MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 353

March 10, 1976

LAMBDA

THE ULTIMATE IMPERATIVE

by

Guy Lewis Steele Jr. and Gerald Jay Sussman

Abstract:

We demonstrate how to model the following common programming constructs in terms of an applicative order language similar to LISP:

Simple Recursion
Iteration
Compound Statements and Expressions
GO TO and Assignment
Continuation-Passing
Escape Expressions
Fluid Variables
Call by Name, Call by Need, and Call by Reference

The models require only (possibly self-referent) lambda application, conditionals, and (rarely) assignment. No complex data structures such as stacks are used. The models are transparent, involving only local syntactic transformations.

Some of these models, such as those for GO TO and assignment, are already well known, and appear in the work of Landin, Reynolds, and others. The models for escape expressions, fluid variables, and call by need with side effects are new. This paper is partly tutorial in intent, gathering all the models together for purposes of context.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Contents

Introduction 1
1. Simple Loops 2
1.1. Simple Recursion 2
1.2. Iteration 2
2. Imperative Programming 5
2.1. Compound Statements 5
2.2. The GO TO Statement 6
2.3. Simple Assignment 7
2.4. Compound Expressions 8
3. Continuations 10
3.1. Continuation-Passing Recursion 10
3.2. Escape Expressions 13
3.3. Dynamic Variable Scoping 14
3.3.1. Free (Global) Variables 15
3.3.2. Dynamic Binding 16
4. Parameter Passing Mechanisms 21
4.1. Call-By-Name 21
4.2. Call-By-Need 22
4.3. Fast Call-By-Name 23
4.4. Assignment by Reference 25
Conclusions 29
Notes 30
Bibliography 36

People who like this sort of thing will find this is the sort of thing they like.
-- Abraham Lincoln

Introduction

We catalogue a number of common programming constructs. For each construct we examine "typical" usage in well-known programming languages, and then capture the essence of the semantics of the construct in terms of a common meta-language.

The lambda calculus {Note Alonzowins} is often used as such a meta-language. Lambda calculus offers clean semantics, but it is clumsy because it was designed to be a minimal language rather than a convenient one. All lambda calculus "functions" must take exactly one "argument"; the only "data type" is lambda expressions; and the only "primitive operation" is variable substitution. While its utter simplicity makes lambda calculus ideal for logicians, it is too primitive for use by programmers. The meta-language we use is a programming language called SCHEME {Note Schemepaper} which is based on lambda calculus.

SCHEME is a dialect of LISP. [McCarthy 62] It is an expression-oriented, applicative order, interpreter-based language which allows one to manipulate programs as data. It differs from most current dialects of LISP in that it closes all lambda expressions in the environment of their definition or declaration, rather than in the execution environment. {Note Closures} This preserves the substitution semantics of lambda calculus, and has the consequence that all variables are lexically scoped, as in ALGOL. [Naur 63] Another difference is that SCHEME is implemented in such a way that tail-recursions execute without net growth of the interpreter stack. {Note Schemenote} We have chosen to use LISP syntax rather than, say, ALGOL syntax because we want to treat programs as data for the purpose of describing transformations on the code. LISP supplies names for the parts of an executable expression and standard operators for constructing expressions and extracting their components. The use of LISP syntax makes the structure of such expressions manifest. We use ALGOL as an expository language, because it is familiar to many people, but ALGOL is not sufficiently powerful to express the necessary concepts; in particular, it does not allow functions to return functions as values. We are thus forced to use a dialect of LISP in many cases.

We will consider various complex programming language constructs and show how to model them in terms of only a few simple ones. As far as possible we will use only three control constructs from SCHEME: LAMBDA expressions, as in LISP, which are just functions with lexically scoped free variables; LABELS, which allows declaration of mutually recursive procedures {Note Labelsdef}; and IF, a primitive conditional expression. For more complex modelling we will introduce an assignment primitive (ASET). We will freely assume the existence of other common primitives, such as arithmetic functions.

The constructs we will examine are divided into four broad classes. The first is *Simple Loops*; this contains simple recursions and iterations, and an introduction to the notion of continuations. The second is *Imperative Constructs*; this includes compound statements, GO TO, and simple variable assignments. The third is *Continuations*, which encompasses the distinction between statements and expressions, escape operators (such as Landin's J-operator [Landin 65] and Reynold's escape expression [Reynolds 72]), and fluid (dynamically bound) variables. The fourth is *Parameter Passing Mechanisms*, such as ALGOL call-by-name and FORTRAN call-by-location.

Some of the models presented here are already well-known, particularly those for GO TO and assignment. [McCarthy 60] [Landin 65] [Reynolds 72] Those for escape operators, fluid variables, and call-by-need with side effects are new.

1. Simple Loops

By *simple loops* we mean constructs which enable programs to execute the same piece of code repeatedly in a controlled manner. Variables may be made to take on different values during each repetition, and the number of repetitions may depend on data given to the program.

1.1. Simple Recursion

One of the easiest ways to produce a looping control structure is to use a recursive function, one which calls itself to perform a subcomputation. For example, the familiar factorial function may be written recursively in ALGOL:

```
integer procedure fact(n); value n; integer n;
   fact := if n=0 then 1 else n*fact(n-1);
```

The invocation *fact*(*n*) computes the product of the integers from 1 to *n* using the identity $n!=n(n-1)!$ $(n>0)$. If *n* is zero, 1 is returned; otherwise *fact* calls itself recursively to compute $(n-1)!$, then multiplies the result by *n* and returns it.

This same function may be written in SCHEME as follows:

```
(DEFINE FACT
   (LAMBDA (N) (IF (= N 0) 1
                   (* N (FACT (- N 1))))))
```

SCHEME does not require an assignment to the "variable" *fact* to return a value as ALGOL does. The IF primitive is the ALGOL if-then-else rendered in LISP syntax. Note that the arithmetic primitives are prefix operators in SCHEME.

1.2. Iteration

There are many other ways to compute factorial. One important way is through the use of iteration.

A common iterative construct is the DO loop. The most general form we have seen in any programming language is the MacLISP DO [Moon 74]. It permits the simultaneous initialization of any number of control variables and the simultaneous stepping of these variables by arbitrary functions at each iteration step. The loop is terminated by an arbitrary predicate, and an arbitrary value may be returned. The DO loop may have a body, a series of expressions executed for effect on each iteration. A version of the MacLISP DO construct has been adopted in SCHEME.

The general form of a SCHEME DO is:

```
(DO ((<var1> <init1> <step1>)
     (<var2> <init2> <step2>)
     . . .
     (<varn> <initn> <stepn>))
    (<pred> <value>)
    <optional body>)
```

The semantics of this are that the variables are bound and initialized to the values of the <init*i*> expressions, which must all be evaluated in the environment outside the DO; then the predicate <pred> is evaluated in the new environment, and if TRUE, the <value> is evaluated and returned. Otherwise the <optional body> is evaluated, then each of the steppers <step*i*> is evaluated in the current environment, all the variables made to have the results as their values, the predicate evaluated again, and so on.

Using DO loops in both ALGOL and SCHEME, we may express FACT by means of iteration.

```
integer procedure fact(n); value n; integer n;
   begin
     integer m, ans;
     ans := 1;
     for m := n step -1 until 0 do ans := m*ans;
     fact := ans;
   end;
```

```
(DEFINE FACT
        (LAMBDA (N)
                (DO ((M N (- M 1))
                     (ANS 1 (* M ANS)))
                    ((= M 0) ANS))))
```

Note that the SCHEME DO loop in FACT has no body -- the stepping functions do all the work. The ALGOL DO loop has an assignment in its body; because an ALGOL DO loop can step only one variable, we need the assignment to step the the variable "manually".

In reality the SCHEME DO construct is not a primitive; it is a macro which expands into a function which performs the iteration by tail-recursion. Consider the following definition of FACT in SCHEME. Although it appears to be recursive, since it "calls itself", it is entirely equivalent to the DO loop given above, for it is the code that the DO macro expands into! It captures the essence of our intuitive notion of iteration, because execution of this program will not produce internal structures (e.g. stacks or variable bindings) which increase in size with the number of iteration steps.

```
(DEFINE FACT
   (LAMBDA (N)
       (LABELS ((FACT1 (LAMBDA (M ANS)
                           (IF (= M 0) ANS
                                  (FACT1 (- M 1)
                                         (* M ANS))))))
               (FACT1 N 1))))
```

From this we can infer a general way to express iterations in SCHEME in a manner isomorphic to the MacLISP DO. The expansion of the general DO loop

```
(DO ((<var1> <init1> <step1>)
     (<var2> <init2> <step2>)
     . . .
     (<varn> <initn> <stepn>))
    (<pred> <value>)
    <body>)
```

is this:

```
(LABELS ((DOLOOP
          (LAMBDA (DUMMY <var1> <var2> ... <varn>)
              (IF <pred> <value>
                  (DOLOOP <body> <step1> <step2> ... <stepn>)))))
        (DOLOOP NIL <init1> <init2> ... <initn>))
```

The identifiers DOLOOP and DUMMY are chosen so as not to conflict with any other identifiers in the program.

Note that, unlike most implementations of DO, there are no side effects in the steppings of the iteration variables. DO loops are usually modelled using assignment statements. For example:

```
for x := a step b until c do <statement>;
```

can be modelled as follows: [Naur 63]

```
begin
    x := a;
 L: if (x-c)*sign(b) > 0 then go to Endloop;
    <statement>;
    x := x+b;
    go to L;
 Endloop:
end;
```

Later we will see how such assignment statements can in general be modelled without using side effects.

2. Imperative Programming

Lambda calculus (and related languages, such as "pure LISP") is often used for modelling the applicative constructs of programming languages. However, they are generally thought of as inappropriate for modelling imperative constructs. In this section we show how imperative constructs may be modelled by applicative SCHEME constructs.

2.1. Compound Statements

The simplest kind of imperative construct is the statement sequencer, for example the compound statement in ALGOL:

```
begin
        S1;
        S2;
end
```

This construct has two interesting properties:
(1) It performs statement S1 <u>before</u> S2, and so may be used for sequencing.
(2) S1 is useful only for its side effects. (In ALGOL, S2 must also be a statement, and so is also useful only for side effects, but other languages have compound expressions containing a statement followed by an expression.) The ALGOL compound statement may actually contain any number of statements, but such statements can be expressed as a series of nested two-statement compounds. That is:

```
begin
        S1;
        S2;
        ...
        Sn-1;
        Sn;
end
```

is equivalent to:

```
begin
        S1;
        begin
                S2;
                begin
                        ...
                                begin
                                        Sn-1;
                                        Sn;
                                end;
                        ...
                end;
        end;
end
```

It is not immediately apparent that this sequencing can be expressed in a purely applicative language. We can, however, take advantage of the implicit sequencing of applicative order evaluation. Thus, for example, we may write a two-statement sequence as follows:

```
((LAMBDA (DUMMY) S2) S1)
```

where DUMMY is an identifier not used in S2. From this it is manifest that the value of S1 is ignored, and so is useful only for side effects. (Note that we did not claim that S1 is expressed in a purely applicative language, but only that the sequencing can be so expressed.) From now on we will use the form (BLOCK S1 S2) as an abbreviation for this expression, and (BLOCK S1 S2 ... Sn-1 Sn) as an abbreviation for

```
(BLOCK S1 (BLOCK S2 (BLOCK ... (BLOCK Sn-1 Sn)...)))
```

2.2. The GO TO Statement

A more general imperative structure is the compound statement with labels and GO TOs. Consider the following code fragment due to Jacopini, taken from Knuth: [Knuth 74]

```
begin
L1:     if B1 then go to L2;
        S1;
        if B2 then go to L2;
        S2;
        go to L1;
L2:     S3;
end
```

It is perhaps surprising that this piece of code can be syntactically transformed into a purely applicative style. For example, in SCHEME we could write:

```
(LABELS ((L1 (LAMBDA ()
                     (IF B1 (L2)
                         (BLOCK S1
                                (IF B2 (L2)
                                    (BLOCK S2 (L1)))))))
         (L2 (LAMBDA () S3)))
        (L1))
```

As with the DO loop, this transformation depends critically on SCHEME's treatment of tail-recursion and on lexical scoping of variables. The labels are names of functions of no arguments. In order to "go to" the labeled code, we merely call the function named by that label.

2.3. Simple Assignment

Of course, all this sequencing of statements is useless unless the statements have side effects. An important side effect is assignment. For example, one often uses assignment to place intermediate results in a named location (i.e. a variable) so that they may be used more than once later without recomputing them:

```
begin
   real a2, sqrtdisc;
   a2 := 2*a;
   sqrtdisc := sqrt(b↑2 - 4*a*c);
   root1 := (- b + sqrtdisc) / a2;
   root2 := (- b - sqrtdisc) / a2;
   print(root1);
   print(root2);
   print(root1 + root2);
end
```

It is well known that such naming of intermediate results may be accomplished by calling a function and binding the formal parameter variables to the results:

```
((LAMBDA (A2 SQRTDISC)
         ((LAMBDA (ROOT1 ROOT2)
                  (BLOCK (PRINT ROOT1)
                         (PRINT ROOT2)
                         (PRINT (+ ROOT1 ROOT2))))
          (/ (+ (- B) SQRTDISC) A2)
          (/ (- (- B) SQRTDISC) A2)))
 (* 2 A)
 (SQRT (- (↑ B 2) (* 4 A C))))
```

This technique can be extended to handle all simple variable assignments which appear as statements in blocks, even if arbitrary GO TO statements also appear in such blocks. {Note Mccarthywins}

For example, here is a program which uses GO TO statements in the form presented before; it determines the parity of a non-negative integer by counting it down until it reaches zero.

```
begin
L1:     if a = 0 then begin parity := 0; go to L2; end;
        a := a - 1;
        if a = 0 then begin parity := 1; go to L2; end;
        a := a - 1;
        go to L1;
L2:     print(parity);
end
```

This can be expressed in SCHEME:

```
(LABELS ((L1 (LAMBDA (A PARITY)
                     (IF (= A 0) (L2 A 0)
                         (L3 (- A 1) PARITY))))
         (L3 (LAMBDA (A PARITY)
                     (IF (= A 0) (L2 A 1)
                         (L1 (- A 1) PARITY))))
         (L2 (LAMBDA (A PARITY)
                     (PRINT PARITY))))
        (L1 A PARITY))
```

The trick is to pass the set of variables which may be altered as arguments to the label functions. {Note Flowgraph} It may be necessary to introduce new labels (such as L3) so that an assignment may be transformed into the binding for a function call. At worst, one may need as many labels as there are statements (not counting the eliminated assignment and GO TO statements).

2.4. Compound Expressions

At this point we are almost in a position to model the most general form of compound statement. In LISP, this is called the "PROG feature". In addition to statement sequencing and GO TO statements, one can return a value from a PROG by using the RETURN statement.

Let us first consider the simplest compound statement, which in SCHEME we call BLOCK. Recall that

`(BLOCK S1 S2)` is defined to be `((LAMBDA (DUMMY) S2) S1)`

Notice that this not only performs S1 before S2, but also returns the value of S2. Furthermore, we defined (BLOCK S1 S2 ... Sn) so that it returns the value of Sn. Thus BLOCK may be used as a compound expression, and models a LISP PROGN, which is a PROG with no GO TO statements and an implicit RETURN of the last "statement" (really an expression).

Most LISP compilers compile DO expressions by macro-expansion. We have already seen one way to do this in SCHEME using only variable binding. A more common technique is to expand the DO into a PROG, using variable assignments instead of bindings. Thus the iterative factorial program:

```
(DEFINE FACT
        (LAMBDA (N)
                (DO ((M N (- M 1))
                     (ANS 1 (* M ANS)))
                    ((= M 0) ANS))))
```

would expand into:

```
(DEFINE FACT
        (LAMBDA (N)
                (PROG (M ANS)
                      (SSETQ M    N
                             ANS  1)
                 LP   (IF (= M 0) (RETURN ANS))
                      (SSETQ M    (- M 1)
                             ANS  (* M ANS))
                      (GO LP))))
```

where SSETQ is a simultaneous multiple assignment operator. (SSETQ is not a SCHEME (or LISP) primitive. It can be defined in terms of a single assignment operator, but we are more interested here in RETURN than in simultaneous assignment. The SSETQ's will all be removed anyway and modelled by lambda binding.) We can apply the same technique we used before to eliminate GO TO statements and assignments from compound statements:

```
(DEFINE FACT
        (LAMBDA (N)
                (LABELS ((L1 (LAMBDA (M ANS)
                                      (LP N 1)))
                         (LP (LAMBDA (M ANS)
                                     (IF (= M 0) (RETURN ANS)
                                         (L2 M ANS))))
                         (L2 (LAMBDA (M ANS)
                                     (LP (- M 1) (* M ANS)))))
                        (L1 NIL NIL))))
```

We still haven't done anything about RETURN. Let's see...
==> the value of (FACT 0) is the value of (L1 NIL NIL)
==> which is the value of (LP 0 1)
==> which is the value of (IF (= 0 0) (RETURN 1) (L2 0 1))
==> which is the value of (RETURN 1) Notice that if RETURN were the identity function (LAMBDA (X) X), we would get the right answer. This is in fact a general truth: if we just replace a call to RETURN with its argument, then our old transformation on compound statements extends to general compound expressions, i.e. PROG.

3. Continuations

Up to now we have thought of SCHEME's LAMBDA expressions as functions, and of a combination such as (G (F X Y)) as meaning "apply the function F to the values of X and Y, and *return a value* so that the function G can be applied and return a value ..." But notice that we have seldom used LAMBDA expressions as functions. Rather, we have used them as control structures and environment modifiers. For example, consider the expression:

```
(BLOCK (PRINT 3) (PRINT 4))
```

This is defined to be an abbreviation for:

```
((LAMBDA (DUMMY) (PRINT 4)) (PRINT 3))
```

We do not care about the value of this BLOCK expression; it follows that we do not care about the value of the (LAMBDA (DUMMY) ...). We are not using LAMBDA as a function at all.

It is possible to write useful programs in terms of LAMBDA expressions in which we never care about the value of *any* lambda expression. We have already demonstrated how one could represent any "FORTRAN" program in these terms: all one needs is PROG (with GO and SETQ), and PRINT to get the answers out. The ultimate generalization of this imperative programming style is *continuation-passing*. {Note Churchwins}

3.1. Continuation-Passing Recursion

Consider the following alternative definition of FACT. It has an extra argument, the *continuation*, which is a function to call with the answer, when we have it, rather than return a value.

```
procedure fact(n, c); value n, c;
    integer n; procedure c(integer value);
    if n=0 then c(1) else
      begin
        procedure temp(a) value a; integer a;
            c(n*a);
        fact(n-1, temp);
      end;
```

```
(DEFINE FACT
   (LAMBDA (N C)
      (IF (= N 0) (C 1)
          (FACT (- N 1)
                (LAMBDA (A) (C (* N A)))))))
```

It is fairly clumsy to use this version of FACT in ALGOL; it is necessary to do something like this:

```
begin
   integer ans;
   procedure temp(x); value x; integer x;
      ans := x;
   fact(3, temp);
   comment Now the variable "ans" has 6;
end;
```

Procedure *fact* does not return a value, nor does *temp*; we must use a side effect to get the answer out.

FACT is somewhat easier to use in SCHEME. We can call it like an ordinary function in SCHEME if we supply an identity function as the second argument. For example, (FACT 3 (LAMBDA (X) X)) returns 6. Alternatively, we could write (FACT 3 (LAMBDA (X) (PRINT X))); we do not care about the value of this, but about what gets printed. A third way to use the value is to write

```
(FACT 3 (LAMBDA (X) (SQRT X)))
```

instead of

```
(SQRT (FACT 3 (LAMBDA (X) X)))
```

In either of these cases we care about the value of the continuation given to FACT. Thus we care about the value of FACT if and only if we care about the value of its continuation!

We can redefine other functions to take continuations in the same way. For example, suppose we had arithmetic primitives which took continuations; to prevent confusion, call the version of "+" which takes a continuation "++", etc. Instead of writing

```
(- (↑ B 2) (* 4 A C))
```

we can write

```
(↑↑ B 2
    (LAMBDA (X)
            (** 4 A C
                (LAMBDA (Y)
                        (-- X Y <the-continuation>)))))
```

where <the-continuation> is the continuation for the entire expression.

This is an obscure way to write an algebraic expression, and we would not advise writing code this way in general, but continuation-passing brings out certain important features of the computation:

[1] The operations to be performed appear in the order in which they are performed. In fact, they <u>must</u> be performed in this order. Continuation-passing removes the need for the rule about left-to-right argument evaluation.

{Note Evalorder}
[2] In the usual applicative expression there are two implicit temporary values: those of (↑ B 2) and (* 4 A C). The first of these values must be preserved over the computation of the second, whereas the second is used as soon as it is produced. These facts are *manifest* in the appearance and use of the variable X and Y in the continuation-passing version.

In short, the continuation-passing version specifies *exactly* and explicitly what steps are necessary to compute the value of the expression. One can think of conventional functional application for value as being an abbreviation for the more explicit continuation-passing style. Alternatively, one can think of the interpreter as supplying to each function an implicit default continuation of one argument. This continuation will receive the value of the function as its argument, and then carry on the computation. In an interpreter this implicit continuation is represented by the control stack mechanism for function returns.

Now consider what computational steps are implied by:

```
(LAMBDA (A B C ...) (F X Y Z ...))
```

When we "apply" the LAMBDA expression we have some values to apply it to; we let the names A, B, C ... refer to these values. We then determine the values of X, Y, Z ... and pass these values (along with "the buck", i.e. control!) to the lambda expression F (F is either a lambda expression or a name for one). Passing control to F is an *unconditional transfer*. {Note Jrsthack} {Note Hewitthack}

Note that we want *values* from X, Y, Z, ... If these are simple expressions, such as variables, constants, or LAMBDA expressions, the evaluation process is trivial, in that no temporary storage is required. In pure continuation-passing style, all evaluations are trivial: no combination is nested within another, and therefore no "hidden temporaries" are required. But if X is a combination, say (G P Q), then we want to think of G as a function, because we want a value from it, and we will need an implicit temporary place to keep the result while evaluating Y and Z. (An interpreter usually keeps these temporary places in the control stack!) On the other hand, we do *not* necessarily need a value from F. This is what we mean by tail-recursion: F is called tail-recursively, whereas G is not. A better name for tail-recursion would be "tail-transfer", since no real recursion is implied. This is why we have made such a fuss about tail-recursion: it can be used for transfer of control without making any commitment about whether the expression expected to return a value. Thus it can be used to model statement-like control structures. Put another way, tail-recursion does not require a control stack as nested recursion does. In our models of iteration and imperative style all the LAMBDA expressions used for control (to simulate GO statements, for example) are called in tail-recursive fashion.

3.2. Escape Expressions

Reynolds [Reynolds 72] defines the construction

escape *x* in *r*

to evaluate the expression *r* in an environment such that the variable *x* is bound to an escape function. If the escape function is never applied, then the value of the escape expression is the value of *r*. If the escape function is applied to an argument *a*, however, then evaluation of *r* is aborted and the escape expression returns *a*. {Note J-operator} (Reynolds points out that this definition is not quite accurate, since the escape function may be called even after the escape expression has returned a value; if this happens, it "returns again"!)

As an example of the use of an escape expression, consider this procedure to compute the harmonic mean of an array of numbers. If any of the numbers is zero, we want the answer to be zero. We have a function *harmsum* which will sum the reciprocals of numbers in an array, or call an escape function with zero if any of the numbers is zero. (The implementation shown here is awkward because ALGOL requires that a function return its value by assignment.)

```
begin
        real procedure harmsum(a, n, escfun);
                real array a; integer n; real procedure escfun(real);
             begin
                  real sum;
                  sum := 0;
                  for i := 0 until n-1 do
                         begin
                              if a[i]=0 then escfun(0);
                              sum := sum + 1/a[i];
                         end;
                  harmsum := sum;
             end;
        real array b[0:99];
        print(escape x in 100/harmsum(b, 100, x));
end
```

If *harmsum* exits normally, the number of elements is divided by the sum and printed. Otherwise, zero is returned from the escape expression and printed without the division ever occurring.

This program can be written in SCHEME using the built-in escape operator CATCH:

```
(LABELS ((HARMSUM
          (LAMBDA (A N ESCFUN)
              (LABELS ((LOOP
                        (LAMBDA (I SUM)
                            (IF (= I N) SUM
                                (IF (= (A I) 0) (ESCFUN 0)
                                    (LOOP (+ I 1)
                                          (+ SUM (/ 1 (A I))))))))))
                      (LOOP 0 0)))))
        (BLOCK (ARRAY B 100)
               (PRINT (CATCH X (/ 100 (HARMSUM B 100 X))))))
```

This actually works, but elucidates very little about the nature of ESCAPE. We can eliminate the use of CATCH by using continuation-passing. Let us do for HARMSUM what we did earlier for FACT. Let it take an extra argument C, which is called as a function on the result.

```
(LABELS ((HARMSUM
          (LAMBDA (A N ESCFUN C)
              (LABELS ((LOOP
                        (LAMBDA (I SUM)
                            (IF (= I N) (C SUM)
                                (IF (= (A I) 0) (ESCFUN 0)
                                    (LOOP (+ I 1)
                                          (+ SUM (/ 1 (A I))))))))))
                      (LOOP 0 0)))))
        (BLOCK (ARRAY B 100)
               (LABELS ((AFTER-THE-CATCH
                         (LAMBDA (Z) (PRINT Z))))
                       (HARMSUM B
                                100
                                AFTER-THE-CATCH
                                (LAMBDA (Y) (AFTER-THE-CATCH (/ 100 Y)))))))
```

Notice that if we use ESCFUN, then C does not get called. In this way the division is avoided. This example shows how ESCFUN may be considered to be an "alternate continuation".

3.3. Dynamic Variable Scoping

In this section we will consider the problem of dynamically scoped, or "fluid", variables. These do not exist in ALGOL, but are typical of many LISP implementations, ECL, and APL. We will see that fluid variables may be modelled in more than one way, and that one of these is closely related to continuation-passing.

3.3.1. Free (Global) Variables

Consider the following program to compute square roots:

```
(DEFINE SQRT
        (LAMBDA (X EPSILON)
                (PROG (ANS)
                      (SETQ ANS 1.0)
                 A    (COND ((< (ABS (-$ X (*$ ANS ANS))) EPSILON)
                             (RETURN ANS)))
                      (SETQ ANS (//$ (+$ X (//$ X ANS)) 2.0))
                      (GO A))))
```

This function takes two arguments: the radicand and the numerical tolerance for the approximation. Now suppose we want to write a program QUAD to compute solutions to a quadratic equation.

```
(DEFINE QUAD
        (LAMBDA (A B C)
                ((LAMBDA (A2 SQRTDISC)
                         (LIST (/ (+ (- B) SQRTDISC) A2)
                               (/ (- (- B) SQRTDISC) A2)))
                 (* 2 A)
                 (SQRT (- (↑ B 2) (* 4 A C)) <tolerance>))))
```

It is not clear what to write for <tolerance>. One alternative is to pick some tolerance for use by QUAD and write it as a constant in the code. This is undesirable because it makes QUAD inflexible and hard to change. Another is to make QUAD take an extra argument and pass it to SQRT:

```
(DEFINE QUAD
        (LAMBDA (A B C EPSILON)
                ...
                (SQRT ... EPSILON) ...))
```

This is undesirable because EPSILON is not really part of the problem QUAD is supposed to solve, and we don't want the user to have to provide it. Furthermore, if QUAD were built into some larger function, and that into another, all these functions would have to pass EPSILON down as an extra argument. A third possibility would be to pass the SQRT function as an argument to QUAD (don't laugh!), the theory being to bind EPSILON at the appropriate level like this:

```
(QUAD 3 4 5 (LAMBDA (X) (SQRT X <tolerance>)))
```

where we define QUAD as:

```
(DEFINE QUAD
        (LAMBDA (A B C SQRT) ...))
```

This is as bad as the second case. The user should no more have to provide a SQRT function than a tolerance for a SQRT function.

We might also consider providing several SQRT functions with several built-in tolerances (versions for single, double, and triple precision...). But then we would have to write several versions of QUAD, and several versions of anything which called QUAD.

Now suppose that not only SQRT but all the arithmetic functions were to take tolerances as arguments (to specify single or double precision, say). It would then be very inconvenient to write QUAD at all using any of the above approaches. The algorithm for QUAD is independent of tolerance considerations. What we would like is a way to say, just before running QUAD (or the larger system which calls QUAD), "I want the tolerance to be *x* from now on until I say otherwise." In some ways this is the approach taken by many compilers, such as those for FORTRAN. We could write QUAD in FORTRAN, and then tell the compiler the tolerance (precision) we want just before compilation. The tolerance would be a *free parameter* in QUAD (and in SQRT, which would take only one argument), a parameter which is not bound anywhere. Thus we would write SQRT like this:

```
(DEFINE SQRT
        (LAMBDA (X)
                (PROG (ANS)
                      (SETQ ANS 1.0)
                 A    (COND ((< (ABS (-$ X (*$ ANS ANS))) EPSILON)
                             (RETURN ANS)))
                      (SETQ ANS (//$ (+$ X (//$ X ANS)) 2.0))
                      (GO A))))
```

The variable EPSILON is *free* in SQRT. What does this mean in a lexically scoped language such as SCHEME? ALGOL provides no clues; such a free variable is not allowed. We will say that free variables in SCHEME are "bound at the top level", i.e. that around all programs is an implicit global environment in which all variables are bound; free variables refer to these global bindings. We can modify these global bindings by using assignments. Thus we might say (ASET 'EPSILON 1.0E-5), and then use QUAD for a while, and SQRT would see EPSILON as being 1.0E-5. Subsequently we might set EPSILON to some other value, and use QUAD some more with the new value in effect. Although perhaps not formally aesthetic, this solution offers a great deal in convenience.

3.3.2. Dynamic Binding

Suppose now we want to write a function FOO which uses SQRT in such a way that for FOO to compute a single-precision result it must calculate square roots in double precision. We could write:

```
(DEFINE FOO
        (LAMBDA ...
                ((LAMBDA (OLDEPSILON)
                         (BLOCK (ASET 'EPSILON (* EPSILON EPSILON))
                                ((LAMBDA (ANSWER)
                                        (BLOCK (ASET 'EPSILON OLDEPSILON)
                                               ANSWER))
                                 (... (SQRT ...) ...))))
                 EPSILON)))
```

That is, we save the current value of EPSILON, square it to double the precision, calculate the answer using SQRT, and then set EPSILON back to its original value. This will work, but is very clumsy. The setting and resetting of EPSILON reminds us of variable binding. What we would like to do is to bind EPSILON across the usage of SQRT within FOO.

We could try writing:

```
(DEFINE FOO
        (LAMBDA ...
                ((LAMBDA (EPSILON)
                         (... (SQRT ...) ...))
                 (* EPSILON EPSILON))))
```

but this will not work. Because SCHEME is a *lexically* scoped language, SQRT must always refer to the "top level" binding of EPSILON; it is not affected by the binding of EPSILON within FOO. In other dialects of LISP this *would* work; this is usually accomplished at the expense of lexical scoping. Thus, while FOO would work "correctly" in such LISP systems, some of our other examples would not. The standard view is that in such dialects functions are closed in the activation environment rather than in the definition environment, and so free variables take on values determined by the caller's environment. Fluid variables are thus considered to be a consequence of the function closing discipline. {Note Funoffun} As a result, some languages offer just lexical scoping (ALGOL and SCHEME) while others offer just dynamic scoping (most LISPs, EL1, and APL).

Some LISP dialects allow a function to be closed in either environment, thus allowing that function's free variables to be either lexical or fluid, using the "funarg device". But suppose we wanted to have two free variables in a function, one lexically scoped and the other fluidly scoped? Consider this example:

```
(DEFINE GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE
        (LAMBDA (FACTOR)
                (LAMBDA (X)
                        ((LAMBDA (EPSILON) (SQRT X))
                         (* EPSILON FACTOR)))))
```

We want GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE to return a function which will always compute a square root to a tolerance which is more precise than the current EPSILON by the factor specified. This generated function is to accept an argument X and compute SQRT in an environment in which EPSILON is dynamically bound to (* EPSILON FACTOR). Here we have a dilemma: in which environment should the (LAMBDA (X) ...) be closed? If it is closed in the definition environment, then in the expression (* EPSILON FACTOR) the variable EPSILON will refer to the top level value and not to the dynamic binding. If it is closed in the activation environment, then the variable FACTOR will refer to its dynamic binding and not to the lexical binding within GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE.

Some LISP dialects provide hybrid scoping, in which lexically bound variables are lexically scoped, and lexically free variables are "dynamically" scoped as in FOO. This is easy for a compiler to do correctly, but fairly difficult to do in an interpreter. Furthermore, it will not generally solve problems of the GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE type.

We want to treat fluid variables as interesting objects in their own right, rather than as consequences of various function closing and variable lookup disciplines. Let us distinguish fluid variables from lexically scoped variables by prefixing them with a colon. Thus :EPSILON is a reference to the fluid variable EPSILON. We can now write GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE as follows:

```
(DEFINE GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE
        (LAMBDA (FACTOR)
                (LAMBDA (X)
                        ((LAMBDA (:EPSILON) (SQRT X))
                         (* :EPSILON FACTOR)))))
```

The (LAMBDA (X) ...) is closed in the definition environment, and so FACTOR is correctly scoped, while the : in front of EPSILON indicates that it is dynamically scoped rather than referring to the top level binding. (For now we will ignore the problem of exactly what (LAMBDA (:EPSILON) ...) means.)

We want the semantics of fluid variables to be "the value of a fluid variable is determined by the caller's environment; or if not there, by his caller's environment, and so on". How can we model these semantics in a purely lexically scoped language such as SCHEME? One way for the caller to specify the values of variables is to pass them down as arguments to the called function. This leads us back around to our original definition of SQRT, in which EPSILON is passed as an argument.

Another way is to provide a way to ask the caller what the value of a fluid variable is. Suppose we let <u>every</u> function take an extra argument FENV which represents the dynamic environment for fluid variables. Then we could replace occurrences of :EPSILON by (LOOKUP 'EPSILON FENV), where LOOKUP is defined as:

```
(DEFINE LOOKUP
        (LAMBDA (VAR FENV)
                (IF (NULL FENV)
                    (TOP-LEVEL-VALUE VAR)
                    (IF (EQ VAR (CAAR FENV))
                        (CDAR FENV)
                        (LOOKUP VAR (CDR FENV))))))
```

The fluid environment FENV is structured here as a standard LISP a-list: a list of association pairs, each of which is a variable name and a value consed together.

In order to make this work we must arrange for every caller to pass its FENV to all the functions it calls, so that they may access fluid variables. Thus we would have to write:

```
(DEFINE GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE
        (LAMBDA (FACTOR FENV)
                (LAMBDA (X FENV)
                        ((LAMBDA (:EPSILON) (SQRT X FENV))
                         (* (LOOKUP 'EPSILON FENV) FACTOR FENV)))))
```

There is still the problem of modelling (LAMBDA (:EPSILON) ...); thus far all we have done is pass the same FENV from caller to caller. But all that is needed to bind a fluid variable is to add a binding to the a-list:

```
(DEFINE GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE
        (LAMBDA (FACTOR FENV)
                (LAMBDA (X FENV)
                        (SQRT X (CONS (CONS 'EPSILON
                                             (* (LOOKUP 'EPSILON FENV)
                                                FACTOR
                                                FENV))
                                      FENV)))))
```

What we have done, in effect, is to bundle all the variables that would have to be passed down into a single data structure which is passed down.

Now functions such as * (or, for that matter, GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE itself) which do not use fluid variables need not have FENV passed to them. But if we define all functions to receive FENV as an extra argument, then in practice we may uniformly suppress this fact in our notation! (This is in fact a good criterion by which to judge a language of any kind: it should allow one to suppress that which carries little information.) This demonstrates how to implement fluid variable primitives in a lexically scoped language without the problems of FOO.

Recall that the interpreter already supplies an implicit extra argument to every function, the default continuation. We stated earlier that this implicit continuation may be identified with the interpreter's control stack; just now we saw that fluid variables are scoped according to control depth rather than lexical depth. {Note Stackfluids} We can combine these two mechanisms.

We have implemented FENV as a data structure and used a separate function, LOOKUP, access it. An alternative would be to let FENV be a lookup function which accepts an identifier and returns its fluid binding. Instead of (LOOKUP 'X FENV), we write (FENV 'X). In order to create new bindings, we create a new function which "knows about" the new bindings, and passes the buck if the given variable is not among them. For example:

```
(DEFINE GENERATE-SQRT-OF-GIVEN-EXTRA-TOLERANCE
        (LAMBDA (FACTOR FENV)
                (LAMBDA (X FENV)
                         (SQRT X ((LAMBDA (EPSILON-VALUE)
                                          (LAMBDA (VAR)
                                                  (IF (EQ VAR 'EPSILON)
                                                      EPSILON-VALUE
                                                      (FENV VAR))))
                                  (* (FENV 'EPSILON)
                                     FACTOR
                                     FENV)))))))
```

The second argument to SQRT is the (LAMBDA (VAR) ...), closed in an environment in which EPSILON-VALUE has the fluid binding for EPSILON, calculated just before SQRT is called, and in which FENV has the old fluid environment.

Now that both the continuations and fluid environments are functions, we may combine them into a single function if we want. The function can take two arguments. The first is RETURN to do the continuation action, or LOOKUP to look up a variables. The second is the return value or the variable to look up. Another way would be to let the continuation take a single argument with the data packaged up: (LOOKUP X) or (RETURN X). We could then extend this set of messages to the continuation to include (ASSIGN X Y), to assign a value to a fluid variable, or (BAKTRACE <output-file>) to print a LISP 1.5 or MacLISP style backtrace. {Note Plasmafluids}

4. Parameter Passing Mechanisms

Parameter passing mechanisms, such as "call-by-name", are not usually considered to be control structures. Such mechanisms may be used to get the effects of complex control structures such as coroutines. We have seen that fluid variables are closely related to control structures. It will be instructive to model these other parameter mechanisms in SCHEME as we have modelled the more conventional control structures.

4.1. Call-By-Name

Consider this example {Note Consgenerators} of a recursive definition of an infinite sequence:

```
list procedure terms(n); value n; integer n;
    terms := cons(1/(n↑2), terms(n+1));
```

Here we have assumed the existence of a list data type in ALGOL and made the appropriate extensions. The function *cons* takes two arguments and returns a data structure such that the function *car*, when applied to the value of *cons*, returns the first argument given to *cons*; similarly the function *cdr* extracts the second argument given to *cons*. The function *terms* is intended to produce an infinite list whose elements are elements of the sequence

$$\frac{1}{1}, \frac{1}{4}, \frac{1}{9}, \ldots, \frac{1}{n^2}, \ldots$$

beginning with the n^{th} term. Thus

car(*cdr*(*cdr*(*terms*(3)))) = 1/25

If *cons* takes its arguments by value, then this function will diverge. If it takes its arguments by name, then it need not diverge. It is possible to implement *cons* in such a way that its arguments are not evaluated until *car* or *cdr* is applied to the data structure which is its value. {Note Funargcons} To explain this requires the use of functions which return functions as values. Here ALGOL fails us, and it will be necessary to use only SCHEME for explanations. For the moment, let us pretend that SCHEME has call-by-name parameters, indicated by writing each parameter, x, called by name, as (NAME X). Later we will see how to simulate call-by-name in an applicative order language.

```
(DEFINE CBN-CONS
        (LAMBDA ((NAME X) (NAME Y))
                (LAMBDA (A)
                        (IF A X Y))))
```

Notice that CBN-CONS returns a value which is a function. The components of the data structure represented by CBN-CONS applied to two arguments are the retained bindings of the variables X and Y. That is, the returned function has associated with it an environment in which X and Y are still bound to the "thunks" [Ingerman 61] for the call-by-name arguments even though CBN-CONS has returned. The reason why the arguments to CBN-CONS are not yet evaluated is that CBN-CONS never referenced them. If, however, we were to apply the returned function, it would then reference X or Y (as necessary) and return the value. Thus we may express *car* and *cdr* in this manner:

```
(DEFINE CBN-CAR (LAMBDA (S) (S T)))

(DEFINE CBN-CDR (LAMBDA (S) (S NIL)))
```

where T and NIL are the true and false Boolean constants.

In SCHEME the *terms* function is written:

```
(DEFINE TERMS
        (LAMBDA (N)
                (CBN-CONS (/ 1 (↑ N 2))
                          (TERMS (+ N 1)))))
```

Because SCHEME really uses applicative order (call-by-value), this function always diverges, but we can simulate call-by-name by use of functional arguments. {Note Landinknewthis}

```
(DEFINE TERMS
        (LAMBDA (N)
                (CBN-CONS (LAMBDA () (/ 1 (↑ N 2)))
                          (LAMBDA () (TERMS (+ N 1))))))

(DEFINE CBN-CONS
        (LAMBDA (X Y)
                (LAMBDA (A)
                        (IF A (X) (Y)))))
```

The trick here is to explicitly pass the "thunk" that an ALGOL compiler implicitly creates to handle a call-by-name parameter. The value is then accessed by calling the thunk. Since SCHEME closes the lambda expression in the lexical environment, the thunk will be evaluated in the lexical environment as it should be.

This implementation of call-by-name is incomplete. We have not yet considered the problem of assignment of a call-by-name parameter. For now we consider only access mechanisms; later we will deal with assignment.

4.2. Call-By-Need

One problem with using call-by-name is that it is inherently inefficient because several references to the same variable will require several re-evaluations of the thunk.

```
begin
   real procedure cube(x); real x;
        cube := x*x*x;
   print(cube(sqrt(5)));
end
```

In this code the square root of 5 will be calculated three times, since *cube* takes its parameter by name and references it three times. The "call-by-need" mechanism {Note Callbyneed} overcomes this difficulty. A call-by-need parameter is passed as if it were call-by-name; but when the thunk is first referenced, after computing the value it replaces itself with the value, and all subsequent references happen as if it were call-by-value. We may express this in SCHEME by:

```
(LABELS ((CUBE (LAMBDA (X) (* (X) (X) (X)))))
        (PRINT (CUBE (NEED-THUNK (LAMBDA () (SQRT 5))))))
```

where NEED-THUNK constructs a call-by-need thunk given a primitive thunk:

```
(DEFINE NEED-THUNK
        (LAMBDA (VALUE)
                ((LAMBDA (FLAG)
                         (LAMBDA ()
                                (BLOCK (IF FLAG
                                           (BLOCK (ASET 'VALUE (VALUE))
                                                  (ASET 'FLAG NIL)))
                                       VALUE)))
                 T)))
```

The function ASET is the primitive SCHEME assignment statement. It produces a <u>true side effect</u> on the value of the variable (as opposed to the assignments we have expressed in terms of binding). The use of ASET reflects the fact that the call-by-need thunk has <u>state</u>.

As before, the value of the parameter is referenced by calling it as a function. The thunk contains two state variables VALUE and FLAG. If FLAG is T, then the thunk has never been referenced, and VALUE contains the "real" (call-by-name style) thunk. When the parameter is first referenced, the real thunk is evaluated and the result stored in VALUE (thereby throwing away the real thunk, which is no longer needed), and FLAG is set to NIL.

4.3. Fast Call-By-Name

Call-by-need does not fully capture the essence of call-by-name. If a side effect occurs between two references of a parameter, the parameter will yield the same value if passed call-by-need, but may yield different values if passed call-by-name. {Note Jensensdevice} For example:

```
begin
    real dx;
    real procedure integral(lower, upper, exp, var)
        value lower, upper;
        real lower, upper, exp, var;
        begin
           real sum;
           sum := 0;
           for var := lower + (dx/2) step dx until upper do
                sum := sum + exp;
           integral := sum;
        end;
    dx := .001;
    print(4 * integral(0, 1, 1/(1+x↑2), x));
end
```

prints an approximation to *pi* by calculating

$$4\int_0^1 \frac{dx}{x^2+1}$$

which is four times the arctangent of 1. It depends on the call-by-name parameter *exp* changing value when the variable *var* is changed. This example in fact brings out two problems. First, call-by-need does not allow the value of a parameter to change when a variable used in the argument expression is modified. Second, the example presses the issue of assignment to call-by-name parameters.

The first problem can be fixed by modifying the call-by-need mechanism to notice side effects and re-evaluate the parameter if its value might have changed. Instead of NEED-THUNK, we use the following function:

```
(DEFINE MEMO-THUNK
        (LAMBDA (THUNK)
            ((LAMBDA (VALUE SAVED-COUNT)
                 (LAMBDA ()
                     (IF (= SAVED-COUNT (GLOBAL-SIDE-EFFECT-COUNT))
                         VALUE
                         (BLOCK (ASET 'SAVED-COUNT
                                      (GLOBAL-SIDE-EFFECT-COUNT))
                                (ASET 'VALUE (THUNK))
                                VALUE)))
             NIL
             -1)))
```

The variable VALUE is used as a cache for the value of the parameter; the counts are used to determine whether the cache data is valid. {Note Muddlevcells} The function GLOBAL-SIDE-EFFECT-COUNT returns a count of all the side effects that have ever occurred which might affect the value of a thunk.

The function ASET is not intended to model the user's assignment statement. It is a SCHEME function we use to model side effects. It is important that the ASETs in MEMO-THUNK do not modify the global side effect count. The user level assignment statement may be modelled by the ASSIGN functions:

```
(DEFINE ASSIGN-CALL-BY-VALUE
        (LAMBDA (VAR VAL)
                (BLOCK (INCREMENT-GLOBAL-SIDE-EFFECT-COUNT)
                       (ASET VAR VAL))))
```

{Note Envproblem} ASSIGN-CALL-BY-VALUE is used for assignment to call-by-value parameters and locally declared variables. Assignment to call-by-name variables is discussed below.

4.4. Assignment by Reference

The second problem, assignment to call-by-name parameters, may be seen in this example:

```
begin
   procedure set3(var); integer var;
        var := 3;
   integer quux;
   set3(quux);
   print(quux);
end
```

ALGOL defines assignment to a call-by-name variable to mean assignment to the object supplied as the argument, in this case *quux*. We would expect the example to print the value 3. The problem is how to cause the assignment to *var* to become an assignment to *quux*; somehow "assignment access" to *quux* must be made available to the procedure *set3*.

This is solved by some ALGOL compilers through the use of two thunks, one for access and one for assignment. We can model this in SCHEME. In order to access a parameter, we write ((CDR X)) instead of (X). In order to set the parameter to a new value A, we write ((CAR X) A). Thus we may define:

```
(DEFINE ASSIGN-CALL-BY-NAME
        (LAMBDA (VAR VAL) ((CAR VAR) VAL)))
```

For arguments which are not variables (i.e. they cannot be assigned to), the argument (say *sqrt*(5)) is modelled as follows:

```
(CONS (LAMBDA (NEWVAL) (ERROR))
      (LAMBDA () (SQRT 5)))
```

If an argument is a variable, say QUUX, which is not itself a call-by-name parameter, we write:

```
(CONS (LAMBDA (NEWVAL) (ASSIGN-CALL-BY-VALUE 'QUUX NEWVAL))
      (LAMBDA () QUUX))
```

If QUUX is a call-by-name parameter, we could write:

```
(CONS (LAMBDA (NEWVAL) ((CAR QUUX) NEWVAL))
      (LAMBDA () ((CDR QUUX))))
```

thereby passing the buck to QUUX's thunks. However, it also works simply to write:

```
QUUX
```

which will also pass the buck correctly! (This was pointed out to the authors by Richard M. Stallman.) Of course, the access thunk for each of these may have a call to MEMO-THUNK wrapped around it to increase its efficiency.

As an example of this call-by-name transformation, consider this ALGOL program:

```
begin
   integer procedure foo(x, y); integer x, y;
        begin
               x := y + 1;
               foo := x + y;
        end;
   integer z;
   z := 4;
   print(foo(z, z + 2));
end
```

The value printed will be 16. When *foo* is called, it first references *y*, which is call-by-name bound to *z*+2; since *z* is 4, this yields 6. This is added to 1, and the resulting 7 is assigned to *x*, which is call-by-name bound to *z*, and so 7 is assigned to *z*. Then both *x* and *y* are referenced, which are *z* and *z*+2 respectively, yielding 7 and 9. The sum, 16, becomes the value of *foo* and this is printed.

Now consider this same program written in SCHEME using the call-by-name transformations we have developed:

```
(LABELS ((FOO (LAMBDA (X Y)
                      (BLOCK ((CAR X) (+ ((CDR Y)) 1))
                             (+ ((CDR X)) ((CDR Y)))))))
        ((LAMBDA (Z)
                 (BLOCK (ASET 'Z 4)
                        (PRINT (FOO (CONS (LAMBDA (NEWVAL) (ASET 'Z NEWVAL))
                                          (LAMBDA () Z))
                                    (CONS (LAMBDA (NEWVAL) (ERROR))
                                          (LAMBDA () (+ Z 2)))))))
         NIL))
```

In executing this, after Z is set to 4, FOO is called with the two sets of thunks as arguments. First (CDR Y), i.e. (LAMBDA () (+ Z 2)), is called as a function, yielding 6. This is added to 1, and (CAR X), i.e. (LAMBDA (NEWVAL) (ASET 'Z NEWVAL)), is called on the result, thereby setting Z to 7. Next both (CDR X) and (CDR Y) are called, yielding 7 and 9 respectively; FOO returns the sum 16, which is then printed. Thus the SCHEME version reflects directly the semantics of the ALGOL version, but using only call-by-value parameters.

The use of two kinds of thunks is similar to the notion of having two kinds of values, called L-values and R-values. The distinction is that an L-value may be assigned to, while an R-value is a pure value. LISP has only R-values. One cannot write (SETQ (CAR X) 'B) to get the effect of (RPLACA X 'B). By the time the CAR operation has happened, the information about where it came from is lost. CPL and related languages {Note Cplstuff} have evaluation modes: most operators evaluate their arguments in R-mode, but assignment evaluates its left argument in L-mode and its right argument in R-mode. The L-mode result is a pointer to the place to store the new value. In ECL [Wegbreit 74a] [Wegbreit 74b], one may write X.CAR←3; X.CAR returns an assignable value, because all expressions are evaluated in L-mode. [Wegbreit 70] This is implemented by always returning a pointer to where the car of X may be found. If this pointer is used for value, the pointer is implicitly followed to get the value; if used in an assignment context, the new value is placed in the location pointed to. BLISS always treats an occurrence of a variable name as an L-value; a special "." operator is used to convert an L-value to an R-value. Thus "X←Y" does not give the variable X the same value as Y, but a value which points to Y; to get the effect of (SETQ X Y) one must write "X←.Y". [BLISS 70] [Wulf 71]

We can easily modify our thunk strategy so that we could write, for example:

```
begin
    procedure clobber3(y); list y;
        y := 3;
    clobber3(car(x));
end
```

and expect the car of *x* to be altered to 3. All we need do is supply appropriate value and assignment thunks:

```
(LABELS ((CLOBBER3
           (LAMBDA (Y) ((CAR Y) 3))))
        (CLOBBER3 (CONS (LAMBDA (NEWCAR) (RPLACA X NEWCAR))
                        (LAMBDA () (CAR X)))))
```

The first thunk handles assignment to the car of X, and the second handles references to it for value.

This works when the function CAR appears explicitly in the actual argument to a called procedure. But suppose we write:

```
begin
    procedure clobber3(y); list y;
        y := 3;
    list procedure fourth(z); list z;
        fourth := car(cdr(cdr(cdr(z))));
    clobber3(fourth(x));
end
```

If we consider only the body of the procedure *clobber3* and the call on it, it is not clear how to write the thunks in SCHEME, since we cannot tell that the last thing *fourth* does is a CAR. The general solution would involve having <u>all</u> values really be two thunks. If *fourth* returned two thunks, then they would be passed to *clobber3*. But this is the same as always passing around a pointer to the value as ECL does; the assignment thunk knows where a datum came from, so that it may assign to it.

Conclusions

We have expressed a number of programming constructs in terms of a simple applicative language, SCHEME, based on lambda calculus. It is not surprising that this is possible, since SCHEME is universal. What is surprising is that the translation is so natural. Most of the translations are syntactically local. The translated program is recognizably equivalent to the original, because the global structure is preserved. The translation process does not increase the size of the program very much.

Landin [Landin 65] and Reynolds [Reynolds 72] have used similar techniques to model programming constructs. However, their modelling languages contained much more machinery than what we have used in SCHEME. For example, Landin introduces a special J-operator to model GO TO, and L-values to model assignment. We show that GO TO and most assignments can be modelled using only the lambda-binding mechanism.

The transformations we provide for escape expressions and general L-values (i.e. L-values for all data structures, not just variables) are not as syntactically local as the others. The complexity of these transformations may indicate that escape expressions and L-values are not subsumed by the mechanism of lambda-binding (except in the trivial sense that lambda-binding is Turing-universal). If they turn out to be desirable constructs, they should be implemented as primitives.

It has been suggested that certain programming language constructs, in particular the GO TO, lend themselves to obscure coding practices. Some language designers have even gone so far as to design languages which purposely omit such familiar constructs as GO TO in an attempt to constrain the programmer to refrain from particular styles of programming thought by the language designer to be "bad" in some sense. {Note Gotophobia} But any language with function calls, functional values, conditionals, correct handling of tail-recursions, and lexical scoping can simulate such "non-structured" constructs as GO TO statements, call-by-name, and fluid variables in a straightforward manner. If the language also has a macro processor or preprocessor, these simulations will even be convenient to use. {Note Features}

No amount of language design can force a programmer to write clear programs. If the programmer's conception of the problem is badly organized, then his program will also be badly organized. The extent to which a programming language can help a programmer to organize his problem is precisely the extent to which it provides features appropriate to his problem domain. The emphasis should not be on eliminating "bad" language constructs, but on discovering or inventing helpful ones.

Notes

{Alonzowins}

The lambda calculus was originally developed by Alonzo Church as a formal axiomatic system of logic. [Church 41] Happily, it may be re-interpreted in several interesting ways as a model for computation.

{Callbyneed}

The term "call-by-need" is due to Wadsworth. [Wadsworth 71] This technique is similar to the "delay rule" of Vuillemin. [Vuillemin 74]

{Churchwins}

Reynolds uses the term "continuation" in [Reynolds 72]. Church clearly understood the use of continuations; it is the *only* way to get anything accomplished at all in pure lambda calculus! For example, examine his definition of ordered pairs and triads on page 30 of [Church 41]. In SCHEME notation, this is:

[M, N] means (LAMBDA (A) (A M N))
2_1 means (LAMBDA (A) (A (LAMBDA (B C) B)))
2_2 means (LAMBDA (A) (A (LAMBDA (B C) C)))

where 2_1 e.g. selects the first element of a pair. (Note that these functions are isomorphic to CONS, CAR, and CDR!)

{Closures}

Most modern LISP systems, such as MacLISP [Moon 74] and InterLISP [Teitelman 74], scope variables dynamically. They often provide a special feature (the FUNARG device) for lexical scoping, but in most implementations this feature is not completely general.

{Consgenerators}

This example is from [Friedman 75]. Landin uses a similar technique to describe streams in [Landin 65]. Henderson and Morris [Henderson 76] present several examples in this vein, including an elegant solution to the samefringe problem of Hewitt [Hewitt 74] [Smith 75].

{Cplstuff}

CPL is described in [Barron 63] and [Buxton 66]. BCPL is a simplified version of CPL intended for systems programming. [Richards 69] [Richards 74] Also related to CPL is the language C, in which UNIX is written.

{Envproblem}

If the variable to be set is VAR or VAL, then this does not work because of the so-called environment problem. However, a compiler can choose the variables VAR and VAL to be different from all other variable names.

{Evalorder}

We can see that continuation-passing removes the need for the left-to-right rule if we consider the form of SCHEME expressions in continuation-passing style. In the style of Church, we can describe a SCHEME expression recursively:

(1) A variable, which evaluates to its bound value in the current environment.
(2) A constant, which evaluates to itself. Primitive operators such as + are constants.
(3) A lambda expression, which evaluates to a closure.
(4) A label expression, which evaluates its body in the new environment. The body may be any SCHEME expression. Only closures of lambda expressions may be bound to labelled variables.
(5) A conditional expression, which must evaluate its predicate recursively before choosing which consequent to evaluate. The predicate and the two consequents may be any SCHEME expressions.
(6) A combination, which must evaluate all its elements recursively before performing the application of function to arguments. The elements may be any SCHEME expressions.

We say that an expression evaluates trivially if it is in category (1), (2), or (3); or in category (4) if the label body evaluates trivially; or in category (5) if the predicate and both consequents of the conditional evaluate trivially.

Lemma: expressions which evaluate trivially have no side effects.

We say that an expression is in continuation-passing form if it is in category (1), (2), (3); or in category (4) if the label body is in continuation-passing form; or in category (5) if the predicate evaluates trivially and the consequents are in continuation-passing form; or in category (6) if all the elements of the combination evaluate trivially, including the function.

Theorem: expressions in continuation-passing form cannot depend on left-to-right argument evaluation.

Proof: all arguments to functions evaluate trivially, and so their evaluations have no side effects. Hence they may be evaluated in any order. QED

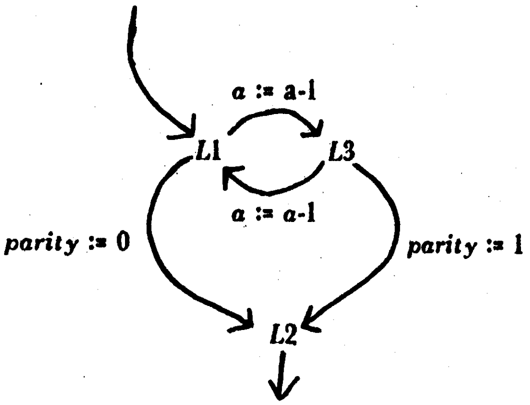
It is not too difficult to prove from this that an evaluator for expressions in continuation-passing form can be iterative; it need not be recursive or use a control stack. Another way to look at it is that continuation-passing style forces the programmer to represent recursive evaluations explicitly. What would be the control stack during evaluation of an ordinary expression is represented in environment structures in continuation-passing style.

{Features}

What if a programming language does not have all these features? Function calls and conditionals are clearly desirable features. Functional values are also valuable. It may be argued that dynamic scoping is just as good as lexical; our view is that both are desirable, and we have shown how to get dynamic scoping given lexical scoping. As for correct handling of tail-recursions, it is not difficult to see that a lexically scoped language which does not handle tail-recursions correctly is holding onto more information than is strictly necessary to execute the program.

{Flowgraph}

The reader may have noticed that the variable PARITY is uselessly passed around between L1 and L3. This could easily be optimized out by a compiler using analysis of data flow graphs. For example, the graph for the parity example would be:



From this it can be deduced that the L1-L3 loop does not alter *parity*, and that after this loop exits to L2 control cannot pass back to the loop, and so that PARITY need not be an argument to L1 or L3 in the SCHEME version.

{Funargcons}

Church understood the problem of divergent arguments; this is evident in his distinction between lambda calculus and lambda-K calculus. Fischer [Fischer 72] specifically discusses the use of functional values to simulate CONS.

{Funoffun}

Moses gives a good description of this dichotomy in [Moses 70].

{Gotophobia}

The great GO TO controversy was started by Dijkstra in 1968 [Dijkstra 68]. This issue was argued heatedly and came to a head at ACM 72. One of the proponents of GO TO-less programming was Wulf, whose language BLISS was purposely designed without GO TO statements. [BLISS 70] He soon discovered that some compensation for the omission was needed, and so exit expressions were introduced, followed by leave expressions. [Wulf 71] [Wulf 72]

The extensible language EL1 was designed before 1970, just before the GO TO statement became a real issue. It had no GO TO statement, but this was more because Wegbreit was more interested in studying extensible data types in his thesis than control structures, and he preferred to omit many control structures from EL1 rather than install a dozen features not well thought out. [Wegbreit 70, p. 417] The EL1 language definition became the basis for the ECL programming system at Harvard. [Wegbreit 71] This implementation was embellished with the GO TO statement. [Wegbreit 72] Partly because of GO TO politics and partly for implementational expediency, the GO TO was later removed from the ECL system. [Wegbreit 74b]

Nowadays it is common for a language designer to omit the GO TO statement as a matter of course. [Liskov 73] [Liskov 74] [Smith 75] Unfortunately, not all new languages which omit the GO TO provide reasonable compensation for the omission.

Knuth presents an extensive history of the GO TO controversy [Knuth 74] and asks, "Will UTOPIA 84, or perhaps we should call it NEWSPEAK, contain GO TO statements?" (p. 264) But perhaps we should ask instead, "Will UTOPIA 84 offer alternatives convenient enough that we won't need the GO TO very often?"

{Hewitthack}

Not only does an unconditional transfer to F occur, but values may be passed. One may think of these values as "messages" to be sent to the lambda expression F. This is precisely what Hewitt is flaming about (except for cells and serializers). [Smith 75]

{Jensensdevice}

The technique of repeatedly modifying a variable passed call-by-name in order to produce side effects on another call-by-name parameter is commonly known as Jensen's device, particularly in the case where the call-by-name parameters are j and $a[j]$. We cannot find any reference to Jensen or who he was, and offer a reward for any information leading to the identification, arrest, and conviction of said Jensen.

{J-operator}

The escape function is analogous to the "program point" returned by Landin's J-operator. [Landin 65] This program point contains the SECD "dump" in exactly the way a SCHEME DELTA expression contains the "clink". [Sussman 75]

{Jrsthack}

This statement is equivalent to the well-known "JRST hack", which states that the sequence of PDP-10 instructions

```
        PUSHJ P,FOO     is equivalent to     JRST FOO
        POPJ P,
```

except no stack slot is used.

{Labelsdef}

The LABELS construct of SCHEME is isomorphic to Landin's *let rec* construct [Landin 65] and Reynold's *letrec* construct [Reynolds 72]. Its purpose is to allow a function to refer to itself. It is more convenient than the more familiar LABEL construct of LISP 1.5 because it allows definition of several mutually referent functions. The general form of a LABELS construct is:

```
(LABELS ((<name1> <lambda-exp1>)
         (<name2> <lambda-exp2>)
         . . .
         (<namen> <lambda-expn>))
        <body>)
```

A new environment is created in which the names <name*i*> are bound to closures of the lambda expresisons <lambda-exp*i*>; the lambda expressions are closed in this new environment, and so may refer to each other. The <body> is then evaluated in this new environment.

The LABEL construct of LISP 1.5:

```
((LABEL <name> <lambda-exp>) <arg1> <arg2> ... <argn>)
```

may be written as a LABELS in SCHEME:

```
(LABELS ((<name> <lambda-exp>))
        (<name> <arg1> <arg2> ... <argn>))
```

{Landinknewthis}
In [Landin 65] Landin uses this same technique to model call-by-name. However, he modelled assignment to call-by-name parameters in a way much different from the one we use later: he uses L-values rather than an extra assignment thunk.

{Mccarthywins}
This was realized as early as 1960 by John McCarthy. In section 6 of [McCarthy 60] he describes a technique for transforming a flowchart into a purely recursive procedure.

{Muddlevcells}
The MDL language (formerly known as MUDDLE) [Galley 75] uses cached value cells, but uses a process number rather than a side effect count to determine the validity of the cache data, the purpose being to share a cache among several processes.

{Plasmafluids}
This indicates an obvious method for implementing fluid variables in PLASMA in a natural way. All that would be required is a slight change to the implicitly supplied continuations.

{Schemenote}
This is discussed in detail in [Sussman 75], where an actual implementation is described. The theoretical justification is described there, and later in this paper also.

{Schemepaper}
SCHEME is fully described in [Sussman 75], which contains a complete reference manual as well as a fully documented implementation of the language in MacLISP [Moon 74].

{Stackfluids}

Real stack-oriented LISP implementations ([Moon 74] [Teitelman 74] cf. [Sussman 75]) in fact either keep fluid bindings on the control stack, or use a separate stack which more or less pushes and pops in parallel with the control stack.

Bibliography

[BLISS 70]
BLISS Reference Manual. Computer Science Dept. Report. Carnegie-Mellon U. (Pittsburgh, January 1970).

[Barron 63]
Barron, D.W., et al. "The Main Features of CPL." The Computer Journal, Vol. 6, 1963, p. 134.

[Buxton 66]
Buxton, Gray, Park, and Strachey. CPL Working Papers. U. of London Institute of Computer Science (1966).

[Church 41]
Church, Alonzo. The Calculi of Lambda Conversion. Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Dijkstra 68]
Dijkstra, Edsger W. "GOTO statement considered harmful." Letter to the Editor. Comm. ACM 11, 3 (March 1968).

[Fischer 72]
Fischer, Michael J. "Lambda Calculus Schemata." Proceedings of ACM Conference on Proving Assertions about Programs. SIGPLAN Notices (January 1972).

[Friedman 75]
Friedman, Daniel P., and Wise, David S. CONS Should Not Evaluate Its Arguments. Technical Report 44. Indiana U. Computer Science Dept. (Bloomington, November 1975).

[Galley 75]
Galley, S.W. and Pfister, Greg. The MDL Language. Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).

[Henderson 75]
Henderson, Peter, and Morris, James H. Jr. "A Lazy Evaluator." SIGPLAN-SIGACT Symposium on Principles of Programming Languages (January 1976).

[Hewitt 74]
Hewitt, Carl, et al. "Behavioral Semantics of Non-recursive Control Structures." Proc. Colloque sur la Programmation. Lecture Notes in Computer Science No. 19. Springer-Verlag (1974).

[Ingerman 61]
Ingerman, P. Z. "Thunks -- A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations." CACM 4, 1 (January 1961) pp. 55-58.

[Landin 65]
Landin, Peter J. "A Correspondence between ALGOL 60 and Church's Lambda-Notation." CACM 8, 2-3 (February and March 1965).

[Liskov 73]
Liskov, Barbara, et al. CLU Design Notes. MIT Lab. for Computer Science (Cambridge, 1973-1976).

[Liskov 74]
Liskov, Barbara, and Zilles, Stephen. "Programming with Abstract Data Types." Proc. Symp. on Very High Level Languages. SIGPLAN Notices, April 1974.

[McCarthy 60]
McCarthy, John. "Recursive functions of symbolic expressions and their computation by machine - I." Comm. ACM 3, 4 (April 1960), 184-195.

[McCarthy 62]
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[Moon 74]
Moon, David A. MACLISP Reference Manual, Revision 0. Project MAC, MIT (Cambridge, April 1974).

[Moses 70]
Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Naur 63]
Naur, Peter (ed.), et al. "Revised Report on the Algorithmic Language ALGOL 60." Comm. ACM 6, 1 (January 1963), 1-20.

[Reynolds 72]
Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." ACM Conference Proceedings 1972.

[Richards 69]
Richards, Martin. "BCPL: A Tool for Compiler Writing and Systems Programming." Proc. AFIPS 1969 SJCC, Vol. 34. AFIPS Press (Montvale, N.J.) pp. 557-566.

[Richards 74]
Richards, Martin, Evans, Arthur Jr., and Mabee, Robert F. The BCPL Reference Manual. TR-141, Project MAC, MIT (Cambridge, December 1974).

[Smith 75]
Smith, Brian C. and Hewitt, Carl. A PLASMA Primer (draft). MIT AI Lab (Cambridge, October 1975).

[Sussman 75]
Sussman, Gerald Jay, and Steele, Guy L. Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Lab Memo 349. MIT (Cambridge, December 1975).

[Teitelman 74]
Teitelman, Warren. InterLISP Reference Manual. Xerox Palo Alto Research Center (Palo Alto, 1974).

[Vuillemin 74]
Vuillemin, Jean. "Correct and Optimal Implementations of Recursion in a Simple Programming Language." Journal of Computer and System Sciences 9, 3 (December 1974).

[Wadsworth 71]
Wadsworth, Christopher. Semantics and Pragmatics of the Lambda-calculus. Ph.D. Thesis. Oxford (1971).

[Wegbreit 70]
Wegbreit, Ben. Studies in Extensible Programming Languages. Ph.D. Thesis. Harvard U. (Cambridge, 1970).

[Wegbreit 71]
Wegbreit, Ben. "The ECL Programming System." Proc. AFIPS 1971 FJCC, Vol. 39. AFIPS Press, Montvale, N.J. pp. 253-262.

[Wegbreit 72]
Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 21-72. Center for Research in Computing Technology, Harvard U. (Cambridge, September 1972).

[Wegbreit 74a]
Wegbreit, Ben. "The Treatment of Data Types in EL1." Comm. ACM 17, 5 (May 1974), 251-264.

[Wegbreit 74b]
Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 23-74. Center for Research in Computing Technology, Harvard U. (Cambridge, December 1974).