MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 452 January 1978

THE REVISED REPORT ON

SCHEME

A DIALECT OF LISP

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman

Abstract:

SCHEME is a dialect of LISP. 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. This has the consequence that variables are normally lexically scoped, as in ALGOL. However, in contrast with ALGOL, SCHEME treats procedures as a first-class data type. They can be the values of variables, the returned values of procedures, and components of data structures. Another difference from LISP is that SCHEME is implemented in such a way that tail-recursions execute without net growth of the interpreter stack. The effect of this is that a procedure call behaves like a GOTO, and thus procedure calls can be used to implement iterations, as in PLASMA.

Here we give a complete "user manual" for the SCHEME language. Some features described here were not documented in the original report on SCHEME (for instance particular macros). Other features have been added, changed, or deleted as our understanding of certain language issues evolved. Annotations to the manual describe the motivations for these changes.

Keywords: LISP, SCHEME, LISP-like languages, lambda-calculus, environments, lexical scoping, dynamic scoping, fluid variables, control structures, macros, extensible syntax, extensible languages

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.

* NSF Fellow

A. The Representation of SCHEME Procedures as S-expressions

SCHEME programs are represented as LISP s-expressions. The evaluator interprets these s-expressions in a specified way. This specification constitutes the definition of the language.

The definition of SCHEME is a little fuzzy around the edges. This is because of the inherent extensibility of LISP-like languages {Note LISP Is a Ball of Mud}. We can define a few essential features which constitute the "kernel" of the language, and also enumerate several syntactic and semantic extensions which are convenient and normally included in a given implementation. The existence of a mechanism for such extensions is a part of the kernel of SCHEME; however, any particular such extension is not necessarily part of the kernel.

For those who like this sort of thing, here is the BNF for SCHEME programs {Note LISP BNF}:

```
<form> ::= <non-symbol atom> | <identifier> | <magic> | <combination>
<identifier> ::= <atomic symbol>
<magic> ::= <lambda-expression>
          | (QUOTE <s-expression> )
          | (IF <form> <form> <form> )
          | (IF <form> <form> )
          | (LABELS ( <labels list> ) <body> )
          | (DEFINE <identifier> <lambda-expression> )
          | (DEFINE <identifier> ( <identifier list> ) <form> )
          | (DEFINE ( <identifier> <identifier list> ) <form> )
          | (ASET' <identifier> <form> )
          | (FLUIDBIND ( <fluidbind list> ) <form> )
          | (FLUID <identifier> )
          | (FLUIDSET' <identifier> <form> )
          | (CATCH <identifier> <form> )
          | <syntactic extension>
<lambda-expression> ::= (LAMBDA ( <identifier list> ) <body>)
<identifier list> ::= <empty> | <identifier> <identifier list>
<body> ::= <form>
<labels list> ::= <empty>
          | ( <identifier> <lambda-expression> ) <labels list>
<fluidbind list> ::= <empty> | ( <identifier> <form> ) <fluidbind list>
<combination> ::= ( <form list> )
<form list> ::= <form> | <form> <form list>
<syntactic extension> ::= ( <magic word> . <s-expression> )
<magic word> ::= <atomic symbol>
<non-symbol atom> ::= <number> | <array> | <string> | ...
```

Atoms which are not atomic symbols (identifiers) evaluate to themselves. Typical examples of such atoms are numbers, arrays, and strings (character arrays). Symbols are treated as identifiers or variables. They

may be lexically bound by lambda-expressions. There is a global environment containing values for (some) free variables. Many of the variables in this global environment initially have as their values primitive operations such as, for example, CAR, CONS, and PLUS. SCHEME differs from most LISP systems in that the atom CAR is not itself an operation (in the sense of being an invocable object, e.g. a valid first argument to APPLY), but only has one as a value when considered as an identifier.

Non-atomic forms are divided by the evaluator into two classes: combinations and "magic (special) forms". The BNF given above is ambiguous; any magic form can also be parsed as a combination. The evaluator always treats an ambiguous case as a magic form. Magic forms are recognized by the presence of a "magic (reserved) word" in the car position of the form. All non-atomic forms which are not magic forms are considered to be combinations. The system has a small initial set of magic words; there is also a mechanism for creating new ones {Note FUNCALL is a Pain}.

A combination is considered to be a list of subforms. These subforms are all evaluated. The first value must be a procedure; it is applied to the other values to get the value of the combination. There are four important points here:

(1) The procedure position is always evaluated just like any other position. (This is why the primitive operators are the values of global identifiers.)

(2) The procedure is never "re-evaluated"; if the first subform fails to evaluate to an applicable procedure, it is an error. Thus, unlike most LISP systems, SCHEME always evaluates the first subform of a combination exactly once.

(3) The arguments are all completely evaluated before the procedure is applied; that is, SCHEME, like most LISP systems, is an applicative-order language. Many SCHEME programs exploit this fact.

(4) The argument forms (and procedure form) may in principle be evaluated in any order. This is unlike the usual LISP left-to-right order. (All SCHEME interpreters implemented so far have in fact performed left-to-right evaluation, but we do not wish programs to depend on this fact. Indeed, there are some reasons why a clever interpreter might want to evaluate them right-to-left, e.g. to get things on a stack in the correct order.)

B. Catalogue of Magic Forms

B.1. Kernel Magic Forms

The magic forms in this section are all part of the kernel of SCHEME, and so must exist in any SCHEME implementation.

(LAMBDA (<identifier list>) <body>)

Lambda-expressions evaluate to procedures. Unlike most LISP systems, SCHEME does not consider a lambda-expression (an s-expression whose car is the atom LAMBDA) to be a procedure. A lambda-expression only evaluates to a procedure. A lambda-expression should be thought of as a partial description of a procedure; a procedure and a description of it are conceptually distinct objects. A lambda-expression must be "closed" (associated with an environment) to produce a procedure object. Evaluation of a lambda-expression performs such a closure operation.

The resulting procedure takes as many arguments as there are identifiers in the identifier list of the lambda-expression. When the procedure is eventually invoked, the intuitive effect is that the evaluation of the procedure call is equivalent to the evaluation of the <body> in an environment consisting of (a) the environment in which the lambda-expression had been evaluated to produce the procedure, plus (b) the pairing of the identifiers of the <identifier list> with the arguments supplied to the procedure. The pairings (b) take precedence over the environment (a), and to prevent confusion no identifier may appear twice in the <identifier list>. The net effect is to implement ALGOL-style lexical scoping [Naur], and to "solve the funarg problem" [Moses].

(IF <predicate> <consequent> <alternative>)

This is a primitive conditional operator. The predicate form is evaluated. If the result is non-NIL {Note IF Is Data-Dependent}, then the consequent is evaluated, and otherwise the alternative is evaluated. The resulting value (if there is one) is the value of the IF form.

(IF <predicate> <consequent>)

As above, but if the predicate evaluates to NIL, then NIL is the value of the IF form. (As a matter of style, this is usually used only when the value of the IF form doesn't matter, for example, when the consequent is intended to cause a side effect.)

(QUOTE <s-expression>)

As in LISP, this quotes the argument form so that it will be passed verbatim as data; the value of this form is <s-expression>. If a SCHEME implementation has the MacLISP read-macro-character feature, then the abbreviation 'FOO may be used instead of (QUOTE FOO).

(LABELS (<labels list>) <body>)
where <labels list> ::= <empty> | (<identifier> <lambda-expression>) <labels list>

This has the effect of evaluating the <body> in an environment where all the identifiers (which, as for LAMBDA, must all be distinct) in the labels list evaluate to the values of the respective lambda-expressions. Furthermore, the procedures which are the values of the lambda-expressions are themselves closed in that environment, and not in the outer environment; this allows the procedures to call themselves and each other recursively. For example, consider a procedure 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 a list count). In order to perform this we define two mutually recursive procedures, 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))))
```

(We have decided not to use the traditional LISP LABEL primitive in SCHEME because it is difficult to define several mutually recursive procedures using only LABEL. Although LABELS is a little more complicated than LABEL, it is considerably more convenient. Contrast this design decision with the choice of IF over the more traditional COND, where the definitional simplicity of IF outweighed the somewhat greater convenience of COND.)

B.2. Side Effects

These magic forms produce side effects in the environment.

(DEFINE <identifier> <lambda-expression>)

This is used for defining a procedure in the "global environment" permanently, as opposed to LABELS, which is used for temporary procedure definitions in a local environment. DEFINE takes a name and a lambda-expression; it evaluates the lambda-expression in the global environment and causes the result to be the global value of the identifier. (DEFINE may perform other implementation-dependent operations as well, such as keeping track of defined procedures for an editor. For this reason it is the preferred way to define a globally available procedure.)

(DEFINE <identifier> (<identifier list>) <form list>)
(DEFINE (<identifier> <identifier list>) <form list>)

These alternative syntaxes permitted by DEFINE are equivalent to:

```
(DEFINE <identifier>
        (LAMBDA ( <identifier list> )
                (BLOCK <form list> )))
```

where BLOCK is a syntactic extension defined below. For example, these three definitions are equivalent:

```
(DEFINE CIRCULATE (LAMBDA (X) (RPLACD X X)))
(DEFINE CIRCULATE (X) (RPLACD X X))
(DEFINE (CIRCULATE X) (RPLACD X X))
```

These forms are provided to support stylistic diversity.

(ASET' <identifier> <form>)

This is analogous to the LISP primitive SETQ. For example, to define a cell [Smith and Hewitt], 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)))
```

Note that ASET' may be used on global identifiers as well as locally bound identifiers {Note ASET Has Disappeared}.

B.3. Dynamic Magic

These magic forms implement escape objects and fluid (dynamic) variables. They are not a part of the essential kernel. For a further explication of their semantics in terms of kernel primitives, see [Imperative].

(FLUIDBIND (<fluidbind list>) <form>)
where <fluidbind list> ::= <empty> | (<identifier> <form>) <fluidbind list>

This evaluates the <form> in the environment of the FLUIDBIND form, with a dynamic environment to which dynamic bindings of the identifiers in the <fluidbind list> have been added. Any procedure dynamically called by the form, even if not lexically apparent to the FLUIDBIND form, will see this dynamic environment (unless modified by further FLUIDBINDs, of course). The dynamic environment is restored on return from the form.

Most LISP systems use a dynamic environment for all variables. A SCHEME which implements FLUIDBIND provides two distinct environments. The fluid variable named FOO is completely unrelated to a normal lexical variable named FOO {Note Global Fluid Environment}, and the access mechanisms for the two are distinct.

(FLUID <identifier>)

The value of this form is the value of the <identifier> in the current dynamic environment. In SCHEME implementations which have the MacLISP read-macro-character feature, (FLUID FOO) may be abbreviated to •FOO.

(FLUIDSET' <identifier> <form>)

The value of the <form> is assigned to the <identifier> in the current dynamic environment.

(STATIC <identifier>)

The value of this is the value of the lexical identifier; writing this is the same as writing just <identifier> {Note What Good Is It?}.

(CATCH <identifier> <form>)

This evaluates the form in an environment where the identifier is bound to an "escape object" [Landin] [Reynolds]. This is a strange object which can be invoked as if it were a procedure of one argument. When the escape object is so invoked, then control proceeds as if the CATCH expression had returned with the supplied argument as its value {Note Multiple Throw}.

If both CATCH and FLUIDBIND are implemented, then their semantics are intertwined. When the escape object is called, then the dynamic environment is restored to the one which was current at the time the CATCH form was evaluated {Note Environment Symmetry}.

For a contorted example, consider the following obscure definition of SQRT (Sussman's least favorite style/Steele's favorite; but see [SCHEME]):

```
(DEFINE SQRT
   (LAMBDA (X EPSILON)
      ((LAMBDA (ANS TAG GO)
          (CATCH RETURN
                 (BLOCK
                  (CATCH M (ASET' TAG M))                      ;CREATE PROG TAG
                  (IF (< (ABS (-$ (*$ ANS ANS) X)) EPSILON)    ;CAMGE
                      (RETURN ANS))                            ;POPJ
                  (ASET' ANS (//$ (+$ (//$ X ANS) ANS) 2.0))   ;MOVEM
                  (GO TAG))))                                  ;JRST
       1.0
       NIL
       (LAMBDA (F) (F NIL)))))
```

This example differs slightly from the version given in [SCHEME]; notice the forms (RETURN ANS) and (GO TAG).

As another example, we can define a THROW function, which may then be used with CATCH much as it is in MacLISP [Moon] (except that in MacLISP the tag is written after the body of the CATCH, not before):

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

An example of its use:

```
(CATCH LOSE
       (MAPCAR (LAMBDA (X) (IF (MINUSP X) (THROW LOSE NIL) (SQRT X)))
               NUMLIST))
```

Indeed, note the similarity between THROW and the definition of GO in the first example.

C. Syntactic Extensions

SCHEME has a syntactic extension mechanism which provides a way to define an identifier to be a magic word, and to associate a function with that word. The function accepts the magic form as an argument, and produces a new form; this new form is then evaluated in place of the original (magic) form. This is precisely the same as the MacLISP macro facility.

C.1. System-Provided Extensions

Some standard syntactic extensions are provided by the system for convenience in ordinary programming. They are distinguished from other magic words in that they are semantically defined in terms of others rather than being primitive {Note FEXPRs Are Okay by Us}. For expository purposes they are described here in a pattern-matching/production-rule kind of language. The matching is on s-expression structure, not on character string syntax, and takes advantage of the definition of list notation: (A B C) = (A . (B . (C . NIL))). Thus the pattern (x . r) matches (A B C), with x = A and r = (B C). The ordering of the "productions" is significant; the first one which matches is to be used.

(BLOCK x_1 x_2 ... x_n)

(BLOCK x) → x
(BLOCK x . r) → ((LAMBDA (A B) (B)) x (LAMBDA () (BLOCK . r)))

BLOCK sequentially evaluates the subforms x_i from left to right. For example:

(BLOCK (ASET' X 43) (PRINT X) (+ X 1))

returns 44 after setting X to 43 and then printing it {Note BLOCK Exploits Applicative Order}.

(LET ((v_1 x_1) (v_2 x_2) ... (v_n x_n)) . body)

→ ((LAMBDA (v_1 v_2 ... v_n) (BLOCK . body)) x_1 x_2 ... x_n)

LET provides a convenient syntax for binding several variables to corresponding quantities. It allows the forms for the quantities to appear textually adjacent to their corresponding variables. Notice that the variables are all bound simultaneously, not sequentially, and that the initialization forms x_i may be evaluated in any order. For convenience, LET also supplies a BLOCK around the forms constituting its body.

```
(DO ((v1 x1 s1) ... (vn xn sn)) (test . done) . body)

      → (LET ((A1 (LAMBDA () x1))
              (B1 (LAMBDA (v1 ... vn) s1))
              ...
              (An (LAMBDA () xn))
              (Bn (LAMBDA (v1 ... vn) sn))
              (TS (LAMBDA (v1 ... vn) test))
              (DN (LAMBDA (v1 ... vn) (BLOCK . done)))
              (BD (LAMBDA (v1 ... vn) (BLOCK . body))))
             (LABELS ((LOOP
                       (LAMBDA (Z1 ... Zn)
                               (IF (TS Z1 ... Zn)
                                   (DN Z1 ... Zn)
                                   (BLOCK (BD Z1 ... Zn)
                                          (LOOP (B1 Z1 ... Zn)
                                                ...
                                                (Bn Z1 ... Zn)))))))
                      (LOOP (A1) ... (An))))
```

This is essentially the MacLISP "new-style" DO loop [Moon]. The variables v_i are bound to the values of the corresponding x_i, and stepped in parallel after every execution of the body by the s_i (by "step" we mean "set to the value of", not "increment by"). If an s_i is omitted, v_i is assumed; this results in the variable not being stepped. If in addition x_i is omitted, NIL is assumed. The loop terminates when the test evaluates non-NIL; it is evaluated before each execution of the body. When this occurs, the done part is evaluated as a BLOCK.

The complexity of the definition shown is due to an effort to avoid conflict of variable names, as for BLOCK. The auxiliary variables Ai, Bi, and Zi must be generated to produce as many as are needed, but they need not be chosen different from all variables appearing in x_i, s_i, body, etc.

The iteration is effected entirely by procedure calls. In this manner the definition of DO exploits the tail-recursive properties of SCHEME [SCHEME] [Imperative].

As an example, here is a definition of a function to find the length of a list:

```
(DEFINE (LENGTH X)
        (DO ((Z X (CDR Z))
             (N 0 (+ N 1)))
            ((NULL Z) N)))
```

The initializations forms x_i may be evaluated in any order, and on each iteration the stepping form s_i may be evaluated in any order. This differs from the MacLISP definition of DO. For example, this definition of NREVERSE (destructively reverse a list) would work in MacLISP but not necessarily in SCHEME:

```
(DEFINE NREVERSE (X)
        (DO ((A X (CDR A))
             (B NIL (RPLACD A B)))
            ((NULL A) B)))
```

This definition depends on the CDR occurring before the RPLACD. In SCHEME we must instead write:

```
(DEFINE NREVERSE (X)
        (DO ((A X (PROG1 (CDR A) (RPLACD A B)))
             (B NIL A))
            ((NULL A) B)))
```

where by (PROG1 x y) we mean ((LAMBDA (P Q) (BLOCK (Q) P)) x (LAMBDA () y)) (but PROG1 is not really a defined SCHEME primitive).

Note also that the effect of an ASET' on a DO variable does not survive to the next iteration; this differs from using SETQ on a DO variable in MacLISP.

(ITERATE name ((v_1 e_1) ... (v_n e_n)) . body)

→ (LABELS ((name (LAMBDA (v_1 ... v_n) (BLOCK . body))))
(name e_1 ... e_n))

This defines a looping construct more general than DO. For example, consider a function to sort out a list of s-expressions into atoms and lists:

```
(DEFINE COLLATE
        (LAMBDA (X)
                (ITERATE COL
                         ((Z X) (ATOMS NIL) (LISTS NIL))
                         (IF (NULL Z)
                             (LIST ATOMS LISTS)
                             (IF (ATOM (CAR Z))
                                 (COL (CDR Z) (CONS (CAR Z) ATOMS) LISTS)
                                 (COL (CDR Z) ATOMS (CONS (CAR Z) LISTS)))))))
```

We have found many situations involving loops where there may be more than one condition on which to exit and/or more than one condition to iterate, where DO is too restrictive but ITERATE suffices. Notice that because each loop has a name, one can specify from an inner loop that the next iteration of any outer loop is to occur. Here is a function very similar to the one used in one SCHEME implementation for variable lookup: there are two lists of lists, one containing names and the other values.

```
(DEFINE (LOOKUP NAME VARS VALUES)
        (ITERATE MAJOR-LOOP
                 ((VARS-BACKBONE VARS)
                  (VALUES-BACKBONE VALUES))
                 (IF (NULL VARS-BACKBONE)
                     NIL
                     (ITERATE MINOR-LOOP
                              ((VARS-RIB (CAR VARS-BACKBONE))
                               (VALUES-RIB (CAR VALUES-BACKBONE)))
                              (IF (NULL VARS-RIB)
                                  (MAJOR-LOOP (CDR VARS-BACKBONE)
                                              (CDR VALUES-BACKBONE))
                                  (IF (EQ (CAR VARS-RIB) NAME)
                                      VALUES-RIB
                                      (MINOR-LOOP (CDR VARS-RIB)
                                                  (CDR VALUES-RIB)))))))))
```

(We had originally wanted to call this construct LOOP, but see {Note FUNCALL is a Pain}. Compare this with looping constructs appearing in [Hewitt].)

It happens that ITERATE is a misleading name; the construct can actually be used for recursion ("true" recursion, as opposed to tail-recursion) as well. If the name is invoked from a non-tail-recursive situation, the argument evaluation in which the call is embedded is not aborted. It just so happens that we have found ITERATE useful primarily to implement complicated iterations. One can draw the rough analogy ITERATE : LABELS :: LET : LAMBDA.

(TEST pred fn alt)

```
→ ((LAMBDA (P F A) (IF P ((F) P) (A)))
   pred
   (LAMBDA () fn)
   (LAMBDA () alt))
```

The predicate is evaluated; if its value is non-NIL then the form fn should evaluate to a procedure of one argument, which is then invoked on the value of the predicate. Otherwise the alternative alt is evaluated.

This construct is of occasional use with LISP "predicates" which return a "useful" non-NIL value. For the consequent of an IF to get at the non-NIL value of the predicate, one might first bind a variable to the value of the predicate, and this variable would then be visible to the alternative as well. With TEST, the use of the variable is restricted to the consequent.

An example:

```
(TEST (ASSQ VARIABLE ENVIRONMENT)
```

```
        CDR
        (GLOBALVALUE VARIABLE))
```

Using an a-list to represent an environment, one wants to use the cdr of the result of ASSQ if it is non-NIL; but if it is NIL, then the variable was not in the environment, and one must look elsewhere.

(COND $(p_1 \cdot e_1) \ldots (p_n \cdot e_n)$)

```
(COND) → 'NIL
(COND (p) . r) → ((LAMBDA (V R) (IF V V (R)))
                  p
                  (LAMBDA () (COND . r)))
(COND (p => f) . r) → (TEST p f (COND . r))
(COND (p . e) . r) → (IF p (BLOCK . e) (COND . r))
```

This COND is a superset of the MacLISP COND. As in MacLISP, singleton clauses return the value of the predicate if it is non-NIL, and clauses with two or more forms treat the first as the predicate and the rest as constituents of a BLOCK, thus evaluating them in order.

The extension to the MacLISP COND made in SCHEME is flagged by the atom =>. (It cannot be confused with the more general case of two BLOCK constituents because having the atom => as the first element of a BLOCK is not useful.) In this situation the form f following the => should have as its value a function of one argument; if the predicate p is non-NIL, this function is determined and invoked on the value returned by the predicate. This is useful for the common situation encountered in LISP:

```
(COND ((SETQ IT (GET X 'PROPERTY)) (HACK IT))
      ...)
```

which in SCHEME can be rendered without using a variable global to the COND:

```
(COND ((GET X 'PROPERTY)
       => (LAMBDA (IT) (HACK IT)))
      ...)
```

or, in this specific instance, simply as:

```
(COND ((GET X 'PROPERTY) => HACK)
      ...)
```

(OR x_1 x_2 ... x_n)

```
(OR) → 'NIL
(OR x) → x
(OR x . r) → (COND (x) (T (OR . r)))
```

This standard LISP primitive evaluates the forms x_i in order, returning the first non-NIL value (and ignoring all following forms). If all forms produce NIL, then NIL is returned {Note Tail-Recursive OR}.

(AND x_1 x_2 ... x_n)

```
(AND) → 'T
(AND x) → x
(AND x . r) → (COND (x (AND . r)))
```

This standard LISP primitive evaluates the forms x_i in order. If any form produces NIL, then NIL is returned, and succeeding forms x_i are ignored. If all forms produce non-NIL values, the value of the last is returned {Note Tail-Recursive AND}.

(AMAPCAR f x_1 x_2 ... x_n)

```
→ (DO ((FN f)
       (V1 x1 (CDR V1))
       (V2 x2 (CDR V2))
       ...
       (Vn xn (CDR Vn))
       (Q 'NIL (CONS (FN (CAR V1) (CAR V2) ... (CAR Vn)) Q)))
      ((OR (NULL V1) (NULL V2) ... (NULL Vn))
       (NREVERSE Q)))
```

AMAPCAR is analogous to the MacLISP MAPCAR function. The function f, a function of n arguments, is mapped simultaneously down the lists x_1, x_2, ..., x_n; that is, f is applied to tuples of successive elements of the lists. The values returned by f are collected and returned as a list. Note that AMAPCAR of a fixed number of arguments could easily be written as a function in SCHEME. It is a syntactic extension only so that it may accommodate any number of arguments, which saves the trouble of defining an entire set of primitive functions AMAPCAR1, AMAPCAR2, ... where AMAPCARn takes n+1 arguments.

(AMAPLIST f x_1 x_2 ... x_n)

```
→ (DO ((FN f)
       (V1 x1 (CDR V1))
       (V2 x2 (CDR V2))
       ...
       (Vn xn (CDR Vn))
       (Q 'NIL (CONS (FN V1 V2 ... Vn) Q)))
      ((OR (NULL V1) (NULL V2) ... (NULL Vn))
       (NREVERSE Q)))
```

AMAPLIST is analogous to the MacLISP MAPLIST function. The function f, a function of n arguments, is applied to tuples of successive tails of the lists. The values returned by f are collected and returned as a list.

(AMAPC f x_1 x_2 ... x_n)

```
→ (DO ((FN f)
       (X1 x1 (CDR X1))
       (X2 x2 (CDR X2))
       ...
       (Xn xn (CDR Xn)))
      ((OR (NULL X1) (NULL X2) ... (NULL Xn))
       'NIL)
      (FN (CAR X1) (CAR X2) ... (CAR Xn)))
```

AMAPC is analogous to the MacLISP MAPC function. The procedure f, a procedure taking n arguments, is mapped simultaneously down the lists x_1, x_2, ..., x_n; that is, f is applied to tuples of successive elements of the lists. Thus AMAPC is similar to AMAPCAR, except that no values are expected from f; therefore f need not be a function, but may be any procedure.

(PROG varlist s_1 s_2 ... s_n)

The SCHEME PROG is like the ordinary LISP PROG. There is no simple way to describe the transformation of PROG syntax into SCHEME primitives. The basic idea is that a large LABELS statement is created, with a labelled procedure (of zero arguments) for each PROG statement. Each statement is transformed in such a way that each one that "drops through" is made to call the labelled procedure for the succeeding statement; each appearance of (GO tag) is converted to a call on the labelled procedure for the statement following the tag; and each appearance of (RETURN value) is replaced by value.

Practical experience with SCHEME has shown that PROG is almost never used. It is usually more convenient just to write the corresponding LABELS directly. This allows one to write LABELS procedures which take

arguments, which tends to clarify the flow of data [Imperative].

The rest of this section (FSUBRs) applies only to the PDP-10 MacLISP implementation of SCHEME.

FSUBRs

As a user convenience, the PDP-10 MacLISP implementation of SCHEME treats FSUBRs specially; any FSUBR provided by the MacLISP system is automatically a SCHEME primitive (but user FSUBRs are not). Of course, if the FSUBR tries to evaluate some form obtained from its argument, the variable references will not refer to SCHEME variables. As a special case, the SCHEME syntactic extension AARRAYCALL is provided to get the effect of the MacLISP FSUBR ARRAYCALL.

C.2. User-Provided Extensions

A SCHEME implementation should have one or more ways for the user to extend the inventory of magic words. The methods provided will vary from implementation to implementation. The following primitive (SCHMAC) is provided in the PDP-10 MacLISP implementation of SCHEME.

(SCHMAC name pattern body)

After execution of this form, a syntactic extension keyed on the atom name is defined. When a form (name . rest) is to be evaluated, rest is matched against the pattern, which is a (possibly "dotted") list of variables. The body is then evaluated in an environment where the variables in the pattern have as values the corresponding parts of rest. This should result in a form to be evaluated in place of the form (name . rest).

The body is not necessarily SCHEME code, but rather code in the same meta-language used to write the evaluator. In the PDP-10 MacLISP SCHEME implementation, the body is MacLISP code.

As an example, here is a definition of TEST:

```
(SCHMAC TEST (PRED FN ALT)
      (LIST '(LAMBDA (P F A) (IF P ((F) P) (A)))
            PRED
            (LIST 'LAMBDA '() FN)
            (LIST 'LAMBDA '() ALT)))
```

The body of a SCHMAC almost always performs a complicated consing-up of a program structure. Often one needs to make a copy of a standard

structure, with a few values filled in. To make this easier, SCHEME provides an "unquoting quote" feature. An expression of the form "<s-expression> is just like '<s-expression>, except that sub-expressions preceded by "," or "@" represent expressions whose values are to be made part of (a copy of) the s-expression at that point. A "," denotes simple inclusion, while "@" denotes "splicing" or "segment" inclusion. (Compare this with the treatment of lists with embedded forms in MUDDLE [Galley and Pfister], which in turn inspired the !" syntax of CONNIVER [McDermott and Sussman], from which SCHEME's " syntax is derived.) Using this, one can define TEST as follows:

```
(SCHMAC TEST (PRED FN ALT)
        "((LAMBDA (P F A) (IF P ((F) P) (A)))
          ,PRED
          (LAMBDA () ,FN)
          (LAMBDA () ,ALT)))
```

Similarly, LET can be defined as:

```
(SCHMAC LET (DEFNS . BODY)
        "((LAMBDA ,(MAPCAR 'CAR DEFNS)
                  (BLOCK . ,BODY))
          @(MAPCAR 'CADR DEFNS)))
```

One could also write (BLOCK @BODY) instead of (BLOCK . ,BODY).

Notice the use of (MAPCAR 'CAR DEFNS) rather than (AMAPCAR CAR DEFNS), and recall that, as stated above, the body of a SCHMAC is MacLISP code, not SCHEME code. Consider too this definition of COND:

```
(SCHMAC COND CLAUSES
        (COND ((NULL CLAUSES) ''NIL)
              ((NULL (CDAR CLAUSES))
               "((LAMBDA (V R) (IF V V R))
                 ,(CAAR CLAUSES)
                 (LAMBDA () (COND . ,(CDR CLAUSES)))))
              ((EQ (CADAR CLAUSES) '=>)
               "(TEST ,(CAAR CLAUSES) ,(CADDAR CLAUSES) (COND . ,(CDR CLAUSES))))
              (T "(IF ,(CAAR CLAUSES)
                      (BLOCK . ,(CDAR CLAUSES))
                      (COND . ,(CDR CLAUSES))))))
```

We have used COND to define COND! The definition is not circular, however; the MacLISP COND is being used to define the SCHEME COND, and indeed the two have slightly different semantics. The definition would have been circular had we written (COND (V) (R)) instead of (IF V V R), for the latter is part of the generated SCHEME code.

(MACRO name pattern body)

This is just like SCHMAC, except that body is SCHEME code rather than MacLISP code. While macros defined with SCHMAC run only in a MacLISP implementation of SCHEME, those defined with MACRO should be completely transportable. (We described SCHMAC first to emphasize the fact that macros are conceptually part of the interpreter, and so conceptually written in the meta-language. It so happens, however, that SCHEME is a good meta-language for SCHEME, and so introducing this meta-circularity provides no serious problems. Contrast this with writing PL/I macros in PL/I!)

The example of defining COND using SCHMAC above would be circular if we changed the word SCHMAC to MACRO. However, we can avoid this by avoiding the use of COND in the definition:

```
(MACRO COND CLAUSES
       (IF (NULL CLAUSES) ''NIL
           (IF (NULL (CDAR CLAUSES))
               "((LAMBDA (V R) (IF V V R))
                 ,(CAAR CLAUSES)
                 (LAMBDA () (COND . ,(CDR CLAUSES))))
               (IF (EQ (CADAR CLAUSES) '=>)
                   "(TEST ,(CAAR CLAUSES)
                          ,(CADDAR CLAUSES)
                          (COND . ,(CDR CLAUSES)))
                   "(IF ,(CAAR CLAUSES)
                        (BLOCK . ,(CDAR CLAUSES))
                        (COND . ,(CDR CLAUSES)))))))
```

We strongly encourage the use of MACRO instead of SCHMAC in practice so that macro definitions will not be dependent on the properties of a specific implementation.

D. Primitive SCHEME Functions

All the usual MacLISP SUBRs are available in SCHEME as procedures which are the values of global variables. The particular primitives CONS, CAR, CDR, ATOM, and EQ are part of the kernel of SCHEME! Others, such as +, -, *, //, =, EQUAL, RPLACA, RPLACD, etc. are quite convenient to have.

Although there is no way in SCHEME to write a LEXPR (a function of a variable number of arguments), MacLISP LSUBRs are also available to the SCHEME user. One may wish to regard these as syntactic extensions in much the same way AMAPCAR is; for example, LIST may be thought of as a syntactic extension such that:

```
(LIST) → 'NIL
(LIST x . r) → (CONS x (LIST . r))
```

Below we also describe some additional primitive functions provided with SCHEME. Their names do not have any special syntactic properties in the way that the magic words for syntactic extensions described in the previous section do. However, they do deal with the underlying implementation, and so could not be programmed directly by the user were they not provided as primitives.

The following primitive functions (PROCP and ENCLOSE) are part of the kernel of SCHEME.

(PROCP thing)

This is a predicate which is true of procedures, and not of anything else. Thus if (PROCP X) is true, then it is safe to invoke the value of X.

More precisely, if PROCP returns a non-NIL value, then the value describes the number of arguments accepted by the procedure. For SCHEME procedures this will be an integer, the number of arguments. For primitive functions, this may be implementation-dependent; in the PDP-10 MacLISP implementation of SCHEME, PROCP of an LSUBR returns the MacLISP ARGS property for that LSUBR. If an object given to PROCP is a procedure but the number of arguments it requires cannot be determined for some reason, then PROCP returns T.

(ENCLOSE fnrep envrep)

ENCLOSE takes two s-expressions, one representing the code for a procedure, and the other representing the (lexical) environment in which the procedure is to run. ENCLOSE returns a (closed) procedure which can be invoked.

The representation of the code is the standard s-expression description (a lambda-expression). The representation of the environment is an association list (a-list) of the traditional kind:

```
((var1 . value1) (var2 . value2) ...)
```

NIL represents the global lexical environment.

This description of ENCLOSE should not be construed as describing how the implementation of SCHEME represents either environment or code internally. Indeed, ENCLOSE could be as simple as CONS, or as complicated as a compiler. All that ENCLOSE guarantees to do is to compute a procedure object given a description of its desired behavior. The description must be in the prescribed form; but the result may be in any form convenient to the implementation, as long as it satisfies the predicate PROCP {Note EVALUATE Has Disappeared} {Note S-expressions Are Not Functions}.

As an example, we can write APPLY using ENCLOSE. One way is to generate a lot of names for the arguments involved:

```
(DEFINE APPLY
        (LAMBDA (FN ARGS)
                (LET ((VARS (AMAPCAR (LAMBDA (X) (GENSYM)) ARGS))
                      (FNVAR (GENSYM)))
                     ((ENCLOSE "(LAMBDA () (,FNVAR @VARS))
                               (CONS (CONS FNVAR FN)
                                     (AMAPCAR CONS VARS ARGS)))))))
```

Here a procedure which will call the procedure FN on the required number of arguments is enclosed in an environment with all the variables bound to the appropriate values. For those who don't like GENSYM, here is another way to do it:

```
(DEFINE APPLY
        (LAMBDA (FN ARGS)
                (DO ((TAIL 'A "(CDR ,TAIL))
                     (REFS NIL (CONS "(CAR ,TAIL) REFS))
                     (COUNT ARGS (CDR COUNT)))
                    ((NULL COUNT)
                     ((ENCLOSE "(LAMBDA (F A) (F @(REVERSE REFS))) NIL)
                      FN ARGS)))))
```

In this version we create a series of forms (CAR A), (CAR (CDR A)), (CAR (CDR (CDR A))), ... to be used to access the arguments. (In a way, these are distinct names for the arguments in the same way that the gensyms were for the first version.) The values FN and ARGS are passed in as arguments to the enclosed procedure, rather than giving a non-NIL environment representation to ENCLOSE.

As another example, we define a function called *LAMBDA:

```
(DEFINE (*LAMBDA VARS BODY)
        (ENCLOSE "(LAMBDA ,VARS ,BODY) NIL))
```

Writing (*LAMBDA '(X Y) '(FOO Y X)) is just like writing (LAMBDA (X Y) (FOO Y X)). However, if there are any free variables in the supplied body, then *LAMBDA will cause them to refer to the global environment, not the current one. We cannot in general simulate LAMBDA by using *LAMBDA, because SCHEME (purposefully) does not provide a general way to get a representation of the current environment. We could, of course, require the user to give *LAMBDA a representation of the current environment, but this hardly seems worthwhile.

The following primitive functions allow for multiprocessing. We do not pretend that they are "right" in any sense, and are not particularly attached to these specific definitions. They are not part of the kernel of SCHEME. (Their primary use in practice is for bootstrapping SCHEME by creating an initial process for the top-level user interface loop.)

There are no primitives for process synchronization, as we have no good theory of how best to do this. However, in the PDP-10 MacLISP implementation of SCHEME we guarantee that SUBRs and LSUBRs execute in an uninterruptible fashion; that is, such functions can be considered "atomic" for synchronization purposes. The user is invited to exploit this fact to invent his own synchronization primitives {Note EVALUATE!UNINTERRUPTIBLY Has Disappeared}.

(CREATE!PROCESS proc)

This is the process generator for multiprocessing. It takes one argument, a procedure of no arguments. If the procedure ever terminates, the entire 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. When started, the procedure will be invoked (with no arguments), and the process will run "in parallel" with all other active processes.

(START!PROCESS procid)

This takes one argument, a process id, and starts up or resumes that process, which then runs.

(STOP!PROCESS procid)

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 global variable **PROCESS** always contains the process id of the currently running process; thus a process can stop itself by doing (STOP!PROCESS **PROCESS**).

(TERMINATE)

This primitive stops and kills the process which invokes it. The process may not be resumed by START!PROCESS. Some other process is selected to run. If the last process is terminated, SCHEME automatically prints a warning message, and then creates a new process running the standard SCHEME "read-eval-print" (actually "read—stick (LAMBDA () .) around—enclose in top-level environment—invoke—print") loop.

An example of the use of TERMINATE:

```
(TERMINATE)
```

Notes

{Note Notes Are in Alphabetical Order}

{ASET Has Disappeared}

The more general primitive ASET described in [SCHEME] has been removed from the SCHEME language. Although the case of a general evaluated expression for the variable name causes no real semantic difficulty (it can be viewed as a syntactic extension representing a large CASE statement, as pointed out in [Declarative]), it can be confusing to the reader. Moreover, in two years we have not found a use for it. Therefore we have replaced ASET with ASET', which requires the name of the variable to be modified to appear manifestly.

We confess to being "cute" when we say that the name of the primitive is ASET'. We have not changed the implementation at all, but merely require that the first argument be quoted. The form (ASET' FOO BAR) is parsed by the MacLISP reader as (ASET (QUOTE FOO) BAR). Of course, a different implementation of SCHEME might actually take ASET' as a single name. We apologize for this nonsense.

{BLOCK Exploits Applicative Order}

The definition shown for BLOCK exploits the applicative order of evaluation of SCHEME to perform this {Note Normal Order Loses}. It does not depend on left-to-right evaluation of arguments to functions! Notice also that in

```
(BLOCK x . r) → ((LAMBDA (A B) (B)) x (LAMBDA () (BLOCK . r)))
```

there can be no conflict between the auxiliary variables A and B and any variables occurring in x and r. It is thus unnecessary to choose variables different from any others appearing in the code. In this respect this definition is an improvement over the one given in [Imperative]. This trick (which is actually a deep property of the lexical scoping rules) is used in a general way in most of the definitions of syntactic extensions: one wraps all the "user code" in lambda-expressions in the outer environment, passes them in bound to internal names, and then invokes them as necessary within the internal code for the definition.

{Environment Symmetry}

One may think of an escape object as being "closed" with respect to a dynamic environment (and here we mean not only fluid variables but the chain of pending procedure calls) in much the same way that an ordinary procedure is closed with respect to a lexical environment. Just as a procedure cannot execute properly except in conjunction with a static environment of the appropriate form, so an escape object cannot properly resume control except in a dynamic environment of the appropriate form.

{EVALUATE Has Disappeared}

The EVALUATE primitive described in [SCHEME] has been removed from the language. We discovered (the hard way) that the straightforward implementation of EVALUATE (evaluate the given expression in the current environment) destroys referential transparency. We then altered it to evaluate the expression in the top-level environment, but were still disturbed by the extent to which one is tied to a particular representation of a procedure to be executed.

We eventually invented an ENCLOSE of one argument (a lambda-expression), which enclosed the procedure in the top-level environment. This allowed one to remove the dependence on representation by making a procedure, and then to pass the procedure around for a while before invoking it. We had no provision for closing in an arbitrary environment, because we did not want to provide the user with direct access to environments as data objects. The excellent idea of allowing ENCLOSE to accept a representation of an environment was suggested to us by R. M. Fano.

{EVALUATE!UNINTERRUPTIBLY Has Disappeared}

The EVALUATE!UNINTERRUPTIBLY primitive described in [SCHEME] has been removed from the language. This primitive was half a joke, and we have since discovered that it had a serious flaw in its definition, namely that the scope of the uninterruptibility is lexical. This worked in our limited examples only by virtue of the fact that SUBRs were atomic operations. In general, this primitive is worse than useless for synchronization purposes. Synchronization is clearly a dynamic and not a static phenomenon. We have no good theory of synchronization (primitives for this were included in [SCHEME] primarily to show that it could be done, however kludgily), and so have defined no replacement for EVALUATE!UNINTERRUPTIBLY. We apologize for any confusion this mistake may have caused.

{FEXPRs Are Okay by Us}

While the syntactic extensions are defined in terms of other constructs, they need not be implemented in terms of them. For example, in the current PDP-10 MacLISP implementation of SCHEME, BLOCK is actually implemented in the same way IF and QUOTE are, rather than as a macro in terms of LAMBDA. This was done purely to speed up the interpreter. The compiler still uses the macro definition (though we could change that too if warranted). The point is that the user doesn't have to know about all this.

It is somewhat an accident that magic forms look like procedure calls (see also {Note FUNCALL is a Pain}): a name appearing in the car of a list may represent either a procedure or a magic word, but not both. (We could, for example, say that magic forms are distinguished by a magic word in the cadr of a list, thus allowing forms such as (FOO := (+ FOO 1)), where := is a magic word for assignment. PLASMA [Smith and Hewitt] allowed just this ability with its "italics" or "reserved word" feature.) Thanks to this accident many LISP interpreters store the magic function definition in the place where an ordinary procedure definition is stored. A special marker (traditionally EXPR/SUBR or FEXPR/FSUBR) distinguishes ordinary functions from magic ones. This allows the lookup for a magic word definition and an ordinary value to be simultaneous, thus speeding up the implementation; it is purely an engineering trick and not a semantic essence. However, this trick has led to a generalization wherein QUOTE and COND are regarded as functions on an equal basis with CAR and CONS; to be sure, they take their arguments in a funny way — unevaluated — but they are still regarded as functions. This leads to all manner of confusion, which has its roots in a confusion between a procedure and its representation.

It is helpful to consider a simple thought experiment. Let us postulate a toy language called "Number-LISP". Programs in this language are written as s-expressions, as usual; the kernel primitives LAMBDA, IF, LABELS, etc. are all present. However, the primitive functions CONS, CAR, and CDR are absent; one has only +, -, *, //, and =. QUOTE is not available; the only constants one can write are numbers.

Now Number-LISP can be used to perform all kinds of arithmetic, but it is clearly a poor language in which to write a LISP interpreter. Now consider the magic form processors and syntactic extension functions for Number-LISP. They are procedures on s-expressions or functions from s-expressions to s-expressions which transform one form into another. Whatever processes IF or LABELS or BLOCK is clearly not a Number-LISP procedure, because it must deal with the text of a Number-LISP procedure, not just the data to be operated on by that procedure. The IF-processor (a "FEXPR") for Number-LISP must be coded in the meta-language for Number-LISP, whatever that may be.

Now it is one of the great features of ordinary LISP that it can serve as its own meta-language. This provides great power, but also permits great confusion. If the implementation allows mixing of levels of

definition, we must keep them separate in our minds. For this reason we don't mind using the "FEXPR hack" to implement syntactic forms, but we do mind thinking of them as functions just like EXPRs.

{FUNCALL is a Pain}

The ambiguity between magic forms and combinations could be eliminated by reserving a special subclass of lists to represent combinations, and allowing all others to represent magic forms. For example, we might say that all lists beginning with the atom CALL are combinations. Then we would write (CALL CONS A B) rather than (CONS A B). One could then have a procedure named LAMBDA, for example; there could be no confusion between (LAMBDA (A) (B A)) and (CALL LAMBDA (CALL A) (CALL B A)), as there would be between (LAMBDA (A) (B A)) as a combination and as a magic form denoting a procedure. Notice that CALL is intended to be merely a syntactic marker, like LAMBDA or IF, and not a function as FUNCALL is in MacLISP [Moon].

If this CALL convention were adopted, there could be no confusion between combinations and other kinds of forms. Not all expressions would have meaningful interpretations; for example (FOO A B) would not mean anything (certainly not a call to the function FOO, which would be written as (CALL FOO A B)). The space of meaningful s-expressions would be a very sparse subset of all s-expressions, rather than a dense one. It would also make writing SCHEME code very clumsy. (These two facts are of course correlated.) Combinations occur about as often as all other non-atomic forms put together; we would like to write as little as possible to denote a call. As in traditional LISP, we agree to tolerate the ambiguity in SCHEME as the price of notational convenience. Indeed, this ambiguity is sometimes exploited; it is convenient not to have to know whether AMAPCAR is a function or a magic word.

This compromise does lead to difficulties, however. For example, we had wanted to define an iteration feature:

```
(LOOP name varspecs body)
```

Unfortunately, there is a great deal of existing code written in SCHEME of the form:

```
(LABELS ((LOOP (LAMBDA ... (LOOP ...) ...)))
        (LOOP ...))
```

because LOOP has become a standard name for use in a LABELS procedure which implements an iteration (see, for example, our definition of DO!). If LOOP were to become a new magic word, then all this existing code would no longer work. We were therefore forced to name it ITERATE instead (after verifying that no existing code used the name ITERATE for another purpose!).

There would have been no problem if all this code had been written as:

```
(LABELS ((LOOP (LAMBDA ... (CALL LOOP ...) ...)))
        (CALL LOOP ...))
```

To this extent SCHEME has unfortunately, despite our best intentions, inherited a certain amount of referential opacity.

{Global Fluid Environment}

There is a question as to the meaning of the global fluid environment. In the PDP-10 MacLISP implementation of SCHEME, the global lexical and fluid environments coincide, but this was an arbitrary choice of convenience influenced by the structure of MacLISP. We recommend that the two global environments be kept distinct.

{IF Is Data-Dependent}

We should note that the usefulness of the definition of IF explicitly depends on the particular kinds of data types and the particular primitive functions available; we expect to use IF with primitive predicates such as ATOM and EQ. This is in contrast to other kernel forms such as LAMBDA and LABELS expressions, whose semantics are independent of the data.

We erred in [SCHEME] when we stated that a practical interpreter must have a little of each of call-by-value and call-by-name in it. The argument was roughly that a call-by-name interpreter must become call-by-value when a primitive operator is to be applied, and a call-by-value interpreter must have some primitive conditional such as IF. We did mention the trick of eliminating IF in a call-by-name interpreter by defining predicates to return (LAMBDA (X Y) X) for TRUE and (LAMBDA (X Y) Y) for FALSE, whereupon one typically writes:

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

but noted that it depends critically on the use of normal order evaluation.

What we had not fully understood at that point was the trick of simulating call-by-name in terms of call-by-value by using lambda-expressions (our use of it in the TRY!TWO!THINGS!IN!PARALLEL example notwithstanding!); this trick was described generally in [Imperative]. A special case of this trick is to define the primitive predicates in a call-by-value interpreter to return (LAMBDA (X Y) (X)) for TRUE and (LAMBDA (X Y) (Y)) for FALSE. Then one can write things like:

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

and so eliminate a call-by-name-like magic form such as IF. One can make the dependence of the conditional on the primitive data operations even more explicit by defining predicates not to return any particular value, but to require two "continuations" as arguments, of which it will invoke one:

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

We were correct when we said that a practical interpreter must have call-by-name to some extent in that there must be some way to designate two pieces of as yet uninterpreted program text of which only one is to be evaluated. We simply did not notice that LAMBDA provides this ability, and so a separate primitive such as IF is not necessary. We have chosen to retain IF in the language because it is traditional, because its implementation is easy to understand, and because it allows us to take advantage of many existing predicates in the host language in the PDP-10 MacLISP implementation.

{LISP BNF}

These rules refer to the following rules for LISP s-expressions:

```
<atomic symbol> ::= <alphanumeric string> <letter> <alphanumeric string>
<alphanumeric string> ::= <empty> | <alphanumeric character> <alphanumeric string>
<alphanumeric character> ::= <letter> | <digit> | / <character>
<letter> ::= A | B | ... | Y | Z | * | $ | % | ...
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> ::= <VERY implementation dependent>
<string> ::= " <character string> "
<character string> ::= <empty> | <literal character> <character string>
<literal character> ::= <any character except " or /> | / <character>
```

(In the PDP-10 MacLISP implementation of SCHEME, we use the " character for another purpose because PDP-10 MacLISP does not have a string data type. We mention strings only as a familiar example other than numbers of an atomic data type other than identifiers.)

In addition, we assume the usual interchangeability of list notation and dot notation for s-expressions: (A B C) = (A . (B . (C . NIL))). Thus the pattern (<magic word> . <s-expression>) may match the list (COND (A B) (T C)). It is the s-expression representation we care about, not particular

character strings.

{LISP Is a Ball of Mud}

LISP is extensible in two ways. First, there is the simple ability to define new functions; these are used in a way that is both syntactically and semantically identical to built-in primitive functions like CAR. This is in contrast to "algebraic" programming languages, which either make an arbitrary syntactic distinction between addition, say, and a user function, or wander into the quagmire of extensible parsers. Second, there is a uniform macro facility for transforming one syntactic form into another; this facility is based on internal data structures rather than external character-string syntax.

Joel Moses (private communication) once made some remarks on the difference between LISP and APL, which we paraphrase here: "APL is like a diamond. It has a beautiful crystal structure; all of its parts are related in a uniform and elegant way. But if you try to extend this structure in any way — even by adding another diamond — you get an ugly kludge. LISP, on the other hand, is like a ball of mud. You can add any amount of mud to it (e.g. MICRO-PLANNER or CONNIVER) and it still looks like a ball of mud!"

{Multiple Throw}

A full implementation of SCHEME allows this to work even if the CATCH has already been "returned from"; that is, the escape object can be used to "return from the catch" several times. However, we allow the possibility that an implementation may allow the escape object to be invoked only from within the execution of the form inside the catch. (This is essentially the restriction that MacLISP makes on its CATCH construct [Moon].) This restriction permits a stack discipline for the allocation of continuations, and also greatly simplifies the flow analysis problem for a compiler [RABBIT].

{Normal Order Loses}

Our definition of BLOCK exploits the fact that SCHEME is an applicative-order (call-by-value) language in order to enforce sequencing. Sussman has proved that one cannot do a similar thing in a normal-order (call-by-name) language:

Theorem: Normal order, as such, is incapable of enforcing sequencing (whereas applicative order is) in the form of the BLOCK construct.
(Informal) proof: The essence of (BLOCK a b) is that a is evaluated before b, and that the value of b is the value of the BLOCK (or, more correctly, the value or meaning of the BLOCK is independent of a; a is executed only for its side effects). Now we know that if (BLOCK a b) has any value at all, it can be found by using normal order (normal order terminates if any order does). Now suppose that the computation of a does not terminate, but the computation of b does. Then (BLOCK a b) must terminate under normal order, because the value of the block is the value of b; but this contradicts the requirement that a finish before b is calculated. QED

This is an informal indication that normal order is less useful (or at least less powerful) in a programming language than applicative order. We also noted in [SCHEME] that normal order iterations tend to consume more space that applicative order iterations, because of the buildup of thunk structure. Given that one can simulate normal order in applicative order by explicitly creating closures [Imperative], there seems to be little to recommend normal order over applicative order in a practical programming language.

{Notes Are in Alphabetical Order}

The notes are ordered alphabetically by name, not in order of reference within the text.

{S-expressions Are Not Functions}

Recall that a lambda-expression (i.e. an s-expression whose car is the atomic symbol LAMBDA) is not itself a valid procedure. It is necessary to ENCLOSE it in order to invoke it.

Moreover, the particular representations we have chosen for procedures and environments are arbitrary. In principle, one could have several kinds of ENCLOSE, each transforming instances of a particular representation into procedures. For example, someone might actually want to implement a primitive ALGOL-ENCLOSE, taking a string and a 2-by-N array representing code and environment for an ALGOL procedure:

```
(ALGOL-ENCLOSE "integer procedure fact(n); value n; integer n;
                fact := if n=0 then 1 else n*fact(n-1)"
               NULL-ARRAY)
```

This could return a factorial function completely acceptable to SCHEME. Of course, the implementation of the primitive ALGOL-ENCLOSE would have to know about the internal representations of procedures used by the implementation of SCHEME; but this is hidden from the user.

Similarly, one could have APL-ENCLOSE, BASIC-ENCLOSE, COBOL-ENCLOSE, FORTRAN-ENCLOSE, RPG-ENCLOSE ...

{Tail-Recursive AND}

The definition of AND has three rules, not only for the same reasons OR does, but because AND is not a precise dual to OR. OR, on failure, returns NIL, but AND does not just return non-NIL on success. It must return the non-NIL thing returned by its last form.

{Tail-Recursive OR}

We might have defined OR with only two rules:

```
(OR) → 'NIL
(OR x . r) → (COND (x) (T (OR . r)))
```

However, because of the way OR is sometimes used, it is a technical convenience to be able to guarantee to the user that the last form in an OR is evaluated without an extra "stack frame"; that is, a function called as the last form in an OR will be invoked tail-recursively. For example:

```
(DEFINE NOT-ALL-NIL-P
        (LAMBDA (X)
                (LABELS ((LOOP
                          (LAMBDA (Z)
                                  (OR (CAR Z) (LOOP (CDR Z))))))
                        (LOOP X))))
```

executes iteratively in SCHEME, but would not execute iteratively if the two-rule definition of OR were used.

{What Use Is It?}

We should perhaps say instead that <identifier> is treated the same as (STATIC <identifier>). The STATIC construction is included in SCHEME primarily for pedagogical purposes, to provide symmetry to (FLUID <identifier>). The fact that lone atomic symbols are interpreted as lexical

variables rather than dynamic ones is in some sense arbitrary. Some critics of SCHEME [personal communications] have expressed a certain horror that there are two kinds of variables, perhaps imagining some confusion in the interpretation of simple identifiers. We can have as many kinds of variables as we like (though we have so far discovered only two kinds of any great use), as long as we can distinguish them. In SCHEME we distinguish them with a special marker, such as STATIC or FLUID; then, as a convenience, we prescribe that simple atomic symbols, not marked by such a keyword, shall also be interpreted as lexical variables, because that is the kind we use most often in SCHEME. We could as easily have defined simple symbols to be interpreted as fluid variables, or for that matter as constants (as numbers and strings are). We could also have prescribed a different method of distinguishing types, e.g. "all variables beginning with I, J, K, L, M, or N shall be fluid". (This is not as silly as it sounds. A fairly wide-spread LISP convention is to spell global variables with leading and trailing *, as in *FOO*, and some programmers have wished that the compiler would automatically treat variables so spelled as SPECIAL.) Indeed, given the read-macro-character facility, we effectively have the syntactic rule "all variables beginning with • shall be fluid". We have settled on the current definition of SCHEME as being the most convenient both to implement and to use.

Compare the use of syntactic markers and read-macro-characters to the constructions <GVAL X> = ,X = "global value of X" and <LVAL X> = .X = "local value of X" in MUDDLE [Galley and Pfister]. Indeed, in MUDDLE a simple atomic symbol is regarded as a constant, not as an identifier.

All this suggests another solution to the problem posed in {Note FUNCALL is a Pain} (the confusion of magic forms with combinations). The real problem is distinguishing a magic word from a variable. Let us abbreviate (STATIC FOO) to ≡FOO, just as (FLUID FOO) can be abbreviated as •FOO. Then (≡LOOP A B) would have to be a call on the function LOOP, and not a magic form. Similarly, we could write (MAGICWORD COND) instead of COND, and invent an abbreviation for that too. This all raises as many problems as it solves by becoming too clumsy; but then again, maybe it isn't asking too much to require the user to write all magic words in boldface (as in the ALGOL reference language) or in italics (as in an early version of PLASMA [Smith and Hewitt])...

Acknowledgements

Comments by Carl Hewitt and Berthold Horn were of considerable value in preparing this paper. Ed Barton (who is also helping to maintain the PDP-10 MacLISP SCHEME implementation) made important contributions to the revisions of the language definition.

References

[Declarative]
Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

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

[Hewitt]
Hewitt, Carl. "Viewing Control Structures as Patterns of Passing Messages." AI Journal 8, 3 (June 1977), 323-364.

[Imperative]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

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

[McDermott and Sussman]
McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER Reference Manual. AI Memo 295a. MIT AI Lab (Cambridge, January 1974).

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

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

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

[RABBIT]
Steele, Guy Lewis Jr. Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto. S.M. thesis. MIT (Cambridge, May 1977).

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

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

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