Higher-Order and Symbolic Computation, 11, 405–439 (1998)

Scheme: A Interpreter for Extended Lambda Calculus

GERALD JAY SUSSMAN gjs@mit.edu
Massachusetts Institute of Technology, 545 Tech Square, Room 428, Cambridge, MA 02139, USA

GUY L. STEELE JR. guy.steele@east.sun.com
Sun Microsystems Labs, 2 Elizabeth Drive, MS UCHL03-207, Chelmsford, MA 01824, USA

Abstract. Inspired by ACTORS [7, 17], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [2], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

1. alleviate the confusion caused by Micro-PLANNER, CONNIVER, etc., by clarifying the embedding of non-recursive control structures in a recursive host language like LISP.
2. explain how to use these control structures, independent of such issues as pattern matching and data base manipulation.
3. have a simple concrete experimental domain for certain issues of programming semantics and style.

This paper is organized into sections. The first section is a short "reference manual" containing specifications for all the unusual features of SCHEME. Next, we present a sequence of programming examples which illustrate various programming styles, and how to use them. This will raise certain issues of semantics which we will try to clarify with lambda calculus in the third section. In the fourth section we will give a general discussion of the issues facing an implementor of an interpreter for a language based on lambda calculus. Finally, we will present a completely annotated interpreter for SCHEME, written in MacLISP [13], to acquaint programmers with the tricks of the trade of implementing non-recursive control structures in a recursive language like LISP.

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.

1. The SCHEME Reference Manual

SCHEME is essentially a full-funarg LISP. `LAMBDA` expressions need not be `QUOTE`d, `FUNCTION`ed, or `*FUNCTION`ed when passed as arguments or returned as values; they will evaluate to closures of themselves.

All LISP functions (i.e., `EXPR`s, `SUBR`s, and `LSUBR`s, but *not* `FEXPR`s, `FSUBR`s, or `MACRO`s) are primitive operators in SCHEME, and have the same meaning as they have in LISP. Like `LAMBDA` expressions, primitive operators and numbers are self-evaluating (they evaluate to trivial closures of themselves).

There are a number of special primitives known as `AINT`s which are to SCHEME as `FSUBR`s are to LISP. We will enumerate them here.

`IF` This is the primitive conditional operator. It takes three arguments. If the first evaluates to non-`NIL`, it evaluates the second expression, and otherwise the third.

`QUOTE` As in LISP, this quotes the argument form so that it will be passed verbatim as data. The abbreviation "`'FOO`" may be used instead of "`(QUOTE FOO)`".

DEFINE This is analogous to the MacLISP DEFUN primitive (but note that the LAMBDA must appear explicitly!). It is used for defining a function in the "global environment" permanently, as opposed to LABELS (see below), which is used for temporary definitions in a local environment. DEFINE takes a name and a lambda expression; it closes the lambda expression in the global environment and stores the closure in the LISP value cell of the name (which is a LISP atom).

LABELS We have decided not to use the traditional LABEL primitive in this interpreter because it is difficult to define several mutually recursive functions using only LABEL. The solution, which Hewitt [17] also uses, is to adopt an ALGOLesque block syntax:

```
(LABELS <function definition list> <expression>)
```

This has the effect of evaluating the expression in an environment where all the functions are defined as specified by the definitions list. Furthermore, the functions are themselves closed in that environment, and not in the outer environment; this allows the functions to call themselves *and each other* recursively. For example, consider a function which counts all the atoms in a list structure recursively to all levels, but which doesn't count the NILs which terminate lists (but NILs in the CAR of some list count). In order to perform this we use two mutually recursive functions, one to count the car and one to count the cdr, as follows:

```
(DEFINE COUNT
    (LAMBDA (L)
       (LABELS ((COUNTCAR
                 (LAMBDA (L)
                    (IF (ATOM L) 1
                        (+ (COUNTCAR (CAR L))
                           (COUNTCDR (CDR L))))))
                (COUNTCDR
                 (LAMBDA (L)
                    (IF (ATOM L)
                        (IF (NULL L) 0 1)
                        (+ (COUNTCAR (CAR L))
                           (COUNTCDR (CDR L)))))))
          (COUNTCDR L))))           ;Note: COUNTCDR is defined here.
```

ASET This is the side effect primitive. It is analogous to the LISP function SET. For example, to define a *cell* [17], we may use ASET as follows:

```
(DEFINE CONS-CELL
   (LAMBDA (CONTENTS)
      (LABELS ((THE-CELL
                (LAMBDA (MSG)
                   (IF (EQ MSG 'CONTENTS?) CONTENTS
                       (IF (EQ MSG 'CELL?) 'YES
                           (IF (EQ (CAR MSG) '<-)
                               (BLOCK (ASET 'CONTENTS (CADR MSG))
                                      THE-CELL)
                               (ERROR '|UNRECOGNIZED MESSAGE - CELL|
                                      MSG
                                      'WRNG-TYPE-ARG)))))))
         THE-CELL)))
```

Those of you who may complain about the lack of ASETQ are invited to write (ASET' foo bar) instead of (ASET 'foo bar).

EVALUATE This is similar to the LISP function EVAL. It evaluates its argument, and then evaluates the resulting s-expression as SCHEME code.

CATCH This is the "escape operator" which gives the user a handle on the control structure of the interpreter. The expression:

```
(CATCH <identifier> <expression>)
```

evaluates <expression> in an environment where <identifier> is bound to a continuation which is "just about to return from the CATCH"; that is, if the continuation is called as a function of one argument, then control proceeds as if the CATCH expression had returned with the supplied (evaluated) argument as its value. For example, consider the following obscure definition of SQRT (Sussman's favorite style/Steele's least favorite):

```
(DEFINE SQRT
   (LAMBDA (X EPSILON)
      ((LAMBDA (ANS LOOPTAG)
          (CATCH RETURNTAG
                 (PROGN
                  (ASET 'LOOPTAG (CATCH M M))         ;CREATE PROG TAG
                  (IF (< (ABS (-$ (*$ ANS ANS) X)) EPSILON)
                      (RETURNTAG ANS)                 ;RETURN
                      NIL)                            ;JFCL
                  (ASET 'ANS (//$ (+$ (//$ X ANS) ANS) 2.0))
                  (LOOPTAG LOOPTAG))))                ;GOTO
       1.0
       NIL)))
```

Anyone who doesn't understand how this manages to work probably should not attempt to use CATCH.[1]

As another example, we can define a THROW function, which may then be used with CATCH much as they are in LISP:

```
(DEFINE THROW (LAMBDA (TAG RESULT) (TAG RESULT)))
```

CREATE!PROCESS This is the process generator for multiprocessing. It takes one argument, an expression to be evaluated in the current environment as a separate parallel process. If the expression ever returns a value, the process automatically terminates. The value of CREATE!PROCESS is a process id for the newly generated process. Note that the newly created process will not actually run until it is explicitly started.

START!PROCESS This takes one argument, a process id, and starts up that process. It then runs.

`STOP!PROCESS` This also takes a process id, but stops the process. The stopped process may be continued from where it was stopped by using `START!PROCESS` again on it. The magic global variable `**PROCESS**` always contains the process id of the currently running process; thus a process can stop itself by doing `(STOP!PROCESS **PROCESS**)`. A stopped process is garbage collected if no live process has a pointer to its process id.

`EVALUATE!UNINTERRUPTIBLY` This is the synchronization primitive. It evaluates an expression uninterruptibly; i.e., no other process may run until the expression has returned a value. Note that if a funarg is returned from the scope of an `EVALUATE!UNINTERRUPTIBLY`, then that funarg will be uninterruptible when it is applied; that is, the uninterruptibility property follows the rules of variable scoping. For example, consider the following function:

```
(DEFINE SEMGEN
   (LAMBDA (SEMVAL)
      (LIST (LAMBDA ()
               (EVALUATE!UNINTERRUPTIBLY
                   (ASET' SEMVAL (+ SEMVAL 1))))
            (LABELS (P (LAMBDA ()
                         (EVALUATE!UNINTERRUPTIBLY
                             (IF (PLUSP SEMVAL)
                                 (ASET' SEMVAL (- SEMVAL 1))
                                 (P)))))
               P))))
```

This returns a pair of functions which are V and P operations on a newly created semaphore. The argument to `SEMGEN` is the initial value for the semaphore. Note that `P` busy-waits by iterating if necessary; because `EVALUATE!UNINTERRUPTIBLY` uses variable-scoping rules, other processes have a chance to get in at the beginning of each iteration. This busy-wait can be made much more efficient by replacing the expression `(P)` in the definition of `P` with

```
((LAMBDA (ME)
    (BLOCK (START!PROCESS (CREATE!PROCESS '(START!PROCESS ME)))
           (STOP!PROCESS ME)
           (P)))
 **PROCESS**)
```

Let's see you figure this one out! Note that a `STOP!PROCESS` within an `EVALUATE!UNINTERRUPTIBLY` forces the process to be swapped out even if it is the current one, and so other processes get to run; but as soon as it gets swapped in again, others are locked out as before.

Besides the `AINT`s, SCHEME has a class of primitives known as `AMACRO`s These are similar to MacLISP `MACRO`s, in that they are expanded into equivalent code before being executed. Some `AMACRO`s supplied with the SCHEME interpreter:

`COND` This is like the MacLISP `COND` statement, except that singleton clauses (where the result of the predicate is the returned value) are not allowed.

`AND, OR` These are also as in MacLISP.

`BLOCK` This is like the MacLISP `PROGN`, but arranges to evaluate its last argument without an extra net control frame (explained later), so that the last argument may involved in an iteration. Note that in SCHEME, unlike MacLISP, the body of a `LAMBDA` expression is *not* an implicit `PROGN`.[2]

`DO` This is like the MacLISP "new-style" `DO`; old-style `DO` is not supported.

`AMAPCAR, AMAPLIST` These are like `MAPCAR` and `MAPLIST`, but they expect a SCHEME lambda closure for the first argument.

To use SCHEME, simply incant at DDT (on MIT-AI):[3]

```
:LISP LIBLSP;SCHEME
```

which will load up the current version of SCHEME, which will announce itself and give a prompt. If you want to escape to LISP, merely hit `^G`. To restart SCHEME, type `(SCHEME)`. Sometimes one does need to use a LISP `FSUBR` such as `UREAD`; this may be accomplished by typing, for example,[4]

```
(EVAL' (UREAD FOO BAR DSK LOSER))
```

After doing this, typing `^Q` will, of course, cause SCHEME to read from the file.

This concludes the SCHEME Reference Manual.

2. Some SCHEME Programming Examples

2.1. Traditional Recursion

Here is the good old familiar recursive definition of factorial, written in SCHEME.

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

2.2. What About Iteration?

There are many other ways to compute factorial. One important way is through the use of *iteration*. Consider the following definition of `FACT`. Although it appears to be recursive, since it "calls itself," 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. This surprising fact will be explained in two ways.

(1) We will consider programming styles in terms of substitution semantics of the lambda calculus (Section 3).

(2) We will show how the SCHEME interpreter is implemented (Sections 4, 5).

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

A common iterative construct is the DO loop. The most general form we have seen in any programming language is the MacLISP DO [13]. 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.

The general form of a MacLISP DO is:

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

The semantics of this are that the variables are bound and initialized to the values of the <initi> 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 body is evaluated, then each of the steppers <stepi> is evaluated in the current environment, all the variables made to have the results as their values, and the predicate evaluated again, and so on.

For example, the following MacLISP function:

```
(DEFUN REV (L)
   (DO ((L1 L (CDR L1))
        (ANS NIL (CONS (CAR L1) ANS)))
       ((NULL L1) ANS)))
```

computes the reverse of a list. In SCHEME, we could write the same function, in the same iterative style, as follows:

```
(DEFINE REV
   (LAMBDA (L)
      (LABELS ((DOLOOP (LAMBDA (L1 ANS)
                         (IF (NULL L1) ANS
                             (DOLOOP (CDR L1)
                                     (CONS (CAR L1) ANS))))))
         (DOLOOP L NIL))))
```

From this we can infer a general way to express iterations in SCHEME in a manner isomorphic to the MacLISP DO:

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

This is in fact what the supplied `DO AMACRO` expands into. Note that there are no side effects in the steppings of the iteration variables.

2.3. Another Way To Do Recursion

Now consider the following alternative definition of `FACT`. It has an extra argument, the *continuation* [16], which is a function to call with the answer, when we have it, rather than return a value; that is, rather than ultimately reducing to the desired value, it reduces to a combination which is the application of the continuation to the desired value.

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

Note that we can call this like an ordinary function if we supply `(LAMBDA (X) X)` as the second argument. For example, `(FACT 3 (LAMBDA (X) X))` returns 6.

2.4. Apparently "Hairy" Control Structure

A classic problem difficult to solve in most programming languages, including standard (stack-oriented) LISP, is the *samefringe* problem [17]. The problem is to determine whether the fringes of two trees are the same, even if the internal structures of the trees are not. This problem is easy to solve if one merely computes the fringe of each tree separately as a list, and then compares the two lists. We would like to solve the problem so that the fringes are generated and compared incrementally. This is important if the fringes of the trees are very large, but differ, say, in the first position.

Consider the following obscure solution to *samefringe*, which is in fact isomorphic to the one written by Shrobe and presented by Smith and Hewitt. Note that SCHEME does not have the packagers of PLASMA, and so we were forced to use continuations; rather than using packages and a *next* operator, we pass a fringe a continuation (called the "getter") which will get the next and the rest of the fringe as its two arguments

```
(DEFINE FRINGE
   (LAMBDA (TREE)
      (LABELS ((FRINGEN
                (LAMBDA (NODE ALT)
                   (LAMBDA (GETTER)
                      (IF (ATOM NODE)
                          (GETTER NODE ALT)
```

```
                          ((FRINGEN (CAR NODE)
                                    (LAMBDA (GETTER1)
                                       ((FRINGEN (CDR NODE)
                                                 ALT)
                                        GETTER1)))
                           GETTER))))))
          (FRINGEN TREE
                   (LAMBDA (GETTER)
                      (GETTER '(EXHAUSTED) NIL))))))

  (DEFINE SAMEFRINGE
     (LAMBDA (TREE1 TREE2)
        (LABELS ((SAME
                  (LAMBDA (S1 S2)
                     (S1 (LAMBDA (X1 R1)
                            (S2 (LAMBDA (X2 R2)
                                   (IF (EQUAL X1 X2)
                                       (IF (EQUAL X1 '(EXHAUSTED))
                                           T
                                           (SAME R1 R2))
                                       NIL))))))))
          (SAME (FRINGE TREE1)
                (FRINGE TREE2)))))
```

Now let us consider an alternative solution to the *samefringe* problem. We believe that this solution is clearer for two reasons:

1. the implementation of SAMEFRINGE is more clearly iterative;
2. rather than returning an object which will return both the *first* and the *rest* of a fringe to a given continuation, FRINGE returns an object which will deliver up a component in response to a request for that component.

```
  (DEFINE FRINGE
     (LAMBDA (TREE)
        (LABELS ((FRINGE1
                  (LAMBDA (NODE ALT)
                     (IF (ATOM NODE)
                         (LAMBDA (MSG)
                            (IF (EQ MSG 'FIRST) NODE
                                (IF (EQ MSG 'NEXT) (ALT) (ERROR))))
                         (FRINGE1 (CAR NODE)
                                  (LAMBDA ()
                                     (FRINGE1 (CDR NODE) ALT)))))))
          (FRINGE1 TREE
                   (LAMBDA ()
                      (LAMBDA (MSG)
                         (IF (EQ MSG 'FIRST) '*EOF* (ERROR))))))))

  (DEFINE SAMEFRINGE
```

```
(LAMBDA (T1 T2)
   (DO ((C1 (FRINGE T1) (C1 'NEXT))
        (C2 (FRINGE T2) (C2 'NEXT)))
       ((OR (NOT (EQ (C1 'FIRST) (C2 'FIRST)))
            (EQ (C1 'FIRST) '*EOF*)
            (EQ (C2 'FIRST) '*EOF*))
        (EQ (C1 'FIRST) (C2 'FIRST))))))
```

A much simpler and more probable problem is that of building a pattern matcher with backtracking for segment matches. The matcher presented below is intended for matching single-level list structure patterns against lists of atoms. A pattern is a list containing three types of elements:

1. constant atoms, which match themselves only.

2. `(THV x)`, which matches any single element in the expression consistently. We may abbreviate this as `?x` by means of a LISP reader macro character.[5]

3. `(THV* x)`, which matches any segment of zero or more elements in the expression consistently. We may abbreviate this as `!x`.

The matcher returns either `NIL`, meaning no match is possible, or a list of two items, an alist specifying the bindings of the match variables, and a continuation to call, if you don't like this particular set of bindings, which will attempt to find another match. Thus, for example, the invocation

```
(MATCH '(A !B ?C ?C !B !E)
       '(A X Y Q Q X Y Z Z X Y Q Q X Y R))
```

would return the result

```
(((E (Z Z X Y Q Q X Y R))
  (C Q)
  (B X Y))
 <continuation1>)
```

where calling `<continuation1>` as a function of no arguments would produce the result

```
(((E (R))
  (C Z)
  (B (X Y Q Q X Y)))
<continuation2>)
```

where calling `<continuation2>` would produce `NIL`.

The `MATCH` function makes use of two auxiliary functions called `NFIRST` and `NREST`. The former returns a list of the first n elements of a given list, while the latter returns the tail of the given list after the first n elements.

```
(DEFINE NFIRST
   (LAMBDA (E N)
      (IF (= N 0) NIL
          (CONS (CAR E) (NFIRST (CDR E) (- N 1))))))

(DEFINE NREST
   (LAMBDA (E N)
      (IF (= N 0) E
          (NREST (CDR E) (- N 1)))))
```

The main MATCH function also uses a subfunction called MATCH1 which takes four arguments: the tail of the pattern yet to be matched; the tail of the expression yet to be matched; the alist of match bindings made so far; and a continuation to call if the match fails at this point. A subfunction of MATCH, called MATCH*, handles the matching of segments of the expression against THV* match variables. It is in the matching of segments that the potential need for backtracking enters, for segments of various lengths may have to be tried. After MATCH* matches a segment, it calls MATCH1 to continue the match, giving it a failure continuation which will back up and try to match a longer segment if possible. A failure can occur if a constant fails to match, or if one or the other of pattern and expression runs out before the other one does.

```
(DEFINE MATCH
  (LAMBDA (PATTERN EXPRESSION)
    (LABELS ((MATCH1
      (LAMBDA (P E ALIST LOSE)
        (IF (NULL P)
            (IF (NULL E) (LIST ALIST LOSE) (LOSE))
            (IF (ATOM (CAR P))
                (IF (NULL E) (LOSE)
                    (IF (EQ (CAR E) (CAR P))
                        (MATCH1 (CDR P) (CDR E) ALIST LOSE)
                        (LOSE)))
                (IF (EQ (CAAR P) 'THV)
                    (IF (NULL E) (LOSE)
                        ((LAMBDA (V)
                           (IF V (IF (EQ (CAR E) (CADR V))
                                     (MATCH1 (CDR P) (CDR E) ALIST LOSE)
                                     (LOSE))
                             (MATCH1 (CDR P) (CDR E)
                                     (CONS (LIST (CADAR P) (CAR E)) ALIST)
                                     LOSE)))
                         (ASSQ (CADAR P) ALIST)))
                    (IF (EQ (CAAR P) 'THV*)
                        ((LAMBDA (V)
                           (IF V
                              (IF (< (LENGTH E) (LENGTH (CADR V))) (LOSE)
                                  (IF (EQUAL (NFIRST E (LENGTH (CADR V)))
                                             (CADR V))
                                      (MATCH1 (CDR P)
                                              (NREST E (LENGTH (CADR V)))
                                              ALIST
                                              LOSE)
                                      (LOSE)))
```

```
                          (LABELS ((MATCH*
                            (LAMBDA (N)
                              (IF (> N (LENGTH E)) (LOSE)
                                  (MATCH1 (CDR P) (NREST E N)
                                          (CONS (LIST (CADAR P)
                                                      (NFIRST E N))
                                                ALIST)
                                          (LAMBDA ()
                                            (MATCH* (+ N 1)))))))))
                                   (MATCH* 0))))
                             (ASSQ (CADAR P) ALIST))
                            (LOSE))))))))
    (MATCH1 PATTERN
            EXPRESSION
            NIL
            (LAMBDA () NIL)))))
```

2.5. A Useless Multiprocessing Example

One thing we might want to use multiprocessing for is to try two things in parallel, and terminate as soon as one succeeds. We can do this with the following function.

```
(DEFINE TRY!TWO!THINGS!IN!PARALLEL
   (LAMBDA (F1 F2)
      (CATCH C
         ((LAMBDA (P1 P2)
             ((LAMBDA (F1 F2)
                 (EVALUATE!UNINTERRUPTIBLY
                    (BLOCK (ASET 'P1 (CREATE!PROCESS '(F1)))
                           (ASET 'P2 (CREATE!PROCESS '(F2)))
                           (START!PROCESS P1)
                           (START!PROCESS P2)
                           (STOP!PROCESS **PROCESS**))))
              (LAMBDA()
                 ((LAMBDA (VALUE)
                     (EVALUATE!UNINTERRUPTIBLY
                        (BLOCK (STOP!PROCESS P2) (C VALUE))))
                  (F1)))
              (LAMBDA ()
                 ((LAMBDA (VALUE)
                     (EVALUATE!UNINTERRUPTIBLY
                        (BLOCK (STOP!PROCESS P1) (C VALUE))))
                  (F2)))))
          NIL NIL))))
```

`TRY!TWO!THINGS!IN!PARALLEL` takes two functions of no arguments (in order to pass an unevaluated expression and its environment in for later use, so as to avoid variable conflicts). It creates two processes to run them, and returns the value of whichever completes first.

As an example of how to misuse `TRY!TWO!THINGS!IN!PARALLEL`, here is a function which determines the sign of an integer using only `ADD1`, `SUB1`, and `EQUAL`.

```
(DEFINE SIGN
   (LAMBDA (N)
      (IF (EQUAL N 0) 'ZERO
          (TRY!TWO!THINGS!IN!PARALLEL
             (LAMBDA ()
                (DO ((I 0 (ADD1 I)))
                    ((EQUAL I N) 'POSITIVE)))
             (LAMBDA ()
                (DO ((I 0 (SUB1 I)))
                    ((EQUAL I N) 'NEGATIVE)))))))
```

3. Substitution Semantics and Programming Styles

In the previous section we showed several different SCHEME programs for computing the factorial function. How are they different? We intuitively distinguish recursive from iterative programs, for example, by noting that recursive programs "call themselves" but in the last section we claimed to do iteration with a seemingly recursive program. Experienced programmers "know" that recursion uses up "stack" so a program implemented recursively will run out of stack on a sufficiently large problem. Can we make these ideas more precise? One traditional approach is to model the computation with lambda calculus.

3.1. Reviewing the Lambda Calculus

Traditionally language constructs are broken up into two distinct classes: imperative constructs and those with side-effects—such as assignment and go-to; and applicative constructs—those executed for value—such as arithmetic expressions. In addition, compiled languages often require a third class, declarative constructs, but these are provided primarily to guide the compilation process and do not directly affect the semantics of execution, and so will not concern us here.

Lambda calculus is a model for the applicative component of programming languages. It models all non-imperative constructs as applications of functions and specifies the semantics of such expressions by a set of axioms or rewrite rules. One axiom states that a combination, i.e., an expression formed by a function applied to some arguments, is equivalent to the body of that function with the appropriate arguments substituted for the free occurrences of the formal parameters of the function in its body:

```
((LAMBDA <vars> <body) <args>) = Subst[<args> <vars> <body>]
```

Another axiom requires that the meaning of an expression be independent of the names of the formal parameters bound in the expression:

```
(LAMBDA <vars> <body>) = (LAMBDA <newvars> Subst[<newvars> <vars> <body>])
```

provided that none of `<newvars>` appears free in `<body>`.

These constraints force `Subst` to be defined in such a way that an important kind of *referential transparency* is obtained. Besides these "structural" axioms, others are provided which specify the result of certain primitive functions applied to specific arguments. We

shall not be concerned with these problems here—we will assume a small reasonable set of primitive functions.

3.2. Recursive programs

Now, let's see how lambda calculus may be used (informally) to model a computation. Consider the standard definition of the factorial function:

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

We are being *very* informal—lambda calculus as presented by Church [2] does not include such constucts as `DEFINE`, `IF`, or `=`, `*`, or even `1`! The "usual" lambda calculus construct for defining recursive functions is a rather obscure object called the "fixed-point" operator. We have been lax to avoid the hassle of "rigor mortis" in this tutorial paper. Similarly, `IF` is the SCHEME conditional construct we will use for convenience; it reduces to its second or third argument depending on whether the first reduces to `TRUE` or `FALSE`. The objects `*`, `=`, `0`, `1`, etc., may be thought of as abbreviations for complex lambda expressions (such as Church numerals) whose details we are not interested in. On the other hand, we may think of them as primitive expressions, defined by additional axioms; this viewpoint leads to practical interpreter implementations.

Now let's reduce the expression `(FACT 3)`. We will perform the expression reductions, except for the `IF` primitive, in Applicative Order (call by value), though this is not necessary, as we will discuss later. We display a "trace" of the substitutions:

```
=>  (FACT 3)
=>  (IF (= 3 0) 1 (* 3 (FACT (- 3 1))))
=>  (* 3 (FACT (- 3 1)))
=>  (* 3 (FACT 2))
=>  (* 3 (IF (= 2 0) 1 (* 2 (FACT (- 2 1)))))
=>  (* 3 (* 2 (FACT (- 2 1))))
=>  (* 3 (* 2 (FACT 1)))
=>  (* 3 (* 2 (IF (= 1 0) 1 (* 1 (FACT (- 1 1))))))
=>  (* 3 (* 2 (* 1 (FACT (- 1 1)))))
=>  (* 3 (* 2 (* 1 (FACT 0))))
=>  (* 3 (* 2 (* 1 (IF (= 0 0) 1 (* 0 (FACT (- 0 1)))))))
=>  (* 3 (* 2 (* 1 1)))
=>  (* 3 (* 2 1))
=>  (* 3 2)
=>  6
```

You will note that we have calculated `(fact 3)` by a process wherein *each expression is replaced* by an expression which is provably equivalent to it via an axiom or which is produced by application of a primitive function.

3.3. Now, What About Iteration?

Consider the "iterative" definition of FACT. Although it appears to be recursive, since it "calls itself", we will see that it captures the essence of our intuitive notion of iteration.

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

Let us now compute (fact 3).

```
=>  (FACT 3)
=>  (FACT1 3 1)
=>  (IF (= 3 0) 1
        (FACT1 (- 3 1) (* 3 1)))
=>  (FACT1 (- 3 1) (* 3 1))
=>  (FACT1 2 (* 3 1))
=>  (FACT1 2 3)
=>  (IF (= 2 0) 3
        (FACT1 (- 2 1) (* 2 3)))
=>  (FACT1 (- 2 1) (* 2 3))
=>  (FACT1 1 (* 2 3))
=>  (FACT1 1 6)
=>  (IF (= 1 0) 6
        (FACT1 (- 1 1) (* 1 6)))
=>  (FACT1 (- 1 1) (* 1 6))
=>  (FACT1 0 (* 1 6))
=>  (FACT1 0 6)
=>  (IF (= 0 0) 6
        (FACT1 (- 0 1) (* 0 6)))
=>  6
```

Notice that the expressions involved have a fixed maximum size independent of the argument to FACT! In fact, as Marvin Minsky pointed out, successive reductions produce a cycle of expressions which are identical except for the numerical quantities involved. Looking back, we may note by way of comparison that the recursive version caused creation of expressions proportional in size to the argument. This is why we think that this version of FACT is iterative rather than recursive. At each stage of the iterative version the "state" of the computation is summarized in two variables, the counter and the answer accumulator, while at each stage of the recursive version the "state" contains a chain of pieces each of which contains a component of the state. In the recursive version of FACT, for example, the state contains the sequence of multiplications to be performed upon return from the bottom. It is true that the iterative factorial also can produce expressions of arbitrary size, since the number of bits needed to express factorial of n grows with n; but this is a property of the numbers calculated by the function which is implemented in iterative style, and

not of the iterative control structure itself. A recursive control structure *inherently* creates expressions of unbounded size as a function of the recursion depth, while an iterative control structure produces a cycle of equivalent expressions, and so the expressions are of approximately the same size no matter how many iteration steps are taken. This is the essence of the difference between the notions of iteration and recursion. Hewitt [15, p. 234] made a similar observation in passing, expressing the difference in terms of storage used in program execution rather than in terms of intermediate expressions produced by substitution semantics.

3.4. Continuation Passing Recursion

Remember the other way to compute factorials?

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

This looks iterative on the surface! but in fact it is recursive. Let's compute `(FACT 3 ANSWER)`, where `ANSWER` is a continuation which is to receive the result of `FACT` applied to `3`; that is, the last thing `FACT` should do is apply the continuation `ANSWER` to its result.

```
=> (FACT 3 ANSWER)
=> (IF (= 3 0) (ANSWER 1)
       (FACT (- 3 1) (LAMBDA (A) (ANSWER (* 3 A)))))
=> (FACT (- 3 1) (LAMBDA (A) (ANSWER (* 3 A))))
=> (FACT 2 (LAMBDA (A) (ANSWER (* 3 A))))
=> (IF (= 2 0) ((LAMBDA (A) (ANSWER (* 3 A))) 1)
       (FACT (- 2 1)
             (LAMBDA (A)
                ((LAMBDA (A) (ANSWER (* 3 A)))
                 (* 2 A)))))
=> (FACT (- 2 1)
         (LAMBDA (A)
            ((LAMBDA (A) (ANSWER (* 3 A)))
             (* 2 A))))
=> (FACT 1
         (LAMBDA (A)
            ((LAMBDA (A) (ANSWER (* 3 A)))
             (* 2 A))))
```

```
=> (IF (= 1 0)
       ((LAMBDA (A)
           ((LAMBDA (A) (ANSWER (* 3 A)))
            (* 2 A)))
        1)
       (FACT (- 1 1)
             (LAMBDA (A)
                ((LAMBDA (A)
                    ((LAMBDA (A)
                        (ANSWER (* 3 A)))
                     (* 2 A)))
                 (* 1 A)))))
=> (FACT (- 1 1)
         (LAMBDA (A)
            ((LAMBDA (A)
                ((LAMBDA (A)
                    (ANSWER (* 3 A)))
                 (* 2 A)))
             (* 1 A))))
=> (FACT 0
         (LAMBDA (A)
                 ((LAMBDA (A)
                          ((LAMBDA (A)
                                   (ANSWER (* 3 A)))
                           (* 2 A)))
                  (* 1 A))))
=> (IF (= 0 0)
       ((LAMBDA (A)
           ((LAMBDA (A)
               ((LAMBDA (A)
                   (ANSWER (* 3 A)))
                (* 2 A)))
            (* 1 A)))
        1)
        (FACT (- 0 1)
              ((LAMBDA (A)
                  ((LAMBDA (A)
                      ((LAMBDA (A)
                          ((LAMBDA (A)
                              (ANSWER (* 3 A)))
                           (* 2 A)))
                       (* 1 A)))
                   (* 0 A))))))
```

```
=> ((LAMBDA (A)
       ((LAMBDA (A)
           ((LAMBDA (A)
               (ANSWER (* 3 A)))
            (* 2 A)))
        (* 1 A)))
    1)
=> ((LAMBDA (A)
       ((LAMBDA (A)
           (ANSWER (* 3 A)))
        (* 2 A)))
    (* 1 1))
=> ((LAMBDA (A)
       ((LAMBDA (A)
           (ANSWER (* 3 A)))
        (* 2 A)))
    1)
=> ((LAMBDA (A)
       (ANSWER (* 3 A)))
    (* 2 1))
=> ((LAMBDA (A)
       (ANSWER (* 3 A)))
    2)
=> (ANSWER (* 3 2))
=> (ANSWER 6)     Whew!
```

Note that we have computed factorial of 3 (and are about to give this result to the continuation), but in the process no combination with FACT in the first position has ever been reduced except as the outermost expression. If we think of the computation in terms of evaluation rather than substitution, this means that *we never returned a value from any application of the function* FACT! It is always possible, if we are willing to specify explicitly what to do with the answer, to perform any calculation in this way: rather than reducing to its value, it reduces to an application of a continuation to its value (cf. [4]). That is, in this continuation-passing programming style,[6] *a function always "returns" its result by "sending" it to another function*. This is the key idea.

We also note that by our previous observation, this program is essentially recursive in that the expressions produced as intermediate results of the substitution semantics grow to a size proportional to the depth. In fact, the same information is being stored in the nested continuations produced by this program as in the nested products produced by the traditional recursion—what to do with the result.

One might object that this FACT is not the same kind of object as the previous definition, since we can't use it as a function in the same manner. Note, however, that if we supply the continuation (LAMBDA (X) X), the resulting combination (FACT 3 (LAMBDA (X) X)) will reduce to 6, just as with traditional recursion.

One might also object that we are using function values—the primitives =, -, and * are functions which return values, for example. But this is just a property of the primitives; consider a new set of primitives ==, --, and ** which accept continuations (indeed, let ==

take two continuations: if the predicate is TRUE call the first, otherwise call the second). We can then define FACT as follows:

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

We can see here that no functional application returns a value in a computation of factorial in this situation. We believe that functional usage, where applicable (pun intended), is more perspicuous than continuation-passing. We shall therefore use functional primitives such as * rather than ** wherever possible, keeping in mind that we could use ** instead if we wished.

4. Some Implementation Issues

The key problem is *efficiency*. Although it is easy to build an inefficient interpreter which straightforwardly performs expression substitutions, such an interpreter performs much unnecessary copying of intermediate expressions. The standard solution to this problem is to use an auxiliary structure, called the *environment*, which represents a set of *virtual substitutions*. Thus, when evaluating an expression of the form

```
((LAMBDA <vars> <body>) <args>)  in environment E
```

instead of reducing it by performing

```
Subst[<args> <vars> <body>]
```

we reduce it to

```
<body>  in environment E′=Pairlis[<vars> <args>* E]
```

where Pairlis creates a new environment E′ in which the <vars> are logically paired with (i.e., "bound to") the corresponding <args>* (the precise meaning of <args>* will be explained presently), and in which any variables not in <vars> are bound as they were in E.

When using environments, it is necessary to keep them straight. For example. the following expression should manage to evaluate to 7:

```
(((LAMBDA (X) (LAMBDA (Y) (+ X Y))) 3) 4)
```

A substitution interpreter would cause the free occurrence of X in the inner lambda expression to be replaced by 3 before applying that lambda expression to 4. An interpreter which uses environments must arrange for the expression (+ X Y) to be evaluated in an environment such that X is bound to 3 and Y is bound to 4. This implies that when the inner

lambda expression is applied to 4, there must be associated with it an environment in which X is bound to 3. In order to solve this problem we introduce the notion of a *closure* [11, 14] which is a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments. We will notate a closure using the *beta* construct (our own notation, but isomorphic to the LISP *funarg* construct) as follows:

```
(BETA (LAMBDA <vars> <body>) <environment>)
```

When a lambda expression is to be evaluated, because it was passed as an argument, it evaluates to a closure of that lambda expression in the environment it is evaluated in (i.e., the environment it was passed from):

```
(LAMBDA <vars> <body>)  in environment E
```

evaluates to

```
(BETA (LAMBDA <vars> <body>) E)  in environment E
```

When a closure is to be applied to some arguments:

```
((BETA (LAMBDA <vars> <body>) E1) <args>)  in environment E2
```

this is performed by reducing the application expression to

```
<body>  in environment Pairlis[<vars> <args in E2> E1]
```

That is, any free variables in the closed lambda expression refer to the environment as of the time of closure (E1), not as of the time of application (E2), whereas the arguments are evaluated in the application environment as expected.

This notion of *closure* has gone by many other names in other contexts. In LISP, for example, such a closure has been traditionally known as a *funarg*. ALGOL has several related ideas. Every ALGOL procedure is, at the time of its invocation, essentially a "downward funarg". In addition, expressions which are passed by name instead of by value are closed through the use of mechanisms called *thunks* [8]. It turns out that an *actor* (other than a cell or a serializer) is also a closure. Hewitt [17] describes an actor as consisting of a *script*, which is code to be executed, and a *set of acquaintances*, which are other actors which it knows about. We contend that the script is in fact identical to the lambda expression in a closure, and that the set of acquaintances is in effect an environment. As an example, consider the following code for *cons* taken from [17] (cf. [4]):

```
[CONS ≡
  (≡> [=A =B]
    (CASES
      (≡> FIRST?
       A)
      (≡> REST?
       B)
      (≡> LIST?
       YES)))]
```

When the form (CONS X Y) is evaluated, the result is an (evaluated) CASES statement, which is a receiver ready to accept a message such as "FIRST?" or "REST?". This resulting receiver evidently *knows about* the actors X and Y as being bound to the variables A and B; it is evidently a *closure* of the cases script plus a set of acquaintances which includes X and Y (as well as "FIRST?" and "REST?" and: "YES", for example; PLASMA considers such "constant acquaintances" to be part of the set, whereas in SCHEME we might prefer to consider them part of the script). Once we realize that it is a closure and nothing more, we can see easily how to express the same semantics in SCHEME:

```
(DEFINE CONS
   (LAMBDA (A B)
      (LAMBDA (M)
         (IF (EQ M 'FIRST?) A
             (IF (EQ M 'REST?) B
                 (IF (EQ M 'LIST?) 'YES
                     (ERROR '|UNRECOGNIZED MESSAGE - CONS|
                            M
                            'WRNG-TYPE-ARG)))))))
```

Note that we have used explicit *eq* tests on the incoming message rather than the implicit pattern-matching of the *cases* statement, but the underlying semantics of the approach are the same.

There are several important consequences of closing every lambda expression in the environment from which it is passed (i.e., in its "lexical" or "static" environment). First, the axioms of lambda calculus are automatically preserved. Thus, referential transparency is enforced. This in turn implies that there are no "fluid" variable bindings (as there are in standard stack implementations of LISP such as MacLISP). Second, the upward funarg problem [14] requires that the environment structure be potentially tree-like. Finally, the environment at any point in a computation can never be deeper than the lexical depth of the expression being evaluated at that time; i.e., the environment contains bindings only for variables bound in lambdas lexically surrounding the expression being evaluated. This is true *even if recursive functions are involved*. It follows that if list structures are used to implement environments, the time to look up a variable is proportional only to the lexical distance from the reference to the binding and not to the depth of recursion or any other dynamic parameter. Therefore it is not necessarily as expensive as many people have been led to believe. Furthermore, it is not even necessary to scan the environment for the variable, since its value must be in a known position relative to the top of the environment structure; this position can be computed by a compiler at compile time on the basis of lexical scope. The tree-like structure of an environment prevents one from merely indexing into it, however; it is necessary to *cdr* down it. (Some ALGOL compilers use a similar technique involving base registers pointing to sets of variables for each level of block nesting; it is necessary to determine the base pointer for the block desired for a variable reference, but then the variable is at a known offset from the base address.) It also follows that an iterative programming style *will* lead to no net accumulation of environment structures in the interpreter. The recursive style, with or without continuation-passing, *will* lead to accumulation of environment structures as a function of the recursion depth, not because any environment becomes arbitrarily deep, but rather because at each level of recursion it is

necessary to save the environment at that point. It is saved by the interpreter in the case of traditional recursion, so that computation can continue in the correct environment on return from the recursive call; it is saved as part of the continuation closure in continuation-passing.

Another problem is concerned with control. This is a consequence of the *functional interpretation* of the lambda calculus, i.e., the view that an "expression" (combination) represents a value to be "returned" (to replace the combination) to its "caller" (the process evaluating the combination containing the original one). The interpreter must provide for correctly resuming the caller when the caller has returned its value. The state of the computation at the time of the call must therefore be preserved. We see, then, that part of the state of the computation must be (a pointer to) the preserved state of its caller; we will call this component of the state the *clink* [12, 1]. Just before the evaluation of a subexpression, the state of the current computation, including the *clink*, must be gathered together into a single data structure, which we will call a *frame*; the *clink* is then altered to point to this new frame. The evaluation of the subexpression then returns by restoring the state of the process from the current *clink*. Note that the value of the subexpression had better not be part of the state, for otherwise it would be lost by the state restoration. Thus, we only build a new frame if further computation would result in losing information which might be necessary. This only occurs if we must somehow return to that state. This in turn can only occur if we must evaluate an expression whose value must be obtained in order to continue computation in the current state.

This implies that no frame need be created in order to *apply* a lambda expression to its arguments. This in turn implies that the iterative and continuation-passing styles lead to *no net creation of frames*, because they are implemented *only* in terms of explicit lambda applications, whereas the recursive style leads to the creation of one net frame per level of recursive depth, because the recursive invocation involves the evaluation of a expression containing the recursive lambda application as a subexpression.

A *clink* in a lambda calculus-based interpreter is in fact equivalent to a low-level default continuation as created by the PLASMA interpreter. Such a continuation is a (closed) lambda expression of one argument whose script will carry on the computation after receiving the value of the subexpression. The *clink* mechanism is therefore not necessary, if we are willing to transform all our programs into pure continuation-passing style. We could do this explicitly, by requiring the user to write his programs in this form; or implicitly, as PLASMA does, by creating these one-argument continuations as necessary, passing them as hidden extra arguments to lambda expressions which behave like functions. On the other hand, we may think of a *clink* as a highly optimized continuation, whose "script" is that carefully coded portion of the lambda calculus interpreter which restores the frame and then carries on. We find this notion useful in defining a primitive, `CATCH` (named for the `CATCH` construct in MacLISP [13]), for "hairy control structure", similar to Reynolds' `ESCAPE` operator [16], which makes these low-level continuations available to the user. Note that PLASMA has a similar facility for getting hold of the low-level continuations, namely the "≡≡>" receiver construct.

Another problem for the implementor of an interpreter of a lambda calculus based language is the order in which to perform reductions. There are two standard orders of evaluation (and several other semi-standard ones, which we will not consider here). The first is *Normal Order*, which corresponds roughly to ALGOL's "call by name", and the

second is *Applicative Order*, which corresponds roughly to ALGOL's "call by value" or to LISP functional application.

Under a call-by-name implementation. the `<args>*` mentioned above are in fact the actual argument expressions, each paired with the environment E. The evaluator has two additional rules:

1. when a variable x is to be evaluated in environment E1, then its associated expression-environment pair [A, E2] (which is equivalent to an ALGOL thunk) is looked up in E1, and then A is evaluated in E2.

2. when a "primitive operator" is to be applied, its arguments must be evaluated at that time, and then the operator applied in a call-by-value manner.

Under a call-by-value implementation, the `<args>*` are the *values* of the argument expressions; i.e., the argument expressions are evaluated in environment E, and only then is the lambda expression applied. Note that this leads to trouble in defining conditionals. Under call-by-name one may define predicates to return `(LAMBDA (X Y) X)` for `TRUE` and `(LAMBDA (X Y) Y)` for `FALSE`. and then one may simply write

```
((= A B) <do this if TRUE> <do this if FALSE>)
```

This trick depends implicitly on the order of evaluation. It will not work under call-by-value, nor in general under any other reductive order except Normal Order. It is therefore necessary to introduce a special primitive operator (such as "if") which is applied in a *call-by-name* manner. This leads us to the interesting conclusion that a practical lambda calculus interpreter cannot be *purely* call-by-name *or* call-by-value; it is necessary to have at least a little of each.

There, is a fundamental problem, however, with using Normal Order evaluation in a lambda calculus interpreter, which is brought out by the iterative programming style. We already know that no net frames are created by iterative programs, and that no net environment structures are created either. The problem is that under a *call-by-name* implementation there may be a net *thunk* structure created proportional in size to the number of iteration steps. This problem is inherent in Normal Order, because Normal Order substitution semantics exhibit the same phenomenon of increasing expression size. Therefore iteration cannot be effectively modeled in a call-by-name interpreter. An alternative view is that a call-by-name interpreter remembers more than is logically necessary to perform the computations indicated by the original expressions. This is indicated by the fact that the Applicative Order substitution semantics lead to expressions of fixed maximum size independent of the number of iteration steps.

It turns out that this conflict between call-by-name and iteration is resolved by the use of continuation-passing. If we use a pure continuation-passing programming style, then Normal Order and Applicative Order are the same order! In pure continuation-passing no combination is ever a subcombination of another combination. (This is the justification for the fact mentioned above that no clinks are needed if pure continuation-passing style is used.) Thus, if we wish to model iteration in pure lambda calculus without even an *if* primitive, we can use Normal Order substitutions and express the iteration in the continuation-passing style.

Under *any* reductive order, whether Normal Order, Applicative Order, or any other order, it is in practice convenient to introduce a means of terminating the evaluation process on a given form; in order to do this we introduce three different and equally useful notions. The first is the *primitive operator* such as +; the evaluator can apply such an operator directly, without substituting a lambda expression for the operator and reducing the resulting form. The second is the *self-evaluating constant*; this is used for primitive objects such as numbers, which effectively behave as if always "bound to themselves" in any environment. The third is the *quoting function*, which protects its argument from reductions so that it is returned as is; this is used for treating forms as data in the usual LISP manner.

These three ideas are not logically necessary, since the evaluation process will (eventually) terminate when no reductions can be made, but they are a great convenience for introducing various functions and data into the lambda calculus. Note too that some are easily defined in terms of the others; for example, instead of letting 3 be a self-evaluating constant, we could let 3 be a primitive operator of no arguments which returned 3, or we could merely quote it; similarly, instead of quoting forms we could let forms be a self-evaluating data type, as in MDL [5] (better known as MUDDLE), written with different parentheses. Because, as we have said, these constructs are all strictly for convenience, we will not strive for any kind of minimality, but will continue to use all three notions in our interpreter, as we already have in our examples. We provide an interface so that all MacLISP `SUBRS` may be used as primitive operators; we define numbers to be self-evaluating; and we will use `QUOTE` to quote forms as in LISP (and thus we may use the "`'`" character as an abbreviation)

One final issue which the implementor of a lambda calculus based interpreter should consider is that of extensions to the language, such as primitives for side effects, multiprocessing, and synchronization of processes. Note that these are ideas which are very hard, if not impossible, to model using the substitution semantics of the lambda calculus, but which are easily incorporated in other semantic models, including the environment interpreter and, perhaps more notably, the ACTORS model [6, 7]. The fundamental problem with modelling such concepts using substitution semantics is that substitution produces *copies* of expressions, and so cannot model the notion of *sharing* very well. In an interpreter which uses environments, all instances of a variable scoped in a given environment refer to the same virtual substitution contained in that environment, and so may be thought of as sharing a *value cell* in that environment. We can take advantage of this sharing by introducing a primitive operator which modifies the contents of a value cell; since all occurrences refer to the same value cell, changing the contents of that value cell will change the result of future references to that value cell (i.e., occurrences of the variable which invoke the virtual substitution mechanism). Such a primitive operator would then be similar to the `SET` function of LISP, or the `:=` of ALGOL. We include such an operator, `ASET`, in our interpreter.

Introducing multiprocessing into the interpreter is fairly straightforward; all that is necessary is to introduce a mechanism for time-slicing the interpreter among several processes. One can even model this in substitution semantics by supposing that there can be more than one expression and at each step an expression is randomly chosen to perform a reduction within. (On the other hand, *synchronizing* of the processes is very hard to model using substitution semantics!)

Since our value cells effectively solve the readers and writers problem (i.e., reads and writes of variables are indivisible) no more than our side effect primitive is necessary to

synchronize our processes [3, 9, 10]. However, the techniques for achieving synchronization using only `:=` are quite cumbersome and opaque. It behooves the implementor to make things easier for the user by introducing a more tractable synchronization primitive (e.g., P+V or monitors or path expressions or ...). Machine language programmers have long known that the easiest way to synchronize processes is to turn off the scheduling clock during the execution of critical code. We have introduced such a primitive, `EVALUATE!UNINTERRUPTIBLY` (which is a sort of "over-anxious serializer", because it locks out the whole world), into our interpreter.

5. The Implementation of the Interpreter

Here we present a real live SCHEME interpreter. This particular version was written primarily for expository purposes; it works, but not as efficiently as possible. The "production version" of SCHEME is coded somewhat more intricately, and runs about twice as fast as the interpreter presented below.

The basic idea behind the implementation is *think machine language*. In particular, we must not use recursion in the implementation language to implement recursion in the language being interpreted. This is a crucial mistake which has screwed many language implementations (e.g., Micro-PLANNER [18]). The reason for this is that if the implementation language does not support certain kinds of control structures, then we will not be able to effectively interpret them. Thus, for example, if the control frame structure in the implementation language is constrained to be stack-like, then modelling more general control structures in the interpreted language will be very difficult unless we divorce ourselves from the constrained structures at the outset.

It will be convenient to think of an implementation machine which has certain operations, which are "micro-coded" in LISP; these are used to operate on various "registers", which are represented as free LISP variables. These registers are:

`**EXP**`
: The expression currently being evaluated.

`**ENV**`
: A pointer to the environment in which to evaluate `**EXP**`.[7]

`**CLINK**`
: A pointer to the frame for the computation of which the current one is a subcomputation.

`**PC**`
: The "program counter". As each "instruction" is executed, it updates `**PC**` to point to the next instruction to be executed.

`**VAL**`
: The returned value of a subcomputation. This register is *not* saved and restored in `**CLINK**` frames; in fact, its sole purpose is to pass values back safely across the restoration of a frame.

`**UNEVLIS**,**EVLIS**`
: These are utility registers which are part of the state of the interpreter (they are saved in `**CLINK**` frames). They are used primarily for evaluation of components of combinations, but may be used for other purposes also.

`**TEM**`
: A super-temporary register, used for random purposes and not saved across interrupts or in `**CLINK**` frames. It therefore may not be used to pass information between "instructions" of the "machine", and so is best thought of as an internal hardware register.

`**QUEUE**`
: A list of all processes other than the one currently being interpreted.

`**TICK**`
: A magic register which a "hardware clock" sets to `T` every so often, used to drive the scheduler.

`**PROCESS**`
: This register always contains the name of the process currently swapped in and running.

The following declarations and macros are present only to make the MacLISP compiler happy. and to make the version number of the SCHEME implementation available in the global variable `VERSION`.

```
(DECLARE (SPECIAL **EXP** **UNEVLIS** **ENV** **EVLIS** **PC** **CLINK**
                  **VAL** **TEM** **TOP** **QUEUE** **TICK** **PROCESS**
                  **QUANTUM**
                  VERSION LISPVERSION))

(DEFUN VERSION MACRO (X)
       (COND (COMPILER-STATE (LIST 'QUOTE (STATUS UREAD)))
             (T (RPLACA X 'QUOTE)
                (RPLACD X (LIST VERSION))
                (LIST 'QUOTE VERSION))))

(DECLARE (READ))

(SETQ VERSION ((LAMBDA (COMPILER-STATE) (VERSION)) T))
```

The function `SCHEME` initializes the system driver. The two `SETQ`'s merely set up version numbers. The top level loop itself is written in SCHEME, and is a `LABELS` which binds the function `**TOP**` to be a read-eval-print loop. The LISP global variable `**TOP**` is initialized to the closure of the `**TOP**` function for convenience and accessibility to user-defined functions.

```
(DEFUN SCHEME ()
       (SETQ VERSION (VERSION)  LISPVERSION (STATUS LISPVERSION))
       (TERPRI)
       (PRINC '|This is SCHEME |)
       (PRINC VERSION)
       (PRINC '| running in LISP |)
```

```
       (PRINC LISPVERSION)
       (SETQ **ENV** NIL  **QUEUE** NIL
             **PROCESS** (CREATE!PROCESS '(**TOP** '|SCHEME - Toplevel|)))
       (SWAPINPROCESS)
       (ALARMCLOCK 'RUNTIME **QUANTUM**)
       (MLOOP))

(SETQ **TOP**
      '(BETA (LAMBDA (**MESSAGE**)
               (LABELS ((**TOP1**
                         (LAMBDA (**IGNORE1** **IGNORE2** **IGNORE3**)
                            (**TOP1** (TERPRI) (PRINC '|==> |)
                                      (PRINT (SET '* (EVALUATE (READ))))))))
                  (**TOP1** (TERPRI) (PRINC **MESSAGE**) NIL))
             NIL))
```

When the LISP alarmclock tick occurs, the global register `**TICK**` is set to `T`. `**QUANTUM**`, the amount or runtime between ticks, is measured in micro-seconds.

```
(DEFUN SETTICK (X) (SETQ **TICK** T))

(SETQ **QUANTUM** 1000000.  ALARMCLOCK 'SETTICK)
```

`MLOOP` is the main loop of the interpreter. It may be thought of as the instruction dispatch in the micro-code of the implementation machine. If an alarmclock tick has occurred, and interrupts are allowed, then the scheduler is called to switch processes. Otherwise the "instruction" specified by `**PC**` is executed via `FASTCALL`.

```
(DEFUN MLOOP ()
       (DO ((**TICK** NIL)) (NIL)   ;DO forever
           (AND **TICK** (ALLOW) (SCHEDULE))
           (FASTCALL **PC**)))
```

`FASTCALL` is essentially a `FUNCALL` optimized for compiled "microcode". Note the way it pulls the SUBR property to the front of the property list if possible for speed.

```
(DEFUN FASTCALL (ATSYM)
       (COND ((EQ (CAR (CDR ATSYM)) 'SUBR)
              (SUBRCALL NIL (CADR (CDR ATSYM))))
             (T ((LAMBDA (SUBR)
                         (COND (SUBR (REMPROP ATSYM 'SUBR)
                                     (PUTPROP ATSYM SUBR 'SUBR)
                                     (SUBRCALL NIL SUBR))
                               (T (FUNCALL ATSYM))))
                 (GET ATSYM 'SUBR)))))
```

Interrupts are allowed unless the variable `*ALLOW*` is bound to `NIL` in the current environment. This is used to implement the `EVALUATE!UNINTERRUPTIBLY` primitive.

```
(DEFUN ALLOW ()
       ((LAMBDA (VCELL)
                (COND (VCELL (CADR VCELL))
                      (T T)))
        (ASSQ '*ALLOW* **ENV**)))
```

Next comes the scheduler. It is apparently interrupt-driven, but in fact is not. The key here is to *think microcode*! There is one place in the microcoded instruction interpretation loop which checks to see if there is an interrupt pending; in our "machine", this occurs in MLOOP, where **TICK** is checked on every cycle. This is another case where we must beware of using too much of the power of the host language; just as we must avoid using host recursion directly to implement recursion, so we must avoid using host interrupts directly to implement interrupts. We may not modify any register during a host language interrupt, except one (such as **TICK**) which is specifically intended to signal interrupts. Thus, if we were to add an interrupt character facility to SCHEME similar to that in MacLISP [13], the MacLISP interrupt character function would merely set a register like **TICK** and dismiss; MLOOP would eventually notice that this register had changed and dispatch to the interrupt handler. All this implies that the "microcode" for the interrupt handlers does not itself contain critical code that must be protected from host language interrupts.

When the scheduler is invoked, if there is another process waiting on the process queue, then the current process is swapped out and put on the end of the queue, and a new process swapped in from the front of the queue. The process stored on the queue consists of an atom which has the current frame and **VAL** register on its property list. Note that the **TEM** register is *not* saved, and so cannot be used to pass information between instructions.

```
(DEFUN SCHEDULE ()
       (COND (**QUEUE**
              (SWAPOUTPROCESS)
              (NCONC **QUEUE** (LIST **PROCESS**))
              (SETQ **PROCESS** (CAR **QUEUE**)
                    **QUEUE** (CDR **QUEUE**))
              (SWAPINPROCESS)))
       (SETQ **TICK** NIL)
       (ALARMCLOCK 'RUNTIME **QUANTUM**))

(DEFUN SWAPOUTPROCESS ()
       ((LAMBDA (**CLINK**)
                (PUTPROP **PROCESS** (SAVEUP **PC**) 'CLINK)
                (PUTPROP **PROCESS** **VAL** 'VAL))
       **CLINK**))

(DEFUN SWAPINPROCESS ()
       (SETQ **CLINK** (GET **PROCESS** 'CLINK)
             **VAL** (GET **PROCESS** 'VAL))
       (RESTORE))
```

Primitive operators are LISP functions, i.e., SUBRs, EXPRs, and LSUBRs.

```
(DEFUN PRIMOP (X) (GETL X '(SUBR EXPR LSUBR)))
```

SAVEUP conses a new frame onto the **CLINK** structure. It saves the values of all important registers. It takes one argument, RETAG, which is the instruction to return to when the computation is restored.

```
(DEFUN SAVEUP (RETAG)
  (SETQ **CLINK**
        (LIST **EXP** **UNEVLIS** **ENV** **EVLIS** RETAG **CLINK**)))
```

RESTORE restores a computation from the CLINK. The use of TEMP is a kludge to optimize the compilation of the "microcode".

```
(DEFUN RESTORE ()
  (PROG (TEMP)
        (SETQ TEMP (OR **CLINK**
                       (ERROR '|PROCESS RAN OUT - RESTORE|
                              **EXP**
                              'FAIL-ACT))
              **EXP** (CAR TEMP)
               TEMP (CDR TEMP)
              **UNEVLIS** (CAR TEMP)
               TEMP (CDR TEMP)
              **ENV** (CAR TEMP)
               TEMP (CDR TEMP)
              **EVLIS** (CAR TEMP)
               TEMP (CDR TEMP)
              **PC** (CAR TEMP)
               TEMP (CDR TEMP)
              **CLINK** (CAR TEMP))))
```

AEVAL is the central function of the SCHEME interpreter. This "instruction" expects **EXP** to contain an expression to evaluate, and **ENV** to contain the environment for the evaluation. The fact that we have arrived here indicates that **PC** contains 'AEVAL, and so we need not change **PC** if the next instruction is also to be AEVAL. Besides the obvious objects likes numbers, identifiers, LAMBDA expressions, and BETA expressions (closures), there are also several other objects of interest. There are primitive operators (LISP functions); AINTs (which are to SCHEME as FSUBRs like COND are to LISP); and AMACROs, which are used to implement DO, AND, OR, COND, BLOCK, etc.[8]

```
(DEFUN AEVAL ()
       (COND ((ATOM **EXP**)
              (COND ((NUMBERP **EXP**)
                     (SETQ **VAL** **EXP**)
                     (RESTORE))
                    ((PRIMOP **EXP**)
                     (SETQ **VAL** **EXP**)
                     (RESTORE))
                    ((SETQ **TEM** (ASSQ **EXP** **ENV**))
                     (SETQ **VAL** (CADR **TEM**))
                     (RESTORE))
                    (T (SETQ **VAL** (SYMEVAL **EXP**))
                       (RESTORE))))
             ((ATOM (CAR **EXP**))
              (COND ((SETQ **TEM** (GET (CAR **EXP**) 'AINT))
```

```
                (SETQ **PC** **TEM**))
               ((EQ (CAR **EXP**) 'LAMBDA)
                (SETQ **VAL** (LIST 'BETA **EXP** **ENV**))
                (RESTORE))
               ((SETQ **TEM** (GET (CAR **EXP**) 'AMACRO))
                (SETQ **EXP** (FUNCALL **TEM** **EXP**)))
               (T (SETQ **EVLIS** NIL
                        **UNEVLIS** **EXP**
                        **PC** 'EVLIS))))
        ((EQ (CAAR **EXP**) 'LAMBDA)
         (SETQ **EVLIS** (LIST (CAR **EXP**))
               **UNEVLIS** (CDR **EXP**)
               **PC** 'EVLIS))
        (T (SETQ **EVLIS** NIL
                 **UNEVLIS** **EXP**
                 **PC** 'EVLIS))))
```

We come to EVLIS when a combination is encountered. The intention is to evaluate each component of the combination and then apply the resulting function to the resulting arguments. We use the register **UNEVLIS** to hold the list of components yet to be evaluated, and the register **EVLIS** to hold the list of evaluated components. We assume that these have been set up by AEVAL. Note that in the case of an explicit LAMBDA expression in the CAR of a combination **UNEVLIS** is initialized to be the list of unevaluated arguments and **EVLIS** is initialized to be the list containing the lambda expression.

EVLIS checks to see if there remain any more components yet to be evaluated. If not, it applies the function. Note that the primitive operators are applied using the LISP function APPLY. Note also how a BETA expression controls the environment in which its body is to be evaluated. DELTA expressions are CATCH tags (see CATCH). It is interesting that the evaluated components are collected in the reverse order from that which we need them in, and so we must reverse the list before applying the function. Do you see why we must not use side effects (e.g., the NREVERSE function) to reverse the list? Think about CATCH!

If there remain components yet to be evaluated, EVLIS saves up a frame, so that execution can be resumed at EVLIS1 when the evaluation of the component returns with a value. It then sets up **EXP** to point to the component to be evaluated and dispatches to AEVAL.[9]

```
(DEFUN EVLIS ()
       (COND ((NULL **UNEVLIS**)
              (SETQ **EVLIS** (REVERSE **EVLIS**))
              (COND ((ATOM (CAR **EVLIS**))
                     (SETQ **VAL** (APPLY (CAR **EVLIS**) (CDR **EVLIS**)))
                     (RESTORE))
                    ((EQ (CAAR **EVLIS**) 'LAMBDA)
                     (SETQ **ENV** (PAIRLIS (CADAR **EVLIS**)
                                            (CDR **EVLIS**)
                                            **ENV**)
                           **EXP** (CADDAR **EVLIS**)
                           **PC** 'AEVAL))
                    ((EQ (CAAR **EVLIS**) 'BETA)
                     (SETQ **ENV** (PAIRLIS (CADR (CADAR **EVLIS**))
                                            (CDR **EVLIS**)
```

```
                                        (CADDAR **EVLIS**))
                              **EXP** (CADDR (CADAR **EVLLS**))
                              **PC** 'AEVAL))
                        ((EQ (CAAR **EVLIS**) 'DELTA)
                         (SETQ **CLINK** (CADAR **EVLIS**))
                         (RESTORE))
                        (T (ERROR '|BAD FUNCTION - EVARGLIST|
                                  **EXP**
                                  'FAIL-ACT))))
     (T (SAVEUP 'EVLIS1)
        (SETQ **EXP** (CAR **UNEVLIS**)
              **PC** 'AEVAL))))
```

The purpose of EVLIS1 is to gobble up the value, passed in the **VAL** register, of the subexpression just evaluated. It saves this value on the list in the **EVLIS** register, pops off the unevaluated subexpression from the **UNEVLIS** register, and dispatches back to EVLIS.

```
(DEFUN EVLIS1 ()
       (SETQ **EVLIS** (CONS **VAL** **EVLIS**)
             **UNEVLIS** (CDR **UNEVLIS**)
             **PC** 'EVLIS))
```

Here is the code for the various AINTs. On arrival at the instruction for an AINT, **EXP** contains the expression whose functional position contains the name of the AINT. None of the arguments have been evaluated, and no new control frame has been created. Note that each AINT is defined by the presence of an AINT property on the property list of the LISP atom which is its name. The value of this property is the LISP function which is the first "instruction" of the AINT.

EVALUATE is similar to the LISP function EVAL; it evaluates its argument, which should result in a s-expression, which is then fed back into the SCHEME expression evaluator (AEVAL).

```
(DEFPROP EVALUATE EVALUATE AINT)

(DEFUN EVALUATE ()
       (SAVEUP 'EVALUATE1)
       (SETQ **EXP** (CADR **EXP**)
             **PC** 'AEVAL))

(DEFUN EVALUATE1 ()
      (SETQ **EXP** **VAL**
            **PC** 'AEVAL))
```

IF evaluates its first argument, with a return address of IF1. IF1 examines the resulting **VAL**, and gives either the second or third argument to AEVAL depending on whether the **VAL** was non-NIL or NIL.

```
(DEFPROP IF IF AINT)
```

```
(DEFUN IF ()
       (SAVEUP 'IF1)
       (SETQ **EXP** (CADR **EXP**)
             **PC** 'AEVAL))

(DEFUN IF1 ()
       (COND (**VAL** (SETQ **EXP** (CADDR **EXP**)))
             (T (SETQ **EXP** (CADDDR **EXP**))))
       (SETQ **PC** 'AEVAL))
```

As it was in the beginning, is now, and ever shall be: QUOTE without end. (Amen, amen.)

```
(DEFPROP QUOTE AQUOTE AINT)

(DEFUN AQUOTE ()
       (SETQ **VAL** (CADR **EXP**))
       (RESTORE))
```

LABELS merely feeds its second argument to AEVAL after constructing a fiendishly clever environment structure. This is done in two stages: first the skeleton of the structure is created, with null environments in the closures of the bound functions; next the created environment is clobbered into each of the closures.

```
(DEFPROP LABELS LABELS AINT)

(DEFUN LABELS ()
       (SETQ **TEM** (MAPCAR '(LAMBDA (DEF)
                                  (LIST (CAR DEF)
                                        (LIST 'BETA (CADR DEF) NIL)))
                             (CADR **EXP**)))
       (MAPC '(LAMBDA (VC) (RPLACA (CDDADR VC) **TEM**)) **TEM**)
       (SETQ **ENV** (NCONC **TEM** **ENV**)
             **EXP** (CADDR **EXP**)
             **PC** 'AEVAL))
```

We now come to the multiprocess primitives.

CREATE!PROCESS temporarily creates a new set of machine registers (by the lambda-binding mechanism of the *host* language), establishes the new process in those registers, swaps it out, and returns the new process id; returning causes the old machine registers to be restored.

```
(DEFUN CREATE!PROCESS (EXP)
     ((LAMBDA (**PROCESS** **EXP** **ENV** **UNEVLIS** **EVLIS**
                           **PC** **CLINK** **VAL**)
              (SWAPOUTPROCESS)
              **PROCESS**)
      (GENSYM)
      EXP
      **ENV**
      NIL
      NIL
```

```
       'AEVAL
       (LIST NIL NIL NIL NIL 'TERMINATE NIL)
       NIL))

(DEFUN START!PROCESS (P)
       (COND ((OR (NOT (ATOM P)) (NOT (GET P 'CLlNK)))
              (ERROR '|BAD PROCESS -- START!PROCESS| **EXP** 'FAIL-ACT)))
       (OR (EQ P **PROCESS**) (MEMQ P **QUEUE**)
           (SETQ **QUEUE** (NCONC **QUEUE** (LIST P))))
       P)

(DEFUN STOP!PROCESS (P)
       (COND ((MEMQ P **QUEUE**)
              (SETQ **QUEUE** (DELQ P **QUEUE**)))
             ((EQ P **PROCESS** (TERMINATE)))
       P)
```

TERMINATE is an internal microcode routine which terminates the current process. If the current process is the only one, then all processes have been stopped, and so a new SCHEME top level is created; otherwise TERMINATE pulls the next process off the scheduler queue and swaps it in. Note that we cannot use SWAPINPROCESS because a RESTORE will happen in EVLIS as soon as TERMINATE completes (this is a very deep global property of the interpreter, and a fine source of bugs; much care is required).

```
(DEFUN TERMINATE ()
    (COND ((NULL **QUEUE**)
           (SETQ **PROCESS**
                 (CREATE!PROCESS '(**TOP** '|SCHEME - QUEUEOUT|))))
          (T (SETQ **PROCESS** (CAR **QUEUE**)
                   **QUEUE** (CDR **QUEUE**))))
    (SETQ **CLINK** (GET **PROCESS** 'CLINK))
    (SETQ **VAL** (GET **PROCESS** 'VAL))
    'TERMINATE-VALUE)
```

EVALUATE!UNINTERRUPTIBLY merely binds the variable *ALLOW* to NIL, and then evaluates its argument. This is why this primitive follows the scoping rules for variables!

```
(DEFPROP EVALUATE!UNINTERRUPTIBLY EVALUATE!UNINTERRUPTIBLY AINT)

(DEFUN EVALUATE!UNINTERRUPTIBLY ()
       (SETQ **ENV** (CONS (LIST '*ALLOW* NIL) **ENV**)
             **EXP** (CADR **EXP**)
             **PC** 'AEVAL))
```

DEFINE closes the function to be defined in the null environment, and installs the closure in the LISP value cell.

```
(DEFPROP DEFINE DEFINE AINT)

(DEFUN DEFINE ()
   (SET (CADR **EXP**) (LIST 'BETA (CADDR **EXP**) NIL))
   (SETQ **VAL** (CADR **EXP**))
   (RESTORE))
```

ASET looks up the specified variable in the current environment, and clobbers the value cell in the environment with the new value. If the variable is not bound in the current environment, the LISP value cell is set. Note that ASET does not need to be an AINT, since it does not fool with order of evaluation; all it needs is access to the "machine register" **ENV**.

```
(DEFUN ASET (VAR VALU)
       (SETQ **TEM** (ASSQ VAR **ENV**))
       (COND (**TEM** (RPLACA (CDR **TEM**) VALU))
             (T (SET VAR VALU)))
       VALU)
```

CATCH binds the tag variable to a DELTA expression which contains the current CLINK. When AEVAL applies such an expression as a function (of one argument), it makes the **CLINK** in the DELTA expression be the **CLINK**, places the value of the argument in **VAL**, and does a RESTORE. The effect is to return from the CATCH expression with the argument to the DELTA expression as its value (can you see why?).

```
(DEFPROP CATCH ACATCH AINT)

(DEFUN ACATCH ()
  (SETQ **ENV** (CONS (LIST (CADR **EXP**) (LIST 'DELTA **CLINK**))
      **ENV**)
**EXP** (CADDR **EXP**)
**PC** 'AEVAL))
```

PAIRLIS is as in the LISP 1.5 Programmer's Manual [11].[10]

```
(DEFUN PAIRLIS (X V Z)
       (DO ((I X (CDR I))
            (J Y (CDR J))
            (L Z (CONS (LIST (CAR I) (CAR J)) L)))
           ((AND (NULL I) (NULL J)) L)
          (AND (OR (NULL I) (NULL J))
               (ERROR '|WRONG NUMBER OF ARGUMENTS - PAIRLIS|
                      **EXP**
                      'WRNG-NO-ARGS))))
```

AMACROs are fairly complicated beasties, and have very little to do with the basic issues of the implementation of SCHEME per se, so the code for them will not be given here. AMACROs behave almost exactly like MacLISP macros [13].

This is the end of the SCHEME interpreter!

6. Acknowledgments

This paper would not have happened if Sussman had not been forced to think about lambda calculus by having to teach 6.031, nor would it have happened had not Steele been forced to understand PLASMA by morbid curiosity.

This work developed out of an initial attempt to understand the actorness of actors. Steele thought he understood it, but couldn't explain it; Sussman suggested the experimental approach of actually building an "ACTORS interpreter". This interpreter attempted to intermix the use of actors and LISP lambda expressions in a clean manner. When it was completed, we discovered that the "actors" and the lambda expressions were identical in implementation. Once we had discovered this, all the rest fell into place, and it was only natural to begin thinking about actors in terms of lambda calculus. The original interpreter was call-by-name for various reasons having to do with 6.031; we subsequently experimentally discovered how call-by-name screws iteration, and rewrote it to use call-by-value. Note well that we did *not* bring forth a clean implementation in one brilliant flash of understanding; we used an experimental and highly empirical approach to bootstrap our knowledge.

We wish to thank the staff of 6.031, Mike Dertouzos, and Steve Ward, for precipitating this intellectual adventure. Carl Hewitt spent many hours explaining the innards and outards of PLASMA to Steele over the course of several months; Marilyn McClennan was also helpful in this respect. Brian Smith and Richard Zippel helped a lot. We wish to thank Seymour Papert, Ben Kuipers, Marvin Minsky, and Vaughan Pratt for their excellent suggestions.

Notes

These notes were not part of the original paper. We have added them to illuminate certain points that would otherwise not be clear to readers in 1998. We should also remark that we have reformatted the original text for journal publication, making use of a greater variety of type faces and styles as well as changing the line breaks and indentation of the text and, where necessary, of the code. We have also quietly corrected minor errors in punctuation throughout the paper, but we have not changed any of the words, though we were sorely tempted to change most of the occurrences of "which" to "that" to accord with the more rigorous rules of usage that we adopted in our later writings.—GJS/GLS, 1998

1. In addition to the funny structure of this program there are a number of strange features and jokes that are incidental to its structure. Though this program worked correctly, PROGN is a MacLISP procedure that should not have appeared here. We should have used BLOCK to be consistent with the rest of the SCHEME examples. PROGN worked because it is a MacLISP LSUBR and thus its arguments were evaluated in left-to-right order. PROGN returns its last argument, and so it is equivalent to BLOCK as far as the value produced, but BLOCK executes its last form "tail-recursively" and PROGN does not. That difference does not matter for this implementation of SQRT, because the expression (LOOPTAG LOOPTAG) always performs a nonlocal transfer of control, so the procedure PROGN is never actually invoked—indeed, we could just as well have used + or APPEND or any other procedure that accepts four arguments! Another strange feature of this program is the use of the "fast" MacLISP floating-point–specific math procedures -$, *$, etc. Of course, this program ran so slowly under our SCHEME interpreter that it was of no consequence for us to use these obscure primitives! Finally, the comment ;JFCL refers to the fastest instruction on the DEC PDP-10 that does nothing. Indeed, we came very close to writing JRST (the fastest unconditional PDP-10 jump instruction) instead of GOTO in the last comment; we fully expected our original intended audience, our colleagues at the MIT Artificial Intelligence Laboratory, to be fluent in PDP-10 assembly language. Our intention was to be terse, not cryptic.
2. In the initial definition of SCHEME, we tried to make each primitive construct as simple as possible. In modern versions of SCHEME, the body of a LAMBDA expression is indeed an implicit BLOCK.
3. MIT-AI was the name of the DEC PDP-10 computer that we were using for our research. DDT was a command shell and assembly language debugger rolled into one.
4. There is a tiny joke here: we wrote (EVAL' (UREAD FOO BAR DSK LOSER)) instead of the more customary (EVAL '(UREAD FOO BAR DSK LOSER)) so that the operation could be pronounced "evalquote".

5. Both (THV x) and ?x are bits of syntax from the MicroPLANNER language, which used pattern matching as an essential part of its control structure.
6. We believe that this was the first occurrence of the term "continuation-passing style" in the literature. It has turned out to be an important concept in source code analysis and transformation for compilers and other metaprogramming tools. It has also inspired a set of other "styles" of program expression.
7. That is, the expression in **EXP**.
8. In the following code, the special treatment for a LAMBDA expression as the first form of a combination is inessential. Such expressions would be treated correctly if the special case (the penultimate clause of the outermost COND) were eliminated. The special case was added as a speed-up hack, to avoid the construction of a BETA closure. This is discussed further in the next note.
9. Here again the special treatment of a LAMBDA expression as a function object is a semantically inessential speedup hack. If the LAMBDA expression had been evaluated in AEVAL to make a BETA closure, the environment that would be extracted by EVLIS from that closure would be exactly the environment in **ENV**. The special handling of LAMBDA expressions in EVLIS exploits this fact.
10. Actually, this is false. We used LIST to make the pairs, where the original PAIRLIS used CONS; and, of course, the original PAIRLIS was not written using the MacLISP new-style DO loop!

References

1. Bobrow, Daniel G. and Wegbreit, Ben. A model and stack implementation of multiple environments. *CACM 16, 10*, pages 591–603, October 1973.
2. Church, Alonzo. The calculi of lambda conversion. *Annals of Mathematics Studies Number 6*, Princeton University Press, 1941. Reprinted by Klaus Reprint Corp. (New York), 1965.
3. Dijkstra, Edsger W. Solution of a problem in concurrent programming control. *CACM 8,9*, page 569, September 1965.
4. Fischer, Michael J. Lambda calculus schemata. In *Proceedings of ACM Conference on Proving Assertions about Programs, SIGPLAN Notices*, January 1972.
5. Galley, S.W. and Pfister, Greg. The MDL language. In *Programming Technology Division Document SYS.11.01*. Project MAC, MIT (Cambridge), November 1975.
6. Greif, Irene. *Semantics of Communicating Parallel Processes.* Ph.D. thesis. Technical Report MAC-TR-154, Project MAC, MIT (Cambridge), September 1975.
7. Greif, Irene and Hewitt, Carl. Actor semantics of Planner-73. Technical Report Working Paper 81, MIT AI Lab (Cambridge), 1975.
8. Ingerman, P.Z. Thunks—a way of compiling procedure statements with some comments on procedure declarations. *CACM 4,1*, pages 55–58, January 1961.
9. Knuth, Donald E. Additional comments on a problem in concurrent programming control. *CACM 9,5*, pages 321–322, May 1966.
10. Lamport, Leslie. A new solution of Dijkstra's concurrent programming problem. *CACM 17,8*, pages 453–455, August 1974.
11. McCarthy, John, et al. *LISP 1.5 Programmer's Manual*. The MIT Press (Cambridge), 1965.
12. McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER reference manual. Technical Report AI Memo 295a, MIT AI Lab (Cambridge), January 1974.
13. Moon, David A. *MACLISP Reference Manual, Revision 0*. Project MAC, MIT (Cambridge), 1974.
14. Moses, Joel. The function of FUNCTION in LISP. Technical Report AI Memo 199, MIT AI Lab (Cambridge), June 1970.
15. Project MAC, MIT. (Cambridge). *Project MAC Progress Report XI (July 1973–July 1974)*, 1974.
16. Reynolds, John C. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, 1972.
17. Smith, Brian C. and Hewitt, Carl. A PLASMA primer (draft). Technical report, MIT AI Lab (Cambridge), October 1975.
18. Sussman, Gerald Jay, Winograd, Terry and Charniak, Eugene. Micro-PLANNER reference manual. Technical Report AI Memo 203A, MIT AI Lab (Cambridge), December 1971.