MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 453 May 1978

The Art of the Interpreter or, The Modularity Complex (Parts Zero, One, and Two)

by

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Abstract:

We examine the effects of various language design decisions on the programming styles available to a user of the language, with particular emphasis on the ability to incrementally construct modular systems. At each step we exhibit an interactive meta-circular interpreter for the language under consideration. Each new interpreter is the result of an incremental change to a previous interpreter.

We explore the consequences of various variable binding disciplines and the introduction of side effects. We find that dynamic scoping is unsuitable for constructing procedural abstractions, but has another role as an agent of modularity, being a structured form of side effect. More general side effects are also found to be necessary to promote modular style. We find that the notion of side effect and the notion of equality (object identity) are mutually constraining; to define one is to define the other.

The interpreters we exhibit are all written in a simple dialect of LISP, and all implement LISP-like languages. A subset of these interpreters constitute a partial historical reconstruction of the actual evolution of LISP.

Keywords: abstraction, actors, applicative order, bindings, control structures, debugging, dynamic scoping, environments, fluid variables, FUNARG problem, functional objects, interactive programming, lambda-calculus, lexical scoping, LISP, modularity, procedural data, recursion equations, referential transparency, SCHEME, side effects, static scoping, structured programming

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

** Jolly Good Fellow

© Massachusetts Institute of Technology 1978

Contents

Introduction 1
Modularity 1
LISP-like Languages 2
Structure of the Paper 2
Part Zero — LISP and Interpreters 4
Recursion Equations 4
An Interpreter for LISP Recursion Equations 7
Part One — Variable Scoping Disciplines 13
Procedures as Data 13
Local Procedures 18
Lexical Scoping 21
Top Levels versus Referential Transparency 25
Part Two — State 31
Decomposition of State 31
Side Effects and Local State 32
Side Effects in the Interpreter 34
Equipotency of SETQ and RPLACA 38
Side Effects and Equality 39
Dynamic Scoping as a State-Decomposition Discipline 43
Summary 51
Acknowledgements 52
Notes 53
{Can George do Better?} 53
{Debugging} 53
{Driver Loop with Side Effects} 54
{EVALQUOTE} 55
{Gaussian} 56
{LABELS} 57
{LABELS with Side Effects}* 59
{Primitive Operators} 60
{PROGN Wizardry} 61
{QUOTE Mapping} 63
{QUOTE Shafts the Compiler} 63
{RPLACA Can Alter CAR Instead} 64
{S-expression Postulates and Notation} 65
{This ain't A-lists} 66
{Value Quibble} 67
{Weber} 68
{Y-operator} 70
References 71

Introduction

Modularity

The entities constructed by programming are extremely complex. Accurate construction of large programs would be impossible without specific techniques for controlling this complexity. Most such techniques are based on finding ways to decompose a problem into almost independently solvable subproblems, allowing a programmer to concentrate on one subproblem at a time, ignoring the others. When the subproblems are solved, the programmer must be able to combine the solutions with a minimum of unanticipated interactions. To the extent that a decomposition succeeds in breaking a programming problem into manageable pieces, we say that the resulting program is modular; each part of the solution is called a module. Well-designed programming languages provide features which support the construction of modular programs.

One decomposition strategy is the packaging of common patterns of the use of a language. For example, in Algol a *for* loop captures a common pattern of *if* and *goto* statements. Packages of common patterns are not necessarily merely abbreviations to save typing. While a simple abbreviation has little abstraction power because a user must know what the abbreviation expands into, a good package encapsulates a higher level concept which has meaning independent of its implementation. Once a package is constructed the programmer can use it directly, without regard for the details it contains, precisely because it corresponds to a single notion he uses in dealing with the programming problem.

A package is most useful if its behavior is independent of the context of its use, thus reducing possible interference with other packages. Such a package is called *referentially transparent*. Intuitively, referential transparency requires that the meanings of parts of a program be apparent and not change, so that such meanings can be reliably depended upon. In particular, names internal to one module should not affect or be affected by other modules — the external behavior of a module should be independent of the choice of names for its local identifiers.

To make a modular program, it is often necessary to think of a computational process as having state. In such cases, if the state can be naturally divided into independent parts, an important decomposition may be the division of the program into pieces which separately deal with the parts of the state.

We will discuss various stylistic techniques for achieving modularity. One would expect these techniques to complement each other. We will instead discover that they can come into conflict. Pushing one to an extreme in a language can seriously compromise others.

LISP-like Languages

Of the hundreds or thousands of computer languages which have been invented, there is one particular family of languages whose common ancestor was the original LISP, developed by McCarthy and others in the late 1950's. [LISP History] These languages are generally characterized by a simple, fully parenthesized ("Cambridge Polish") syntax; the ability to manipulate general, linked-list data structures; a standard representation for programs of the language in terms of these structures; and an interactive programming system based on an interpreter for the standard representation. Examples of such languages are LISP 1.5 [LISP 1.5M], MacLISP [Moon], InterLISP [Teitelman], CONNIVER [McDermott and Sussman], QA4 [Rulifson], PLASMA [Smith and Hewitt] [Hewitt and Smith], and SCHEME [SCHEME] [Revised Report]. We will call this family the LISP-like languages.

The various members of this family differ in some interesting and often subtle ways. These differences have a profound impact on the styles of programming each may encourage or support. We will explore some of these differences by examining a series of small ("toy") evaluators which exhibit these differences without the clutter of "extra features" provided in real, production versions of LISP-like language systems.

The series of evaluators to be considered partially constitute a reconstruction of what we believe to be the paths along which the family evolved. These paths can be explained after the fact by viewing the historical changes to the language as being guided by the requirements of various aspects of modularity.

Structure of the Paper

Our discussion is divided into several parts, which form a linear progression. In addition, there are numerous large digressions which explore interesting side developments. These digressions are placed at the end as notes, cross-referenced to and from the text.

We exhibit a large number of LISP interpreters whose code differs from one to another in small ways (though their behavior differs greatly!). In order to avoid writing identical pieces of code over and over, each figure exhibits only routines which differ, and also contains cross-references to preceding figures from which missing routines for that figure are to be drawn.

Part Zero introduces the restricted dialect of the LISP language in which most of our examples are written. It also discusses the basic structure of an interpreter, and exhibits a meta-circular interpreter for the language.

Part One introduces procedural data as an abstraction mechanism, and considers its impact on variable scoping disciplines in the language. We are forced through a series of such disciplines as unexpected interactions are uncovered and fixed. Interpreters are exhibited for dynamic scoping and lexical scoping.

Part Two considers the problems associated with the decomposition

of state. Side effects are introduced as a mechanism for effecting such decompositions. We find that the notion of side effect is inextricably wound up with the notion of identity. Dynamic scoping is retrospectively viewed as a restricted kind of side effect.

With this we summarize and conclude with many tantalizing questions yet unanswered.

In Part Three (in a separate paper) we will find that the introduction of side effects forces the issue of the order of evaluation of expressions. We will contrast call-by-name and its variants with call-by-value, and discuss how these control disciplines arise as a consequence of different models of packaging. In particular, call-by-name arises naturally from the syntactic nature of the Algol 60 copy rule. As before, many little interpreters for these disciplines will be exhibited.

In Part Four we will be led to generalize the notion of a syntactic package. We will discuss meta-procedures, which deal with the representations of procedures. The distinction between a procedure and its representation will be more carefully considered. Macro processors, algebraic simplifiers, and compilers will be considered as meta-procedures. Various interpreters, compilers, and simplifiers will be exhibited.

Part Zero

LISP and Interpreters

Recursion Equations

Contrary to popular belief, LISP was not originally derived from Church's λ-calculus [Church] [LISP History]. The earliest LISP did not have a well-defined notion of free variables or procedural objects. Early LISP programs were similar to recursion equations, defining functions on symbolic expressions ("S-expressions"). They differed from the equations of pure recursive function theory [Kleene] by introducing the conditional expression construction (often called the "McCarthy conditional"), to avoid "pattern-directed invocation". That is, in recursive function theory one would define the factorial function by the following two equations:

```
factorial(0) = 1
factorial(successor(x)) = successor(x) * factorial(x)
```

In early LISP, however, one would have written:

```
factorial[x] = [x=0 → 1; T → x*factorial[x-1]]
```

where "[a → b; T → c]" essentially means "if a then b else c". The recursive function theory version depends on selecting which of two equations to use by matching the argument to the left-hand sides (such a discipline is actually used in the PROLOG language [Warren]); the early LISP version represents this decision as a conditional expression.

The theory of recursion equations deals with functions over the natural numbers. In LISP, however, one is interested in being able to manipulate algebraic expressions, programs, and other symbolic expressions as data structures. While such expressions can be encoded as numbers (using the technique of "arithmetization" developed by Kurt Gödel), such an encoding is not very convenient. Instead, a new kind of data called "S-expressions" (for "symbolic expressions") is introduced specifically to provide convenient encodings. S-expressions can be defined by a set of formal inductive axioms analogous to the Peano postulates used to define natural numbers. Here we will give only an informal and incomplete definition of S-expressions; for a more complete description, see {Note S-expression Postulates and Notation}.

For our purposes we will need only the special cases of S-expressions called atoms and lists. An atom is an "indivisible" data object, which we denote by writing a string of letters and digits; if only digits are used, then the atom is considered to be a number. Many special characters such as "-" and "+" are considered to be letters; we will see below that it is not necessary to specially reserve them for use as operator symbols. A list is a (possibly empty) sequence of S-expressions, notated by writing the S-expressions in order, between a set of parentheses

and separated by spaces. A list of the atoms "FOO", "43", and "BAR" would be written "(FOO 43 BAR)". Notice that the definition of a list is recursive. For example,

```
(DEFINE (SECOND X) (CAR (CDR X)))
```

is a list of three things: the atomic symbol DEFINE, a list of the two atomic symbols SECOND and X, and another list of two other things.

We can use S-expressions to represent algebraic expressions by using "Cambridge Polish" notation, essentially a parenthesized version of prefix Polish notation. Numeric constants are encoded as numeric atoms; variables are encoded as non-numeric atoms (which henceforth we will call atomic symbols); and procedure invocations are encoded as lists, where the first element of the list represents the procedure and the rest represent the arguments. For example, the algebraic expression "a*b+c*d" can be represented as "(+ (* a b) (* c d))". Notice that LISP does not need the usual precedence rules concerning whether multiplication or addition is performed first; the parentheses explicitly define the order. Also, all procedure invocations have a uniform syntax, no matter how many arguments are involved. Infix, superscript, and subscript notations are not used; thus the expression "$J_p(x^2+1)$" would be written "(J p (+ (↑ x 2) 1))".

To encode a conditional expression

$$[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \ldots\ ;\ p_N \rightarrow e_N]$$

(which means to evaluate the predicates p_j in order until a true one is found, at which point the value of e_j is taken to be the value of the conditional) we write the S-expression

(COND (p_1 e_1) (p_2 e_2) ... (p_n e_n))

We can now encode sets of LISP recursion equations as S-expressions. For the equation

```
factorial[x] = [x=0 → 1; T → x*factorial[x-1]]
```

we write the S-expression

```
(DEFINE (FACTORIAL X)
        (COND ((= X 0) 1)
              (T (* X (FACTORIAL (- X 1))))))
```

(We could also have written

```
(DEFINE (FACTORIAL X) (COND ((=
X 0) 1) (T (* X (FACTORIAL (- X
1))))))
```

but we conventionally lay out S-expressions so that they are easy to read.)

We now have a complete encoding for algebraic expressions and LISP recursion equations in the form of S-expressions. Suppose that we now want to write a LISP program which will take such an S-expression and perform some useful operation on it, such as determining the value of an algebraic expression. We need some procedures for distinguishing, decomposing, and constructing S-expressions.

The predicate ATOM, when applied to an S-expression, produces true when given an atom and false otherwise. The empty list is considered to be an atom. The predicate NULL is true of only the empty list; its argument need not be a list, but may be any S-expression. The predicate NUMBERP is true of numbers and false of atomic symbols and lists. The predicate EQ, when applied to two atomic symbols, is true if the two atomic symbols are identical. It is false when applied to an atomic symbol and any other S-expression. (We have not defined EQ on two lists yet; this will not become important, or even meaningful, until we discuss side effects.)

The decomposition operators for lists are traditionally called CAR and CDR for historical reasons. [LISP History] CAR extracts the first element of a list, while CDR produces a list containing all elements but the first. Because compositions of CAR and CDR are commonly used in LISP, an abbreviation is provided: all the C's and R's in the middle can be squeezed out. For example, "(CDR (CDR (CAR (CDR X))))" can be written as "(CDDADR X)".

The construction operator CONS, given an S-expression and a list, produces a new list whose car is the S-expression and whose cdr is the list. The operator LIST can take any number of arguments (a special feature), and produces a list of its arguments.

We can now write some interesting programs in LISP to deal with S-expressions. For example, we can write a predicate EQUAL, which determines whether two S-expressions have the same CAR-CDR structure:

```
(DEFINE (EQUAL X Y)
        (COND ((NUMBERP X)
               (COND ((NUMBERP Y) (= X Y))
                     (T NIL)))
              ((ATOM X) (EQ X Y))
              ((ATOM Y) NIL)
              ((EQUAL (CAR X) (CAR Y))
               (EQUAL (CDR X) (CDR Y)))))
```

Here we have used the standard names T and NIL to represent true and false. (Traditionally NIL is also considered to be the empty list, but we will avoid this here, writing "()" for the empty list.)

Because LISP programs are represented as LISP data structures (S-expressions), there is a difficulty with representing constants. For example, suppose we want to determine whether or not the value of the variable X is the atomic symbol "FOO". We might try writing:

```
(EQ X FOO)
```

This doesn't work. The occurrence of "FOO" does not refer to the atomic symbol FOO as a constant; it is treated as a variable, just as "X" is.

The essential problem is that we want to be able to write any S-expression as a constant in a program, but some S-expressions must be used to represent other things, such as variables and procedure invocations. To solve this problem we invent a new notation: (QUOTE x) in a program represents the constant S-expression x. (Note QUOTE Mapping) Thus we can write our test as "(EQ X (QUOTE FOO))". Similarly,

```
(EQUAL X (LIST Y Z))
```

constructs a list from the values of Y and Z, and compares the result to the value of X, while

```
(EQUAL X (QUOTE (LIST Y Z)))
```

compares the value of X to the constant S-expression "(LIST Y Z)". Because the QUOTE construction is used so frequently in LISP, we use an abbreviated notation: "'FOO" is equivalent to "(QUOTE FOO)". This is only a notational convenience; the two notations denote the same S-expression. (S-expressions are not character strings, but data objects with a certain structure. We use character strings to notate S-expressions on paper, but we can use other notations as well, such as little boxes and arrows. We can and do allow several different character strings to denote the same S-expression.)

An Interpreter for LISP Recursion Equations

We now have enough machinery to begin our examination of the genetic history of LISP. We first present a complete interpreter for LISP recursion equations. The language interpreted is a dialect of LISP which allows no free variables except for names of primitive or defined procedures, and no definitions of procedures within other procedures.

The driver loop reads in definitions of procedures of the form:

```
(DEFINE (F A B C ...) <expression in A B C ... and F G H ...>)
```

and saves them. It can also read in requests to apply some defined procedure to some arguments (or, more generally, to evaluate any expression), in which case it prints the resulting value. An expression may consist of variable references, constants (numbers and quoted s-expressions), procedure calls, and conditional expressions (COND). The defined procedures may refer to each other and to initially supplied primitive procedures (such as CAR, CONS, etc.). Definitions may contain "forward references", as long as all necessary definitions are present at the time of a request for a computation. The interpreter itself is presented here as a set of such definitions, and so is meta-circular.

The language is intended to be evaluated in applicative order;

that is, all arguments to a procedure are fully evaluated before an attempt is made to apply the procedure to the arguments. (It is necessary to state this explicitly here, as it is not inherent in the form of the meta-circular definition. See [Reynolds] for an explication of this problem.)

The driver loop (see Figure 1) is conceptually started by a request to invoke DRIVER with no arguments. Its task is to first print the message "LITHP ITH LITHTENING" (a tradition of sorts) and then invoke DRIVER-LOOP. The expression <THE-PRIMITIVE-PROCEDURES> is intended to represent a constant list structure, containing definitions of primitive procedures, to be supplied to DRIVER-LOOP.

```
(DEFINE (DRIVER)
        (DRIVER-LOOP <THE-PRIMITIVE-PROCEDURES> (PRINT '|LITHP ITH LITHTENING|)))

(DEFINE (DRIVER-LOOP PROCEDURES HUNOZ)
        (DRIVER-LOOP-1 PROCEDURES (READ)))

(DEFINE (DRIVER-LOOP-1 PROCEDURES FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP PROCEDURES (PRINT (EVAL FORM '() PROCEDURES))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP (BIND (LIST (CAADR FORM))
                                  (LIST (LIST (CDADR FORM) (CADDR FORM)))
                                  PROCEDURES)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP PROCEDURES (PRINT (EVAL FORM '() PROCEDURES))))))
```

Figure 1
Top Level Driver Loop for a Recursion Equations Interpreter

DRIVER-LOOP reads an S-expression from the input stream and passes it, along with the current procedure definitions, to DRIVER-LOOP-1. This procedure in turn determines whether the input S-expression is a definition. If it is, then it uses BIND (described below) to produce an augmented set of procedure definitions, prints the name of the defined procedure, and calls DRIVER-LOOP to repeat the process. The augmented set of procedures is passed to DRIVER-LOOP, and so the variable PROCEDURES always contains all the accumulated definitions ever read. If the input S-expression is not a definition, then it is given to the evaluator EVAL, whose purpose is to determine the values of expressions. {Note Value Quibble} The set of currently defined procedures is also passed to EVAL.

The process carried on by the driver loop is often called the "top level"; all user programs and requests are run "under" it. The growing set of procedure definitions is called the "top-level environment"; this environment changes in the course of the user interaction, and contains the state of the machine as perceived by the user. It is within this environment that user programs are executed.

```
(DEFINE (EVAL EXP ENV PROCEDURES)
        (COND ((ATOM EXP)
               (COND ((EQ EXP 'NIL) 'NIL)
                     ((EQ EXP 'T) 'T)
                     ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV PROCEDURES))
              (T (APPLY (VALUE (CAR EXP) PROCEDURES)
                        (EVLIS (CDR EXP) ENV PROCEDURES)
                        PROCEDURES))))

(DEFINE (APPLY FUN ARGS PROCEDURES)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              (T (EVAL (CADR FUN)
                       (BIND (CAR FUN) ARGS '())
                       PROCEDURES))))

(DEFINE (EVCOND CLAUSES ENV PROCEDURES)
        (COND ((NULL CLAUSES) (ERROR))
              ((EVAL (CAAR CLAUSES) ENV PROCEDURES)
               (EVAL (CADAR CLAUSES) ENV PROCEDURES))
              (T (EVCOND (CDR CLAUSES) ENV PROCEDURES))))

(DEFINE (EVLIS ARGLIST ENV PROCEDURES)
        (COND ((NULL ARGLIST) '())
              (T (CONS (EVAL (CAR ARGLIST) ENV PROCEDURES)
                       (EVLIS (CDR ARGLIST) ENV PROCEDURES)))))
```

Figure 2
Evaluator for a Recursion Equations Interpreter

The evaluator proper (see Figure 2) is divided into two conceptual components: EVAL and APPLY. EVAL classifies expressions and directs their evaluation. Simple expressions (such as constants and variables) can be evaluated directly. For the complex case of procedure invocations (technically called "combinations"), EVAL looks up the procedure definition, recursively evaluates the arguments (using EVLIS), and then calls APPLY. APPLY classifies procedures and directs their application. Simple procedures (primitive operators) are applied directly. For the complex case of user-defined procedures, APPLY uses BIND to build an environment, a kind of symbol table, associating the formal parameters from the procedure definition with the actual argument values provided by EVAL. The body of the procedure definition is then passed to EVAL, along with the environment just constructed, which is used to determine the values of

variables occurring in the body.

In more detail, EVAL is a case analysis on the structure of the S-expression EXP. If it is an atom, there are several subcases. The special atoms T and NIL are defined to evaluate to T and NIL (this is strictly for convenience, because they are used as truth values). Similarly, for convenience numeric atoms evaluate to themselves. (These cases could be eliminated by requiring the user to write lots of QUOTE forms: 'T, 'NIL, '43, etc. This would have been quite inconvenient in early LISP, before the "'" notation had been introduced; one would have had to write (QUOTE 43), etc.) Atomic symbols, however, encode variables; the value associated with that symbol is extracted from the environment ENV using the function VALUE (see below).

If the expression to be evaluated is not atomic, then it may be a QUOTE form, a COND form, or a combination. For a QUOTE form, EVAL extracts the S-expression constant using CADR. Conditionals are handled by EVCOND, which calls EVAL on a predicate expression; if the predicate is true, EVCOND evaluates the corresponding result expression (by calling EVAL, of course); if the predicate is false, EVCOND calls itself to test the predicate of the next clause of the COND body. For combinations, the procedure is obtained, the arguments evaluated (using EVLIS), and APPLY called as described earlier. Notice that VALUE is used to get the procedure definition from the set PROCEDURES; we can do this because, as an engineering trick, we arrange for ENV and PROCEDURES to have the same structure, because they are both symbol tables.

EVLIS is a simple recursive function which calls EVAL on successive arguments in ARGLIST and produces a list of the values in order.

APPLY distinguishes two kinds of procedures: primitive and user-defined. For now we avoid describing the precise implementation of primitive procedures by assuming the existence of a predicate PRIMOP which is true only of primitive procedures, and a function PRIMOP-APPLY which deals with the application of such primitive procedures. (See {Note Primitive Operators} for the details of a possible implementation of PRIMOP and PRIMOP-APPLY.) We consider primitive procedures to be a kind of atomic S-expression other than numbers and atomic symbols; we define no particular written notation for them here. However, primitive procedures are not to be confused with the atomic symbols used as their names. The result of (VALUE 'CAR PROCEDURES) is not the atomic symbol CAR, but rather some bizarre object which is meaningful only to PRIMOP-APPLY.

User-defined procedures are represented here as lists. These lists are constructed by DRIVER-LOOP-1. The car of the list is the list of formal parameters, and the cadr is the body of the definition.

```
(DEFINE (BIND VARS ARGS ENV)
        (COND ((= (LENGTH VARS) (LENGTH ARGS))
               (CONS (CONS VARS ARGS) ENV))
              (T (ERROR))))

(DEFINE (VALUE NAME ENV)
        (VALUE1 NAME (LOOKUP NAME ENV)))

(DEFINE (VALUE1 NAME SLOT)
        (COND ((EQ SLOT '&UNBOUND) (ERROR))
              (T (CAR SLOT))))

(DEFINE (LOOKUP NAME ENV)
        (COND ((NULL ENV) '&UNBOUND)
              (T (LOOKUP1 NAME (CAAR ENV) (CDAR ENV) ENV))))

(DEFINE (LOOKUP1 NAME VARS VALS ENV)
        (COND ((NULL VARS) (LOOKUP NAME (CDR ENV)))
              ((EQ NAME (CAR VARS)) VALS)
              (T (LOOKUP1 NAME (CDR VARS) (CDR VALS) ENV))))
```

Figure 3
Utility Routines for Maintaining Environments

The interpreter uses several utility procedures for maintaining symbol tables (see Figure 3). A symbol table is represented as a list of buckets; each bucket is a list whose car is a list of names and whose cdr is a list of corresponding values. (Note This ain't A-lists) If a variable name occurs in more than one bucket, the leftmost such bucket has priority; in this way new symbol definitions added to the front of the list can supersede old ones.

BIND takes a list of names, a list of values, and a symbol table, and produces a new symbol table which is the old one augmented by an extra bucket containing the new set of associations. (It also performs a useful error check — LENGTH returns the length of a list.)

VALUE is essentially an interface to LOOKUP. We define it because later, in Part Three, we will want to use different versions of VALUE1 without changing the underlying algorithm in LOOKUP. The check for &UNBOUND catches incorrect references to undefined variables.

LOOKUP takes a name and a symbol table, and returns that portion of a bucket whose car is the associated value. (This definition will be more useful later than one in which the value itself is returned.)

Note carefully the use of the variable PROCEDURES in the interpreter. When DRIVER-LOOP-1 calls EVAL it passes the current list of defined procedures (both primitive and user-defined). DRIVER-LOOP-1 is the only routine which augments the value of PROCEDURES, and this value is only

used in EVAL, when it is passed to VALUE. However, all of the routines APPLY, EVCOND, and EVLIS have to know about PROCEDURES, and dutifully pass it along so that it may be eventually used by EVAL. The set of definitions must be passed along because there is no provision for free variables or side effects; there is no way to have "memory" or "state" other than in passed variables. The absence of free variables effectively causes our language to be referentially transparent. However, we sense a disturbing lack of modularity in the use of PROCEDURES (and, to a lesser extent, in the use of ENV — look at EVCOND and EVLIS). We will return to this point later.

Our recursion equations language has no special iteration or looping constructs, such as the Algol for statement or the FORTRAN DO loop. All loops are constructed by arranging for recursive procedures to call themselves or each other. For example, EVCOND (see Figure 2) iterates over the clauses of a COND by calling itself on successive "tails" of the list of clauses. Now such recursive calls may strike the reader familiar with other languages (such as Algol, FORTRAN, PL/I, etc.) on an intuitive level as being rather inefficient for implementing real programs. Even granted that calls might be made fast, they would seem to consume space in the form of return addresses and other control information. Examination of the recursion equations evaluator will show, however, that this phenomenon does not have to occur. This is because no extra information is saved if there is nothing left to do on return from a recursive call. See [SCHEME] and [Debunking] for a more thorough discussion of this.

Part One

Variable Scoping Disciplines

Procedures as Data

The simple LISP described in Part Zero can be a pleasant medium for encoding rather complex algorithms, including those of symbolic mathematics. Often lists are used for representing such structures as the set of coefficients of a polynomial or coordinates of a space vector. Many problems require one to perform an operation on each element of a list and produce a new list of the results. For example, it may be useful to make a list of the squares of each of the elements in a vector. We would write this as follows:

```
(DEFINE (SQUARELIST L)
        (COND ((NULL L) '())
              (T (CONS (SQUARE (CAR L))
                       (SQUARELIST (CDR L))))))
```

We find ourselves writing this pattern over and over again:

```
(DEFINE (fLIST L)
        (COND ((NULL L) '())
              (T (CONS (f (CAR L))
                       (fLIST (CDR L))))))
```

where f is a function defined on the elements of our list. It would be nice to be able to define an entity of the programming language which would capture this abstract pattern. The "obvious" solution is to write the variable function as a functional variable which can be accepted as an argument:

```
(DEFINE (MAPCAR F L)
        (COND ((NULL L) '())
              (T (CONS (F (CAR L))
                       (MAPCAR F (CDR L))))))
```

(MAPCAR is the traditional name of this abstraction.) Using this we could say:

```
(MAPCAR SQUARE '(1 2 3))
```

Unfortunately, this will not work in our recursion equations interpreter. Why not?

The essence of the problem is that our interpreter segregates procedures from other kinds of objects. We refer to F as a procedure but it was passed in as a variable. Procedures are only looked up in the

PROCEDURES symbol table, but variables are bound in ENV. Moreover, in the call to MAPCAR, SQUARE is used as a variable, which is looked up in ENV, but its definition is only available in PROCEDURES.

Let's merge the two symbol tables... How could that hurt?

```
(DEFINE (DRIVER-LOOP-1 ENV FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP (BIND (LIST (CAADR FORM))
                                  (LIST (LIST '&PROCEDURE (CDADR FORM) (CADDR FORM)))
                                  ENV)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))))
```

For DRIVER-LOOP see Figure 1.
For EVAL see Figure 5.
For BIND see Figure 3.

Figure 4
Modified Driver Loop for Treating Procedures as Objects

We will eliminate PROCEDURES, and use ENV to contain both procedures and other objects. The driver loop requires no particular changes (see Figure 4), except for eliminating the argument '() in the calls to EVAL. We will change the name PROCEDURES to ENV throughout as well, but of course that isn't logically necessary, because our language is referentially transparent. (Snicker!) {Note EVALQUOTE}

(We have introduced a funny object &PROCEDURE which we use to flag procedural objects. In the previous interpreter it was impossible for the user to request application of an object which was not either a primitive operator or a procedure produced by a DEFINE form. Now that procedures mingle freely with other data objects, it is desirable to be able to distinguish them, e.g. for error checking in APPLY. We also have some deeper motivations having to do with avoiding the confusion of a procedure with its textual representation, but we do not want to deal with this issue yet.)

To fix up the evaluator, we eliminate all occurrences of PROCEDURES. In EVAL, where the name of a procedure in a combination is looked up, we change it to perform the lookup in ENV. Finally, there is a problem in APPLY: if the call to EVAL to evaluate the body is simply

```
(EVAL (CADDR FUN)
      (BIND (CADR FUN) ARGS '()))
```

then the new ENV given to EVAL does not have the procedure definitions in it. Moreover, APPLY does not even have access to an environment which

contains the procedure definitions (because its parameter PROCEDURES was deleted)! We can easily fix this. When APPLY is called from EVAL, ENV can be passed along (as PROCEDURES used to be), and the call to EVAL from APPLY can be changed to

```
(EVAL (CADDR FUN)
      (BIND (CADR FUN) ARGS ENV))
```

In this way the environment passed to EVAL will contain the new variable bindings added to the old environment containing the procedure definitions. (See Figure 5.) This is indeed a good characteristic: if the name of a defined procedure is used as a local variable (procedural or otherwise), the new binding takes precedence locally, temporarily superseding the global definition.

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              (T (APPLY (VALUE (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)
                        ENV)))))

(DEFINE (APPLY FUN ARGS ENV)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              ((EQ (CAR FUN) '&PROCEDURE)
               (EVAL (CADDR FUN)
                     (BIND (CADR FUN) ARGS ENV)))
              (T (ERROR))))

(DEFINE (EVCOND CLAUSES ENV)
        (COND ((NULL CLAUSES) (ERROR))
              ((EVAL (CAAR CLAUSES) ENV)
               (EVAL (CADAR CLAUSES) ENV))
              (T (EVCOND (CDR CLAUSES) ENV))))

(DEFINE (EVLIS ARGLIST ENV)
        (COND ((NULL ARGLIST) '())
              (T (CONS (EVAL (CAR ARGLIST) ENV)
                       (EVLIS (CDR ARGLIST) ENV)))))
```

For VALUE and BIND see Figure 3.

Figure 5
Evaluator for Treating Procedures as Objects

Another good thing about this version of the interpreter is that the gross non-modularity of the scattered occurrences of PROCEDURES has disappeared. The problem has not been solved, of course, but we certainly feel relieved that the particular manifestation has been removed!

By the way, we also eliminated the explicit tests for T and NIL in EVAL, assuming that we can simply put their initial values in the initial environment provided by DRIVER.

An interesting property of this interpreter is that free variables now have been given a meaning, though we originally did not intend this. Indeed, in the original recursion equations interpreter, there were free variables in a sense: all procedural variables were free (but they could be used only in operator position in a combination). In our new

interpreter, thanks to the merging of the procedural and variable environments, we may have not only bound procedure names, but also free variable names, for after all the two kinds of names are now one.

This interpreter differs in only small details from the one in LISP 1.5 [LISP 1.5M]. Both have dynamically scoped free variables (we will elaborate on this point later). We might note that the reference to VALUE in EVAL when computing the first argument for APPLY can be replaced by a reference to EVAL; this does the same thing if a variable appears in the operator position of a combination, and allows the additional general ability to use any expression to compute the procedure. This difference in fact appears in the LISP 1.5 interpreter. There are other slight differences, such as the representation of primitive operators and the handling of procedures which are not primitive or user-defined. Aside from these, the greatest difference between our interpreter and LISP 1.5's is the use of lambda notation. This we will meet in the next section.

Local Procedures

We now have the ability to define and use the MAPCAR procedure. After some more experience in programming, however, we find that, having abstracted the common pattern from our loops, that the remaining part (the functional argument) tends to be different for each invocation of MAPCAR. Unfortunately, our language for all practical purposes requires that we use a name to refer to the functional arguments, because the only way we have to denote new procedures is to DEFINE names for them. We soon tire of thinking up new unique names for trivial procedures:

```
(DEFINE (FOOBAR-43 X) (* (+ X 4) 3))

... (MAPCAR FOOBAR-43 L)
```

We run the risk of name conflicts; also, it would be nice to be able to write the procedure definition at the single point of use.

More abstractly, given that procedures have become referenceable objects in the language, it would be nice to have a notation for them as objects, or rather a way to write an S-expression in code that would evaluate to a procedure. LISP [LISP 1M] adapted such a notation from the λ-calculus of Alonzo Church [Church]:

```
(LAMBDA <variables> <body>)
```

Comparing this with the DEFINE notation, we see that it has the same parts: a keyword so that it can be recognized; a list of parameters; and a body. The only difference is the omission of an irrelevant name. It is just the right thing.

Given this, we can simply write

```
(MAPCAR (LAMBDA (X) (* X X)) L)
```

rather than having to define SQUARE as a separate procedure. An additional benefit is that this notation makes it very easy for a compiler to examine this code and produce an efficient iterative implementation, because all the relevant code is present locally (assuming the compiler knows about MAPCAR).

Installing this notation requires only a two-line change in EVAL (see Figure 6).

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              ((EQ (CAR EXP) 'LAMBDA)
               (CONS '&PROCEDURE (CDR EXP)))
              (T (APPLY (EVAL (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)
                        ENV))))
```

For VALUE see Figure 3.
For APPLY, EVCOND, and EVLIS see Figure 5.

Figure 6
Evaluator for LAMBDA-notation (Dynamically Scoped)

(The reader might have noticed that all EVAL does for a LAMBDA-expression is replace the word LAMBDA with the word &PROCEDURE, and that we could avoid that work by uniformly using LAMBDA instead of &PROCEDURE as the flag for a procedural object. Given then that EVAL on a LAMBDA-expression is an identity operation, we can eliminate the handling of LAMBDA in EVAL merely by requiring the user to write '(LAMBDA ...) instead of (LAMBDA ...). Although the implementors of most LISPs have in fact done just this ever since LISP 1, it is a very bad idea. EVAL is supposed to process expressions and produce their values, and the fact that it might be implemented as an identity operation is no business of the user. The confusion between a procedural object and an expression having that object as its value will lead to serious trouble. (Imagine confusing 15 with (+ 7 8), and trying to take the car of the former instead of the latter, or trying to add 3 to the latter instead of the former!) The quoted LAMBDA-expression engineering trick discourages the implementation of a referentially transparent LISP. In Part Four we will see the extreme difficulties for a LISP compiler (or other program-understander) caused by the blatant destruction of referential transparency. {Note QUOTE Shafts the Compiler})

The ability to use free variables and local procedures gives us additional freedom to express interesting procedures. For example, we can define a procedure SCALE which multiplies a vector of arbitrary length by a scalar. If the vector is represented as a list of components, then we can use MAPCAR and a local procedure with a free variable:

```
(DEFINE (SCALE S V)
        (MAPCAR (LAMBDA (X) (* X S))
                V))
```

Everything would be just peachy keen, except for one small glitch. Suppose that the programmer who wrote SCALE for some reason chose the name L rather than S to represent the scalar:

```
(DEFINE (SCALE L V)
        (MAPCAR (LAMBDA (X) (* X L))
                V))
```

Although the version with S works, the version with L does not work. This happens because MAPCAR also uses the name L for one of its arguments (that is, a "local" variable). The reference to L in the LAMBDA-expression in SCALE refers to the L bound in MAPCAR and not to the one bound by SCALE. In general, free variable references in one procedure refer to the bindings of variables in other procedures higher up in the chain of calls. This discipline is called dynamic scoping of variables, because the connection between binding and reference is established dynamically, changing as different procedures are executed.

That the behavior of the SCALE program depends on the choice of names for its local variables is a violation of referential transparency. The modularity of the MAPCAR abstraction has been destroyed, because no one can use that abstraction without understanding the details of its implementation. This is the famous "FUNARG problem" [Moses] [LISP History].

If we are to avoid such conflicts between different uses of the same name, we must arrange our language so that the choice of names locally cannot have global repercussions. More specifically, we must have the ability to bind a variable in such a way that it will have a truly local meaning (though in general we might not want all variables to be strictly local — we will consider later the possibility of having several types of variables).

Lexical Scoping

We now construct an interpreter in which all variables have strictly local usage. This discipline is called lexical scoping of variables, and has been used in many programming languages, including Algol 60 [Naur]. The term "lexical" refers to the fact that all references to a local variable binding are textually apparent in the program. The term static binding is also used, indicating that the connection between binding and reference is unchanging at run time.

The difficulty in SCALE is that the body of the LAMBDA-expression (* X L) is evaluated using the ENV which was available to EVAL (and so passed to APPLY) when it was working on the body of MAPCAR. But we want the (* X L) to be evaluated using the ENV which was available when the body of SCALE was being evaluated. Somehow we must arrange for this environment to be available for evaluating (* X L).

The correct environment was available at the time the LAMBDA-expression was evaluated to produce a &PROCEDURE-object. Why not just tack the environment at that point onto the end of the &PROCEDURE-object so that it can be used when the procedure is applied?

This is in fact the right thing to do. The object we want to give to MAPCAR must be not just the text describing the computation to be performed, but also the meanings of the free variables referenced in that text. Only the combination of the two can correctly specify the computation which reflects the complete meaning of the abstract function to be mapped. This is the first place where we find it crucial to distinguish the three ideas: (1) The program — the text describing a procedure, e.g. in the form of an S-expression; (2) The procedure which is executed by the computer; and (3) The mathematical function or other conceptual operation computed by the execution of the procedure.

To install lexical scoping in our interpreter, we must change the treatment of LAMBDA-expressions in EVAL to make the current environment ENV part of the &PROCEDURE-object. We say that the procedure is closed in the current environment, and the &PROCEDURE-object is therefore called a closure of the procedure, or a closed procedure. We must also change APPLY to bind the new variable-value associations onto the environment in the &PROCEDURE-object, rather than onto that passed by EVAL. When we have done this, we see that in fact the environment passed by EVAL is not used, so we can eliminate the parameter ENV from the definition of APPLY, and change the invocation of APPLY that occurs in EVAL. Thus, while the handling of LAMBDA-expressions has become more complicated, the handling of ENV has been correspondingly simplified. (See Figure 7.)

Had we previously adopted the trick described in the preceding section, wherein the user was required to write '(LAMBDA ...) rather than (LAMBDA ...), it would have been more difficult to adjust the interpreter to accommodate lexical scoping — it would have involved a large change rather than a small tweak. (The change from dynamic scoping to lexical scoping does involve a gross change of programming style, and this is undoubtedly why, once dynamic scoping had historically become the standard discipline, the quotation problem was never cleared up. We will see later that dynamic

scoping is a valuable technique for producing modularity, but we see no virtue at all in the confusion produced by quoted LAMBDA-expressions. While quoted LAMBDA-expressions do produce dynamic scoping, the support of dynamic scoping does not depend on the quotation of LAMBDA-expressions.)

While lexical scoping solves our problems of referential transparency, we will see later that we must in turn pay a large price for it — but it is <u>not</u> a price of run-time efficiency (contrary to popular belief)!

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              (T (APPLY (EVAL (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)))))

(DEFINE (APPLY FUN ARGS)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              ((EQ (CAR FUN) '&PROCEDURE)
               (EVAL (CADDR FUN)
                     (BIND (CADR FUN) ARGS (CADDDR FUN))))
              (T (ERROR))))
```

For VALUE and BIND see Figure 3.
For EVCOND and EVLIS see Figure 5.

Figure 7
Evaluator for Lexically Scoped LAMBDA-notation

Let's see what we have bought. One thing we can do is generalize MAPCAR. After yet more programming experience we find that we write many MAPCAR-like procedures. For example, we might need a kind of MAPCAR where the function F always returns a list, and we want to produce not a list of the lists, but the concatenation of the lists. We might also want to take the sum or the product of all the numbers in a list, or the sum of the cars of all elements in a list. The general pattern is that we look at each element of a list, do something to it, and then somehow combine the results of all these elementwise operations. Another application might be to check for duplicates in a list; for each element we want to see whether another copy follows it in the list. We further generalize the pattern to look at successive trailing segments of the list; we can always take the car to

get a single element.

We could simply add more procedural parameters to MAPCAR:

```
(DEFINE (MAP F OP ID L)
    (COND ((NULL L) ID)
          (T (OP (F L)
                 (MAP F OP ID (CDR L))))))
```

Using this, we can make a copy of the list L:

```
(MAP CAR CONS '() L)
```

We can simulate (MAPCAR F L):

```
(MAP (LAMBDA (X) (F (CAR X))) CONS '() L)
```

Indeed, we can write:

```
(DEFINE (MAPCAR F L)
        (MAP (LAMBDA (X) (F (CAR X))) CONS '() L))
```

We can sum the elements of L:

```
(MAP CAR + 0 L)
```

We can take the product of the elements of L:

```
(MAP CAR * 1 L)
```

We can count the pairs of duplicate elements of L:

```
(MAP (LAMBDA (X) X)
     (LAMBDA (Y N) (COND ((MEMBER (CAR Y) (CDR Y))
                          (+ N 1))
                         (T N)))
     0
     L)
```

If we have occasion to take the sum over lots of lists in different places, we might want to package the operation "sum over list" — we get awfully tired of writing "CAR + 0". We can write:

```
(DEFINE (MAPGEN F OP ID)
        (LAMBDA (L) (MAP F OP ID L)))
```

The result of (MAPGEN CAR + 0) we might call SUM — it is a procedure of one argument which will sum the elements of a list. The reason we wrote a procedure to construct SUM, rather than just writing:

```
(DEFINE (SUM L)
        (MAP CAR + 0 L))
```

is that MAPGEN serves as a generalized constructor of such procedures, thus capturing an interesting abstraction — we might call the result of (MAPGEN CAR * 1), for example, PRODUCT, and so on.

What is interesting about this is that we can write procedures which construct other procedures. This is not to be confused with the ability to construct S-expression representations of procedures; that ability is shared by all of the interpreters we have examined. The ability to construct procedures was not available in the dynamically scoped interpreter. In solving the violation of referential transparency we seem to have stumbled across a source of additional abstractive power. While the MAP example may seem strained, this example is quite natural: given a numerical function, to produce a new function which numerically approximates the derivative of the first.

```
(DEFINE (DERIVATIVE F ΔX)
        (LAMBDA (X)
                (/ (- (F (+ X ΔX))
                      (F X))
                   ΔX)))
```

Notice that this is not a symbolic process dealing with the representation of F. The DERIVATIVE procedure knows nothing about the internal structure of F. All it does is construct a new procedure which uses F only by invoking it. The program DERIVATIVE captures (in approximation) the abstraction of "derivative" as a mapping from the space of numerical (and reasonably well-behaved!) functions to itself.

The ability to define procedures which construct other procedures is powerful. We can use it to construct procedures which behave like data objects. For example, since the only constraints which CONS must (so far) obey are the algebraic identities:

(CAR (CONS α β)) = α and (CDR (CONS α β)) = β

the value of (CONS α β) can be thought of as a procedure which produces α or β on demand (cf. [Hewitt and Smith] [Fischer]). We can write this as follows:

```
(DEFINE (CONS A D)
        (LAMBDA (M)
                (COND ((= M 0) A)
                      ((= M 1) D))))

(DEFINE (CAR X) (X 0))

(DEFINE (CDR X) (X 1))
```

Here we have envisioned the value of (CONS α β) as a vector of two elements, with zero-origin indexing. However, this definition of CONS makes use of the primitive operator =. We can define the "primitive operators" CONS, CAR, and CDR without using another primitive operator at all! Following [Church], we write:

```
(DEFINE (CONS A D)
        (LAMBDA (M) (M A D)))

(DEFINE (CAR X)
        (X (LAMBDA (A D) A)))

(DEFINE (CDR X)
        (X (LAMBDA (A D) D)))
```

Rather than using 0 and 1 (i.e. data objects) as selectors, we instead use (LAMBDA (A D) A) and (LAMBDA (A D) D) (i.e. procedures).

We can think of the LAMBDA-expression which appears as the body of the definition of DERIVATIVE or of CONS as a prototype for new procedures. When DERIVATIVE or CONS is called, this prototype is instantiated as a closure, with certain variables free to the prototype bound to the arguments given to the constructor.

At this point it looks like we have solved all our problems. We started with a referentially transparent but expressively weak language. We augmented it with procedural objects and a notation for them in order to capture certain notions of abstraction and modularity. In doing this we lost the referential transparency. We have now regained it, and in the process uncovered even more powerful abstraction capabilities.

Top Levels versus Referential Transparency

"The Three Laws of Thermodynamics:
1. You can't win.
2. You can't break even.
3. You can't get out of the game."
— Unknown

There is no free lunch. We have ignored a necessary change to the top level driver loop. We have changed the format of &PROCEDURE-objects. DRIVER-LOOP-1 constructs &PROCEDURE-objects; it must be rewritten to accommodate the change. We must include an environment in each such object. The obvious fix is shown in Figure 8.

```
(DEFINE (DRIVER-LOOP-1 ENV FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP (BIND (LIST (CAADR FORM))
                                  (LIST (LIST '&PROCEDURE
                                              (CDADR FORM)
                                              (CADDR FORM)
                                              ENV))
                                  ENV)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))))
```

For DRIVER-LOOP see Figure 1.
For BIND see Figure 3.
For EVAL see Figure 7.

Figure 8
Modified Driver Loop for Lexically Scoped LAMBDA-notation

It doesn't work. This patch does put the finishing touch on the preservation of referential transparency. It does it so well, that each new definition can only refer to previously defined names! We have lost the ability to make forward references. We can't redefine a procedure which had a bug in it and expect old references to use the new definition. In fact, we cannot use DEFINE to make a recursive procedure. {Note Y-operator} The &PROCEDURE-object for each defined procedure contains an environment having only the previously defined procedures.

We are finally confronted with the fact that we have been seeking the impossible. We have tried to attain complete referential transparency (in the expectation that modularity would be enhanced), while trying also to retain the notion of an incremental, interactive top-level loop for reading definitions. But the very existence of such a top level inherently constitutes a violation of referential transparency. A piece of code can be read in which refers to an as yet undefined identifier (the name of a procedure, for example), and then later a definition for that identifier read in (thereby altering the meaning of the reference).

If we stubbornly insist on maintaining absolute referential transparency in our language, we are forced to eliminate the incremental top level loop. A program must be constructed monolithically. We must read in all our procedure definitions at once, close them all together, and then take one or more shots at running them. (This is the way many Algol implementations work; development of large systems can be very difficult if parts cannot be separately constructed and compiled.) We are forced to give up interactive debugging, because we cannot redefine erroneous procedures easily. We are forced to give up incremental compilation of separate modules.

We have thrown the baby out with the bath water. The very purpose of referential transparency is to permit programs to be divided into parts so that each part can be separately specified without a description of its implementation. The desirable result is that pieces can be separately written and debugged. {Note Debugging}

On the other hand, if we give up absolute referential transparency, we can fix the top level loop. The basic problem is that we really want procedures defined at top level to be able to refer to procedures defined later. The problem with pure lexical scoping is that the &PROCEDURE-objects are created too early, when the desired environment is not yet available. We must arrange for them to be constructed at a later time. We could simply use the environment in use by the caller at the time of invocation (reverting to dynamic scoping). But dynamic scoping would lose a great deal of referential transparency and abstractive power. Procedures must not be allowed to refer to variables internal to other procedures, but only to top-level variables existing at the time they are called. Therefore only the future top-level environment is to be included in the &PROCEDURE-object when it is eventually constructed. In this way free variable references will be dynamic only with respect to the top-level environment.

Considering our dynamically-scoped interpreter above (see Figure 5), we would be led to modify APPLY again, to combine the best properties of the dynamically and lexically scoped interpreters. Indeed, the two kinds of function can easily coexist. We borrow the code involving the passing of PROCEDURES (including the DRIVER-LOOP, modified to initialize ENV to PROCEDURES) from the recursion-equations interpreter (Figures 1 and 2), the code for using this top-level environment from the dynamically-scoped interpreter (Figure 5), and the code for constructing &PROCEDURE-objects for LAMBDA-expressions from the lexically-scoped interpreter (Figure 7). The result appears in Figure 9.

```
(DEFINE (EVAL EXP ENV PROCEDURES)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV PROCEDURES))
              (T (APPLY (EVAL (CAR EXP) ENV PROCEDURES)
                        (EVLIS (CDR EXP) ENV PROCEDURES)
                        PROCEDURES))))

(DEFINE (APPLY FUN ARGS PROCEDURES)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              ((EQ (CAR FUN) '&PROCEDURE)
               (EVAL (CADDR FUN)
                     (BIND (CADR FUN) ARGS (CADDDR FUN))
                     PROCEDURES))
              (T (EVAL (CADR FUN)
                       (BIND (CAR FUN) ARGS PROCEDURES)
                       PROCEDURES))))

(DEFINE (DRIVER-LOOP-1 PROCEDURES FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP PROCEDURES
                            (PRINT (EVAL FORM PROCEDURES PROCEDURES))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP (BIND (LIST (CAADR FORM))
                                  (LIST (LIST (CDADR FORM) (CADDR FORM)))
                                  PROCEDURES)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP PROCEDURES
                              (PRINT (EVAL FORM PROCEDURES PROCEDURES))))))
```

For DRIVER-LOOP see Figure 1.
For VALUE and BIND see Figure 3.
For EVCOND and EVLIS see Figure 2.

Figure 9
An Evaluator for Local Lexical Scoping
and Dynamic Top-Level References

Ugh bletch, PROCEDURES is back! Also, there are two kinds of user-defined procedural objects floating around. There happens to be another way to fix the top level, which yields additional flavor. We note that

during any one processing cycle of EVAL/APPLY, PROCEDURES remains constant. We can thus choose to associate the top level environment with a top-level procedure at a time earlier than invocation time in APPLY. We also note that LOOKUP1 will have its hands on the top-level environment anyway just before it locates the definition of a top-level procedure. Exploiting this idea yields an alternate solution. {Note LABELS}

In the new driver (see Figure 10) loop we no longer use BIND to augment the top-level environment whenever a new definition is made. We instead have all of the top-level definitions in one frame of the environment. When a new definition is to be made we extract the list of names and the list of values for the old definitions from the old environment and make a new top-level environment with the lists of names and values separately augmented.

Instead of creating &PROCEDURE-objects, this driver loop creates &LABELED-objects, which have the same format except that they contain no environment. A &LABELED-object is purely internal and can never be seen by a user program. When LOOKUP1 encounters such an object as the value of a variable, it immediately creates the corresponding &PROCEDURE-object, using the environment at hand, which turns out to be the top-level environment.

```
(DEFINE (DRIVER-LOOP-1 ENV FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP (LIST (CONS (CONS (CAADR FORM) (CAAR ENV))
                                        (CONS (LIST '&LABELED
                                                    (CDADR FORM)
                                                    (CADDR FORM))
                                              (CDAR ENV))))
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP ENV (PRINT (EVAL FORM ENV))))))

(DEFINE (LOOKUP1 NAME VARS VALS ENV)
        (COND ((NULL VARS)
               (LOOKUP NAME (CDR ENV)))
              ((EQ NAME (CAR VARS))
               (COND ((ATOM (CAR VALS)) VALS)
                     ((EQ (CAAR VALS) '&LABELED)
                      (LIST '&PROCEDURE (CADAR VALS) (CADDAR VALS) ENV))
                     (T VALS)))
              (T (LOOKUP1 NAME (CDR VARS) (CDR VALS) ENV))))
```

For DRIVER-LOOP see Figure 1.
For LOOKUP see Figure 3.
For EVAL see Figure 7.

Figure 10
An Alternative Solution for Local Lexical Scoping
and Dynamic Top-Level References
(Modified Top-Level Driver Loop and Environment Lookup)

Part Two

State

Decomposition of State

We saw in Part One that an interactive top-level loop necessarily violates referential transparency. We wish to deal with the computer as an entity with state, which changes over time by interacting with a user. In particular, we want the computer to change over time by accumulating procedure definitions.

Just as the user wishes to think of the computer as having state, he may find it conceptually convenient to organize a program similarly: one part may deal with another part having state. Often programs are written for the purpose of analyzing or simulating a physical system. If modules of the program are to reflect the conceptual divisions of the physical system, then the program modules may well need to have independent state variables. Thus the notion of state is not just a programming trick, but may be required by the nature of the problem domain.

A simpler example of the use of state involves the use of a pseudo-random number generator. A LISP version of one might be:

```
(DEFINE (RANDOM SEED)
        ((LAMBDA (Z)
                 (COND ((> Z 0) Z)
                       (T (+ Z -32768.))))
         (* SEED 899.)))
```

This version of RANDOM uses the power-residue method for a 16-bit two's-complement number representation; the value produced is a pseudo-random integer, and also is the seed for the next call. The caller of RANDOM is required to save this value and supply it on the next call to RANDOM.

This fact is unfortunate. The caller really has no interest in the workings of RANDOM, and would much prefer to simply call it as "(RANDOM)", for example, and get back a random number — because this would reflect most precisely the abstract notion of "random number generator". Such a generator would have to have state.

Suppose we are willing to live with this nuisance. Consider now building some larger program using RANDOM. Many levels up, the programmer who writes some high-level routine very likely does not care at all that a low-level routine uses RANDOM; he may not even know about the existence of that routine. However, if the state of the pseudo-random number generator is to be preserved, that programmer will have to deal with some state quantity he knows nothing about, for the sake of a program ten levels removed from his thinking. Just as PROCEDURES had to be passed all around for the sake of EVAL in Figure 2, so the state of RANDOM must be passed up and down and all around by programs which don't really care. This clearly violates our principle of modularity. (For an example of how bad this can

get, see {Note Gaussian}.)

As another example, suppose that George writes MAPCAR, and Harry uses it. Harry complains that MAPCAR is too slow. George then decides to collect some statistics about the use of MAPCAR, such as the number of times called, the average length of the second argument, and so on. He first writes an experimental MAPCAR to count number of calls:

```
(DEFINE (MAPCAR F L N)
        (CONS (OLDMAPCAR F L) (+ N 1)))

(DEFINE (OLDMAPCAR F L)
        (COND ((NULL L) '())
              (T (CONS (F (CAR L))
                       (OLDMAPCAR F (CDR L))))))
```

and asks Harry to use it for a while in his program. "I had to add an extra argument to keep track of the count," says George, "and in order to return both the result and the count, I had to cons them together. Please rewrite your program to keep track of the count and pass it on from one call on MAPCAR to the next." Harry's reply is "**unprintable**".

Now Bruce comes along and asks Harry how to use Harry's program. Harry says, "Just write (DIFFERENTIATE EXP VAR N), where EXP is the expression to be differentiated, VAR is the variable with respect to which to differentiate, and N is George's statistics counter — but that may go away next week." Bruce gives Harry a funny look, then goes away and writes his own DIFFERENTIATE, using George's documentation for the old MAPCAR, of course, unaware that the new one has been installed...

George's new MAPCAR conceptually has state. The state information should be local to the definition of MAPCAR, because that information is not anyone else's business, and George has no business requiring everyone else to keep track of it for him. George and Harry and Bruce all wish George had a way to maintain local state information in MAPCAR.

Side Effects and Local State

Traditionally local state is maintained through some sort of "side effect". We can always avoid the use of side effects if we are willing to pass all state variables around. As we have seen, this requires a monolithic conception of the program structure. If we wish to break a program up into independent modules, each with local state information, we must seek another method.

We claim that any such method effectively constitutes a side effect. If a module has hidden state, then its behavior can potentially change over time.

If only one module in the system has local state, then we can hide the side effect by making it the top-level module of the system, as we have done for DRIVER-LOOP. (For an example of this, see {Note Weber}.) If more than one module has state, however, then each may perceive changes in the

other's behavior. This the essence of side effect.

The concept of side effect is induced by particular choices of boundaries between parts of a larger system. If a system boundary encloses all processes of interest (the system is closed), we need no concept of side effect to describe that system as a whole in vacuo. If, however, we wish to make an abstraction by dividing the system into modules more than one of which has independent state, then we have by this action created the concept of side effect.

We are forced to introduce side effects as a technique for constructing modular systems. But side effects violate referential transparency by altering the meanings of expressions; we expect (+ 3 4) always to mean the same thing, but we cannot say the same for (+ 3 (RANDOM)). Two techniques for achieving modularity have come into direct conflict.

The most common form of side effect in programming languages is the assignment statement, which alters the meaning of a variable. LISP provides this notion in the SETQ construct:

```
(SETQ X 43)
```

returns 43, and as a side effect alters the meaning of X so that subsequent references will obtain 43 also.

With this, George can now write:

```
(DEFINE (MAPCAR F L)
        (MAPCAR1 F L (SETQ N (+ N 1))))

(DEFINE (MAPCAR1 F L HUNOZ)
        (OLDMAPCAR F L))
```

There are still some minor problems here. The function MAPCAR1 and the variable HUNOZ are used solely to throw away the value of the SETQ form. It is so common to use SETQ only for its side effect that another construction, PROGN, is very useful:

(PROGN e_1 e_1 ... e_N)

evaluates each of the forms e_j in order, throwing away the values of all but the last one. Notice that we specifically require them to be evaluated in order; this concept did not occur in the specification of our earlier interpreters, because it was not necessary in the absence of side effects. Similarly, it was not useful to be able to throw away values in the absence of side effects. (We did throw away a value in DRIVER-LOOP, but that was one which resulted from calling PRINT, which of course is assumed to have a side effect!) Using PROGN, George can write:

```
(DEFINE (MAPCAR F L)
        (PROGN (SETQ N (+ N 1))
               (OLDMAPCAR F L)))
```

There remains the problem of the global variable N, which Harry or Bruce might stumble across by accident. George has to have some handle to get at the statistics counter, and any handle George can use intentionally, Bruce and Harry can use accidentally. One thing that George can do is rename N to MAPCAR-STATISTICS-COUNTER, and warn Bruce and Harry not to use a global variable with that name. This is still better than the original situation — at least now Bruce and Harry need not change their programs, and it is George's responsibility to find a name which does not conflict. {Note Can George do better?}

In the case of RANDOM, where the state information is truly local in that no one wants to access it except its owner, we can combine the use of lexical scoping and of side effects to manipulate a completely hidden state variable. For example, suppose we want several independent pseudo-random number generators, initialized with different seeds. We can make a pseudo-random number generator generator as follows:

```
(DEFINE (RGEN SEED)
        (LAMBDA () (PROGN (SETQ SEED
                                ((LAMBDA (Z) (COND ((> Z 0) Z)
                                                   (T (+ Z -32768.))))
                                 (* SEED 899.)))
                          SEED)))
```

Each call to RGEN delivers as its value a new pseudo-random number generator which is an instance of the prototype described by the LAMBDA-expression which is the body of RGEN. Each one has a state variable which is its seed. The state of each instance is distinct from that of every other instance. This gives one the power of the own variables of ALGOL 60 without any additional mechanism.

Side Effects in the Interpreter

In order to write a simple interpreter which implements the side effect SETQ, we will postulate the existence of two side effect operators which alter S-expressions:

(RPLACA X Y) and (RPLACD X Y)

return the value of X (which must not be atomic), but as a side effect alters X so that its car or cdr, respectively, is the value of Y. (The introduction of operators which modify S-expressions causes a number of nasty problems, which we will consider presently.) We will use these operators to alter the structure of the environment ENV. We modify EVAL to recognize the SETQ construct (see Figure 11). On seeing "SETQ" in the "operator position" of the expression, EVAL dispatches to EVSETQ, after recursively evaluating the value to be assigned. EVSETQ uses LOOKUP to find the effective binding of the variable mentioned in the SETQ. If there is such a binding, RPLACA is used to change the value associated with the

variable. If there is no such binding, then the intent is to initialize a top-level variable; EV-TOP-LEVEL-SETQ locates the top-level environment (which is always at the end of any environment) and creates a new binding by altering the environment structure.

We also modify EVAL to recognize PROGN. EVPROGN is a tail-recursive loop which evaluates each subform of the PROGN form in turn, throwing away each value but the last. {Note PROGN Wizardry}

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'SETQ)
               (EVSETQ (CADR EXP) (EVAL (CADDR EXP) ENV) ENV))
              ((EQ (CAR EXP) 'PROGN)
               (EVPROGN (CDR EXP) ENV NIL))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              (T (APPLY (EVAL (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)))))

(DEFINE (EVSETQ VAR VAL ENV)
        ((LAMBDA (SLOT)
                 (COND ((EQ SLOT '&UNBOUND)
                        (EV-TOP-LEVEL-SETQ VAR VAL ENV))
                       (T (CAR (RPLACA SLOT VAL)))))
         (LOOKUP VAR ENV)))

(DEFINE (EV-TOP-LEVEL-SETQ VAR VAL ENV)
        (COND ((NULL (CDR ENV))
               (CADAR (RPLACA ENV
                              (CONS (CONS VAR (CAAR ENV))
                                    (CONS VAL (CDAR ENV))))))
              (T (EV-TOP-LEVEL-SETQ VAR VAL (CDR ENV)))))

(DEFINE (EVPROGN EXPS ENV HUNOZ)
        (COND ((NULL (CDR EXPS)) (EVAL (CAR EXPS) ENV))
              (T (EVPROGN (CDR EXPS) ENV (EVAL (CAR EXPS) ENV)))))
```

For VALUE, LOOKUP, and BIND see Figure 3.
For EVCOND and EVLIS see Figure 5.
For APPLY see Figure 7.
For LOOKUP1 see Figure 10 (not Figure 3).

Figure 11
Evaluator with User Side Effects (Assignment to Variables)

Because EVSETQ can be used to initialize new top-level variables, it is convenient for DRIVER-LOOP-1 to call EVSETQ when defining a new function (see Figure 12). Unlike the DRIVER-LOOP-1 of Figure 10, this one has no special knowledge about the structure of environments; as before, such knowledge is hidden in environment specialists such as BIND, VALUE, and now EVSETQ. (The value of EVSETQ is not used, but thrown away; we introduce an extra throwaway parameter into the definition of DRIVER-LOOP for this purpose.)

```
(DEFINE (DRIVER)
        (DRIVER-LOOP <THE-PRIMITIVE-PROCEDURES>
                     NIL
                     (PRINT '|LITHP ITH LITHTENING|)))

(DEFINE (DRIVER-LOOP ENV HUNOZ HUKAIRZ)
        (DRIVER-LOOP-1 ENV (READ)))

(DEFINE (DRIVER-LOOP-1 ENV FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP ENV NIL (PRINT (EVAL FORM ENV))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP ENV
                            (EVSETQ (CAADR FORM)
                                    (LIST '&LABELED
                                          (CDADR FORM)
                                          (CADDR FORM))
                                    ENV)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP ENV NIL (PRINT (EVAL FORM ENV))))))
```

For EVSETQ see Figure 11.

Figure 12
Driver Loop for Evaluator with User Side Effects
(Assignment to Variables)

(Once we have side effects, we don't really need the &LABELED device to permit incremental definition of recursive functions; we can just perform a side effect on the top-level environment. We left the &LABELED device in Figure 12 for continuity with the previous examples. "Real" LISP systems use the side effect method. See {Note Driver Loop with Side Effects}, and also {Note LABELS with Side Effects}.)

Equipotency of SETQ and RPLACA

We pulled a fast one when we introduced RPLACA and RPLACD for the sake of implementing SETQ (though we actually only used RPLACA). We used a side effect to define the implementation of side effects. While this makes a fine meta-circular description, it doesn't constitute a definition of side effects founded on the original meta-circular recursion equations interpreter.

We could implement an interpreter which would define a side effect without itself using side effects. Such a definition would encapsulate the entire state of the user's data structures into a single interpreter data structure which is passed around by a top-level loop. Constructing such an interpreter would involve turning a regular interpreter inside out (in much the same way GAUSSIAN was everted in {Note Weber}). This is extremely difficult and lengthy, and the module boundaries within the interpreter are so destroyed that the resulting interpreter is nearly impossible to understand. We will spare the reader the details.

We settle for a meta-circular description of side effects. Now that we have seen how to implement SETQ in terms of RPLACA and RPLACD, we can also do the reverse, completing the meta-circle (see Figure 13). We use the procedural version of CONS shown earlier, modified to provide two "setting procedures" SA and SD, which provide the ability to alter the car and cdr.

```
(DEFINE (CONS A D)
        (LAMBDA (M)
                (M A D (LAMBDA (Z) (SETQ A Z)) (LAMBDA (Z) (SETQ D Z)))))

(DEFINE (CAR X)
        (X (LAMBDA (A D SA SD) A))

(DEFINE (CDR X)
        (X (LAMBDA (A D SA SD) D))

(DEFINE (RPLACA X Y)
        (X (LAMBDA (A D SA SD)
                   (PROGN (SA Y) X))))

(DEFINE (RPLACD X Y)
        (X (LAMBDA (A D SA SD)
                   (PROGN (SD Y) X))))
```

Figure 13
Procedural ("Actors-like") Implementation of CONS and Friends

We originally introduced side effects such as SETQ to help us build modules such as RANDOM which have local state. Now, using the technique of

constructing procedures, we find that CONS can be viewed as a constructor of modules, just as MAPGEN was. CONS constructs modules ("cons cells") which use SETQ to maintain a local state.

Side Effects and Equality

> "Things are seldom what they seem,
> Skim milk masquerades as cream..."
> — Gilbert and Sullivan
> (H.M.S. Pinafore)

> "Plus ça change, plus c'est la même chose."
> — Alphonse Karr

Our descriptions of SETQ and RPLACA, both informal and meta-circular, are imprecise. They admit a number of drastically different interpretations of the behavior of the system. We would all agree that for RPLACA to mean anything at all like what we want, the expression:

```
((LAMBDA (X)
         (PROGN (RPLACA X 'Z)
                (CAR X)))
 (CONS 'A '(B C)))
```

Puzzle #1

should evaluate to Z. But what about this case:

```
((LAMBDA (X Y)
         (PROGN (RPLACA X 'Z)
                (CAR Y)))
 (CONS 'A '(B C))
 (CONS 'A '(B C)))
```

Puzzle #2

Should this evaluate to A or Z? Nearly all LISP systems would produce A, but there are arguments for both possibilities. Similarly, should this:

```
((LAMBDA (X)
         ((LAMBDA (U V)
                  (PROGN (RPLACA U 'Z)
                         (CAR V)))
          X X))
 (CONS 'A '(B C)))
```

Puzzle #3

evaluate to A or Z? Again there are arguments for both possibilities.

Before we can meaningfully consider these questions, we must have a more precise notion of what we mean by "RPLACA". Let us review its description:

> If X has as its value a non-atomic S-expression, and we evaluate the expression (RPLACA X Y), then after this evaluation, the value of the expression (CAR X) is Y.

This description depends upon a critical assumption. We have a notion of a thing which is the value of X, such that several references to the variable X all refer to the same thing. But what the ₹╬#◎ do we mean by "same"??

The concept of side effect is inseparable from the notion of equality/identity/sameness. The only way one can observationally determine that a side effect has occurred is when the same object behaves in two different ways at different times. {Note RPLACA Can Alter CAR Instead} Conversely, the only way one can determine that two objects are the same is to perform a side effect on one and look for an appropriate change in the behavior of the other.

In order to determine the answers to the Puzzles above, we must determine what properties are required of "sameness". There may be different points of view regarding sameness, which may lead to different answers to the Puzzles.

If we agree that the answer to Puzzle #1 is Z, then we have implicitly adopted the notion of consistency of variable reference, because we have referred to the variable X twice. As a property of the sameness predicate ≡, we write: (≡ X X). We can say that referring to a variable does not make a copy of its value (because if it did, the RPLACA in Puzzle #1 would have changed only a copy of the value of X, and (CAR X) would extract the car of a different copy, producing A).

Given this, and given that we accept the interpreter of Figure 11 and believe in its meta-circularity, we are forced to conclude that the answer to Puzzle #3 is also Z. We must consider all access paths and show that no copying can occur which would allow the answer to be A. The meta-circularity requires that any property of the interpreted language also hold for the text of the interpreter, and vice versa. The answer to Puzzle #1 requires that variable references not produce implicit copies, and so neither can variable references in the text of the interpreter.

(Consistent with this, our particular interpreter has no explicit code in LOOKUP which specifies copying.) The other place in Puzzle #3 where copying might occur is in the binding of U and V. Examining the text of our particular meta-circular interpreter shows that BIND also has no explicit code for copying. There remains the possibility that binding does implicitly copy in the text of the meta-circular interpreter; this would consistently cause copying in the bindings of the interpreted code, because ENV would be copied whenever bound in the text of the interpreter. This, however, would cause the answer to Puzzle #1 to be A, because ENV is bound at other places which would cause incorrect copying. We therefore conclude that no implicit copying can occur, and so the answer to Puzzle #3 is Z.

We emphasize that this result rests on our acceptance of a particular class of meta-circular interpreters. (These interpreters, however, closely model what real LISP systems do.) There are other languages which do implicitly copy structured values when binding variables, such as Algol 60 when using call-by-value. For such a language, the answer to Puzzle #3 would be A (if we represented the list (A B C) as an Algol 60 array, for example), even though the answer to Puzzle #1 would still be Z.

One can argue both for and against copying during binding on the basis of modularity. Copying isolates the caller from the called routine by preventing the called routine from performing under-the-table side-effects on the caller's data objects. Not copying allows data objects to encapsulate independent pieces of state which can be operated on by low-level routines whose details need not be understood by their caller (an example of such a data object is the symbol table of an assembler, with its insertion and lookup routines).

We now consider Puzzle #2. If we accept that binding and variable referencing do not makes copies, then Puzzle #2 is a question about the nature of CONS: if CONS is called twice with arguments which are the same, are the two results the same? (Note that this is the inverse of Postulate 4 for S-expressions in {Note S-expression Postulates and Notation}.) If the answer is consistently A (as in most real LISP systems), then CONS must generate a new object every time it is called. (It must produce different results if the two sets of arguments differ, and an answer of A to Puzzle #2 requires different results if the two sets of arguments are the same.) CONS perforce contains a side effect. Calls to it are not referentially transparent.

The other possibility, given that variable binding and variable referencing do not make copies, is that the answer to Puzzle #2 is Z. In this case, CONS of the same arguments must always produce the same result. This choice leads to galloping non-modularity of data structures without compensation. Suppose, for example, we represent arrays as lists of numbers (a reasonable LISP representation), and want to alter the last element of one such array (using RPLACA). Under this scheme, all arrays whatsoever with the same last element would be magically altered! A language with such characteristics would be extremely difficult to control.

Supposing now that binding does make copies as in Algol 60, the answer to Puzzle #2 must be A. Here it does not matter whether CONS of the

same arguments produces the same result, since the bindings of X and Y will make copies anyway. We may, however, consider this variant:

```
(PROGN (RPLACA (CONS 'A '(B C)) 'Z)
       (CAR (CONS 'A '(B C))))
```

Puzzle #2a

Here we have simply substituted the expressions (CONS 'A '(B C)) for the occurrences of X and Y. If CONS always returns the same object for the same inputs, then Puzzle #2 and Puzzle #2a have different answers if bindings copy, but may have the same answers if bindings do not copy (they may not have the same answer if CONS notices that we have pulled the rug out from under it and produces a new version because the old one was changed!). There is also a quibble as to whether the passing of an argument to RPLACA in itself constitutes a binding — if so, RPLACA must be completely ineffectual, because it always receives a copy! We must then regard RPLACA as a built-in system primitive; the user would have no way to define such a thing. This would be most unfortunate.

We have examined many of the design decisions for the meaning of RPLACA, CONS, and equality. If side effects are to be usable at all, the references to things denoted by variables must not make copies of those things. If the user is to be able to write procedures which produce lasting side effects on their arguments (as system-supplied primitive operators do), then there must be a variable binding mechanism which does not make copies. (LISP's binding mechanism in fact does not copy. Algol 60's call-by-value mechanism does copy structured data, but its call-by-name mechanism does not; we will study this in Part Three.) If the variable binding (or assignment) mechanism does not make copies, then CONS must generate a new, distinct object on each call.

The reader may have noted that we have been talking in circles for the last several paragraphs: in attempting to elucidate the meaning of sameness, we have discussed side effects, and in so doing used the word "same" nearly every other sentence. The point is that it is not possible to define them separately; The meanings of "equality" and "side effect" simultaneously constrain each other. With this in mind, we will investigate the choice of a primitive equality predicate.

The equality predicate we choose should be sufficiently finely grained to distinguish any two objects which have potentially distinct behavior, yet should not be so finely grained as to distinguish entities which otherwise would have the same behavior. Thus we have two desiderata:

> [1] Two objects which are observed to behave differently must not be equal.
> [2] Conversely, we would like two objects which are adjudged unequal to exhibit differing behaviors under suitable circumstances.

Any useful equality predicate must satisfy [1]. Unfortunately, satisfying [2] also may be too difficult; the equivalence of behavior for procedural objects is an unsolvable problem. We are thus forced to settle for an equality predicate which may make more distinctions than are strictly necessary.

LISP has two standard equality predicates: EQUAL and EQ. We exhibited a definition of EQUAL in Part Zero. In Part Zero we also gave a description of EQ, but defined it only on atoms; LISP usually extends EQ to all S-expressions in such a way as to distinguish the results of different calls to CONS (regardless of the arguments given to CONS). Variable references and variable binding "preserve EQness".

In the absence of RPLACA ("pure LISP"), EQ and EQUAL both satisfy desideratum [1]. EQUAL, however, makes fewer unnecessary distinctions than EQ. By desideratum [2], EQUAL is therefore preferred to EQ. (The technique of "hash-consing" [Goto] can be used in this situation to make EQ and EQUAL effectively the same.)

In the presence of side effects such as RPLACA, EQUAL fails to make sufficiently many distinctions. Each call to CONS produces distinct objects, which EQUAL may fail to distinguish. In this case, EQUAL fails desideratum [1]. Thus, in the presence of RPLACA, EQ is the preferred equality predicate.

In summary, indeed "the more things change, the more they remain the same". Two distinct objects may look the same because one masquerades as the other; they can be operationally distinguished only by purposely altering the behavior of just one of them. Thus the ability to decide whether two objects are the same is directly correlated with the ability to perform side effects on them.

Dynamic Scoping as a State-Decomposition Discipline

As we saw in the preceding section, side effects can become rather complicated. To help keep this complexity under control, we ought to abstract and package common patterns of their use.

Suppose we have a procedure PRINT-NUMBER which prints numbers:

```
(DEFINE (PRINT-NUMBER N)
        ((LAMBDA (Q R)
                 (COND ((ZEROP Q) (PRINT-DIGIT R))
                       (T (PROGN (PRINT-NUMBER Q)
                                 (PRINT-DIGIT R)))))
         (/ N 10.)
         (REMAINDER N 10.)))
```

Now people find this program very useful and use it in all their programs.

Normally we want to print numbers in radix 10 (decimal), but occasionally (for example, in a debugging aid) we want to print numbers in other radices, such as 8 or 16. One might generalize the PRINT-NUMBER program to take the radix as an extra argument:

```
(DEFINE (PRINT-NUMBER N RADIX)
        ((LAMBDA (Q R)
                 (COND ((ZEROP Q) (PRINT-DIGIT R))
                       (T (PROGN (PRINT-NUMBER Q)
                                 (PRINT-DIGIT R)))))
         (/ N RADIX)
         (REMAINDER N RADIX)))
```

Of course, then everyone who uses PRINT-NUMBER must supply the radix. This is mildly annoying, because most of the time one wants decimal printing, and one tires of writing "10." all the time. One might write another program for most people to use:

```
(DEFINE (PRINT-10 N)
        (PRINT-NUMBER N 10.))
```

This example is simple, but a real PRINT procedure in a real LISP system may be controlled by dozens of parameters like RADIX: format parameters for printing floating-point numbers, which file to print to, file-dependent format parameters such as line width and page length, file-dependent processing routines (e.g. scrolling for display terminals), abbreviation format parameters for S-expressions, etc. All these extra parameters to PRINT are really determined by the larger context in which PRINT is used, but this context is usually not determined by the immediate caller of PRINT. A program which generates and prints successive prime numbers should not have to deal with the complexities of output files; in particular, one does not want to have to rewrite the program just to direct the output to a line printer instead of a disk file. Context decisions are usually made at a much higher level (perhaps interactively by the user). Therefore the solution of using procedures like PRINT-10 is not acceptable; such procedures only serve as abbreviations, binding the many parameters to constants at too low a decision level.

Another idea is to pass the extra parameters for print control through the intermediate levels of the program. But this violates the modularity of the intermediate modules, which generally have no interest in PRINT's screwy parameters. On the other hand, an occasional intermediate module will be interested in dealing with a few of the parameters (but probably not all of them!). We would like a mechanism for dealing with only the parameters of interest, without having to deal with all of them all of the time.

Side effects can do the job. We can make all the parameters globally available variables (in the top-level environment), initialized to reasonable default values, and invite all interested parties to perform SETQ as necessary. This technique has disadvantages. If every program just changes the parameters at will, then each program must re-set all the parameters (even the ones not of interest) for its own uses of PRINT. This is even worse than just passing PRINT all the parameters!

We can require a convention whereby the parameters normally have their initial default values, and any program which modifies a parameter

must eventually restore it to its previous value. For example, a procedure to print in octal might look like:

```
(DEFINE (PRINT-8 N)
        ((LAMBDA (OLDRADIX)
                 (PROGN (SETQ RADIX 8)
                        (PRINT-NUMBER N)
                        (SETQ RADIX OLDRADIX)))
         RADIX))
```

This convention allows PRINT-8 to locally alter the radix, in a manner transparent to its caller; it does not interfere with the way in which its caller may be using PRINT.

This convention is a standard pattern of use. It is a stack discipline on the values of RADIX (or whatever other variables). We would like to capture this pattern as an abstraction in our language.

Surprise! We have seen this abstraction before: dynamically scoped variables behave in precisely this way. Dynamically scoped variables conceptually have a built-in side effect — we took advantage of this at the end of Part One to fix the problem with the top-level loop. Binding a dynamically scoped variable such as RADIX can be said to cause a side effect because it alters the behavior of a (superficially) unrelated procedure such as PRINT in a referentially opaque manner. Such binding is a particularly structured kind of side effect, because it guarantees that the side effect will be properly undone when the binder has finished executing. Thus with dynamic scoping we could write:

```
(DEFINE (PRINT-8 N)
        ((LAMBDA (RADIX)
                 (PRINT-NUMBER N))
         8))
```

We saw in Part One that, precisely because dynamically scoped variables are referentially opaque, we do not want all variables to be dynamically scoped. But we have newly rediscovered dynamic variables in another context and found them desirable. We therefore consider an interpreter which supplies both lexical and dynamic variables (see Figure 14).

Here we have merged the dynamically scoped variable evaluator (Figure 5) with the lexically scoped evaluator (Figure 11). We changed APPLY to have an extra case, wherein an "open LAMBDA-expression" is effectively closed at the time of its application using the environment of its caller. EVAL is changed to once again supply the environment to APPLY. This interpreter is almost identical to that of LISP 1.5 [LISP 1.5M], with the difference that we write simply (LAMBDA ...) to get a closed procedure where in LISP 1.5 one must write (FUNCTION (LAMBDA ...)); in both cases one must write '(LAMBDA ...) to get an open LAMBDA-expression.

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'SETQ)
               (EVSETQ (CADR EXP) (EVAL (CADDR EXP) ENV) ENV))
              ((EQ (CAR EXP) 'PROGN)
               (EVPROGN (CDR EXP) ENV NIL))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              (T (APPLY (EVAL (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)
                        ENV))))

(DEFINE (APPLY FUN ARGS ENV)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              ((EQ (CAR FUN) '&PROCEDURE)
               (EVAL (CADDR FUN)
                     (BIND (CADR FUN) ARGS (CADDDR FUN))))
              ((EQ (CAR FUN) 'LAMBDA)
               (EVAL (CADDR FUN)
                     (BIND (CADR FUN) ARGS ENV)))
              (T (ERROR))))
```

For VALUE, LOOKUP, and BIND see Figure 3.
For EVCOND and EVLIS see see Figure 5.
For LOOKUP1 see Figure 10 (not Figure 3).

Figure 14
Interpreter with Both Open and Closed Procedures

Although this is the tradition, it doesn't work very well. The problem is that the lexical variables are not really lexical. Although lexical references cannot incorrectly refer to dynamically intended bindings, the reverse is not true. Dynamic variable references can be captured by bindings intended to be strictly lexical.

For example, we might want to write a procedure which packages up information about dealing with RADIX:

```
(DEFINE (RADIX-10 FUN)
        ((LAMBDA (RADIX) (FUN))
         10.))
```

This is more general than PRINT-10 in that it allows us to wrap a binding of RADIX around any piece of code, not just a call to PRINT. (In a more realistic example, we might package up the bindings of a dozen parameters in a similar manner.)

There are two possibilities: should the argument to RADIX-10 be a closed procedure or an open LAMBDA-expression? If closed:

```
(DEFINE (DO-SOMETHING-INTERESTING X FUN)
        (RADIX-10 (LAMBDA () (FORMAT-HAIR 'FOO (CADR X) FUN))))
```

(FORMAT-HAIR takes several arguments, one of them a procedure and presumably calls PRINT at some level), then the binding of RADIX in RADIX-10 will not be apparent to PRINT, because the environment of the call to FORMAT-HAIR is that of the closed procedure, which in turn is that of the call to RADIX-10 within DO-SOMETHING-INTERESTING. Thus it fails to work at all. If the argument to RADIX-10 is left open:

```
(DEFINE (DO-SOMETHING-INTERESTING X FUN)
        (RADIX-10 '(LAMBDA () (FORMAT-HAIR 'FOO (CADR X) FUN))))
```

then this fails to work at all because of a variable naming conflict with FUN. The third argument passed to FORMAT-HAIR will evaluate to the argument which was passed to RADIX-10, namely the quoted lambda expression. This is similar to the MAPCAR bug that originally got us thinking about lexical scoping in Part One.

A solution to this problem is to maintain separate environments for lexical and dynamic variables; this will guarantee that the two kinds cannot interfere with each other. This will require a special syntax for distinguishing references to and bindings of the two kinds of variables. We will choose to encode lexical variables as atomic symbols, as before, and dynamic variables as lists of the form (DYNAMIC x), where x is the name of the dynamic variable. (This choice is completely arbitrary. We could have chosen to encode the two kinds as (LEXICAL x) and x; or as (LEXICAL x) and (DYNAMIC x), leaving atomic symbols as such free to encode yet something else; but we have chosen this because in practice most variable references, even in a purely dynamically scoped LISP, are lexical, or can be considered so.)

In our new interpreter (see Figure 15) we call the two environments ENV (lexical) and DENV (dynamic). The syntax of LAMBDA-expressions is extended to accommodate two kinds of bindings; for example,

```
(LAMBDA (X Y (DYNAMIC Z) W) ...)
```

takes four arguments, and binds the parameters X, Y, and W lexically, and Z dynamically. Using this syntax, we could write RADIX-10 in this way:

```
(DEFINE (RADIX-10 FUN)
        ((LAMBDA ((DYNAMIC RADIX)) (FUN))
         10.))
```

The code for PRINT-NUMBER would then be written:

```
(DEFINE (PRINT-NUMBER N)
        ((LAMBDA (Q R)
                 (COND ((ZEROP Q) (PRINT-DIGIT R))
                       (T (PROGN (PRINT-NUMBER Q)
                                 (PRINT-DIGIT R)))))
         (/ N (DYNAMIC RADIX))
         (REMAINDER N (DYNAMIC RADIX))))
```

Most of the extra complexity in Figure 15 is devoted to the parsing of LAMBDA-expression binding lists upon application by APPLY-PROCEDURE. (For the sake of brevity we have omitted the parts of the interpreter which deal with SETQ and PROGN; they could easily be re-inserted.)

```
(DEFINE (EVAL EXP ENV DENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE) (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV DENV))
              ((EQ (CAR EXP) 'DYNAMIC) (VALUE (CADR EXP) DENV))
              (T (APPLY (EVAL (CAR EXP) ENV DENV)
                        (EVLIS (CDR EXP) ENV DENV)
                        DENV))))

(DEFINE (APPLY FUN ARGS DENV)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS DENV))
              ((EQ (CAR FUN) '&PROCEDURE)
               (APPLY-PROCEDURE (CADR FUN) ARGS '() '() '() '()
                                (CADDDR FUN) DENV (CADDR FUN)))
              (T (ERROR))))

(DEFINE (APPLY-PROCEDURE VARS ARGS LVARS LARGS DVARS DARGS ENV DENV BODY)
        (COND ((NULL VARS)
               (COND ((NULL ARGS)
                      (EVAL BODY
                            (BIND LVARS LARGS ENV)
                            (BIND DVARS DARGS DENV)))
                     (T (ERROR))))
              ((NULL ARGS) (ERROR))
              ((ATOM (CAR VARS))
               (APPLY-PROCEDURE (CDR VARS) (CDR ARGS)
                                (CONS (CAR VARS) LVARS) (CONS (CAR ARGS) LARGS)
                                DVARS DARGS
                                ENV DENV BODY))
              ((EQ (CAAR VARS) 'DYNAMIC)
               (APPLY-PROCEDURE (CDR VARS) (CDR ARGS)
                                LVARS LARGS
                                (CONS (CAR VARS) DVARS) (CONS (CAR ARGS) DARGS)
                                ENV DENV BODY))
              (T (ERROR))))
```

For EVCOND and EVLIS see Figure 2.
For VALUE, BIND, and LOOKUP see Figure 3.
For LOOKUP1 see Figure 10.

Figure 15
Interpreter with Separate Lexical and Dynamic Variables

Dynamic scoping provides an important abstraction for dealing with side effects in a controlled way. A low-level procedure may have state variables which are not of interest to intermediate routines, but which must be controlled at a high level. Dynamic scoping allows any procedure to get access to parts of the state when necessary, but permits most procedures to ignore the existence of the state variables. The existence of many dynamic variables permits the decomposition of the state in such a way that only the part of interest need be dealt with.

If dynamic variables are integrated with the lexical environment, intractable dilemmas are encountered. (We have not considered here all possible such integration schemes, but the authors have found such difficulties with every such scheme they have examined.) We have therefore presented an interpreter in which environments for the two kinds of variable are separated.

Summary

We examined the effects of various language design decisions on the programming styles available to a user of the language, with particular emphasis on the ability to incrementally construct modular systems. At each step we exhibited an interactive meta-circular interpreter for the language under consideration. Each new interpreter was the result of an incremental change to the previous interpreter.

We started with a simple interpreter for LISP recursion equations. In order to capture certain abstractions we were forced to introduce procedural data. This in turn forced consideration of the meanings of free variables in a procedure, for the simplest extension unexpectedly introduced dynamic scoping of variables.

We were compelled to turn from dynamic scoping to lexical scoping to preserve the integrity of procedural abstractions. The referentially transparent language thus obtained is richer than expected. It allows the definition of procedures which construct other procedures by instantiation of a prototype. Unfortunately, we found that complete referential transparency in a language makes it impossible to construct an interactive interface to the interpreter. But such an interface is necessary to satisfy another requirement of modular construction: that parts of a program can be independently defined, replaced, and debugged. We were forced to give up absolute referential transparency to admit an interactive interface.

The problems of the interactive interface led us to consider the notion of state as a dimension of abstraction. Just as we didn't want to have textually monolithic programs, we wanted to avoid programs which manipulate a monolithic representation of the state. The decomposition of the state of a system into several independent parts induces the notion of a side effect. Side effects only make sense relative to a definition of equality on the space of data objects. But the definition of equality itself depends simultaneously on the notion of side effect. Only a few of the choices of equality predicate and side effect notion are consistent with the requirements of modular construction.

The introduction of side effects is inconsistent with referential transparency. But since both are important to support modular construction we must accept an engineering trade-off between them. We were led to look for controlled patterns of side effects which can be easily understood and safely applied. We discovered that one such pattern is equivalent to the use of dynamically scoped variables we discussed earlier. We investigated how to construct a system which integrates lexical and dynamic scoping in a smooth way.

There are many issues yet to be explored. The introduction of side effects raises questions about order of evaluation. An interesting order provided by Algol 60 is call-by-name. This discipline, so unlike LISP's, is induced from a different notion of procedure, expressed as the "copy rule". This idea is a syntactic one, and so differs in flavor from the

procedural ideas embodied by the interpreters we have presented. Consideration of syntactic transformations leads to the notion of meta-procedures, such as macros, compilers, and simplifiers. We will explore all of this in Parts Three and Four.

Acknowledgements

We would like to thank Johan De Kleer, Daniel L. Weinreb, Julie Sussman, Carl Hewitt, Richard Stallman, Jon Doyle, and Mitch Marcus for reading our draft. They found a few bugs and helped us refine our presentation. We also want to thank Hal Abelson and Robert Fano for help and encouragement. Finally, we must thank John McCarthy. Besides responding to our messages and answering questions about the early history of LISP, it was all his idea in the first place and we are continually amazed at the beauty and power of his conception.

Notes

{Can George do better?} Page 34

The problem here is that George needs access to the statistics counter without giving that access to anyone else. As described in the next example George can make the counter an own variable, but how can he get access to it? One idea is that George can define MAPCAR in the following manner:

```
((LAMBDA (N)
        (PROGN (SETQ MAPCAR
                     (LAMBDA (F L)
                             (PROGN (SETQ N (+ N 1))
                                    (OLDMAPCAR F L))))
               (LAMBDA () N)))
 0)
```

This expression defines MAPCAR by SETQing (See {Note Driver Loop with Side Effects}.) it to an appropriate procedure. It then returns, as a value, an anonymous procedure which accesses the value of the statistics counter. If George saves this value and uses it to get at the counter when he needs it, he will have isolated it completely from everyone else!

{Debugging} Page 27

It has been suggested that it is possible always to write correct programs. Such a situation would eliminate the need for debugging. The problem with this idea is that a crucial part of the problem-solving strategy is the decomposition of problems into presumably independent subproblems. There is no guarantee that this is possible in general, but even when it is not possible, there are often general strategies for approximating a solution to a problem by composing the solutions to almost independent subproblems. Often one can make progress on the solution to a hard problem by considering the solution of a simplified version of the problem which is similar in some essential aspect to the original one but which differs from it in detail. Once the solutions to the subproblems are obtained, they must be fitted together, and the details of the interactions smoothed out. The fixing of unanticipated interactions is debugging.

Even in those cases where a decomposition into completely independent subproblems is possible, it is not always feasible. In order to be sure that the solutions to the subproblems are really independent it is necessary to understand both the problem and the possible implementations and interactions of subsolutions so completely that one must effectively solve the entire problem before choosing the correct decomposition. This compromises the decomposition strategy.

{Driver Loop with Side Effects} Pages 37, 53, 59

This driver loop (Figure N1) is similar to the one in Figure 8 (which didn't work). This one does work because, although top-level procedure definitions are closed in the current top-level environment, that environment is changed using a side effect when new definitions are made.

```
(DEFINE (DRIVER-LOOP-1 ENV FORM)
        (COND ((ATOM FORM)
               (DRIVER-LOOP ENV NIL (PRINT (EVAL FORM ENV))))
              ((EQ (CAR FORM) 'DEFINE)
               (DRIVER-LOOP ENV
                            (EVSETQ (CAADR FORM)
                                    (LIST '&PROCEDURE
                                          (CDADR FORM)
                                          (CADDR FORM)
                                          ENV)
                                    ENV)
                            (PRINT (CAADR FORM))))
              (T (DRIVER-LOOP ENV NIL (PRINT (EVAL FORM ENV))))))
```

For EVAL and EVSETQ see Figure 11.
For LOOKUP1 see Figure 3 (not Figure 10, despite Figure 11!).

Figure N1
Implementation of DRIVER-LOOP Using Side Effects

{EVALQUOTE}

Page 14

The top level of LISP 1 [LISP 1M] and LISP 1.5 [LISP 1.5M] actually was not at all like the one presented here. Rather than reading one S-expression and giving it to EVAL, it read two S-expressions and gave them to APPLY. Such a top level is called an EVALQUOTE top level (see Figure N2).

```
(DEFINE (DRIVER-LOOP-1 PROCEDURES FORM1)
        (DRIVER-LOOP-2 PROCEDURES FORM1 (READ)))

(DEFINE (DRIVER-LOOP-2 PROCEDURES FORM1 FORM2)
        (COND ((EQ FORM1 'DEFINE)
               (DRIVER-LOOP (BIND (LIST (CAAR FORM2))
                                  (LIST (LIST '&PROCEDURE (CDAR FORM2) (CADR FORM2)))
                                  PROCEDURES)
                            (PRINT (CAAR FORM2))))
              (T (DRIVER-LOOP PROCEDURES
                              (PRINT (APPLY FORM1 FORM2 PROCEDURES))))))
```

For DRIVER-LOOP see Figure 1.
For APPLY see Figure 2.
For BIND see Figure 3.

Figure N2
Driver Loop for an EVALQUOTE Top Level

This driver loop is somewhat nicer than the one in Figure 1, because the one in Figure 1 had an essentially useless COND clause. The case of typing an atom was not useful, because there were no top-level values for variables. Once we introduce procedural objects, this is no longer true. But EVALQUOTE requires an inconsistency of notation: at the top level one must write CAR((A . B)), whereas in the middle of a program one would write (CAR '(A . B)).

The notion of EVALQUOTE also has some theoretical motivation, if one thinks of LISP as a universal machine akin to a universal Turing machine. In this model one takes a description of a machine to be simulated and a description of its input data, and gives them to the universal machine to process. In LISP, the universal machine is APPLY.

{Gaussian}

Pages 32, 68

A typical example of the use of a pseudo-random number generator is to construct a generator for pseudo-random numbers with a Gaussian distribution by adding up a large number of uniformly distributed pseudo-random numbers. We would like to write it in roughly as in Figure N3.

```
(DEFINE (GAUSSIAN)
        (WEBER 0 43))

(DEFINE (WEBER X N)
        (COND ((= N 0) X)
              (T (WEBER (+ X (RANDOM)) (- N 1)))))
```

Figure N3
"Gaussian" Pseudo-Random Number Generator

This code should add up 43 pseudo-random numbers obtained by calling RANDOM. We cannot write such a RANDOM without side effects, however. We can arrange to pass the seed around, as in Figure N4.

```
(DEFINE (GAUSSIAN SEED)
        (WEBER 0 43 SEED))

(DEFINE (WEBER X N SEED)
        (COND ((= N 0) (CONS X SEED))
              (T ((LAMBDA (NEWSEED)
                          (WEBER (+ X NEWSEED) (- N 1) NEWSEED))
                  (RANDOM SEED)))))
```

Figure N4
"Gaussian" Pseudo-Random Number Generator, Passing SEED

This is much more complicated. The user of GAUSSIAN must maintain the seed. Moreover, GAUSSIAN and WEBER each need to return two values; here we cons them together, and the user must take them apart.

{LABELS}

Pages 29, 59

This technique can be generalized to allow the definition of recursive local procedures. (Although the Y-operator discussed in {Note Y-operator} can be used to implement recursive local procedures, it is extremely painful to construct several mutually recursive procedures. Although mutually recursive procedures can be theoretically eliminated (by procedure integration), this process destroys the conceptual structure of the program.)

Consider writing a procedure to construct the reverse of a given list:

```
(DEFINE (REVERSE L)
        (REVERSE1 L '()))

(DEFINE (REVERSE1 OLD NEW)
        (COND ((NULL OLD) NEW)
              (T (REVERSE1 (CDR OLD) (CONS (CAR OLD) NEW)))))
```

The procedure REVERSE1 is irrelevant to the outside world; we would like to hide it inside REVERSE.

Let us invent a new construction to permit the definition of local procedure definitions with names:

```
(LABELS (((f1 v11 v12 ...) body1)
         ((f2 v21 v22 ...) body2)
         ...
         ((fN vN1 vN2 ...) bodyN))
        body)
```

means the value of body when evaluated in an environment where the specified procedure definitions are available. For example:

```
(DEFINE (REVERSE L)
        (LABELS (((REVERSE1 OLD NEW)
                  (COND ((NULL OLD) NEW)
                        (T (REVERSE1 (CDR OLD) (CONS (CAR OLD) NEW))))))
                (REVERSE1 L '())))
```

The same trick works for LABELS as for the top level: when LOOKUP1 has found a LABELS-defined function, it has the correct environment in hand for constructing a &PROCEDURE-object. We need only add a test in EVAL for the LABELS construct, and arrange for the appropriate &LABELED-objects to be constructed (see Figure N5).

```
(DEFINE (EVAL EXP ENV)
        (COND ((ATOM EXP)
               (COND ((NUMBERP EXP) EXP)
                     (T (VALUE EXP ENV))))
              ((EQ (CAR EXP) 'QUOTE)
               (CADR EXP))
              ((EQ (CAR EXP) 'LAMBDA)
               (LIST '&PROCEDURE (CADR EXP) (CADDR EXP) ENV))
              ((EQ (CAR EXP) 'LABELS)
               (EVLABELS (CADR EXP) EXP '() '() ENV))
              ((EQ (CAR EXP) 'COND)
               (EVCOND (CDR EXP) ENV))
              (T (APPLY (EVAL (CAR EXP) ENV)
                        (EVLIS (CDR EXP) ENV)))))

(DEFINE (EVLABELS DEFINITIONS EXP NAMES FNS ENV)
        (COND ((NULL DEFINITIONS)
               (EVAL (CADDR EXP) (BIND NAMES FNS ENV)))
              (T (EVLABELS (CDR DEFINITIONS)
                           EXP
                           (CONS (CAAAR DEFINITIONS) NAMES)
                           (CONS (LIST '&LABELED
                                       (CDAAR DEFINITIONS)
                                       (CADAR DEFINITIONS))
                                 FNS)
                           ENV))))
```

For VALUE, LOOKUP, and BIND see Figure 3.
For EVCOND and EVLIS see Figure 5.
For APPLY see Figure 7.
For LOOKUP1 see Figure 10 (not Figure 3).

Figure N5
An Evaluator For Local Lexical Scoping,
Dynamic Top-Level References,
and Local Definition of Recursive Procedures

{LABELS with Side Effects} Page 37

This implementation of LABELS (see Figure N6) applies the technique of {Note Driver Loop with Side Effects} to the implementation of LABELS in {Note LABELS}. This is in fact how LABELS (or its cousin LABEL) is usually implemented in "real" LISP systems.

```
(DEFINE (EVLABELS DEFINITIONS EXP NAMES FNS ENV)
        (COND ((NULL DEFINITIONS)
               (EVLABELS-CLOSE (CADR EXP) EXP NIL (BIND NAMES FNS ENV)))
              (T (EVLABELS (CDR DEFINITIONS)
                           EXP
                           (CONS (CAAR DEFINITIONS) NAMES)
                           (CONS '&UNASSIGNED FNS)
                           ENV))))

(DEFINE (EVLABELS-CLOSE DEFINITIONS EXP VALS ENV)
        (COND ((NULL DEFINITIONS)
               (EVLABELS-CLOBBER NIL EXP (CDAR ENV) VALS ENV))
              (T (EVLABELS-CLOSE (CDR DEFINITIONS)
                                 EXP
                                 (CONS (LIST '&PROCEDURE
                                             (CDAAR DEFINITIONS)
                                             (CADAR DEFINITIONS)
                                             ENV)
                                       VALS)
                                 ENV))))

(DEFINE (EVLABELS-CLOBBER HUNOZ EXP SLOTS VALS ENV)
        (COND ((NULL VALS)
               (EVAL (CADDR EXP) ENV))
              (T (EVLABELS-CLOBBER (RPLACA SLOTS (CAR VALS))
                                   EXP
                                   (CDR SLOTS)
                                   (CDR VALS)
                                   ENV))))
```

For EVAL and EVSETQ see Figure 11.
For LOOKUP1 see Figure 3 (not Figure 10, despite Figure 11!).

Figure N6
Implementation of LABELS Using Side Effects

{Primitive Operators} Page 10

A primitive operator might be a very complicated object in a "real" LISP implementation; it would probably have machine-language code within it. We are not interested in the details of a particular host machine here; we wish only to present a simple meta-circular definition of PRIMOP and PRIMOP-APPLY. We will notate the procedural object which is the value of CAR (say) in the initial top-level environment <THE-PRIMITIVE-PROCEDURES> as "&CAR". This object has no interesting properties except that it is EQ to itself and not to any other object. The initial top-level environment therefore looks like:

```
(((CAR CDR EQ ATOM NULL NUMBERP + - * ...)
  &CAR &CDR &EQ &ATOM &NULL &NUMBERP &+ &- &* ...))
```

Given this, we can define PRIMOP and PRIMOP-APPLY as in Figure N7.

```
(DEFINE (PRIMOP FUN)
        (COND ((EQ FUN '&CAR) T)
              ((EQ FUN '&CDR) T)
              ((EQ FUN '&EQ) T)
              ((EQ FUN '&ATOM) T)
              ((EQ FUN '&NULL) T)
              ((EQ FUN '&NUMBERP) T)
              ((EQ FUN '&+) T)
              ((EQ FUN '&-) T)
              ((EQ FUN '&*) T)
              ...
              (T NIL)))

(DEFINE (PRIMOP-APPLY FUN ARGS)
        (COND ((EQ FUN '&CAR) (CAR (CAR ARGS)))
              ((EQ FUN '&CDR) (CDR (CAR ARGS)))
              ((EQ FUN '&EQ) (EQ (CAR ARGS) (CADR ARGS)))
              ((EQ FUN '&ATOM) (ATOM (CAR ARGS)))
              ((EQ FUN '&NULL) (NULL (CAR ARGS)))
              ((EQ FUN '&NUMBERP) (NUMBERP (CAR ARGS)))
              ((EQ FUN '&+) (+ (CAR ARGS) (CADR ARGS)))
              ((EQ FUN '&-) (- (CAR ARGS) (CADR ARGS)))
              ((EQ FUN '&*) (* (CAR ARGS) (CADR ARGS)))
              ...
              (T (ERROR))))
```

Figure N7
Meta-Circular Definition of PRIMOP and PRIMOP-APPLY

{PROGN Wizardry}

Page 35

We defined EVPROGN in the way shown in Figure 11 rather than in this "more obvious" way:

```
(DEFINE (EVPROGN EXPS ENV LASTVAL)
        (COND ((NULL EXPS) LASTVAL)
              (T (EVPROGN (CDR EXPS) ENV (EVAL (CAR EXPS) ENV)))))
```

for a technical reason: we would like the tail-recursive properties of the code being interpreted to be reflected in the interpretation process. We specifically want recursive calls as the last subform of a PROGN form to be tail-recursive if the PROGN form itself is in a tail-recursive situation. For example, we might write a loop such as:

```
(DEFINE (PRINTLOOP X)
        (COND ((= X 0) 'BLASTOFF)
              (T (PROGN (PRINT X)
                        (PRINTLOOP (- X 1))))))
```

We would like this loop to be iterative, but it can be iterative only if the recursive call to PRINTLOOP is tail-recursive. Our point is that if the "obvious" version of EVPROGN is used in the interpreter, then interpretation of PRINTLOOP will not be tail-recursive because of the "stacking up of EVPROGN frames" (the last call to EVAL from EVPROGN is not tail-recursive). This is unnecessary because EVPROGN does nothing with the last value but return it anyway.

By the way, the use of PROGN in a COND clause as shown above in PRINTLOOP is a very common situation, as is the use of a PROGN as the body of a procedure (cf. George's last experimental version of MAPCAR). As a convenience, most real LISP implementations define extended versions of COND and LAMBDA which implicitly treat clauses (resp. bodies) as PROGN forms (see Figure N8). This allows us to write such things as:

```
(DEFINE (PRINTLOOP X)
        (SLEEP 1)
        (COND ((= X 0) 'BLASTOFF)
              (T (PRINT X)
                 (PRINTLOOP (- X 1)))))
```

```
(DEFINE (EVCOND CLAUSES ENV)
        (COND ((NULL CLAUSES) (ERROR))
              ((EVAL (CAAR CLAUSES) ENV)
               (EVPROGN (CDAR CLAUSES) ENV NIL))
              (T (EVCOND (CDR CLAUSES) ENV))))

(DEFINE (APPLY FUN ARGS)
        (COND ((PRIMOP FUN) (PRIMOP-APPLY FUN ARGS))
              ((EQ (CAR FUN) '&PROCEDURE)
               (EVPROGN (CDDR FUN)
                        (BIND (CADR FUN) ARGS (CADDDR FUN))
                        NIL))
              (T (ERROR))))
```

For EVAL and EVPROGN see Figure 11.

Figure N8
Treating COND Clauses and Procedure Bodies as Implicit PROGN Forms

Finally, we note that PROGN is unnecessary except as a programming convenience. Because the language is defined to be executed in applicative order (cf. {Note Normal Order Loses} in [Revised Report]), we can force the sequencing of evaluation, as well as throw away unwanted values, by using LAMBDA-expressions. We first note that

$$(\text{PROGN } e_1\ e_2\ \cdots\ e_{N-1}\ e_N) \equiv (\text{PROGN } e_1\ (\text{PROGN } e_2\ \ldots\ (\text{PROGN } e_{N-1}\ e_N)\ \ldots\ ))$$

so that we need worry only about PROGN with two subforms:

$$(\text{PROGN } e_1\ e_2) \equiv ((\text{LAMBDA } (\text{HUNOZ F})\ (\text{F}))\ e_1\ (\text{LAMBDA } ()\ e_2))$$

(see [Imperative] and [Revised Report]).

{QUOTE Mapping} Page 7

What the QUOTE notation achieves is a simple mapping of the entire set of S-expressions into a subset of itself; this mapping is trivially invertible. This is necessary in order to leave some S-expressions left over to represent other things.

This idea may be applied to natural numbers as well. We can "quote" a number by doubling it. In this way every even number represents half of itself, just as the S-expression (QUOTE α) represents the S-expression in its cadr. This leaves all the odd numbers for other purposes. For example, we can define an ordered set of variables and let 3^N encode the N'th variable, for N>10. We can also let 3^1 mean COND, 3^2 mean LAMBDA, etc. We can then encode a procedure call as $5^f 7^x 11^y 13^z \ldots$ where f is the encoding of the procedure and x, y, z, ... are the encodings of the arguments; COND forms and LAMBDA-expressions can be similarly encoded. For example,

```
(COND ((NULL A) 3) (T 6))
```

might be encoded as the number

$$5^{3^1}{}_7{}^{(5^{(5^{3^{1421}}{}_7{}^{3^{10}})}{}_7{}^6)}{}_{11}{}^{(5^{3^{48}}{}_7{}^{12})}$$

In this manner we can encode all of the LISP language as natural numbers. This is an example of the technique of "Gödelization".

{QUOTE Shafts the Compiler} Page 19

We emphasize that it is not the presence of dynamically scoped variables which makes standard LISP difficult for compilers, but the very fact that the LAMBDA-expressions are quoted. It is impossible in general to determine whether a quoted S-expression is intended to be code or just some constant data. Most LISP systems provide another kind of QUOTE called FUNCTION. In LISP 1 [LISP 1M] and LISP 1.5 [LISP 1.5M] this used to produce FUNARG objects (we call them &PROCEDURE objects), but in more recent LISP systems [Moon] [Teitelman] an ordinary FUNCTION-expression has been made equivalent to a quoted expression, serving only as a flag to the compiler that the quoted expression is intended as code. However, the introduction of the "'" notation for quoted expressions has led many programmers to prefer the use of QUOTE to FUNCTION for reasons of conciseness. This in turn has required changes to the compiler to specially recognize standard situations where this is used (e.g. the functional argument to MAPCAR), but this patch doesn't solve the problem generally.

{RPLACA Can Alter CAR Instead}

Page 40

We have implicitly thought of the RPLACA operation as modifying a cons so as to have a different car. However, there is an interpretation in which RPLACA is thought of as modifying the CAR operator. Taking the car of an object always involves both the CAR operator and the object. When we perform an RPLACA on object denoted by FOO, all we can say is that the value of (CAR FOO) may have changed. It is not necessarily clear what aspect of that expression has changed. Using this idea, we can express RPLACA in terms of SETQ as in Figure N9. Note that we depend on EQ to distinguish different results of CONS.

```
(DEFINE (RPLACA X Y)
        (PROGN ((LAMBDA (OLDCAR)
                        (SETQ CAR
                              (LAMBDA (Z)
                                      (COND ((EQ Z X) Y)
                                            (T (OLDCAR X))))))
                CAR)
               X))
```

Figure N9
RPLACA in Terms of SETQ Which Modifies CAR

{S-expression Postulates and Notation) Pages 4, 41

S-expressions form a number system analogous to that for the natural numbers. Each can be used to encode arbitrary strings of symbols by means of "Gödelization", but the S-expression encoding is usually far more convenient than the numerical encoding.

We repeat here the informal characterization of Peano's postulates and the analogous postulates for S-expressions from [Levin]:

The Postulates of Arithmetic

1. Zero is a number.
2. The successor of a number is a number.
3. Zero is not the successor of any number.
4. No two numbers have the same successor.
5. (Induction Principle) Any property which is true for zero, and is such that if it is true for some number it is also true for the successor of that number, it is true for all numbers.

Zero is notated as 0, and the successor of any number N is notated as N'. As a convenience we define alternative notations for numbers other than zero, such as decimal place-value notation. Thus for 0'''''''''''''' we often write 13.

The Postulates for S-expressions

1. Atoms are S-expressions.
2. The cons of any two S-expressions is an S-expression.
3. An atom is not the cons of any two S-expressions.
4. If α differs from β, or if γ differs from δ, then cons of α and γ differs from cons of β and δ.
5. (Induction Principle) Any property which is true of all atoms, and is such that if it is true for two S-expressions it is also true for their cons, is true for all S-expressions.

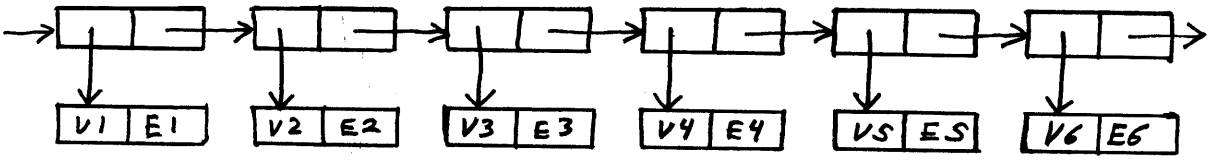
Atoms are notated as strings of letters and digits. The cons of two S-expressions α and β is notated $(\alpha . \beta)$. As a convenience, we define alternative notations for some commonly used forms of S-expression, such as list notation. The atom NIL is called the "empty list"; we write it as (). If $(\alpha \beta \gamma \ldots \delta)$ is (the notation for) a list π (where the "..." is meant as a meta-syntactic ellipsis), then the cons of ϵ and π is written $(\epsilon \alpha \beta \gamma \ldots \delta)$. We also define quotation notation, in which (QUOTE α) is written as 'α.

(This definition of S-expressions applies to "pure LISP", which has no side effects. In Part Two, when the RPLACA and RPLACD operators are introduced, the phrase "the cons of" will not be well-defined.)

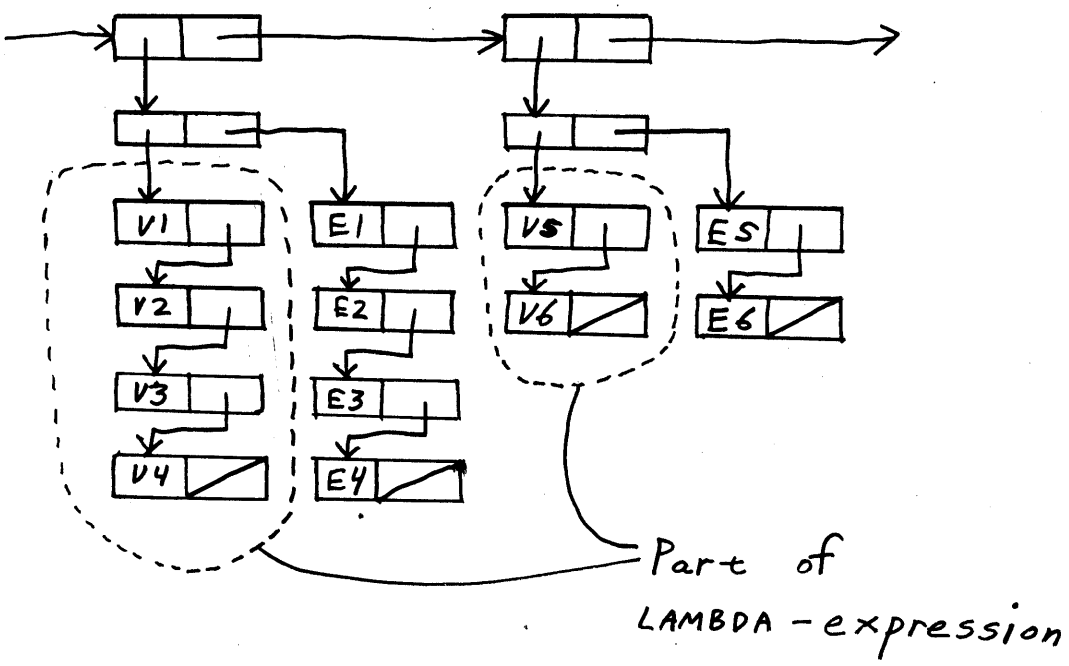
{This ain't A-lists}

Page 11

Our symbol table routines are not the same as those in LISP 1.5. Their behavior is approximately the same, but the data structures involved differ. The LISP 1.5 routines (PAIRLIS and ASSOC) use the traditional "association list" format:



Our routines (BIND and LOOKUP), besides having nicer names, are more efficient because the number of conses performed to bind a given number of variables is usually smaller (we arrange for the environment structure to share the variable lists already contained in LAMBDA-expressions). Morever, the environment is organized into "frames" or "contours", which will be of some utility later. The environment is represented in this form:



{Value Quibble}

"Did he ever return?
No, he never returned,
And his fate is still unlearned..."
— The Man Who Never Returned
(Charlie on the MTA)

We said "EVAL's purpose is to determine the values of expressions". But what is the value of the expression (DRIVER)? It is certainly not an illegal or useless expression to evaluate, yet it has no value. The purpose of the expression is to cause a certain process to be evolved; it is an "infinite loop", which never returns. This process includes side effects (READ and PRINT) through which it interacts with the user. This situation arises because the system of interest is broken into two parts with independent state: the computer and the user. We will have more to say about this later.

{Weber}

Pages 32, 38

To continue our GAUSSIAN example (see {Note Gaussian}), we can try to remove the side-effect from RANDOM while avoiding the passing around of SEED by pushing RANDOM up to the top level (see Figure N10). RANDOM-DRIVER takes a function F and an initial seed (reminiscent of <THE-PRIMITIVE-PROCEDURES>), and continually stuffs random numbers into F. Each call to F must produce a new F (a kind of continuation [Reynolds]). Using this, we can arrange for numbers with a "Gaussian" distribution to be generated.

```
(DEFINE (RANDOM-DRIVER F SEED)
        ((LAMBDA (NEWSEED)
                 (RANDOM-DRIVER (F NEWSEED) NEWSEED))
         ((LAMBDA (Z)
                  (COND ((> Z 0) Z)
                        (T (+ Z -32768.))))
          (* SEED 899.))))

(DEFINE (GAUSSIAN G)
        (WEBER 0 43 G))

(DEFINE (WEBER X N H)
        (COND ((= N 0) (H X))
              (T (LAMBDA (R)
                         (WEBER (+ X R) (- N 1) H)))))

(DEFINE (DRIVER USERFN INITSEED)
        (RANDOM-DRIVER (GAUSSIAN USERFN) INITSEED))
```

Figure N10
"Gaussian" Pseudo-Random Number Generator without Passing SEED Around

In this way, a user function can be provided to DRIVER (along with the initial seed), and the user function will have "Gaussian" numbers stuffed into it. For example:

```
(DEFINE (P R) (PROGN (PRINT R) P))

(DRIVER P 11)
```

will print an interminable sequence of "Gaussian" numbers. Notice the structure of the program: the RANDOM procedure calls GAUSSIAN, which in turn calls the user procedure. We have completely everted the overall system. The more layers in the original system piled on top of GAUSSIAN, the more layers will appear inside-out in this version.

Now there are two other funny things about this. One is that we had to use a side effect (PRINT) to get the answer out; the other is that

it's hard to make it stop! These problems are related. The structure of RANDOM-DRIVER is an infinite loop, as with all drivers. Because RANDOM-DRIVER never returns a value, there is no way to get an answer out without a side-effect like PRINT.

We can arrange to signal RANDOM-DRIVER that no more values are desired, and to return a value (see Figure N11).

```
(DEFINE (RANDOM-DRIVER F SEED)
        (COND ((CAR F) (CDR F))
              (T ((LAMBDA (NEWSEED)
                          (RANDOM-DRIVER ((CDR F) NEWSEED) NEWSEED))
                  ((LAMBDA (Z)
                           (COND ((> Z 0) Z)
                                 (T (+ Z -32768.))))
                   (* SEED 899.))))))

(DEFINE (GAUSSIAN G)
        (WEBER 0 43 G))

(DEFINE (WEBER X N H)
        (COND ((= N 0) (H X))
              (T (CONS NIL
                       (LAMBDA (R)
                               (WEBER (+ X R) (- N 1) H))))))

(DEFINE (DRIVER USERFN)
        (RANDOM-DRIVER (GAUSSIAN USERFN) 43))
```

Figure N11
"Gaussian" Random-Number Generator "Top Level" without Side Effects

Using this new definition, we can write:

```
(DEFINE (P R) (CONS T R))

(DRIVER P 11)
```

which eventually returns one "Gaussian" number. (Doing something with more than one "Gaussian" number takes a little more work...)

Notice that in order to make this work, RANDOM-DRIVER had to know an awful lot about its functional argument; a fairly complicated protocol had to be developed for handshaking. We might argue that this exercise, while it has indeed removed all obvious side effects, has somewhat tarnished the modularity of the RANDOM program. In any case, the structure of our final program is not exactly what we had in mind when we started.

{Y-operator}

Pages 26, 57

While the interpreter of Figure 8 cannot DEFINE recursive procedures, it is possible to define recursive procedures by using a variant of the "paradoxical combinator", also known as the Y-operator:

```
(DEFINE (Y F)
        ((LAMBDA (G)
                 (LAMBDA (X)
                         ((F (G G)) X)))
         (LAMBDA (G)
                 (LAMBDA (X)
                         ((F (G G)) X)))))
```

Using this we define the doubly-recursive algorithm for computing the Fibonacci function:

```
(DEFINE (FIB K)
        ((Y (LAMBDA (F)
                    (LAMBDA (N)
                            (COND ((= N 0) 1)
                                  ((= N 1) 1)
                                  (T (+ (F (- N 1)) (F (- N 2))))))))
         K))
```

That this manages to work is truly remarkable. Notice that this is almost identical to the LABEL construct which was actually introduced by LISP 1, though at the time it was invented the implementors didn't realize this correspondence [LISP History].

References

(Entries marked with a "*" are not referenced in the text.)

[Church] Pages 4, 18, 25
Church, Alonzo. The Calculi of Lambda Conversion. Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Debunking] Page 12
Steele, Guy Lewis Jr. "Debunking the 'Expensive Procedure Call' Myth." Proc. ACM National Conference (Seattle, October 1977),153-162. Revised as MIT AI Memo 443 (Cambridge, October 1977).

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

[Fischer] Page 24
Fischer, Michael J. "Lambda Calculus Schemata." Proceedings of ACM Conference on Proving Assertions about Programs. SIGPLAN Notices (January 1972).

[Goto] Page 43
Goto, Eiichi. Monocopy and Associative Algorithms in an Extended LISP. Information Science Laboratory, University of Tokyo (May 1974).

[Hewitt and Smith] Pages 2, 24
Hewitt, Carl, and Smith, Brian. "Towards a Programming Apprentice." IEEE Transactions on Software Engineering SE-1, 1 (March 1975), 26-45.

[Imperative] Page 62
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Kleene] Page 4
Kleene, Stephen Cole. Introduction to Metamathematics. Von Nostrand (Princeton, 1950).

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

[Levin] Page 65
Levin, Michael. Mathematical Logic for Computer Scientists. MIT Project MAC TR-131 (Cambridge, June 1974).

[LISP 1M] Pages 18, 55, 63
McCarthy, J., Brayton, R., Edwards, D., Fox, P., Hodes, L., Luckham, D., Maling, K., Park, D., and Russell, S. LISP 1 Programmer's Manual. Artifical Intelligence Group, Computation Center and Research Laboratory of Electronics, MIT (Cambridge, March 1960).

[LISP 1.5M] Pages 2, 17, 45, 55, 63
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[LISP History] Pages 4, 20, 70
McCarthy, John. "History of LISP." To appear in Proceedings of the SIGPLAN History of Programming Languages Conference, June 1978.

[McDermott and Sussman] Page 2
McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER Reference Manual. AI Memo 295a. MIT AI Lab (Cambridge, January 1974).

[Moon] Pages 2, 63
Moon, David A. MacLISP Reference Manual, Revision 0. Project MAC, MIT (Cambridge, April 1974).

[Moses] Page 20
Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Naur] Page 21
Naur, Peter (ed.), et al. "Revised Report on the Algorithmic Language ALGOL 60." Comm. ACM 6, 1 (January 1963), 1-20.

[Revised Report] Pages 2, 62
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME. MIT AI Memo 452 (Cambridge, January 1978).

[Reynolds] Pages 8, 68
Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." ACM Conference Proceedings 1972.

[Rulifson] Page 2
Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. QA4: A Procedural Calculus for Intuitive Reasoning. Techical Note 73. Artificial Intelligence Center, Stanford Research Institute (Menlo Park, California, November 1972).

[SCHEME] Pages 2, 12
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] Page 2
Smith, Brian C. and Hewitt, Carl. A PLASMA Primer (draft). MIT AI Lab (Cambridge, October 1975).

[Teitelman] Pages 2, 63
Teitelman, Warren. InterLISP Reference Manual. Revised edition. Xerox Palo Alto Research Center (Palo Alto, 1975).

[Warren] Page 4
Warren, David H.D., and Pereira, Luis. "PROLOG: The Language and Its Implementation Compared with LISP." Proceedings of the Symposium on Artifical Intelligence and Programming Languages (Rochester, New York, August 1977). SIGPLAN Notices 12, 8, SIGART Newsletter 64 (August 1977), 109-115.