
quoted expression		2
car-cdr chain		3
other expression	1	4

$CLASS1(U, V) = (c_N, u_N), \dots, ((c_2, u_2), ((c_1, u_1), V))$.

Proof. Structural induction on U . Basis step: $NULL$ u gives V . Induction step: $CLASS1(CDR U, (c_1, u_1), V) = (c_N, u_N), \dots, ((c_2, u_2), ((c_1, u_1), V))$. Note that u_1 in $CCCHAIN$ u_1 is non-atomic since the first test for $ATOM$ u_1 failed. For the special case $V = NIL$ the result reduces to the list of pairs $((c_N, u_N) \dots (c_2, u_2) (c_1, u_1))$, •

Lemma 6 [Definition of $CLASS2(U, V, FLG)$]. Input assumptions:

U is a list of pairs $((c_N, u_N) \dots (c_2, u_2) (c_1, u_1))$ with c_i as defined in $CLASS1$.

V is an S-expression.

$FLG = T$ or NIL .

Result. Let j be the greatest integer, if any, such that $c_j = 4$ in U .

FLG	Result
T	$((c_1, u_1), ((c_2, u_2) \dots ((c_N, u_N), V)))$ with c_j now 5
NIL	$((c_1, u_1), ((c_2, u_2) \dots ((c_N, u_N), V)))$ with c_j still 4

In words, the list of pairs is reversed and the first 4 is changed to 5.

Proof. Structural induction on U . Basis step: $NULL$ u gives v . Induction step: If $FLG = T$ and $c_N = 4$ then $CLASS2(CDR U, (5, u_N), V, NIL) = ((c_1, u_1), ((c_2, u_2) \dots ((5, u_N), V)))$ with c_1, c_2, \dots, c_{N-1} as in U . If $FLG \neq T$ or $c_N \neq 4$ then $CLASS2(CDR U, (c_N, u_N), V, FLG) = ((c_1, u_1), ((c_2, u_2) \dots ((c_N, u_N), V)))$ with the c_i 's as in the table of the result. Again, when $V = NIL$, the result reduces to the list of pairs $((c_1, u_1) (c_2, u_2) \dots (c_N, u_N))$, •

Lemma 7 [Definition of $CLASSIFY(U)$]. Assume $U = (u_1 u_2 \dots u_N)$. Let d_i be the classifying integer of u_i as in $CLASS1$ except the last other expression has d_i of 5 instead of 4. Then $CLASSIFY(U) = ((d_1, u_1) (d_2, u_2) \dots (d_N, u_N))$, •

Proof. Composition of $CLASS1$ with V as NIL and $CLASS2$ with V as NIL and FLG as T . •

Theorem 8 [Definition of COMPLIS(Z, M, K, VPR)], Input assumptions:

Z is a CLASSIFY'ed list of pairs ((dK,uK) (d[K+1],u[K+1])... (dN,uN)). Conditions D1-D7 of COMPLIS of Theorem 2.

Result. Let e1, ..., e[J-1] denote those subscripts, if any, in Z for which di is equal to 4, and let ej denote the one di, if any, equal to 5.

```
COMPLIS = ((Instructions to leave V u[e1] in ac1)
           (PUSH P 1)
           '
           (Instructions to leave V u[e[J-1]] in ac1)
           (PUSH P 1)
           (Instructions to leave V u[ej] in ac[ej]))
```

Note that this COMPLIS is a new function from that of Theorem 2. The function STACKUP(U, M, VPR) is identical to the old COMPLIS.

Proof. Structural Inductfon on Z. Basis step: NULL Z gives NIL. Induction step: If dK = 4 then e1 = K, COMPEXP(uK, M, VPR) inductively produces

```
(Instructions to leave V u[e1] in ac1)
```

In view of the (PUSH P 1), then COMPLIS(((d[K+1],u[K+1])... (dN,uN)), M-1, K+1, VPR) inductively completes the desired result.

If dK = 5 then ej = K and there are no (more) 4's, COMPEXP(uK, M, VPR) inductively Produces

```
(Instructions to leave V u[ej] in ac1)
```

If K = 1 (i.e. ej = 1), no further instruction is needed nor generated because V u[ej] is already in ac1. Otherwise if K ≠ 1, the instruction (MOVE K 1) is generated to leave V u[ej] in ac[ej] = ac[K].

If dK is neither 4 nor 5, COMPLIS(((d[K+1],u[K+1]) ... (dN,uN)), M, K+1, VPR) inductively gives the desired result. •

Theorem 9 [Definition of COMPC(EXP, N2, M, VPR)], Input assumptions:

EXP is a car-cdr chain (C@1R (C@2R (... (C@NR α))) where N ≥ 1; each βi is either A or D; and α is an atom ≠ T, NIL, numeric-atom, Conditions C2-C6 and C7 for afrom COMPEXP of Theorem 1,

```
Result. COMPC = ((ac[N2] := C@1R ac[N2])
                  (ac[N2] := C@2R ac[N2])
```

a ..

(ac[N2] := CPNR α))

Only accumulator N2 is used,

Proof, Induction on the number J of β's in EXP. Define ei to be L or R according as βi is A or D, basis step: If N = 1 then EXP = (CP1R α). Since ATOM α, COMPC produces

((He1RZ@ N2 M+CDR ASSOC(α, VPR) P))

which is ((ac[N2] := CP1R α)), the last line of the result, Induction step: If N ≥ 2 then NOT ATOM (CP2R (... (CPNR α))). Hence COMPC produces

(He1RZ@ N2 N2)
COMPC((CP2R(... (CPNR α))), N2, M, VPR)

which, invoking COMPC inductively, becomes

((ac[N2] := CP1R ac[N2 3])
(ac[N2] := CP2R ac[N2]))
((ac[N2] := CPNR α))

Incidentally, the assumption that EXP is a car-cdr chain makes unnecessary the error check at the first line of COMPC.

Theorem 10 [Definition of LOADAC(Z, M2, N2, M, VPR)], Input assumptions:

Z is a CLASSIFY'ed list of pairs.

Z = ((d[N2], u[N2]) (d[N2+1], u[N2+1]) ... (dN, uN))

Conditions D1-D7 of COMPLIS of Theorem 2.

Let e1, e2, ..., e[1-M2] denote those subscripts, if any, in Z for which d[i] is equal to 4. The stack P contains the values of the 1-M2 u[e]'s as follows

P | V u[e1] V u[e2] ... V u[1-M2]

Let ej, with j > 1-M2, denote the one, if any, equal to 5. Assume ac[ej] holds V u[ej].

Result. LOADAC = ((Instructions to leave V u[N2]) In ac[N2])
(Instructions to leave V u[N2+1] In ac[N2+1])
...
(Instructions to leave V uN In acN))

Each line of instructions uses only the accumulator mentioned. The stack P is unaltered. (The ej-th line involving ac[ej] is missing.)

Proof, Structural induction on Z. Basis step: NULL Z gives NIL. Induction step: Six cases based on the classifying integer d[N2]. If d[N2] = 1 then u[N2] is an other atom, LOADAC produces


```
(MOVE N2 M+CDR ASSOC(u[N2], VPR) P)
, LOADAC(((d[N2+1].u[N2+1]) ... (dN.uN)), M2, N2+1, M, VPR)
```

The MOVE instruction leaves $V\ u[N2]$ in $ac[N2]$ using only $ac[N2]$. Inductively the LOADAC part completes the result including the unalteration of the stack. The use of the infix dot follows the conventions that the value of LOADAC is a list of instructions.

If $d[N2] = 0$ or 2 then $u[N2]$ is either T, NIL, or numeric-atom; or a quoted expression. The proofs are each similar to the case $d[N2] = 1$. The generated instructions are, respectively,

```
(MOVEI N2 (QUOTE u[N2]))
```

and

```
(MOVEI N2 u[N2])
```

with each followed by the same LOADAC term as in the first case. Both MOVEI instructions leave $V\ u[N2]$ in $ac[N2]$ using only $ac[N2]$, and again the LOADAC term inductively completes the result.

If $d[N2] = 3$ then $u[N2]$ is a car-cdr chain. Syntactically correct input implies the atom a at the end of the chain is neither T, NIL, nor numeric-atom. Thus COMPC may be invoked. Since a car-cdr chain is executed from right to left, the REVERSE function is needed. LOADAC produces

```
((ac[N2] := CDR a)
 (ac[N2] := CDR ac[N2])
 (ac[N2] := CDR ac[N2])
 (same LOADAC term as first case))
```

The first N lines are

```
(Instructions to leave  $V\ u[N2]$  in  $ac[N2]$ )
```

and the LOADAC term inductively completes the result.

If $d[N2] = 5$ then $ac[N2]$ is not altered. LOADAC(((d[N2+1].u[N2+1]) ... (dN.uN)), 1, N2+1, M, VPR) inductively gives the result. (The constant 1 as the second argument in this call to LOADAC means $1-M2 = 1-1 = 0$, i.e. the stack input condition of LOADAC is vacuous.)

Finally, if $d[N2] = 4$ then the last test of LOADAC produces

```
(MOVE N2 M2 P)
```

which, using only $ac[N2]$, leaves $V\ u[N2]$ in $ac[N2]$ because there are $1-M2 = -M2+1$ of the $(V\ u[i])$'s in the stack. LOADAC(((d[N2+1].u[N2+1]) ... (dN.uN)), M2+1, N2+1, M, VPR)

inductively completes the result since there is now one fewer 4 in the remaining $d[N2+1] \dots dN$. Even though the stack is unaltered, the stack segment of interest is now from $Vu[e2]$ to $Vu[1-M2]$ which the stack input condition inductively rennumbers as $Vu[e1]$ to $Vu[-M2]$. •

Lemma 8 [Definition of $CCOUNT(Z)$]. Assume Z is a CLASSIFY'ed list of pairs $((d1, u1) (d2, u2) \dots (dN, uN))$. $CCOUNT$ gives the number of d_i 's that are 4. This number is denoted by #4.

Proof. Structural induction on Z . **Basis step:** NULL Z gives 0. **Induction step:** If $d1 = 4$ then $1 + CCOUNT((d2, u2) \dots (dN, uN))$ inductively gives the result. If $d1 \neq 4$ then $CCOUNT((d2, u2) \dots (dN, uN))$ inductively gives the result. •

Lemma 9. If $N \geq 0$ then $SUBSTACK N$ is the same function as $LIST LIST('SUB, 'P, LIST('C, 0, 0, N, N))$.

Proof. If $N = 0$ then NIL is $LIST LIST('SUB, 'P, LIST('C, 0, 0, 0, 0))$. If $N > 0$ then it is clear. •

Theorem 11 [Definition of $COMPLISA(U, M, VPR)$]. **Input assumptions:**

$U = (u1 u2 \dots uN)$ is a list of arguments,
Conditions D2-D7 of $COMPLIS$ of Theorem 2,

Result. $ac1$ holds Vu_i for $1 \leq i \leq N$. The stack P is safe over the output of $COMPLISA$,

Proof. $COMPLIS(CLASSIFY U, M, 1, VPR)$ places the class 4 arguments on the stack in the order required for $LOADAC$. $COMPLIS$ also leaves the class 5 argument, say u_j , in ac_j . It is permissible to invoke

$LOADAC(((d1, u1) (d2, u2) \dots (dN, uN)), 1 - \#4, 1, M - \#4, VPR)$

since (i) there are now $-(M - \#4) = -M + \#4$ accessible stack locations, (ii) there are $1 - (1 - \#4) = \#4$ of the d_i 's which are 4, (iii) the stack P contains the class 4 arguments in the proper order by the result of $COMPLIS$, and (iv) ac_j holds Vu_j by the last line of the result of $COMPLIS$. After $SUBSTACK \#4$, the result is established,

The order of first $COMPLIS$ and then $LOADAC$ avoids the need to stack a non-class 4 argument since after the class 5 argument is computed by $COMPLIS$, $LOADAC$ may assume the safety of all $ac_i, 1 \leq i \leq N2$. •

Theorem 12 [Definition of $COMPANDOR1(U, M, L, L2, FLG, VPR)$]. **Input assumptions:**

$U = (u_1 u_2 \dots u_N)$,
 Conditions E1-E9 of COMPANDOR of Theorem 4,
 L_2 is a label different from L .

Result. COMPANDOR1 produces a list, I , of instructions given by

```

FLG | Algor equivalent of I
-----
NIL | If NOT  $u_1$  then go to  $L$ ;
    | If NOT  $u_2$  then go to  $L$ ;
    |
    | If NOT  $u_{[N-1]}$  then go to  $L$ ;
    | If  $u_N$  then go to  $L_2$ ;
    -----
    T | If  $u_1$  then go to  $L$ ;
      | If  $u_2$  then go to  $L$ ;
      |
      | If  $u_{[N-1]}$  then go to  $L$ ;
      | If NOT  $u_N$  then go to  $L_2$ ;
  
```

If, however, U is NULL then the Algor equivalent produced is "go to L_2 ;" The statements labeled L and L_2 are not in I . P is safe over the execution of I .

Proof. Structural Induction on U . NULL U gives "go to L_2 ;"
 induction step: Assume $FLG = T$. If NULL $(u_2 \dots u_N)$, i.e. $N = 1$,
 then

COMPANDOR1 = COMBOOL(u_1 , M , L_2 , NIL, VPR)

= if NOT u_1 then go to L_2 ;

as required. if NOT NULL $(u_2 \dots u_N)$, i.e. $N \geq 2$, then

```

((COMBOOL( $u_1$ ,  $M$ ,  $L$ ,  $FLG$ ,  $VPR$ ))
 (COMPANDOR1( $(u_2 \dots u_N)$ ,  $M$ ,  $L$ ,  $L_2$ ,  $FLG$ ,  $VPR$ ))
  
```

inductively gives the result. Note that $(u_2 \dots u_N)$ is not NULL in the inductive call. The uniqueness of the label generation mechanism will help show that the labels L and L_2 are outside I . The case $FLG = NIL$ is essentially identical.*

Theorem 13 [Definition of COMBOOL(P , M , L , FLG , VPR)]. Input assumptions are the same as COMBOOL of Theorem 5. COMBOOL produces a list, I , of instructions given by (the same as Theorem 5)

```

FLC | Algor equivalent of I
-----
NIL | If NOT  $P$  then go to  $L$ ;
    |
    -----
    T | If  $P$  then go to  $L$ ;
  
```

with the statement labeled L not in I, P is safe over the execution of I,

Proof. (Modifications to the proof of theorem 5.) Assume FLG = T. Add a case P = T which from case 0.1 produces (JRST 0 L) as required. Add a case P = (EQ α β) with α and β expressions. Inductively invoke COMPLISA((α β), M, VPR). COMBOOL produces from case 1.1

```
((ac1 holds V  $\alpha$ )
 (ac2 holds V  $\beta$ )
 (CAMN 1 2)
 (JRST 0 L))

= ((if (EQ  $\alpha$   $\beta$ ) then go to L))

= ((if P then go to L;))
```

Modify the P = (AND α) case. If α is non-NULL then after evaluating COMPANDOR1((α), M, L1, L, NIL, VPR), the result follows by noting the equivalence of

```
((If NOT uN then go to L1))
 (JRST L)
 L1)
```

and

```
((if uN then go to L;))
 L1)
```

If α is NULL, then ((JRST L) L1) results in both instances,

Under the assumption FLG = T, the P = (OR α) case is unchanged,

Add the case P = (NULL α) with α an expression, COMBOOL produces from case 4.1

```
((COMPEXP(( $\alpha$ ), M, VPR))
 (JUMPE 1 L))

= ((Instructions to leave V  $\alpha$  in ac1)
 (JUMPE 1 L))

= ((if P then go to L;))
```

These cases with FLG = NIL are proved similarly. The tests in COMBOOL are slightly different: T is treated separately rather than as an atom; the EQ and NULL functions are treated separately rather than as arbitrary functions in the last test. These differences do not affect the result of COMBOOL. *

Theorem 14 [Definition of COMCOND(U, M, L, VPR)], Same as COMCOND of Theorem 6,

Proof, To the proof of Theorem 6 add two cases to the induction step corresponding to the second and third tests of COMCOND. The second test asks if the pair (u1 u2) is the pair ((NULL α) NIL), If so COMCOND produces

```
((COMPEXP( $\alpha$ , M, VPR))
 (JUMPE 1 L)
 (COMCOND((u3 u4) ... (u[2N-1] u[2N])), M, L, VPR)))

= ((instructions to leave V  $\alpha$  in ac1)
 (JUMPE 1 L)
 (ac1 := if u3 then u4 ... else if u[2N-1] then u[2N]; L))
    by inductively invoking COMPEXP and COMCOND

= (ac1 := if NULL  $\alpha$  then NIL else if u3 then u4 ... else
    if u[2N-1] then u[2N]; L;)
    by checking two cases on NULL at If NULL  $\alpha$ 
    then ac1 already holds  $\emptyset = V \text{ NIL}$ .
```

The third test asks if (u1 u2) is (T u2). If so any succeeding pairs may be ignored. COMCOND produces

```
((COMPEXP(u2, M, VPR))
 L)
```

as required. •

Theorem 15 [Definition of COMPEXP(EXP, M, VPR)], Same as Theorems 1 and 7.

Proof, (Modifications to the proofs of Theorems 1 and 7.) Add a case for EXP = (CAR α). By correct syntax, $\alpha \neq T, \text{NIL}, \text{numeric-atom}$. If α is an atom case 3.1a produces

```
(HLRZ@ 1 M+CDR ASSOC( $\alpha$ , VPR) P)
```

As in Theorem 1, case 3, M+CDR ASSOC(α , VPR) is correct; by the definition of HLRZ@, ac1 holds V EXP. If α is not an atom, then case 3.1b holds. Invoking COMPEXP(α , M, VPR) inductively leaves V α in ac1, from which (HLRZ@ 1 1) produces CAR V α = V EXP in ac1 as required. The additional case for EXP = (CDR α) is identical to the case for CAR except for HLRZ@.

Case 4, The first case of Theorem 7 also handles the function EQ since Theorem 13 handles EQ.

Case 7, EXP = (fname α) where α consists of N arguments, COMPEXP produces

```
((COMPLISA(( $\alpha$ ), M, VPR))
  (CALL N (E fname)))
```

This is correct, i.e. ac1 holds V EXP in view of the definitions of COMPLISA and CALL.

Case 8, STACKUP is identical with COMPLIS of Theorem 2. Use Lemma 9 on SUBSTACK. •

Theorem 16 [Correctness of the compiler], Same as Theorem 3.

Proof, Same as Theorem 3 but using Lemma 9. •

Termination of C4 follows by essentially the same argument as used for C0. CLASSIFY and SUBSTACK join COMP as exceptions since neither is recursive. COMPLISA can be shown to terminate by replacing its two calls (in COMPEXP, case 7 and COMBOOL, case 1.1) by the body of COMPLISA; this substitution will allow the body to reference substructures directly. This completes the proof P4 of the compiler C4.

The process of constructing P4 uncovered six errors in C4 as originally written, in addition to the numeric-atom problem in C0. Three were found early on by attempting to show that CARs and CDRs in C4 were always well-defined, i.e. not applied to atoms. Although no further errors were expected, the other three surfaced after carefully stating the theorems and then discovering where the proof could not be completed. Each case that failed led very quickly to the construction of a counter-example to the statement of correctness, and furthermore showed what changes to C4 would be sufficient. These changes were made (by London) and the proof was completed.

The changes made to C4 are shown in the listing of the compiler in Appendix 2. Each change is now elaborated!

- (i) COMPEXP, case 2, Same change to C0 for numeric-atoms.
- (ii) COMCOND, line 2 and COMBOOL, case 1, Found by checking CARs and CDRs for being well (-defined). Counter-examples are Boolean atomic variables.
- (iii) COMPANDOR1, lines 1-2. Found as in (ii). Only counter-examples are (AND) and (OR). Incorrectness in the first proposed change [IF NULL U THEN NIL ELSE], which seems correct, was only discovered by checking the case $N = 0$ in $P = (AND \alpha)$ of Theorem 13.
- (iv) LOADAC, case CAAR Z = 0 and CLASS1, lines 3-5, Found by considering the case of T, NIL, and numeric-atoms as actual parameters to a function in the atom case for LOADAC in Theorem 10.

(v) LOADAC, case CAAR $Z = 5$. Found by noting that the result for LOADAC in Theorem 10 did not Inductively follow if $d[N2] = 5$. Counter-examples are function calls with a class 5 argument; all succeeding arguments failed to be compiled at all.

(vi) COMBOOL, case 5. Found by reconsidering the case of a LAMBDA expression in Boolean context (for example an argument to AND, OR, or COND) at the last case of Theorem 5 which case failed in Theorem 13.

As a check on the changes and the completed proof P4, London used the changed C4 to compile some of McCarthy's test functions and also a set of representative counter-examples. The test functions gave identical output as the original C4 (another use of the file comparison utility program). The counter-examples gave correct output as determined by a hand inspection.

ACKNOWLEDGMENTS

As noted, John McCarthy made the compilers available to me. Rod M. Burstall and Whitfield Diffie provided many stimulating discussions and suggestions.

REFERENCES

- Burstall, R. M., 1969, Proving properties of programs by structural induction, *Computer J.*, 12, 1, February, pp. 41-48.
- Burstall, R. M. & Landin, P. J., 1969, Programs and their proofs: An algebraic approach, *Machine Intelligence 4*, B. Meltzer & D. Michie (eds.), American Elsevier, pp. 17-43.
- Dijkstra, E. W., 1970, Notes on structured programming, T.H.-Report 70-WSK-03, Technological University Eindhoven, The Netherlands, Second Edition, April.
- Hearn, A. C., 1973, REDUCE 2 user's manual, Artificial Intelligence Memo AIM-133, Stanford University, October.
- Hoare, C. A. R., 1971, Proof of a program: FIND, *Comm. ACM* 14, 1, January, pp. 39-45.
- Kanlan, D. M., 1967, Correctness of a compiler for Algol-like programs, *Artificial Intelligence Memo No. 48*, Stanford University, July.
- London, R. L., 1970, Proving programs correct: Some techniques and examples, *BIT*, 10, 2, pp. 168-182.
- McCarthy, J. & Painter, J. A., 1967, Correctness of a compiler for arithmetic expressions, *Proceedings of a Symposium in Applied Mathematics*, Vol. 19, J. T. Schwartz (ed.), American Mathematical Society, pp. 33-41.
- McGowan, C. L., 1971, An inductive proof technique for interpreter equivalence, *Formal Semantics Of Programming Languages*, R. Rustin (ed.), Prentice-Hall, to appear.
- Milner, R., 1972, Implementation and applications of Scott's logic for computable functions, *Proceedings of a Conference on Proving Assertions about Programs*, Association for Computing Machinery, to appear.
- Painter, J. A., 1967, Semantic correctness of a compiler for an Algol-like language, *Artificial Intelligence Memo No. 44* [also Ph.D. thesis], Stanford University, March.
- Weissman, C., 1967, *Lisp 1.5 Primer*, Dickenson Publishing Co.
- Wirth, N., 1971, Program development by stepwise refinement, *Comm. ACM*, 14, 4, April, pp. 221-227.



APPENDIX 1 - A LISTING OF THE COMPILER C0

```

FEXPR COMPL FILE ← BEGIN SCALAR Z;
  EVAL('OUTPUT,('DSK: , LIST (CARFILE,'LAP)))$
  EVAL('INPUT , ('DSK: , FILE))$
  INC('T ,NIL)$
  OUTC(T,NIL)$
LOOP:  Z ← ERRSET(READ())$
  IF ATOM Z THEN GO TO DONE$
  Z ← CAR Z$
  IF CAR Z EQ 'DE THEN
BEGIN SCALAR PROG;
  PROG ← COMP(CADR Z,CADDR Z,CADDR Z)$
  MAPC(FUNCTION(PRINT),PROG)$
  OUTC(NIL,NIL)$
  PRINT LIST(CADR Z,LENGTH PROG)$
  OUTC(T,NIL)$
END
  ELSE PRINT z$
  GO TO LOOP$
DONE:  OUTC(NIL,T)$
  INC(NIL,T)$
  RETURN 'ENDCOMP END;

```

 For the purposes of this paper, the compiler starts here; above here
 may be ignored.

```

COMP(FN,VARS,EXP) ←
  (LAMBDA N;
    APPEND(
      LIST LIST('LAP,FN,'SUBR ),
      MKPUSH(N,1),
      COMPEXP(EXP,-N,PRUP(VARS,1)),
      LIST LIST ('SUB , 'P ,LIST('C,0,0,N,N)),
      '((POP P) NIL)))
    LENGTH VARS;
  PRUP(VARS,N) ← IF NULL VARS THEN NIL
    ELSE (CAR VARS , N) . PRUP(CDR VARS,N+1);
  MKPUSH(N,M) ← IF NCM THEN NIL ELSE LIST('PUSH , 'P ,M),MKPUSH(N,M+1);
  COMPEXP(EXP,M,VPR) ←
  [1] IF NULL EXP THEN ' ( (MOVEI 1 0))
  [2] ELSE IF EXP EQ 'T OR NUMBERP EXP THEN
    LIST LIST('MOVEI, 1, (LIST('QUOTE, EXP)))
  [3] ELSE IF ATOM EXP THEN
    LIST LIST('MOVE ,1,M+CDR ASSOC(EXP,VPR),'P )
  [4] ELSE IF CAR EXP EQ 'AND OR CAR EXP EQ 'OR OR
    C A R EXP EQ 'NOT THEN

```

```

        (LAMBDA L1,L2; APPEND(COMBOOL(EXP,M,L1,NIL,VPR),
            LIST('MOVEI 1 (QUOTE T)),LIST('JRST ,0,L2),
            L1,('MOVEI 1 0),L2)))
        (GENSYM(),GENSYM())
[5]     ELSE IF CAR EXP EQ 'COND THEN
        COMCOND(CDR EXP,M,GENSYM(),VPR)
[6]     ELSE IF CAR EXP EQ 'QUOTE THEN LIST LIST('MOVEI,1,EXP)
[7]     ELSE IF ATOM CAR EXP THEN
        (LAMBDA N; APPEND(COMPLIS(CDR EXP,M,VPR),
            LOADAC(1-N,1),
            LIST LIST('SUB,'P ,LIST('C,0,0,N,N)),
            LIST LIST('CALL ,N,
            LIST('E ,CAR EXP))))
        LENGTH CDR EXP
[8]     ELSE IF CAAR EXP EQ 'LAMBDA THEN
        (LAMBDA N; APPEND(COMPLIS(CDR EXP,M,VPR),
            COMEXP(CADDAR EXP,M-N,
            APPEND(PRUP(CADAR EXP,1-M),VPR)),
            LIST LIST('SUB,'P ,LIST('C ,0,0,N,N))))
        LENGTH CDR EXP;

COMPLIS(U,M,VPR) ←
    IF NULL U THEN NIL
    ELSE APPEND(COMEXP(CAR U,M,VPR),
        '((PUSH P 1)),
        COMPLIS(CDR U,M-1,VPR));

LOADAC(N,K) ← IF N>0 THEN NIL ELSE LIST('MOVE ,K,N,'P ),
    LOADAC(N+1,K+1);

COMCOND(U,M,L,VPR) .
    IF NULL U THEN LIST L
    ELSE (LAMBDA L1; APPEND(
        COMBOOL(CAAR U,M,L1,NIL,VPR),
        COMEXP(CADAR U,M,VPR),
        LIST(LIST('JRST ,L),L1),
        COMCOND(CDR U,M,L,VPR)))
    GENSYM();

COMBOOL(P,M,L,FLG,VPR) ←
[1]     IF ATOM P THEN APPEND(COMEXP(P,M,VPR),
        LIST LIST(IF FLG THEN 'JUMPN
        ELSE 'JUMPE ,1,L))

c2     ELSE IF CAR P EQ 'AND THEN
2 a3     (IF NOT FLG THEN COMPANDOR(CDR P,M,L,NIL,VPR)
[b]     ELSE (LAMBDA L1; APPEND(
        COMPANDOR(CDR P,M,L1,NIL,VPR),
        LIST LIST('JRST ,0,L),
        LIST L1))
        GENSYM())
[3]     ELSE IF CAR P EQ 'OR THEN
[a]     (IF FLG THEN COMPANDOR(CDR P,M,L,T,VPR)

```

```

[6]          ELSE (LAMBDA L1;  APPEND(
                                COMPANDOR(CDR P,M,L1,T,VPR),
                                LIST LIST('JRST ,0,L),
                                LIST L1))
                                GENSYM() )
[4]  ELSE IF CAR P EQ 'NOT THEN
      COMBOOL(CADR P,M,L,NOT FLG,VPR)
[5]  ELSE APPEND(COMPEXP(P,M,VPR),
                LIST LIST(IF FLG THEN 'JUMPN
                          ELSE 'JUMPE ,1,L));

COMPANDOR(U,M,L,FLG,VPR) . IF NULL U THEN NIL
  ELSE APPEND(COMECOL(CAR U,M,L,FLG,VPR),
              COMPANDOR(CDR U,M,L,FLG,VPR));

```


APPENDIX 2 - A LISTING OF THE MORE OPTIMIZING COMPILER C4

The changes needed to complete the proof of correctness of C4 are shown in this listing - deletions enclosed between the symbols < and > and additions enclosed between the symbols [and] with the latter two also being used to number cases. The eight changes are at COMPEXP, case 2; COMCOND, line 2; LOADAC, cases CAAR Z = 0 and CAARZ 5. CLASS1, lines 3-5; COMBOOL, cases 1 and 5; and COMPANDOR1, lines 1-2:

```

FEXPRC GMPL FILE ← BEGINS CALARZ;
    EVAL('OUTPUT', ('DSK:', LIST(CAR FILE, 'LAP')));
    EVAL('INPUT', ('DSK:', FILE));
    INC('T', NIL);
    OUTC('T', NIL);
LOOP:   Z ← ERRSET(READ());
        I F ATOM Z THEN GOT O DONE;
        Z ← CAR Z;
        IF CAR Z EQ? 'DE THEN
BEGINS CALAR PROG;
    PROG ← COMP(CADR Z, CADDR Z, CADDDR Z);
    MAPC(FUNCTION(PRINT), PROG);
    OUTC(NIL, NIL);
    PRINTLIST(CADR Z, LENGTH PROG);
    OUTC('T', NIL);
END
        ELSE PRINT Z;
        GO TO LOOPS;
DONE:   OUTC(NIL, T);
        INC(NIL, T);
        RETURN 'ENDCOMP END;

```

 For the purposes of this paper, the compiler starts here; above here
 may be ignored,

```

COMP(FN, VARS, EXP) ←
    (LAMBDA VPR, N;
        APPEND(
            LISTLIST('LAP, FN, 'SUBR),
            MKPUSH(N, 1),
            COMPEXP(EXP, -N, VPR),
            SUBSTACK N,
            '((POPJP) NIL)))
        (PRUP(VARS, 1), LENGTH VARS);

SUBSTACK N ← 1 FN = 0 THEN NIL
    ELSE LIST LIST('SUB', 'P', LIST('C', 0, 0, N, N));

```

```

PRUP(VARS,N) ← IF NULL VARS THEN NIL
                ELSE (CAR VARS , N) , PRUP(CDR VARS,N+1);

MKPUSH(N,M) ← IF N<M THEN NIL ELSE LIST('PUSH','P,M),MKPUSH(N,M+1);

COMPEXP(EXP,M,VPR) ←
[1]      IF NULL EXP THEN '((MOVEI 1 0))
[2]      ELSE IF EXP EQ 'T THEN '((MOVEI 1 (QUOTE T)))
                [OR NUMBERP EXP THEN
[3]      ELSE IF ATOM EXP THEN
                LIST LIST('MOVEI, 1, (LIST('QUOTE, EXP)))
[3.1]    ELSE IF CAR EXP EQ 'CAR THEN
    Cal    (IF ATOM CADR EXP THEN
                LIST LIST('HLRZ@,1,
                M+CDR ASSOC(CADR EXP,VPR),'P )
    [b]    ELSE APPEND(COMPEXP(CADR EXP,M,VPR),
                '((HLRZ@ 1 1)))
[3.2]    ELSE IF CAR EXP EQ 'COR THEN
    [a]    (IF ATOM CADR EXP THEN
                LIST LIST('HRRZ@ ,1,
                M+CDR ASSOC(CADR EXP,VPR),'P )
    [b]    ELSE APPEND(COMPEXP(CADR EXP,M,VPR),
                '((HRRZ@ 1 1)))
[4]      ELSE IF CAR EXP EQ 'AND OR CAR EXP EQ 'OR OR
                CAR EXP EQ 'NOT OR CAR EXP EQ 'EQ THEN
                (LAMBDA L1,L2; APPEND(
                COMBOOL(EXP,M,L1,NIL,VPR),
                LIST('MOVEI 1 (QUOTE T)),LIST('JRST,0,L2),
                L1,'(MOVEI 1 0),L2)))
                (GENSYM(),GENSYM())
[5]      ELSE IF CAR EXP EQ 'COND THEN
                COMCOND(CDR EXP,M,GENSYM(),VPR)
[6]      ELSE IF CAR EXP EQ 'QUOTE THEN LIST LIST('MOVEI,1,EXP)
[7]      ELSE IF ATOM CAR EXP THEN
                APPEND(COMPLISA(CDR EXP,M,VPR),
                LIST LIST('CALL,LENGTH CDR EXP,
                LIST('E,CAR EXP)))
[8]      ELSE IF CAAR EXP EQ 'LAMBDA THEN
                (LAMBDA N; APPEND(STACKUP(CDR EXP,M,VPR),
                COMPEXP(CADDR EXP,M-N,
                APPEND(PRUP(CADDR EXP,1-M),VPR)),
                SUBSTACK N))
                LENGTH CDR EXP;

STACKUP(U,M,VPR) ← IF NULL U THEN NIL
                ELSE APPEND(COMPEXP(CAR U,M,VPR),
                '((PUSH P 1)),
                STACKUP(CDR U,M-1,VPR));

```



```
CCCHAINEXP ← (CAREXPEQ'CAR OR CAREXP EQ'CDR) AND
              (ATOMCADREXP OR CCCHAINCADREXP);
```

```
COMPC(EXP,N2,M,VPR) ←
  IF ATOMEXP THEN ERROR'COMPC
  ELSE IF CAREXPEQ'CAR THEN
    (IF ATOMCADREXP THEN
      LIST LIST('HLRZ@ ,N2,M+CDR ASSOC(CADREXP,VPR),'P )
    ELSE LIST('HLRZ@ ,N2,N2).COMPC(CADREXP,N2,M,VPR))
  ELSE IF ATOMCADREXP THEN
    LIST LIST('HRRZ@ ,N2,M+CDRASSOC(CADREXP,VPR),'P )
    ELSE LIST('HRRZ@ ,N2,N2).COMPC(CADREXP,N2,M,VPR);
```

```
COMCOND(U,M,L,VPR) ←
  IF NULL U THEN LIST L
  ELSE IF (NOT ATOMCAARU AND)
    CAARU ≠ NULL AND NULL CADARU THEN
    APPEND(COMPEXP(CADARU,M,VPR),
      LIST LIST('JUMPE ,1,L),
      COMCOND(CDR U,M,L,VPR))
  ELSE IF CAARUEQ'T THEN
    APPEND(COMPEXP(CADARU,M,VPR),LIST L)
  ELSE (LAMBDA L1; APPEND(
    COMBOOL(CAARU,M,L1,NIL,VPR),
    COMPEXP(CADARU,M,VPR),
    LIST(LIST('JRST ,0,L),L1),
    COMCOND(CDR U,M,L,VPR)))
  GENSYM();
```

```
COMPLISA(U,M,VPR) ←
  (LAMBDA Z; APPEND (
    COMPLIS(Z,M,1,VPR),
    LOADAC(Z,1-CCOUNT Z,1,M-CCOUNT Z,VPR),
    SUBSTACKCOUNT t))
  CLASSIFY U;
```

```
CCOUNT Z ← IF NULL Z THEN 0 ELSE IF CAAR Z = 4 THEN 1+CCOUNT CDR Z
  ELSE CCOUNTCDRZ;
```

```
LOADAC(Z,M2,N2,M,VPR) ←
  IF NULL Z THEN NIL
  ELSE IF CAAR Z = 1 THEN
    LIST('MOVE ,N2,M+CDRASSOC(CDAR Z,VPR),'P )
    .LOADAC(CDR Z,M2,N2+1,M,VPR)
  [ELSE IF CAAR Z = 3 THEN
    LIST('MOVEI ,N2,(LIST('QUOTE,CDAR Z)))
    .LOADAC(CDR Z,M2,N2+1,M,VPR)]
  ELSE IF CAAR Z = 2 THEN
    LIST('MOVEI ,N2,CDAR Z)
    .LOADAC(CDR Z,M2,N2+1,M,VPR)
  ELSE IF CAAR Z = 3 THEN
```

```

        APPEND(REVERSE COMPC(CDAR Z,N2,M,VPR),
                LOADAC(CDR Z,M2,N2+1,M,VPR))
ELSE IF CAAR Z = 5 THEN <NIL> [LOADAC(CDR Z,1,N2+1,M,VPR)]
ELSE LIST('MOVE,N2,M2,'P),
        LOADAC(CDR Z,M2+1,N2+1,M,VPR);

```

```

COMPLIS(Z,M,K,VPR) ←
    IF NULL Z THEN NIL
    ELSE IF CAAR Z = 4 THEN APPEND(
        COMEXP(CDAR Z,M,VPR),
        '((PUSH P 1)),
        COMPLIS(CDR Z,M-1,K+1,VPR))
    ELSE IF CAAR Z = 5 THEN APPEND(
        COMEXP(CDAR Z,M,VPR),
        IF K=1 THEN NIL
        ELSE LIST LIST('MOVE ,K,1))
    ELSE COMPLIS(CDR Z,M,K+1,VPR);

```

```

CLASSIFY U ← CLASS2(CLASS1(U,NIL),NIL,T);

```

```

CLASS1(U,V) ← IF NULL U THEN V
    ELSE IF ATOM CAR U THEN
        [(IF CAR U = 'NIL OR CAR U = 'T OR NUMBERP CAR U THEN
            CLASS1(CDR U, (0 , CAR U),V)
            ELSE] CLASS1(CDR U, (1 , CAR U),V)[)]
    ELSE IF CAAR U = 'QUOTE THEN CLASS1(CDR U,(2 , CAR U),V)
    ELSE IF CCHAIN CAR U THEN CLASS1(CDR U,(3 , CAR U),V)
    ELSE CLASS1(CDR U,(4 , CAR U),V);

```

```

CLASS2(U,V,FLG) ← IF NULL U THEN V
    ELSE IF FLG AND (CAAR U = 4) THEN
        CLASS2(CDR U,(5 , CDAR U),V,NIL)
    ELSE CLASS2(CDR U,CAR U . V,FLG);

```

```

MKJRST L ← LIST LIST('JRST ,0,L);

```

```

COMBOOL(P,M,L,FLG,VPR) ←
[0.1] IF P EQ T THEN (IF FLG THEN MKJRST L ELSE NIL)
C13  [ELSE IF ATOM P THEN APPEND(
        COMEXP(P, M, VPR),
        LIST LIST(IF FLG THEN 'JUMPN
                    ELSE 'JUMPE ,1,L))];
[1.1] ELSE IF CAR P EQ 'EQ THEN APPEND(
        COMPLISA(CDR P,M,VPR),
        IF FLG THEN '((CAMN 1 2)) ELSE '((CAME 1 2)),
        MKJRST L)
[2]  ELSE IF CAR P EQ 'AND THEN
    [a] (IF NOT FLG THEN COMPANDOR(CDR P,M,L,NIL,VPR)
    [b] ELSE (LAMBDA L1; APPEND(
        COMPANDOR1(CDR P,M,L1,L,NIL,VPR),
        LIST L1))
        GENSYM())

```

```

[3]      ELSE IF CAR P EQ 'OR THEN
      [a]      ( IF FLG THEN COMPANDOR(CDR P,M,L,T,VPR)
      [b]      ELSE (LAMBDA L1; APPEND(
                                COMPANDOR1(CDR P,M,L1,L,T,VPR),
                                LIST L1))
                                GENSYM())
[4]      ELSE IF CAR P EQ 'NOT THEN
      COMBOL(CADR P,M,L,NOT FLG,VPR)
[4.1]    ELSE IF CAR P EQ 'NULL THEN APPEND (
      COMEXP(CADR P,M,VPR),
      LIST LIST(IF FLG THEN 'JUMPE
      ELSE 'JUMPN ,1,L))
[5]      E L S E <IF ATOM CAR P THEN> APPEND(
      COMEXP(P,M,VPR),
      LIST LIST(IF FLG THEN 'JUMPN
      ELSE 'JUMPY ,1,L));

COMPANDOR(U,M,L,FLG,VPR) = IF NULL U THEN NIL
      ELSE APPEND(COMBOL(CAR U,M,L,FLG,VPR),
      COMPANDOR(CDR U,M,L,FLG,VPR));

COMPANDOR1(U,M,L,L2,FLG,VPR) = [IF NULL U THEN MKJST L2
      ELSE] IF NULL CDR U THEN COMBOL(CAR U,M,L2,NOT FLG,VPR)
      ELSE APPEND(COMBOL(CAR U,M,L,FLG,VPR),
      COMPANDOR1(CDR U,M,L,L2,FLG,VPR));

```


APPENDIX 3 - SAMPLE OUTPUT OF C0 AND C4 FOR A REVERSE FUNCTION

(DE REV (X Y) (COND ((NULL X) Y) (T (REV (CDR X) (CONS (CAR X) Y))))))

Code from C0	Comments	Code from C4
(LAP REV SUBR)	header	(LAP REV SUBR)
(PUSH P 1)	stack first arg	(PUSH P 1)
(PUSH P 2)	stack second arg	(PUSH P 2)
(MOVE 1 -1 P)	compute x	
(PUSH P 1)	stack it	
(MOVE 1 0 P)	recall X	(MOVE 1 -1 P)
(SUB P (C 0 0 1 1))	adj. stack by 1	
(CALL 1 (E NULL))	call NULL	
(JUMPE 1 L2)	if not NULL jump	(JUMPN 1 L2)
(MOVE 1 0 P)	recall Y	(MOVE 1 0 P)
(JRST L1)	jump for return	(JRST L1)
L2	the label L2	L2
(MOVE 1 (QUOTET))	compute T	
(JUMPE 1 L3)	if not T jump	
(MOVE 1 -1 P)	compute X	
(PUSH P 1)		
(MOVE 1 0 P)	recall X	
(SUB P (C 0 0 1 1))		
(CALL 1 (E CDR))	CDR	
(PUSH P 1)		
(MOVE 1 -2 P)	compute X	
(PUSH P 1)		
(MOVE 1 0 P)	recall X	
(SUB P (C 0 0 1 1))		
(CALL 1 (E CAR))	CAR, resp. CAR X	(HLRZ@ 1 -1 P)
(PUSH P 1)		
(MOVE 1 -2 P)	compute Y	
(PUSH P 1)		
(MOVE 1 -1 P)	recall CAR X	
(MOVE 2 0 P)	recall Y	(MOVE 2 0 P)
(SUB P (C 0 0 2 2))	adj. stack by 2	
(CALL 2 (E CONS))	CONS.	(CALL 2 (E CONS))
(PUSH P 1)		
(MOVE 1 -1 P)	recall CDR X	
(MOVE 2 0 P)	recall CONS, resp. transfer CONS	(MOVE 2 1)
	compute CDR X	(HRRZ@ 1 -1 P)
(SUB P (C 0 0 2 2))		
(CALL 2 (E REV))	REV	(CALL 2 (E REV))
(JRST L1)	jump for return	
L3		
L1		L1
(SUB P (C 0 0 2 2))	return	(SUB P (C 0 0 2 2))
(POPJ P)		(POPJ P)
NIL	end of code	NIL

