Abstract:

This report is a complete software kit. In the form in which it is written it is intended only for readers wishing to construct the kit. That is to say, its style is terse in the extreme. It assumes familiarity with Lisp. The kit implements a variant of the Lisp language, which we call Lispkit Lisp. Lispkit Lisp has proper binding and supports higher order functions. The implementation comprises an abstract machine and a compiler, written in Lispkit Lisp, which translates any Lispkit Lisp program to run on that machine. The implementor must write an emulator for this machine and then copy the compiler from the appendix of this report. Thereupon he is in a position to bootstrap the system to include his own extensions, for both the source and object of the compiler are provided.

The Lispkit System - A Software Kit

By

Peter Henderson

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 129
October 1978

Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

bibliographical details

HENDERSON, Peter.

The Lispkit system: a software kit.
[By] Peter Henderson.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 129).

Added entries UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 129.

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th): 001.6425 001.6424
U. D. C. 681.322.06

Suggested keywords

APPLICATIVE LANGUAGES
COMPILER
FUNCTIONAL PROGRAMMING
λ-CALCULUS
LISP
SECD MACHINE

Abstract

This report is a complete software kit. In the form in which it is written it is intended only for readers wishing to construct the kit. That is to say, its style is terse in the extreme. It assumes familiarity with Lisp. The kit implements a variant of the Lisp language, which we call Lispkit Lisp. Lispkit Lisp has proper binding and supports higher order functions. The implementation comprises an abstract machine and a compiler, written in Lispkit Lisp, which translates any Lispkit Lisp program to run on that machine. The implementor must write an emulator for this machine and then copy the compiler from the appendix of this report. Thereupon he is in a position to bootstrap the system to include his own extensions, for both the source and object of the compiler are provided.

About the author

Dr. Peter Henderson is a lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.

bibliographical details

Introduction

We assume the reader is familiar with Lisp at least to the level of having written some Lisp programs or read the basic references (see bibliography). The variant of Lisp described here differs in many essential details from the standard implementations of Lisp. Firstly, many of the standard functions are not supported. Secondly the way in which function definitions are introduced is made more powerful, allowing nested definitions and finally, proper binding rules are used consistently throughout the language. To emphasise this difference, we refer to the language as Lispkit Lisp. The main aim has been to produce a formally very simple but powerful variant with a high degree of portability. The portability is obtained as follows. An abstract machine, the SECD machine, is defined, an emulation of which must be provided on each machine to which the Lispkit System is to be transported. The emulation, which normally takes the form of a program, basically provides for the input and output of S-expressions, a garbage-collected list space, and the usual instruction execution loop. A compiler, written in Lispkit Lisp will translate Lispkit Lisp programs into SECD machine language programs. In particular it will translate itself. The source of the compiler is some 75 lines of Lisp and the object some 60 lines of SECD machine instructions. By copying these components from the appendices of this report, it is trivial to obtain a working Lispkit system, able to compile Lispkit Lisp programs to run on the SECD machine emulator. In particular, since the compiler can compile itself, subsequent enhancements and modifications to the system are easily possible.

This report has six main sections and two appendices which list additional sources of information and the two versions of the compiler needed to start the bootstrap. The six main sections are intentionally terse since this report is mainly for use as an implementor's guide rather than as a guide for users of Lispkit systems.

The six sections each give definitions of some component of the overall design. The first section simply gives the syntax for S-expressions and one possible concrete realisation of these as internal data structures. The second section defines the well-formed Lispkit Lisp expressions and informally describes their meanings. The third section gives an abstract definition of the SECD machine as a set of machine transitions. The fourth section gives the standard implementation of each SECD machine instruction. The fifth section defines the code skeletons for each well-formed Lispkit expression and the sixth and final section gives a presentation of the Lisp compiler in abstract form. It is not expected that the reader can simply study these sections in order. He must find a place to start which is easiest for him to understand and then iterate through each section, back and forth, until he has derived from this description that information which he seeks.

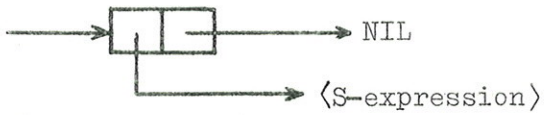
1. S-expressions

The symbolic expressions which are used to represent both data and programs have the following syntax.

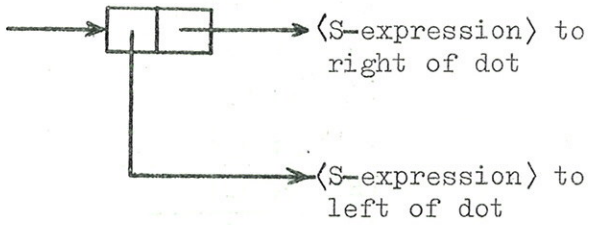
⟨S-expression⟩::= ⟨atom⟩
| (⟨S-expression - list⟩)
⟨S-expression-list⟩::= ⟨S-expression⟩
| ⟨S-expression⟩.⟨S-expression⟩
| ⟨S-expression⟩⟨S-expression-list⟩
⟨atom⟩::= ⟨number⟩
| ⟨identifier⟩

For ⟨number⟩s we shall use sequences of decimal digits, possibly preceded by a minus sign and for ⟨identifier⟩s sequences of letters and digits beginning with a letter. Each ⟨S-expresssion⟩ is represented by a particular internal data structure. An ⟨atom⟩ is represented by a pointer to a numeric or string value according to whether it is a ⟨number⟩ or an ⟨identifier⟩. There are three cases for (⟨S-expression-list⟩) each represented by the corresponding structure given below

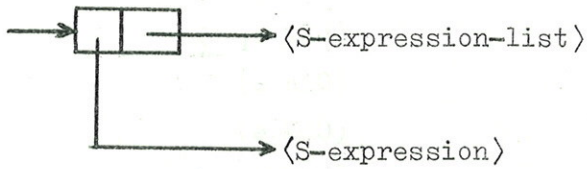
(⟨S-expression⟩)



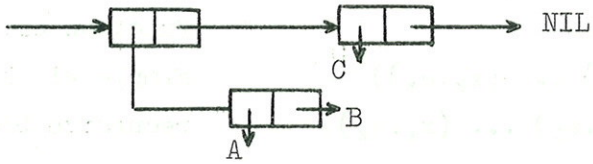
(⟨S-expression⟩.⟨S-expression⟩)



(⟨S-expression⟩ ⟨S-expression-list⟩)



Applying these rules we see that the ⟨S-expression⟩ ((A.B) C) is represented by



It is easy to see that this structure also represents the ⟨S-expressions⟩

((A.B).(C))
((A.B).(C.NIL))

In general we will always write an ⟨S-expression⟩ with the minimum of dots. In practice, a dot followed immediately by an opening parenthesis can be omitted along with that parenthesis and its corresponding closing parenthesis. Similarly a dot followed immediately by the atom NIL can be omitted, along with the NIL.

2. Well-formed Lispkit Lisp expressions

The well-formed expressions, from which syntactically valid programs can be built are defined recursively as follows. If $x, x_1, \ldots, x_k$ are ⟨identifier⟩s, s is an arbitrary ⟨S-expression⟩ and $e, e_1, \ldots, e_k$ are well-formed expressions then each of the following is a well-formed expression.

x	a variable
(QUOTE s)	a constant
(ADD e_1 e_2)	arithmetic expressions
(SUB e_1 e_2)	
(MUL e_1 e_2)	
(DIV e_1 e_2)	
(REM e_1 e_2)	
(EQ e_1 e_2)	relational expressions
(LEQ e_1 e_2)	
(CAR e)	structural expressions
(CDR e)	
(CONS e_1 e_2)	
(ATOM e)	
(IF e_1 e_2 e_3)	conditional form
(LAMBDA (x_1 ... x_k)e)	lambda expression
(e e_1 ... e_k)	function call
(LET e (x_1.e_1) ... (x_k.e_k))	simple block
(LETREC e (x_1.e_1) ... (x_k.e_k))	recursive block

The variable, the constant and the five basic functions EQ, CAR, CDR, CONS, ATOM take their usual syntactic forms. The arithmetic operators are given different names in Lispkit Lisp and the only relational operator apart from EQ, is LEQ, only applicable to numbers. LAMBDA expressions and function calls take their usual form, while the conditional and block forms are peculiar to this variant. We shall describe their intended meaning only briefly.

A well-formed expression always occurs in a particular context in a program and evaluates to give a particular value with reference to that context. A variable x occurs in the body of a function or block in which x has been introduced either as a parameter or a local variable. Its value is that value assigned to it on function call or block entry. The value of the constant (QUOTE s) is always s. The values of the basic operator expressions (arithmetic, relational and structural) is just the appropriate operator applied to the values of

the well formed sub-expressions. The value of the form (IF e_1 e_2 e_3) is just the value of e_2 if the value of e_1 is T and the value of e_3 if e_1 is F. The value of (LAMBDA (x_1... x_k) e) is a function which can be called and to which the values of k actual parameters must be assigned to correspond to the formal parameters $x_1,...,x_k$. The call (e e_1... e_k) yields a value obtained as follows. The value of e must be a function with k parameters. The body of this function is evaluated with each of these parameters assigned the value of the corresponding expression from $e_1,...,e_k$. There are two forms of block. Each introduces k new variables $x_1,...,x_k$ with the value of the corresponding expression from $e_1,...,e_k$ as its value. The value of the block is the value of the bound expression e, evaluated in the context established by the addition of the variables $x_1,...x_k$. The difference between the LET and LETREC forms of block is just the scope of the x_i. In the LET form the scope is just the bound expression e. In the LETREC form the scope is also the defining expressions $e_1,...,e_k$. The LETREC forms must be used for defining recursive or mutually recursive functions. There is a restriction in the implementation given here that in the LETREC form of block, the defining expressions must all be LAMBDA expressions. In general, the defining expression in each block is compound and thus the dot, after the variable being defined, can be dropped along with one set of parentheses from around the defining expression.

A Lispkit Lisp program is a well-formed expression, whose value is a function, and which has no free-variables. Clearly all variables must occur in a context in which they are defined. The following is a sample complete Lispkit Lisp program. It is represented here only to be inspected for clarification of syntax and need not be understood. Another example, the compiler, appears in the appendices.

```
(LETREC (LAMBDA (N T E) (SHAPE N (SUBEXPR T E)))
  (SUBEXPR LAMBDA (T E)
    (IF (ATOM E)
        (IF (MATCHES T E) E (QUOTE NIL))
    (IF (MATCHES T E) E
    (LET (IF (EQ M (QUOTE NIL)) (SUBEXPR T (CDR E)) M)
      (M SUBEXPR T (CAR E))))))
  (MATCHES LAMBDA (T E)
    (IF (ATOM E)
        (IF (ATOM T) (IF (EQ E T) (QUOTE T) (EQ T (QUOTE *)))
        (QUOTE F))
    (IF (ATOM T) (EQ T (QUOTE *))
    (IF (MATCHES (CAR T) (CAR E)) (MATCHES (CDR T)(CDR E))
    (QUOTE F)))))
  (SHAPE LAMBDA (N E)
    (IF (ATOM E) E
    (IF (EQ N (QUOTE 0)) (QUOTE *)
    (CONS (SHAPE (SUB N (QUOTE 1)) (CAR E))
          (SHAPE N (CDR E))))))
)
```

3. The SECD Machine

This machine, first described by Landin, is presented here in a particular abstract form which uses S-expressions to describe the data structures held in each register. There are four principal registers in the machine

s - the stack
e - the environment
c - the control
d - the dump

They each contain (or point to) arbitrarily large data structures. The stack and dump have the role normally afforded to the activation stack in Algol implementations, s holding intermediate results of partially evaluated expressions and d holding the values which must be restored to s,e and c on return from a function call. The environment e is a collection of the values of all accessible variables and the control c is the machine language program currently being executed. A glance at appendix 2 will confirm that by and large a machine language program is a list of integers, mostly operation codes. In this section we will use symbolic operation codes, assigning the numerical values in the next section.

The state of the SECD machine can be represented by writing the contents of its four registers in the order

s e c d

Normally, the stack is a list of values, the top element being leftmost. Similarly the control and the dump are lists, the control containing machine instructions and the dump, previous values of s,e and c which need to be restored. The environment is a list of lists. It is accessed by a pair of indexes (b.n) which selects the n^{th} element of the b^{th} sublist. Elements of lists are numbered from zero for this purpose. The following is therefore a possible machine state

(6 3) ((7 2)(-1 0)) (ADD STOP) NIL

Two values (3 and 6) have been placed on the stack. The environment has four variable values 7, 2, -1 and 0 at locations (0.0), (0.1), (1.0) and (1.1) respectively. Two machine instructions ADD and STOP remain to be accessed and the dump is empty.

The effect of each machine instruction is defined by giving a transition of the form

s e c d → s′ e′ c′ d′

The instruction being defined will be the leftmost instruction in c. The after-state of the machine, on the right of the arrow will be expressed in terms of the components of the before state. The following are the transition rules for the instructions of the basic SECD machine. Each rule has four S-expressions on the left and four on the right of the arrow. The machine executes by performing these transitions until it stops.

s	$((v_{00}\ldots)(v_{10}\ldots)\ldots)$	(LD(i.j).c)	d	$\rightarrow (v_{ij}.s)$	$((v_{00}\ldots)(v_{10}\ldots))$	c d
s	e	(LDC v.c)	d	$\rightarrow$ (v.s)	e	c d
s	e	(LDF c′.c)	d	$\rightarrow$ ((c′.e).s)	e	c d
((c′.e′)v.s)	e	(AP.c)	d	$\rightarrow$ NIL	(v.e′)	c′(s e c.d)
(v)	e″	(RTN)	(s e c.d)	$\rightarrow$ (v.s)	e	c d
s	e	(DUM.c)	d	$\rightarrow$ s	$(\Omega.e)$	c d
((c′.e′)v.s)	$(\Omega.e)$	(RAP.c)	d	$\rightarrow$ NIL	rplaca(e′,v)	c′(s e c.d)
(v.s)	e	(SEL c_T c_F.c)	d	$\rightarrow$ s	e	c_v(c.d)
s	e	(JOIN)	(c.d)	$\rightarrow$ s	e	c d
$((v_1.v_2).s)$		(CAR.c)	d	$\rightarrow (v_1.s)$	e	c d
$((v_1.v_2).s)$	e	(CDR.c)	d	$\rightarrow (v_2.s)$	e	c d
(v.s)	e	(ATOM.c)	d	$\rightarrow$ (atom(v).s)	e	c d
$(v_1\ v_2.s)$	e	(CONS.c)	d	$\rightarrow ((v_1.v_2).s)$	e	c d
$(v_1\ v_2.s)$	e	(EQ.c)	d	$\rightarrow$ $(eq(v_1,v_2).s)$	e	c d
$(v_1\ v_2.s)$	e	(ADD.c)	d	$\rightarrow (v_2 + v_1.s)$	e	c d
$(v_1\ v_2.s)$	e	(SUB.c)	d	$\rightarrow (v_2 - v_1.s)$	e	c d
$(v_1\ v_2.s)$	e	(MUL.c)	d	$\rightarrow (v_2 \times v_1.s)$	e	c d
$(v_1\ v_2.s)$	e	(DIV.c)	d	$\rightarrow (v_2 \div v_1.s)$	e	c d
$(v_1\ v_2.s)$	e	(REM.c)	d	$\rightarrow (v_2 - v_2 \div v_1 \times v_1.s)$	e	c d
$(v_1\ v_2.s)$	e	(LEQ.c)	d	$\rightarrow (v_2 \leq v_1.s)$	e	c d
s	e	(STOP)	d	$\rightarrow$ s	e	(STOP) d

[when the RAP instruction is used, we shall always have $e' = (\Omega.e)$]

There are 21 different instructions. Each rule is identified by the appearance of the instruction mnemonic in the head of the control register. In practice of course this will be a numeric value, in terms of which the instruction execution cycle given in the next section, will select the appropriate case for execution.

Let us consider each of these rules in an order which deals with the simplest first (since this is a reference document, the above list is in what will be order of ascending numerical value of operation code).

The simplest instruction of all is STOP. Formally, its execution leaves the machine unchanged but in practice the top element of the stack will be displayed as the result of the computation thus completed. Next, each of the basic operators CAR, CDR, CONS, ATOM, EQ, ADD, SUB, MUL, DIV, REM, LEQ finds its operands on top of the stack, computes the appropriate result to replace the operands and skips past the instruction in the head of control register. The instructions SEL (select) and JOIN are used together to get the effect of an if-then-else structure. A control list (machine language program) will always use them with the following structure.

```
(.... SEL (...then part...JOIN)
          (...else part...JOIN)....)
```

SEL inspects the top value on the stack and selects one of the two sublists following it. It saves the continuation on the dump. JOIN, which always terminates a control list selected by SEL, restores the dumped continuation.

There are three load instructions. LD (load) has as an operand a pair of integers which it uses to select a variable value from the environment. It places the value from the j^{th} position of the i^{th} sublist on top of the stack. LDC (load constant) simply pushes its operand onto the stack. LDF (load function) builds a "closure" on top of the stack. The closure contains the code of a subroutine c′ and a copy of the environment. This closure value represents a function and will normally find its way into the environment as the value associated with the name of a function. However, to call it, it will again be placed on the stack, with the values of its parameters below it. This is what should have happened when the AP instruction is used. It saves the old values of s, e and c on the dump. The stack is then set empty and a new environment built from the parameter values v and the environment part e′ of the closure. Similarly, the control part c′ of the closure is installed as the new control, thus effecting a branch to the subprogram which it represents. When this subprogram has completed, it should execute the RTN (return) instruction. The value v is returned by pushing it onto the restored stack and the environment and control are simply restored from the dump. The remaining instructions DUM (dummy) and RAP (recursive apply) are used to implement

LETREC blocks. DUM creates a dummy sublist in the environment and RAP, which is very similar to AP in all other respects, updates the environment by replacing this dummy sublist by v. To fully understand these last two instructions it will be necessary to follow very carefully through the code skeletons given for LETREC in section 5.

4. The instruction execution cycle

Each of the machine transitions given above can be implemented as a sequence of transfers of values between the registers of the abstract machine. To describe how this is done, at a reasonably abstract level, we make use of the following construction and selection functions and type testing predicates

constructor	selectors	predicate
z=cons(x,y)	car(z)=x	iscons(z)
	cdr(z)=y	
z=numb(n)	ival(z)=n	isnumb(z)
z=symb(s)	sval(z)=s	issymb(z)

where x,y,z are S-expressions, n an integer and s a string.

Using these functions we obtain the following simple implementation of the twenty one machine instructions given in the previous section.

```
LD :  begin     w:=e;
                for i:=1 until ival(car(car(cdr(c)))) do w:=cdr(w);
      w:=car(w);for i:=1 until ival(cdr(car(cdr(c)))) do w:=cdr(w);
      w:=car(w);     s:= cons(w,s); c:=cdr(cdr(c))
      end
LDC : begin     s:= cons(car(cdr(c)),s); c:=cdr(cdr(c)) end
LDF : begin     s:= cons(cons(car(cdr(c)),e),s);c:=cdr(cdr(c))end
AP  : begin     d:= cons(cdr(cdr(s)),cons(e,cons(cdr(c),d)));
                e:= cons(car(cdr(s)),cdr(car(s)));
                c:= car(car(s)); s:= nil
      end
RTN : begin     s:= cons(car(s),car(d)); e:=car(cdr(d));
                c:= car(cdr(cdr(d))), d:=cdr(cdr(cdr(d)))
      end
DUM : begin     e:= cons(nil,e); c:=cdr(c) end
RAP : begin     d:= cons(cdr(cdr(s)),cons(cdr(e),cons(cdr(c),d)));
                e:= cdr(car(s)); car(e):=car(cdr(s));
                c:= car(car(s)); s:=nil
      end
SEL : begin     d:= cons(cdr(cdr(cdr(c))),d);
                if  sval(car(s))="T" then c:= car(cdr(c))
                                     else c:= car(cdr(cdr(c)));
                s:= cdr(s)
      end
JOIN :begin     c:= car(d); d:= cdr(d) end
```

```
CAR  : begin    s:= cons(car(car(s)),cdr(s)); c:= cdr(c) end
CDR  : begin    s:= cons(cdr(car(s)),cdr(s)); c:=cdr(c) end
ATOM : begin    if issymb (car(s)) or isnumb(car(s))
                     then s:= cons(t,cdr(s))
                     else s:= cons(f,cdr(s)); c:=cdr(c)
       end
CONS : begin    s:= cons(cons(car(s),car(cdr(s))),cdr(cdr(s)));
                c:= cdr(c)
       end
EQ   : begin    if issymb (car(s))and issymb (car(cdr(s)))and
                   sval(car(s)) = sval(car(cdr(s))) or
                   isnumb(car(s))and isnumb(car(cdr(s))) and
                   ival(car(s))=ival(car(cdr(s)))
                then s:= cons(t,cdr(cdr(s)))
                else s:= cons(f,cdr(cdr(s))); c:=cdr(c)
       end
ADD  : begin    s:= cons(numb(ival(car(cdr(s)))+ival(car(s))),
                                cdr(cdr(s)));c:=cdr(c)end
SUB  : begin    s:= cons(numb(ival(car(cdr(s)))-ival(car(s))),
                                cdr(cdr(s)));c:=cdr(c)end
MUL  : begin    s:= cons(numb(ival(car(cdr(s)))×ival(car(s))),
                                cdr(cdr(s)));c:=cdr(c)end
DIV  : begin    s:= cons(numb(ival(car(cdr(s)))÷ival(car(s))),
                                cdr(cdr(s)));c:=cdr(c)end
REM  : begin    s:= cons(numb(ival(car(cdr(s)))rem ival(car(s))),
                                cdr(cdr(s)));c:=cdr(c)end
LEQ  : begin    if ival(car(cdr(s))) ≤ ival(car(s))
                                then s:= cons(t,cdr(cdr(s)))
                                else s:= cons(f,cdr(cdr(s)));
                c:= cdr(c)
       end
```

Here the variables s,e,c and d point to data structures representing the contents of the corresponding register of the abstract machine. The variables t, f and nil told constant values set up by

```
t:= symb("T"); f:= symb("F"); nil:= symb("NIL")
```

and w is simply a temporary variable (called the working register).

Each machine instruction has a numeric operation code as follows (they have always been enumerated in numerical order).

1	LD	6	DUM	11	CDR	16	SUB	21	STOP
2	LDC	7	RAP	12	ATOM	17	MUL		
3	LDF	8	SEL	13	CONS	18	DIV		
4	AP	9	JOIN	14	EQ	19	REM		
5	RTN	10	CAR	15	ADD	20	LEQ		

and hence the instruction execution loop for this implementation of the SECD machine can be written

```
cycle: case ival(car(c))of
       begin
            LD: ...
            LDC: ...
              :
              :
            LEQ: ...
            STOP: go to endcycle;
       end;
       go to cycle;
endcycle:
```

Since the record storage space is used rather indiscriminately in Lisp, and in particular in this implementation of Lispkit Lisp, it is necessary to implement a garbage collector which is called whenever cons(x,y), numb(n) or symb(s) find that the record space is exhausted.

5. The translation rules

For each well-formed expression in Lispkit Lisp we will give a code skeleton which defines the sequence of SECD machine instructions generated for that expression. For each Lispkit Lisp program, we construct a list of lists of atoms being the variables used in that program. This namelist has the property that the position of any variable in it corresponds to the position which the value of that variable will have in the run time environment. If x is a variable

and n a namelist, we use the following function to determine the index-pair corresponding to the position of the variable in the namelist.

```
location(x,n) ≡
  if member (x,car(n)) then cons(0, position(x,car(n))) else
       (incar (location (x,cdr(n)))
            where incar(z) ≡ cons(car(z)+1,cdr(z))}
position(x,a) ≡ if eq(x,car(a)) then 0 else 1+ position(x,cdr(a))
member(x,a) ≡
   if a=nil then F else
   if x=car(a) then T else  member(x,cdr(a))
```

Now for each well-formed expression in Lispkit Lisp, one can denote by e*n the list of SECD machine instructions generated from it. In describing e*n for each well-formed expression we shall make use of the infix operator x|y between lists x and y to mean append y to x. We treat each of the well-formed expressions in the order in which they were enumerated in section 2, using the same conventions over names.

$x*n$ = (LD i) where i = location(x,n)

(QUOTE s)*n = (LDC s)

(ADD e_1 e_2)*n = e_1*n | e_2*n | (ADD)

(SUB e_1 e_2)*n = e_1*n | e_2*n | (SUB)

(MUL e_1 e_2)*n = e_1*n | e_2*n | (MUL)

(DIV e_1 e_2)*n = e_1*n | e_2*n | (DIV)

(REM e_1 e_2)*n = e_1*n | e_2*n | (REM)

(EQ e_1 e_2)*n = e_1*n | e_2*n | (EQ)

(LEQ e_1 e_2)*n = e_1*n | e_2*n | (LEQ)

(CAR e)*n = e*n | (CAR)

(CDR e)*n = e*n | (CDR)

(CONS e_1 e_2)*n = e_2*n | e_1*n | (CONS)

(ATOM e)*n = e*n | (ATOM)

(IF e_1 e_2 e_3)*n = e_1*n | (SEL e_2*n | (JOIN) e_3*n | (JOIN))

(LAMBDA (x_1...x_k)e)*n = (LDF e*((x_1...x_k).n) | (RTN))

(e e_1... e_k)*n = (LDC NIL) | e_k*n | (CONS) | ... | e_1*n | (CONS) | e*n | (AP)

(LET e (x_1.e_1)...(x_k.e_k))*n =

(LDC NIL) | e_k*n | (CONS) | ... | e_1*n | (CONS) | (LDF e*m AP)

where m = ((x_1 ... x_k).n)

(LETREC e (x_1.e_1) ... (x_k.e_k))*n =

(DUM LDC NIL) | e_k*m | (CONS) | ... | e_1*m | (CONS) | (LDF e*m RAP)

where m = ((x_1 ... x_k).n)

In these equations we have taken * as always more binding than |.

6. The compiler

The rules of section 5 can very simply be turned into a function which takes as argument a Lispkit Lisp program and produces as result the SECD machine language program. To avoid the excessive use of append we in fact define a function comp(e,n,c) which compiles the well-formed expression e with respect to the namelist n and appends the instruction list c.

That is

$$comp(e,n,c) = e^*n \mid c$$

The definition of comp(e,n,c) is as follows:

```
comp(e,n,c) ≡
    if atom(e) then cons(LD, cons (location(e, n),c)) else
    if eq(car(e),QUOTE) then cons(LDC, cons(cadr(e),c)) else
    if eq(car(e),ADD) then
                    comp(cadr(e), n, comp(caddr(e),n,cons(ADD,c))) else
            ... similarly for SUB, MUL, DIV, REM, LEQ, EQ ....
    if eq(car(e),CAR) then
                    comp(cadr(e),cons(CAR,c)) else
            ... similarly for CDR, ATOM ....
    if eq(car(e),CONS) then
                    comp(caddr(e),n,comp(cadr(e),n,cons(CONS,c))) else
    if eq(car(e),IF) then
                    {comp(cadr(e),n,cons(SEL,cons(thenpt,cons(elsept,c))))
                        where thenpt = comp(caddr(e),n, (JOIN))
                        and elsept = comp(cadddr(e),n, (JOIN)) }
    if eq(car(e),LAMBDA) then
                    {cons(LDF, cons(body, c))
                        where body = comp(caddr(e),cons(cadr(e),n),(RTN))} else
    if eq(car(e),LET) then
                    {{complis(args,n,cons(LDF,cons(body,cons(AP,c))))
                        where body = comp(cadr(e),m,(RTN))}
                    where m = cons(vars(cddr(e)),n)
                    and args = exprs(cddr(e))} else
    if eq(car(e),LETREC) then
                    {{cons(DUM, complis(args,m,
                        cons(LDF,cons(body,cons(RAP,c)))))
                            where body = comp(cadr(e),m, (RTN))}
                        where m = cons(vars(cddr(e)),n)
                        and args = exprs(cddr(e))} else
    complis(cdr(e),n,comp(car(e),n,cons(AP,c)))
```

```
complis(e,n,c) ≡ if eq(e,NIL) then cons(LDC,cons(NIL,c)) else
                 complis(cdr(e),n,comp(car(e),n,cons(CONS,c)))
     vars(d) ≡ if eq(d,NIL) then NIL else
                      cons(caar(d),vars(cdr(d)))
     exprs(d) ≡ if eq(d,NIL) then NIL else
                      cons(cdar(d),exprs(cdr(d)))
     cadr(x)  ≡ car(cdr(x))
     cddr(x)  ≡ cdr(cdr(x))
     cdar(x)  ≡ cdr(car(x))
     caar(x)  ≡ car(car(x))
     caddr(x) ≡ car(cdr(cdr(x)))
     cadddr(x)≡ car(cdr(cdr(cdr(x))))
```

The function complis (e,n,c) is used to compile a list of well-formed expression into code which computes a list of their values. The functions vars(d) and exprs(d) separate respectively the variables and the defining expressions in a block form of well-formed expressions. The remaining subsiduary functions are the standard composite selectors.

The SECD machine is used to apply a compiled function to its arguments. The following instructions to load the SECD machine registers will accomplish that, where fn is an SECD machine language program and args a list of its arguments.

```
s:= cons(args,nil); e:= nil; c:= fn; d:= nil;
cycle: ....
endcyle:    comment  result is in car(s);
```

To obtain fn in the correct form however, we must ensure that additional instructions are included to apply the computed function and to stop the machine. This is accomplished by the following function:

```
compile (e) ≡ comp(e,NIL, (AP STOP))
```

In the appendix, the Lispkit Lisp (source) form of this function definition is given. In addition, the SECD machine language (object) version of this function which is given is such that, if executed with the source version as argument produce the object version as result. Thus, the reader who implements the instruction execution loop of section 4 will not only be able to compile and execute any Lispkit Lisp program, but to modify and extend the language for himself. A good size for the list space is about 10K list cells, whereupon the self-compilation, requiring about 55K cells, will do about six garbage collections. The best Lispkit Lisp implementation currently available on our 370/168 [that by Mr. M.R. King] requires about 0.5 sec c.p.u. time for self compilation.

Appendix I

[1] Allen, J. Anatomy of Lisp, McGraw-Hill, 1978
[to see how elaborate Lisp has become].

[2] Berkeley, E. and Bobrow, D. The programming language LISP. Information International Cambridge, 1964.
[for everything to do with LISP].

[3] Burge, W. Recursive Programming Techniques Addison Wesley, 1976.
[for alternative description of an SECD machine].

[4] Friedman, D. The Little LISPer. Science Research Associates, 1974.
[for a very basic introduction to LISP].

[5] Landin, P. The Mechanical Evaluation of Expressions, Computer Journal, 1964.
[for a description of an SECD machine].

[6] Landin, P. The next 700 programming languages. Comm. A.C.M. 1966, Vol. 9.
[for the meaning of let, letrec and proper binding]

[7] McCarthy, J. Recursive Functions of Symbolic Expressions and their computation by machine. Comm. A.C.M., 1960.
[for an introduction to LISP and for a simple garbage collector].

[8] McCarthy, J. et.al. Lisp 1.5 Programmer's Manual. M.I.T. Press, 1962.
[the basic reference document].

[9] Knuth, D. Fundamental Algorithms. Addison Wesley, 1973.
[for alternative garbage collection algorithms].

```
(LETREC COMPILE
  (COMPILE LAMBDA (E)
    (COMP E (QUOTE NIL) (QUOTE (4 21))))
  (COMP LAMBDA (E N C)
    (IF (ATOM E)
        (CONS (QUOTE 1) (CONS (LOCATION E N) C))
    (IF (EQ (CAR E) (QUOTE QUOTE))
        (CONS (QUOTE 2) (CONS (CAR (CDR E)) C))
    (IF (EQ (CAR E) (QUOTE ADD))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 15) C)))
    (IF (EQ (CAR E) (QUOTE SUB))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 16) C)))
    (IF (EQ (CAR E) (QUOTE MUL))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 17) C)))
    (IF (EQ (CAR E) (QUOTE DIV))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 18) C)))
    (IF (EQ (CAR E) (QUOTE REM))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 19) C)))
    (IF (EQ (CAR E) (QUOTE LEQ))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 20) C)))
    (IF (EQ (CAR E) (QUOTE EQ))
        (COMP (CAR (CDR E)) N (COMP (CAR (CDR (CDR E))) N (CONS (QUOTE 14) C)))
    (IF (EQ (CAR E) (QUOTE CAR))
        (COMP (CAR (CDR E)) N (CONS (QUOTE 10) C))
    (IF (EQ (CAR E) (QUOTE CDR))
        (COMP (CAR (CDR E)) N (CONS (QUOTE 11) C))
    (IF (EQ (CAR E) (QUOTE ATOM))
        (COMP (CAR (CDR E)) N (CONS (QUOTE 12) C))
    (IF (EQ (CAR E) (QUOTE CONS))
        (COMP (CAR (CDR (CDR E))) N (COMP (CAR (CDR E)) N (CONS (QUOTE 13) C)))
    (IF (EQ (CAR E) (QUOTE IF))
        (LET (COMP (CAR (CDR E)) N (CONS (QUOTE 8)
                                         (CONS THENPT (CONS ELSEPT C))))
          (THENPT COMP (CAR (CDR (CDR E))) N (QUOTE (9)))
          (ELSEPT COMP (CAR (CDR (CDR (CDR E)))) N (QUOTE  (9)))  )
    (IF (EQ (CAR E) (QUOTE LAMBDA))
        (LET (CONS (QUOTE 3) (CONS BODY C))
          (BODY COMP (CAR (CDR (CDR E))) (CONS (CAR (CDR E)) N)
                                         (QUOTE (5))) )
    (IF (EQ (CAR E) (QUOTE LET))
         (LET  (LET  (COMPLIS ARGS N (CONS (QUOTE 3)
                                (CONS BODY (CONS (QUOTE 4) C))))
                     (BODY COMP (CAR (CDR E)) M (QUOTE (5))))
            (M CONS (VARS (CDR (CDR E))) N)
            (ARGS EXPRS (CDR (CDR E))))
    (IF (EQ (CAR E) (QUOTE LETREC))
         (LET  (LET  (CONS (QUOTE 6) (COMPLIS ARGS M
                                (CONS (QUOTE 3) (CONS BODY (CONS (QUOTE 7) C)))))
                     (BODY COMP (CAR (CDR E)) M (QUOTE (5))))
            (M CONS (VARS (CDR (CDR E))) N)
            (ARGS EXPRS (CDR (CDR E))))
    (COMPLIS (CDR E) N (COMP (CAR E) N (CONS (QUOTE 4) C))))))))))))))))))))))))))))
```

```
(COMPLIS LAMBDA (E N C)
  (IF (EQ E (QUOTE NIL)) (CONS (QUOTE 2) (CONS (QUOTE NIL) C))
      (COMPLIS (CDR E) N (COMP (CAR E) N (CONS (QUOTE 13) C)))))
(LOCATION LAMBDA (E N)
  (LETREC
    (IF (MEMBER E (CAR N)) (CONS (QUOTE 0) (POSN E (CAR N)))
        (INCAR (LOCATION E (CDR N))))
  (MEMBER LAMBDA (E N)
        (IF (EQ N (QUOTE NIL)) (QUOTE F)
        (IF (EQ E (CAR N)) (QUOTE T) (MEMBER E (CDR N)))))
  (POSN LAMBDA (E N)
    (IF (EQ E (CAR N)) (QUOTE 0) (ADD (QUOTE 1) (POSN E (CDR N)))))
  (INCAR LAMBDA (L) (CONS (ADD (QUOTE 1) (CAR L)) (CDR L)))))
(VARS LAMBDA (D)
  (IF (EQ D (QUOTE NIL)) (QUOTE NIL)
      (CONS (CAR (CAR D)) (VARS (CDR D)))))
(EXPRS LAMBDA (D)
  (IF (EQ D (QUOTE NIL)) (QUOTE NIL)
      (CONS (CDR (CAR D)) (EXPRS (CDR D)))))
)
```

```
(6 2 NIL 3 (1 (0 . 0) 2 NIL 14 8 (2 NIL 9) (2 NIL 1 (0 . 0) 11 13 1 (1 . 5) 4 1
(0 . 0) 10 11 13 9) 5) 13 3 (1 (0 . 0) 2 NIL 14 8 (2 NIL 9) (2 NIL 1 (0 . 0) 11
13 1 (1 . 4) 4 1 (0 . 0) 10 10 13 9) 5) 13 3 (6 2 NIL 3 (1 (0 . 0) 11 2 1 1 (0
. 0) 10 15 13 5) 13 3 (1 (0 . 0) 1 (0 . 1) 10 14 8 (2 0 9) (2 1 2 NIL 1 (0 . 1)
11 13 1 (0 . 0) 13 1 (1 . 1) 4 15 9) 5) 13 3 (1 (0 . 1) 2 NIL 14 8 (2 F 9) (1
(0 . 0) 1 (0 . 1) 10 14 8 (2 T 9) (2 NIL 1 (0 . 1) 11 13 1 (0 . 0) 13 1 (1 . 0)
4 9) 9) 5) 13 3 (2 NIL 1 (1 . 1) 10 13 1 (1 . 0) 13 1 (0 . 0) 4 9 (2 NIL 1 (1 .
1) 10 13 1 (1 . 0) 13 1 (0 . 1) 4 2 0 13 9) (2 NIL 2 NIL 1 (1 . 1) 11 13 1 (1 .
0) 13 1 (2 . 3) 4 13 1 (0 . 2) 4 9) 5) 7 5) 13 3 (1 (0 . 0) 2 NIL 14 8 (1 (0 .
2) 2 NIL 13 2 2 13 9) (2 NIL 2 NIL 1 (0 . 2) 2 13 13 13 1 (0 . 1) 13 1 (0 . 0)
10 13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 13 1 (1 . 2) 4 9) 5) 13 3 (1 (0
. 0) 12 8 (1 (0 . 2) 2 NIL 1 (0 . 1) 13 1 (0 . 0) 13 1 (1 . 3) 4 13 2 1 13 9)
(1 (0 . 0) 10 2 QUOTE 14 8 (1 (0 . 2) 1 (0 . 0) 11 10 13 2 2 13 9) (1 (0 . 0)
10 2 ADD 14 8 (2 NIL 2 NIL 1 (0 . 2) 2 15 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10
13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10
2 SUB 14 8 (2 NIL 2 NIL 1 (0 . 2) 2 16 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13
1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2
MUL 14 8 (2 NIL 2 NIL 1 (0 . 2) 2 17 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13 1
(1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 DIV
14 8 (2 NIL 2 NIL 1 (0 . 2) 2 18 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13 1 (1
. 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 REM 14
8 (2 NIL 2 NIL 1 (0 . 2) 2 19 13 13 1 (( . 1) 13 1 (0 . 0) 11 11 10 13 1 (1 .
1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 LEQ 14 8
(2 NIL 2 NIL 1 (0 . 2) 2 20 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13 1 (1 . 1)
4 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 EQ 14 8 (2
NIL 2 NIL 1 (0 . 2) 2 14 13 13 1 (0 . 1) 13 1 (0 . 0) 11 11 10 13 1 (1 . 1) 4
13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 CAR 14 8 (2
NIL 1 (0 . 2) 2 10 13 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 .
0) 10 2 CDR 14 8 (2 NIL 1 (0 . 2) 2 11 13 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1
(1 . 1) 4 9) (1 (0 . 0) 10 2 ATOM 14 8 (2 NIL 1 (0 . 2) 2 12 13 13 1 (0 . 1) 13
1 (0 . 0) 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 CONS 14 8 (2 NIL 2 NIL 1 (0 .
2) 2 13 13 13 1 (0 . 1) 13 1 (0 . 0) 11 10 13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0
. 0) 11 11 10 13 1 (1 . 1) 4 9) (1 (0 . 0) 10 2 IF 14 8 (2 NIL 2 NIL 2 (9) 13 1
(0 . 1) 13 1 (0 . 0) 11 11 11 10 13 1 (1 . 1) 4 13 2 NIL 2 (9) 13 1 (0 . 1) 13
1 (0 . 0) 11 11 10 13 1 (1 . 1) 4 13 3 (2 NIL 1 (1 . 2) 1 (0 . 1) 13 1 (0 . 0)
13 2 8 13 13 1 (1 . 1) 13 1 (1 . 0) 11 10 13 1 (2 . 1) 4 5) 4 9) (1 (0 . 0) 10
2 LAMBDA 14 8 (2 NIL 2 NIL 2 (5) 13 1 (0 . 1) 1 (0 . 0) 11 10 13 13 1 (0 . 0)
11 11 10 13 1 (1 . 1) 4 13 3 (1 (1 . 2) 1 (0 . 0) 13 2 3 13 5) 4 9) (1 (0 . 0)
10 2 LET 14 8 (2 NIL 2 NIL 1 (0 . 0) 11 11 13 1 (1 . 5) 4 13 1 (0 . 1) 2 NIL 1
(0 . 0) 11 11 13 1 (1 . 4) 4 13 13 3 (2 NIL 2 NIL 2 (5) 13 1 (0 . 0) 13 1 (1 .
0) 11 10 13 1 (2 . 1) 4 13 3 (2 NIL 1 (2 . 2) 2 4 13 1 (0 . 0) 13 2 3 13 13 1
(2 . 1) 13 1 (1 . 1) 13 1 (3 . 2) 4 5) 4 5) 4 9) (1 (0 . 0) 10 2 LETREC 14 8 (2
NIL 2 NIL 1 (0 . 0) 11 11 13 1 (1 . 5) 4 13 1 (0 . 1) 2 NIL 1 (0 . 0) 11 11 13
1 (1 . 4) 4 13 13 3 (2 NIL 2 NIL 2 (5) 13 1 (0 . 0) 13 1 (1 . 0) 11 10 13 1 (2
. 1) 4 13 3 (2 NIL 1 (2 . 2) 2 7 13 1 (0 . 0) 13 2 3 13 13 1 (1 . 0) 13 1 (1 .
1) 13 1 (3 . 2) 4 2 6 13 5) 4 5) 4 9) (2 NIL 2 NIL 1 (0 . 2) 2 4 13 13 1 (0 .
1) 13 1 (0 . 0) 10 13 1 (1 . 1) 4 13 1 (0 . 1) 13 1 (0 . 0) 11 13 1 (1 . 2) 4
9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 9) 5) 13 3 (2 NIL 2 (4 21) 13 2
NIL 13 1 (0 . 0) 13 1 (1 . 1) 4 5) 13 3 (1 (0 . 0) 5) 7 4 21)
```