

already been determined, all other nodes will eventually enter the state in which they await a reply.

Complexity Analysis

A processor, x , initiates messages along paths of length 2^i only if it is not defeated by a processor within distance 2^{i-1} (in either direction) from x . Within any group of $2^{i-1} + 1$ consecutive processors, at most one can initiate messages along paths of length 2^i . Although possibly all n processors will initiate paths of length 1, at most $\lceil n/2 \rceil$ (read ceiling of $n/2$) processors will initiate paths of length 2, at most $\lceil n/3 \rceil$ of length 4, at most $\lceil n/5 \rceil$ of length 8, etc.

A processor initiating messages along paths of length 2^i causes messages to emanate in both directions, and return. At most $4 \cdot 2^i$ messages will be passed as a result of that initiation. The sum total of all messages passed is therefore at most

$$4 \cdot (1 \cdot n + 2 \cdot \lceil n/2 \rceil + 4 \cdot \lceil n/3 \rceil + 8 \cdot \lceil n/5 \rceil + \dots + 2^i \cdot \lceil n/(2^{i-1} + 1) \rceil + \dots).$$

Each of the terms within the parentheses is less than $2n$. There are no more than $1 + \lceil \log n \rceil$ terms. (No processor will pass messages along paths of length $2n$ or greater since, once a processor initiates paths of at least n length and the message is acceptable all the way around the circle, the processor wins and stops initiating messages.) Thus, the total number of messages passed is less than $8n + 8\lceil n \log n \rceil = O(n \log n)$.

If one defines the time complexity to be the minimum time required for the completion of an election assuming as much message transmission overlap as possible, the worst case time complexity can easily be shown to be linear in the number of processors. The exact formula is

$$\text{Time} = 2 \cdot (1 + 2 + 4 + \dots + 2^i + \dots + n).$$

When n is an exact power of 2 (the best case), $\text{Time} = 4n - 2$; when n is one more than an exact power of 2 (the worst case), $\text{Time} = 6n - 6$. Thus, the savings in worst-case message passages is paid for by an increase in the wall time.

We conjecture that models in which message passing is unidirectional must, in the worst case, have quadratic behavior and that bidirectional capability is necessary in order to achieve $O(n \log n)$ performance. Recently, Burns has shown [3] that $n \log n$ is asymptotically optimal.

Received 6/79; revised 5/80; accepted 7/80

References

1. Chang, E., and Roberts, R. An improved algorithm for decentralized extrema-finding in circularly configurations of processes. *Comm. ACM* 22, 5 (May 1979), 281-283.
2. LeLann, G. Distributed systems—Towards a formal approach. *Inform. Proc. 77*, North-Holland Pub. Co., 1977, Amsterdam, pp. 155-160.
3. Burns, J.E. A formal model for message passing systems. *Tech. Rep. No. 91*, Comput. Sci. Dept., Indiana Univ., May 1980.

Computer Architecture
and Systems

J.P. Hayes
Editor

Design of a LISP-Based Microprocessor

Guy Lewis Steele Jr. and
Gerald Jay Sussman
Massachusetts Institute of Technology

We present a design for a class of computers whose "instruction sets" are based on LISP. LISP, like traditional stored-program machine languages and unlike most high-level languages, conceptually stores programs and data in the same way and explicitly allows programs to be manipulated as data, and so is a suitable basis for a stored-program computer architecture. LISP differs from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of integers or bit fields of fixed size. An instruction set can be designed for programs expressed as trees of record structures. A processor can interpret these program trees in a recursive fashion and provide automatic storage management for the record structures.

We discuss a small-scale prototype VLSI microprocessor which has been designed and fabricated, containing a sufficiently complete instruction interpreter to execute small programs and a rudimentary storage allocator.

Key Words and Phrases: microprocessors, large-scale integration, integrated circuits, VLSI, list structure, linked lists, garbage collection, storage management, direct execution, high-level language architectures, interpreters, tail recursion, LISP, SCHEME

CR Categories: 4.13, 4.21 4.22, 4.34, 6.21, 6.22, 6.33

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

Authors' present addresses: G.L. Steele, Jr., Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213; G.J. Sussman, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139. © 1980 ACM 0001-0782/80/1100-0628 \$00.75.

1. Introduction

An idea which has increasingly gained attention is that computer architectures should reflect specific language structures to be supported. This is an old idea; one can see features in the machines of the 1960s intended to support Cobol, Fortran, Algol, and PL/I. More recently research has been conducted into architectures to support string or array processing as in SNOBOL or APL.

An older and by now well-accepted idea is that of the stored-program computer. In such a computer the program and the data reside in the same memory; that is, the program is itself data which can be manipulated as any other data by the processor. It is this idea which allows the implementation of such powerful and incestuous software as program editors, compilers, interpreters, linking loaders, debugging systems, etc.

One of the great failings of most high-level languages is that they have abandoned this idea. It is extremely difficult, for example, for a PL/I (Pascal, Fortran, Cobol...) program to manipulate PL/I (Pascal, Fortran, Cobol...) programs.

On the other hand, many of these high-level languages have introduced other powerful ideas not present in standard machine languages. Among these are (1) recursively defined, nested data structures; and (2) the use of functional composition to allow programs to contain expressions as well as (or instead of) statements. The LISP language, in fact, has both of these features. It is unusual among high-level languages in that it also explicitly supports the stored-program idea: LISP programs are represented in a standardized way as recursively defined, nested LISP data structures. By contrast with some APL implementations, for example, which allow programs to be represented as arrays of characters, LISP also reflects the structure of program expressions in the structure of the data which represents the program. (An array of APL characters must be *parsed* to determine the logical structure of the APL expressions represented by the array. Similar remarks apply to SNOBOL statements represented as SNOBOL strings.)

It is for this reason that LISP is often referred to as a "high-level machine language." As with standard stored-program machine languages, programs and data are made of the same stuff. In a standard machine, however, the "stuff" is a homogeneous, linear (ordered) vector of fixed-size bit fields; a program is represented as an ordered sequence of bit fields (instructions) within the overall vector. In LISP, the "stuff" is a heterogeneous, unordered set of records linked to form lists, trees, and graphs; a program is represented as a tree (a "parse tree" or "expression tree") of linked records (a subset of the overall set of records). Standard machines usually exploit the linear nature of the "stuff" through such mechanisms as indexing by additive offset and linearly advancing program counters. A computer based on LISP can similarly exploit tree structures. The counterpart of indexing is component selection; the counterpart of lin-

ear instruction execution is evaluation of expressions by recursive tree-walk.

Just as the "linear vector" stored-program-computer model leads to a variety of specific architectures, so does the "linked record" model. For concreteness we present here one specific architecture based on the linked record model which has actually been constructed.

2. List Structure and Programs

One of the central ideas of the LISP language is that storage management should be completely invisible to the programmer, so that he need not concern himself with the issues involved. LISP is an object-oriented language, rather than a value-oriented language. The LISP programmer does not think of variables as the objects of interest, bins in which values can be held. Instead, each data item is itself an object, which can be examined and modified, and which has an identity independent of the variable(s) used to name it.

In this section we discuss LISP data structures at the conceptual level; the precise form of LISP data objects is not of concern here. Later we will discuss specific representations within the machine. LISP data is collectively referred to as "S-expressions" ("S" for "symbolic"). 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 LISP data objects, notated by (recursively) notating the objects in order, between a set of parentheses and separated by blank space. 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 (LAMBDA (X) (CAR (CDR X))))
```

is a list of three things: the atomic symbol `DEFINE`, the atomic symbol `SECOND`, and another list of three things `LAMBDA`, `(X)`, and `(CAR (CDR X))`.

A convenient way to use lists to represent algebraic expressions is to use "Cambridge Polish" notation, essentially a parenthesized version of prefix Polish notation. Numeric constants are encoded as numeric atoms; variables are encoded as nonnumeric atoms (which henceforth we will call *symbols*); and procedure invocations (function calls) 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 (or, rather,

the structure of the lists) 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 "if p then x else y " we write:

```
(IF p x y)
```

Expressions are made into procedures (functions) by the use of Church's lambda-notation [5]. For example,

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

evaluates to a function of two arguments X and Y which computes $3 * Y + X$. The list of names of variables after the LAMBDA indicates how the names of variables in the expression are to be matched positionally to supplied arguments when the function is applied.

We can also encode recursive LISP programs as list data. For example, to compute N factorial ($N!$):

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

Suppose that we now want to write a LISP program which will take such a data structure and perform some useful operation on it, such as determining the value of an algebraic expression represented as a list structure. We need some procedures for categorizing, decomposing, and constructing LISP data.

The predicate ATOM, when applied to a LISP datum, 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 LISP datum. The predicate NUMBERP is true of numbers and false of symbols and lists. The predicate EQ, when applied to two symbols, is true if the two atomic symbols are identical. It is false when applied to two distinct symbols, or to a symbol and any other datum.

The decomposition operators for lists are traditionally called CAR and CDR for historical reasons. 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 Cs and Rs 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 any datum and a list, produces a new list whose car is the datum and whose cdr is the given list; that is, CONS adds a new element to the front of a list. The operator LIST can take any number of arguments (a special feature) and produces a list of its arguments.

Notice that CONS (and LIST) conceptually *create* new data structures. As far as the LISP programmer is

concerned, new data objects are available in endless supply. They can be conveniently called forth to serve some immediate purpose and discarded when they are no longer of use. While creation is explicit, discarding is not; a data object simply disappears into limbo when the program throws away all references (direct or indirect) to that object.

The immense freedom this gives the programmer may be seen by an example taken from current experience. A sort of error message familiar to most programmers is "too many nested DO loops" or "more than 200 declared arrays" or "symbol table overflow". Such messages typically arise within compilers or assemblers which were written in languages requiring data tables to be preallocated to some fixed length. The author of a compiler, for example, might well guess, "No one will ever use more than, say, ten nested DO loops; I'll double that for good measure and make the nested DO loop table 20 long." Inevitably, someone eventually finds some reason to write 21 nested DO loops and finds that the compiler overflows its fixed table and issues an error message (or, worse yet, *does not* issue an error message!). On the other hand, had the compiler writer made the table 100 long or 1,000 long, most of the time most of the memory space devoted to that table would be wasted.

A compiler written in LISP would be much more likely to keep a linked list of records describing each DO loop. Such a list could be grown at any time by creating a new record on demand and adding it to the list. In this way, as many or as few records as needed could be accommodated.

Now one could certainly write a compiler in any language and provide such dynamic storage management with enough programming. The point is that LISP provides automatic storage management from the outset and encourages its use (in much the same way that Fortran provides floating-point numbers and encourages their use, even though the particular processor on which a Fortran program runs may or may not have floating-point hardware).

Using CAR, CDR, and CONS, we can now write some interesting programs in LISP to deal with LISP data. For example, we can write a program APPEND, which when given two lists produces their concatenation as a new list:

```
(DEFINE APPEND
  (LAMBDA (X Y)
    (IF (NULL X) Y
        (CONS (CAR X) (APPEND (CDR X) Y)))))
```

Because LISP programs are represented as LISP data structures, 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 symbol "FOO". We might try writing

```
(EQ X FOO)
```

This does not work. The occurrence of "FOO" does not

refer to the 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* LISP datum as a constant in a program, but some data objects must be used to represent other things, such as variables and procedure invocations. To solve this problem we invent a new notation: (QUOTE *d*) in a program represents the constant datum *d*. Thus, we can write our test as “(EQ X (QUOTE FOO))”. Similarly,

```
(APPEND X (LIST Y Z))
```

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

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

appends to the value of *X* the constant list “(LIST Y Z)”. Because the QUOTE construction is used so frequently in LISP, we use an abbreviated notation: “‘FOO’” (“FOO” with a preceding quote) is equivalent to “(QUOTE FOO)”. This is only a notational convenience; the two notations denote the same list.

3. A LISP Interpreter

Here we exhibit an interpreter for a dialect of LISP, written in that dialect (this fact makes this interpreter *meta-circular* [28]—it can interpret itself, but of course must interpret certain constructs by using those constructs). This dialect is based on SCHEME [33, 38] and differs semantically from most other LISP dialects primarily in that free variables are lexically scoped (as in Algol and Pascal) rather than dynamically scoped (as in APL). Also, SCHEME was intentionally designed to be a *small* version of LISP, which makes it most suitable for implementation by hardware. Finally, SCHEME permits a correct implementation of *tail-recursion*; this means that function calls can be used to express true iteration, because calling a function is equivalent in all respects to a GO TO [13, 31, 32, 34].

```
(DEFINE EVAL
  (LAMBDA (EXP ENV)
    (IF ATOM EXP
      (IF (NUMBERP EXP) EXP
          (VALUE EXP ENV))
      (IF (EQ (CAR EXP) 'QUOTE)
          (CADR EXP)
      (IF (EQ (CAR EXP) 'LAMBDA)
          (LIST '&PROCEDURE
              (CADR EXP)
              (CADDR EXP)
              ENV)
          (IF (EQ (CAR EXP) 'IF)
              (IF (EVAL (CADR EXP) ENV)
                  (EVAL (CADDR EXP) ENV)
                  (EVAL (CADDDR EXP) ENV))
              (APPLY (EVAL (CAR EXP) ENV)
                      (EVLIS (CDR EXP) ENV)))))))
```

```
(DEFINE APPLY
  (LAMBDA (FUN ARGS)
    (IF (PRIMOP FUN)
      (PRIMOP-APPLY FUN ARGS)
      (IF (EQ (CAR FUN) '&PROCEDURE)
          (EVAL (CADR FUN)
              (BIND (CADR FUN)
                    ARGS
                    (CADDR FUN))))
      (ERROR))))))

(DEFINE EVLIS
  (LAMBDA (ARGLIST ENV)
    (IF (NULL ARGLIST) '()
      (CONS (EVAL (CAR ARGLIST) ENV)
              (EVLIS (CDR ARGLIST)
                      ENV)))))
```

The evaluator 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 (see below) to add to the lexical environment, a kind of symbol table, of the procedure, by 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 of the structure of the S-expression EXP. If it is an atom, there are two subcases. Numeric atoms evaluate to themselves. 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 LAMBDA form, an IF form, or a combination. For a QUOTE form, EVAL extracts the S-expression constant using CADR. LAMBDA forms evaluate to procedure objects (here represented as lists whose cars are the atom “&PROCEDURE”) containing the lexical environment and the “text” of the procedure definition. For an IF form, the predicate part is recursively evaluated; depending on whether the result is **true** or **false**, the consequent or alternative is selected for evaluation. For combinations, the procedure is obtained, the arguments evaluated (using EVLIS), and APPLY called as described earlier.

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 PRI-

MOP-APPLY which deals with the application of such primitive procedures. 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 actual procedure involved in the combination (CAR X) is not the atomic symbol CAR, but rather some bizarre object (the *value* of the atomic symbol CAR) which is meaningful only to PRIMOP-APPLY.

The interpreter uses several utility procedures for maintaining environments. An environment 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 that this representation is *not* the same as the association list ("a-list") representation traditionally used in LISP interpreters.) If a variable name occurs in more than one bucket, the most recently added such bucket has priority; in this way new symbol definitions added to the front of the list can supersede old ones. In this way the "shadowing" effect of lexical block structure is obtained. The code for manipulating environments is below.

```
(DEFINE BIND
  (LAMBDA (VARS ARGS ENV)
    (IF (= (LENGTH VARS)
          (LENGTH ARGS))
        (CONS (CONS VARS ARGS) ENV)
        (ERROR))))

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

(DEFINE VALUE1
  (LAMBDA (NAME SLOT
           (IF (EQ SLOT '&UNBOUND) (ERROR)
               (CAR SLOT))))))

(DEFINE LOOKUP
  (LAMBDA (NAME ENV)
    (IF (NULL ENV) '&UNBOUND
        (LOOKUP1
         NAME (CAAR ENV)
         (CDAR ENV) ENV))))

(DEFINE LOOKUP1
  (LAMBDA (NAME VARS VALS ENV)
    (IF (NULL VARS)
        (LOOKUP NAME (CDR ENV))
        (IF (EQ NAME (CAR VARS)) VALS
            (LOOKUP1
             NAME (CDR VARS)
             (CDR VALS) ENV))))))
```

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.)

```
(DEFUN EVAL-DISPATCH ()
  (COND ((ATOM EXP)
         (COND ((NUMBERP EXP)
                 (SETQ VAL EXP)
                 (POPJ-RETURN))
               (T
                 (SETQ VAL (VALUE EXP ENV))
                 (POPJ-RETURN))))
```

VALUE is essentially an interface to 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.

4. State-Machine Implementation

The LISP interpreter we have presented is recursive. It implicitly relies on a hidden control mechanism which retains the state information which must be saved for each recursive invocation. In this section we make this control information explicit. Below we present an interpreter in the form of a state machine controller. The controller manipulates a small set of registers and also issues commands to a list memory system. The recursion control information which is typically kept on a stack will be maintained in the linked-list memory.

This evaluator, written in LISP, has five global variables which are used to simulate the registers of a machine. EXP is used to hold the expression or parts of the expression under evaluation. ENV is used to hold the pointer to the environment structure which is the context of evaluation of the current expression. VAL is used to hold the value developed in evaluation of expressions. It is set whenever a primitive operator is invoked, or whenever a variable is evaluated, a quoted expression is evaluated, or a lambda expression is evaluated. ARGS is used to hold the list of evaluated arguments (the "actual parameters") being accumulated for a combination. Finally, CLINK is the pointer to the top of the list structure which is the control stack. (It is called "CLINK" for historical reasons stemming from CONNIVER [21] and "spaghetti stacks" [4].)

The style of coding here depends on "tail-recursion"; that is, iterative loops are implemented as patterns of function calls. When the last thing a function does is to call another function, then the effect is simply to perform a GO TO to the called function, passing arguments, if any, without pushing anything onto a stack, not even a return address (for in this situation it is not necessary to return [13, 32]).

EVAL-DISPATCH is the procedure which dispatches on the type of an expression—implementing the action of EVAL. When EVAL-DISPATCH is called, EXP contains an expression to be evaluated, ENV contains the environment for the evaluation, and the top element of CLINK is a "return address," i.e., the name of a function to call when the value has been determined and placed in VAL.

```
;If an atomic expression:
; numbers evaluate
; to themselves
; (i.e., are "self-quoting"),
; but symbols must be looked
; up in the environment.
```

((EQ (CAR EXP) 'QUOTE)	;If a QUOTE expression
(SETQ VAL (CADR EXP))	; extract the quoted constant
(POPJ-RETURN))	; and return it
((EQ (CAR EXP) 'LAMBDA)	;If a LAMBDA expression
(SETQ VAL (CADR EXP))	; get the formal parameters,
(SETQ EXP (CADDR EXP))	; get the body,
(SETQ VAL (LIST '&PROCEDURE VAL EXP ENV))	; and construct a closure
(POPJ-RETURN)	; which includes ENV.
((EQ (CAR EXP) 'IF)	;If a conditional,
(SETQ CLINK (CONS ENV CLINK))	; save the environment
(SETQ CLINK (CONS EXP CLINK))	; save the expression,
(SETQ CLINK (CONS 'EVIF-DECIDE CLINK))	;set up a return address,
(SETQ EXP (CADR EXP))	; then extract the predicate
(EVAL-DISPATCH))	; and evaluate it.
((NULL (CDR EXP))	;If a call with no arguments,
(SETQ CLINK (CONS 'APPLY-NO-ARGS CLINK))	; set up a return address,
(SETQ EXP (CAR EXP))	; get the function position
(EVAL-DISPATCH))	; and evaluate it.
(T	;Otherwise,
(SETQ CLINK (CONS ENV CLINK))	; save ENV,
(SETQ CLINK (CONS EXP CLINK))	; save EXP,
(SETQ CLINK (CONS 'EVARGS CLINK))	; set up return address,
(SETQ EXP (CAR EXP))	; get the function position
(EVAL-DISPATCH)))))	; and evaluate it.

When the process evolved by the evaluator has *finished* the evaluation of a subexpression, it must continue executing the rest of the expression. The place in the evaluator to continue executing was pushed onto CLINK when the evaluation of the subexpression was begun. This return address is now at the top of the CLINK, where it can be popped off and called:

(DEFUN POPJ-RETURN ()	
(SETQ EXP (CAR CLINK))	; Return to caller:
(SETQ CLINK (CDR CLINK))	; Save return address in EXP,
(FUNCALL EXP))	; and pop it off CLINK.
	; Transfer control.

After the predicate part of a conditional is evaluated, the process comes back to here to look at VAL to see whether the consequent or the alternative branch is to be taken. One of these is selected and is made the EXP to be further evaluated.

(DEFUN EVIF-DECIDE ()	
(SETQ EXP (CAR CLINK))	;Restore expression
(SETQ CLINK (CDR CLINK))	; and pop it off.
(SETQ ENV (CAR CLINK))	;Restore ENV
(SETQ CLINK (CDR CLINK))	; and pop it off.
(COND (VAL	;If predicate was true,
(SETQ EXP (CADDR EXP)))	; extract consequent.
(T	;Otherwise
(SETQ EXP (CADDR EXP)))))	; extract alternative.
(EVAL-DISPATCH))	;In either case, evaluate it.

The following procedures are the states which the evaluator must go through to evaluate the arguments to procedures before applying them. There is a special-case check in EVAL-DISPATCH for functions with no arguments. In this case, it is not necessary to save the state of the evaluator when evaluating the function position because there are no further arguments to evaluate. One may just apply the procedure which comes back in VAL. This is a case of “*evlis tail-recursion*” (see [39]). We will see this idea again in EVARGS1 where we have a special-case check for evaluation of the last argument.

(DEFUN APPLY-NO-ARGS ()	
(SETQ ARGS NIL)	
(SAPPLY))	;Set up null argument list
	; and apply function in VAL.

General argument evaluations come to EVARGS. This segment of the evaluator incorporates some cleverness in that it checks for the special case of the last argument in a combination. However, for the sake of clarity and uniformity we did not try to remove all unnecessary pushing and popping. There are many, more clever ways to write this code, as we will see later. The following procedure is the initialization of the argument evaluation loop.

```

(DEFUN EVARGS ( )
  (SETQ EXP (CAR CLINK))           ;Restore EXP
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SETQ ENV (CAR CLINK))           ;Restore ENV,
  (SETQ CLINK (CDR CLINK))         ; and pop it.
  (SETQ CLINK (CONS VAL CLINK))    ;Save function.
  (SETQ EXP (CDR EXP))             ;Get rid of function part.
  (SETQ ARGS NIL)                 ;Initialize argument list.
  (EVARGS1))                      ;Evaluate arguments.

```

The function EVARGS1 is the top of the argument evaluation loop.

```

(DEFUN EVARGS1 ( )
  (COND ((NULL (CDR EXP))          ;Is this the last argument?
    (SETQ CLINK (CONS ARGS CLINK)) ;If so, save argument list,
    (SETQ CLINK (CONS 'LAST-ARG CLINK)) ; set up return address,
    (SETQ EXP (CAR EXP))           ; set up last argument,
    (EVAL-DISPATCH))             ; and evaluate it.
    (T
      (SETQ CLINK (CONS ENV CLINK)) ; Otherwise,
      (SETQ CLINK (CONS EXP CLINK)) ; save ENV,
      (SETQ CLINK (CONS ARGS CLINK)) ; save EXP,
      (SETQ CLINK (CONS 'EVARGS2 CLINK)) ; save argument list,
      (SETQ CLINK (CONS 'EVARGS2 CLINK)) ; set up return address,
      (SETQ EXP (CAR EXP))           ; set up next argument,
      (EVAL-DISPATCH))))          ; and evaluate it.

```

The function EVARGS2 is where we end up after each argument is evaluated. The evaluated argument is accumulated into (consed onto) the list growing in ARGS.

```

(DEFUN EVARGS2 ( )
  (SETQ ARGS (CAR CLINK))          Restore argument list,
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SETQ EXP (CAR CLINK))           ;Restore EXP
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SETQ ENV (CAR CLINK))           ;Restore ENV,
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SETQ ARGS (CONS VAL ARGS))      ;Add value to argument list.
  (SETQ EXP (CDR EXP))             ;Flush form just evaluated.
  (EVARGS1))                      ;GO evaluate next argument.

```

When the last argument has been evaluated we come back here. The value is accumulated onto the ARGS and the function is restored from the stack. Execution continues at SAPPLY for application of the procedure to the evaluated arguments.

```

(DEFUN LAST-ARG ( )
  (SETQ ARGS (CAR CLINK))          ;Restore argument list,
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SETQ ARGS (CONS VAL ARGS))      ;Add last value to it.
  (SETQ VAL (CAR CLINK))           ;Retrieve function,
  (SETQ CLINK (CDR CLINK))         ; and pop it off.
  (SAPPLY))                       ;Apply function to arguments.

```

SAPPLY is the state machine analog of APPLY. This procedure checks out what kind of procedure is to be applied. If it is primitive, the appropriate actions occur. If it is a procedural closure, we evaluate the body of the closed procedure in an environment constructed by binding the formal parameters of the closed procedure to the actual parameters (in ARGS) in the environment carried in the closure.

```

(DEFUN SAPPLY ( )
  (COND ((PRIMOP? VAL)            ;Apply function in VAL to ARGS.
    (SETQ VAL (PRIMOP-APPLY VAL ARGS)) ;If a primitive procedure,
    (POPJ-RETURN))                 ; do it!
    ((EQ (CAR VAL) '&PROCEDURE)    ; then return value to caller.
      (SETQ ENV (SETQ CLINK (CDR CLINK))
        (BIND (CADR VAL)
          ARGS
          (CADDR VAL))))           ;If a defined procedure,
      (SETQ EXP (CADDR VAL))       ; set up its environment
      (EVAL-DISPATCH))            ; by binding the formals
      (T (ERROR)))                 ; to the actuals
      (SETQ EXP (CADDR VAL))       ; in the closure environment
      (EVAL-DISPATCH))            ; then get the procedure body
      (T (ERROR)))                 ; and evaluate it.
      (T (ERROR)))                 ; Otherwise, error.

```

In this state-machine code we have avoided functional composition. Each statement is an assignment or a conditional. (We have used the usual LISP COND conditional form, rather than IF, for reasons of convenience. This interpreter is not meta-circular. Instead, it is working MacLISP [25] code which implements a non-MacLISP version of LISP.) An assignment can contain at most one call to a storage management procedure such as CONS or CAR (we allow calls to, e.g., CADDR, realizing that (SETQ X (CADDR Y)) can be considered an abbreviation for the sequence (SETQ X (CDR Y)), (SETQ X (CDR X)), (SETQ X (CAR X))). Also, VALUE and BIND can be considered here as complex storage operations (defined essentially as before).

5. Representing List Data

Lists are normally represented by records, each of which contains two *pointers* to other records. One pointer is the car, and the other is the cdr. In this way a list (A (B C) D) can be visualized by the diagram in Figure 1. Atoms are represented as records of other types.

The exact representation of a pointer is not of interest here. All we really care about is that if we give the pointer to the memory system, it can return the contents of the record pointed to. (In particular, there is nothing at this level requiring the parts of a record to be "contiguous." Later we will discuss ways to represent LISP data within standard linear memories.)

In our particular architecture, we find it convenient to associate with each pointer a *type* field describing the nature of the record pointed to. (This use of typed pointers was inspired by the implementations of such languages as ECL [40] and MUDDLE (now called MDL) [8].) This type field can be exploited for other purposes as well; in particular, we shall use it to encode "opcodes" and "return addresses." We will say that the type field is a part of the pointer, and that the other part of the pointer (that which identifies another record) is the *address* part. The list shown above, with type fields added, is shown in Figure 2.

One efficiency problem with the version of the LISP interpreter given above is that the repeated consecutive tests for atoms, LAMBDA, IF, etc., take time. Conceptually what these tests perform is a dispatch on the syntactic category of the expression. Each expression is distinguished by a special symbol in the car position—except for atoms and procedure calls. The evaluator could be made faster and simpler if it could dispatch in a more uniform way.

Another efficiency problem is that LOOKUP must search for the values of variables. Because our dialect of LISP is lexically scoped like Algol, we can arrange for variable references to specify "*n* levels back, and then *j* over" in much the same way used by the Algol "display" technique, eliminating the search.

To allow these efficiencies, we systematically alter

Fig. 1. The list (A (B C) D).

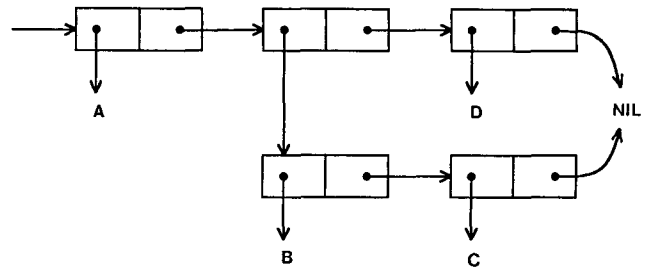


Fig. 2. Typed-pointer representation of (A (B C) D).

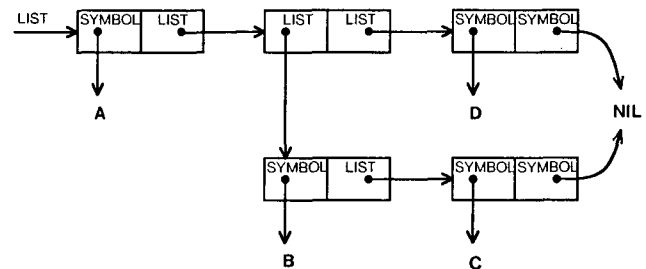
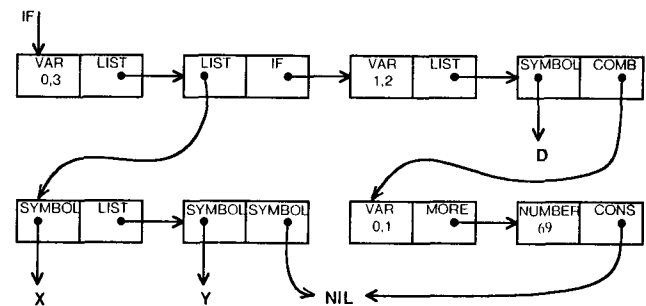


Fig. 3. Code for (IF A '(X Y) (IF C 'D (CONS E 69))).



the representation of programs. They will still be represented as trees of list records, but we encode the syntactic categories in the type fields of pointers to the expressions. EVAL can then simply dispatch on this type code. For a pointer whose type is "variable reference," we use the address part as "immediate data" indicating the values of *n* and *j* for the environment lookup. The representation of the following program fragment

(IF A '(X Y) (IF C 'D (CONS E 69)))

is illustrated in Figure 3.

Because variable references and other constructs have separate types, lists and symbols can be "self-evaluating" in the same way as numbers. Also, we assume that CONS is a "built-in" primitive operator and encode that operator in the type field at the end of the list representing the call to CONS. The encoding of IF forms has been changed to reduce space requirements; the car of a

conditional is the predicate, and the cdr is a cons cell whose car is the consequent and whose cdr is the alternative.

We ought, perhaps, to define a printed representation for all these new data types. We do not do this here, however. We assume that in practice one will write LISP code in the usual style, and a simple "compiler" program will transform it into the typed-pointer representation. (The typed-pointer representation is isomorphic to the original expression. It is easy for another program to reverse-compile a typed-pointer expression to reconstruct the original code. There is a problem with recovering the names of variables, but this can be taken care of by associating with each procedure a description of the names of its variables.)

To describe the evaluator for this new representation, we introduce a construct TYPE-DISPATCH (essentially a CASE statement on the type) which is not part of the LISP language, but which we use to indicate the underlying mechanism. The names in double quotes can be thought of as values of some Pascal-style enumeration type; they represent the distinct values which can occur in the type field. We also use a primitive operator TYPED-CONS, which creates a new listlike cell with a specified type. The new evaluator is very much like the old one: EVCOMB is sort of like EVLIS, combined with the entry point to APPLY. The function DISPLAY-BIND adds a new contour to an Algol-style display

structure; DISPLAY-LOOKUP accesses a value in such a display. (See Program below).

When a non-"MORE-ARGS" type code is seen in EVCOMB, it indicates that a primitive operation is to be performed on the argument values. EVCOMB then calls APPLY to perform this operation. (As shown here, APPLY needlessly duplicates the dispatching operation in EVCOMB; we have done this to exhibit the similarity of this interpreter to the previous one. Later we will remove this duplication.) One of these primitive operations, "FUNCALL", is used to invoke user-defined procedures (closures).

The type codes used to indicate primitive operations such as "CONS" may overlap those used to distinguish syntactic categories, because they are used in different contexts. Compare this to the way in which the same bits in an instruction can be used for different purposes depending on the opcode; for example, in the PDP-11 the same bits of an instruction word can be a register number, part of a branch offset, or condition code bits.

6. Combining These Ideas

The state-machine implementation of a LISP interpreter can be combined with the typed pointer dispatch idea to form a very efficient interpreter for LISP which can be easily implemented directly in hardware. We now

```
(DEFINE EVAL
  (LAMBDA (EXP ENV)
    (TYPE-DISPATCH EXP
      ("NUMBER" EXP)
      ("SYMBOL" EXP)
      ("LIST" EXP)
      ("VARIABLE" (DISPLAY-LOOKUP EXP ENV))
      ("PROCEDURE" (TYPED-CONS "CLOSURE" (CDR EXP) ENV))
      ("IF" (IF (EVAL (CAR EXP) ENV)
                (EVAL (CADR EXP) ENV)
                (EVAL (CDDR EXP) ENV)))
      ("COMBINATION" (EVCOMB (CDR EXP)
                              ENV
                              (TYPED-CONS "LIST"
                                           (EVAL (CAR EXP) ENV)
                                           '( ))))
      (OTHERWISE (ERROR))))))

(DEFINE EVCOMB
  (LAMBDA (EXP ENV ARGS)
    (TYPE-DISPATCH EXP
      ("MORE-ARGS" (EVCOMB (CDR EXP)
                           ENV
                           (TYPED-CONS "LIST"
                                         (EVAL (CAR EXP) ENV)
                                         ARGS)))
      (OTHERWISE (APPLY EXP ARGS)))))

(DEFINE APPLY
  (LAMBDA (FUN ARGS)
    (TYPE-DISPATCH FUN
      ("FUNCALL" (EVAL (CDAAR ARGS)
                       (DISPLAY-BIND (CDR ARGS)
                                      (CDAR ARGS))))
      ("CONS" (CONS (CADR ARGS) (CAR ARGS)))
      ("CAR" (CAAR ARGS))
      ("CDR" (CDAR ARGS))
      ("ATOM" (ATOM (CAR ARGS)))
      ...
      (OTHERWISE (ERROR)))))
```

present such an interpreter, written in a statement-oriented language to emphasize that we are describing a hardware interpreter. As before, the controller manipulates a small set of registers and also issues commands to a list memory system. The recursion-control information is, as before, stored in a push-down control list maintained in linked-list memory. Type fields in the cdr

pointers of the control list will be used to retain "return addresses" within the state machine; in this way, return addresses do not require any extra conses in the CLINK. (Compare this with the previous state-machine interpreter, which used separate tokens in the CLINK as return addresses.) This is possible because the set of return addresses is small.

```

BEGIN "EVALUATOR"
  DECLARE REGISTERS
    EXP      !GENERALLY HOLDS EXPRESSION BEING EVALUATED
    ENV      !HOLDS CURRENT ENVIRONMENT
    VAL      !RESULT OF EVALUATION; ALSO SCRATCH
    ARGS     !ACCUMULATES EVALUATED ARGUMENTS OF A COMBINATION
    CLINK    !"CONTROL LINK": RECURSION CONTROL STACK

  EVAL:     TYPE-DISPATCH ON EXP INTO
    "NUMBER": GOTO SELF
    "SYMBOL": GOTO SELF
    "LIST": GOTO SELF
    "VARIABLE": GOTO LOOKUP
    "PROCEDURE": GOTO PROC
    "IF": GOTO IF1
    "COMBINATION": GOTO EVCOMB
    HCTAPSID-EPYT

  SELF:     VAL := EXP; GOTO RETURN
  PROC:     VAL := TYPED-CONS("CLOSURE", EXP, ENV); GOTO RETURN
  IF1:      VAL := CDR(EXP)
    CLINK := CONS(ENV, CLINK)
    CLINK := TYPED-CONS("IF2", VAL, CLINK)
    EXP := CAR(EXP); GOTO EVAL
  !RECURSIVE EVALUATION OF PREDICATE RETURNS HERE
  IF2:      EXP := CAR(CLINK)
    CLINK := CDR(CLINK)
    ENV := CAR(CLINK)
    CLINK := CDR(CLINK)
    IF NULL(VAL)
      THEN EXP := CDR(EXP); GOTO EVAL
      ELSE EXP := CAR(EXP); GOTO EVAL
    FI

  EVCOMB:   ARGS := '()
  EVCOM1:   TYPE-DISPATCH ON EXP INTO
    "COMBINATION": GOTO EVCOM2
    "FUNCALL": GOTO CALL
    "CONS": GOTO CONS
    "CAR": GOTO CAR
    "CDR": GOTO CDR
    . . .

  EVCOM2:   HCTAPSID-EPYT
    CLINK := CONS(ENV, CLINK)
    CLINK := CONS(ARGS, CLINK)
    VAL := CDR(EXP)
    CLINK := TYPED-CONS("EVCOM3", VAL, CLINK)
    EXP := CAR(EXP); GOTO EVAL
  !RECURSIVE EVALUATION OF ARGUMENT RETURNS HERE
  EVCOM3:   EXP := CAR(CLINK)          !UNWIND STACK
    CLINK := CDR(CLINK)
    ARGS := CAR(CLINK)
    CLINK := CDR(CLINK)
    ENV := CAR(CLINK)
    CLINK := CDR(CLINK)
    ARGS := CONS(VAL, ARGS); GOTO EVCOM1

  CALL:     EXP := CAR(VAL)          !N.B. VAL = CAR(ARGS)
    VAL := CDR(VAL)
    ENV := CONS(ARGS, VAL); GOTO EVAL

  CONS:     ARGS := CDR(ARGS)        !I.E., ARGS := CADDR(ARGS)
    ARGS := CAR(ARGS)              !(ALREADY HAD VAL := CAR(ARGS), IN EFFECT)
    VAL := CONS(ARGS, VAL); GOTO RETURN

  CAR:      VAL := CAR(VAL); GOTO RETURN
  CDR:      VAL := CDR(VAL); GOTO RETURN
  . . .

  RETURN:   TYPE-DISPATCH ON CLINK INTO
    "IF2": GOTO IF2
    "EVCOM3": GOTO EVCOM3
    HCTAPSID-EPYT
  !SESOL ARTSKJID

END "EVALUATOR"

```

In *this* state-machine code we have avoided functional composition rigorously. Each statement is an assignment or a dispatch operation (IF-THEN-ELSE being a kind of dispatch). As assignment can contain at most one call to a simple storage management procedure such as CONS or CAR. Each statement goes to another statement (to the one textually following, if no GOTO clause is present).

We have omitted the details of the LOOKUP operation (it obtains the value from the environment and then goes to RETURN). We have, however, shown DISPLAY-BIND (beginning at CALL). These are not done as subroutines (as they were in the previous state-machine interpreter); they are coded "in-line" as state-machine code.

Recursive evaluation of subexpressions is handled by using an explicit stack. When for an IF or a COMBINATION a recursive evaluation is needed, any required registers (e.g., ENV) are consed onto the control structure CLINK. The last cons onto CLINK uses the type code to encode the "return address" (IF2 or EVCOM3) within the state machine. (These return address codes may be the same codes used as "opcodes" or "primitive operator codes"—this is a third, distinct context in which type bits are used for some funny purpose unrelated to the type of the data.) The expression to be recursively evaluated is put into EXP, and then state EVAL is entered. When the evaluation finishes, the code at RETURN decodes the type field of CLINK and resumes execution of the caller, which retrieves the saved information from CLINK and carries on. Thus, CLINK, though implemented as linked records, behaves as a stack.

7. Storage Management

A complete LISP system, as implied in the previous section, is conveniently divided into two parts: (1) a storage system, which provides an operator for the creation of new data objects and also other operators (such as pointer traversal) on those subjects; and (2) a program interpreter (EVAL), which executes programs expressed as data structures within the storage system. (Note that this memory/processor division characterizes the usual von Neumann architecture also. The differences occur in the nature of the processor and the memory system.)

Most hardware memory systems which are currently available commercially are not organized as sets of linked lists, but rather as the usual linearly indexed vectors. (More precisely, commercially available RAMs are organized as Boolean N-cubes indexed by bit vectors. The usual practice is to impose a total ordering on the memory cells by ordering their bit-vector addresses lexicographically, and then to exploit this total ordering by using indexing hardware typically containing an addition unit (or, more rarely, a subtraction unit, as on the IBM 7094).)

Commercially available memories are, moreover, available only in finite sizes (more's the pity). Now the

free and wasteful throw-away use of data objects would cause no problem if infinite memory were available, but within a finite memory it is an ecological disaster. In order to make such memories usable for our processor we must interpose between EVAL and the storage system a *storage manager* which makes a finite vector memory appear to the evaluation mechanism to be an infinite linked-record memory. This would seem impossible, and it is. The catch is that at no time may more records be *active* than will fit into the finite memory actually provided. The memory is "apparently infinite" in the sense that an indefinitely large number of new records can be "created" using the CONS operator. The storage manager *recycles* discarded records in order to create new ones in a manner completely invisible to the evaluator.

The storage manager therefore consists of routines which implement the operations CAR, CDR, CONS, etc. in terms of the vector memory, plus a *garbage collector* which deals with the finiteness of the memory by locating records which have been discarded and making them available to the CONS routine for recycling.

The method we use for implementing CAR, CDR, and CONS is the usual one of using two consecutive words of linear memory to hold a list cell, one being the car and the other the cdr, where each word of memory can hold a type field and an address field. The address part of a pointer is in turn the address within the linear memory of the record pointed to. (This may seem obvious, but remember that until now we have been non-committal about the precise representation of pointers, as until this point all that was necessary was that the memory system associate records with pointers by any convenient means whatsoever. The evaluator is completely unconcerned with the format or meaning of addresses; it merely accepts them from the memory system and eventually gives them back later to retrieve record components. One may think of an address as a *capability* for accessing a record using certain defined operations.)

Many techniques for garbage collection are well-documented in the literature [2, 6, 7, 10, 12, 17, 20, 23, 26, 29, 30] and will not be discussed here. Suffice it to say here that the storage manager can be implemented as a second state machine. The storage manager runs continuously, performing services for the evaluator. When the storage manager has completed a request, it then advances the evaluator to its next state and dispatches on the new request from the evaluator. The storage manager can also read from or write into the off-chip memory.

8. Our Prototype Microprocessor

In 1978 we used the ideas of the preceding sections to implement a prototype LISP evaluator in the form of a VLSI microprocessor. This prototype was originally called SIMPLE (Small Integrated MicroProcessor for

LISP Expressions), and later renamed SCHEME-78. This implementation was intended only to test the concepts involved and is not a complete working microprocessor. It has 11-bit words, divided into 3 type bits and 8 address bits.

Internally, the processor contains two state machines, EVAL and GC. Each machine has several registers (five for EVAL, four for GC) on a common bus (E bus, G bus). Each state machine is in the form of a read-only memory plus a "micro-PC" which encodes the current state. At each transition a state machine can read one register onto its bus, load one or more other registers from the bus, request some storage operation to occur, and enter some new state (possibly computed by dispatching on bits obtained from the bus). The storage operations requested by the EVAL machine are relayed to the GC machine, and requests from the GC machine are sent to the external memory system. The E bus and G bus can be connected together; this is done to pass pointers between EVAL and GC. Similarly, the G bus and pads can be connected; this is done to pass addresses and data between GC and the external memory.

When the chip is initialized, it takes a given expression and evaluates it in the global (null) environment. If that evaluation ever terminates, the result of the evaluation is stored in memory and the chip halts, with the evaluator looping in a dead state. The chip also halts if it runs out of memory (i.e., after consing the last of the 256 words), with the storage manager looping in a dead state (see below).

The type field can hold one of eight "op codes":

0 = constant list	4 = procedure
1 = constant symbol	5 = conditional (if-then-else)
2 = variable reference	6 = procedure call
3 = constant closure	7 = quoted constant

The evaluation of a type 0 (list), 1 (symbol), or 3 (closure) object simply results in that object; such objects are "self-evaluating." (Notice that symbols are *not* the same as variables here; this usage has been "compiled out.")

The evaluation of type 7 (quote) returns the cdr of the object. In this way any object whatsoever, not just a list, symbol, or closure, can be included as a constant datum in a program. (Given this operation, opcodes 0, 1, and 3 are functionally redundant; however, they are faster and occur frequently in programs. Also, these codes distinguish between types of data when used as data types *per se*.)

The evaluation of type 2 (variable reference) treats the address part of the pointer as a two's-complement number, which is the negative of the number of times to cdr the current environment; the car of this is then the value of the variable. (To simplify the lookup hardware (at the expense of speed) environments are maintained as a simple flat list of values; the contours are concatenated rather than chained.)

The evaluation of type 4 (procedure) results in a

pointer to a newly allocated word pair. This pointer has type 3 (closure). The car of the pair contains the cdr of the procedure; this is the code body of the procedure. The cdr of the pair contains the current environment (the environment within which the procedure object is being evaluated). In this way the code and the environment are bound up together (as a closure or "FUNARG" [27]) for later application. The car of the procedure object is not used by the evaluator. The compiler can use this slot to store documentation about the procedure, such as the name of the procedure, the name of the file it was read from (for the text editor), the names of its variables (for the decompiler and run-time debugger), etc.

A conditional (type 5) points to a two-word cell, the cdr of which points to another two-word cell. The car of the conditional object is a predicate expression (IF), the cadr is a consequent expression (THEN), and the caddr is an alternative expression (ELSE). The predicate expression is evaluated first; depending on whether the result is non-NIL or NIL, then the consequent or alternative is evaluated, throwing away the other one, to produce the value of the conditional.

A procedure call (type 6) is the most complicated of the lot. It is a list of indefinite length, chained together by cdr pointers. Each cdr pointer except the last must have type 0 (list). The last cdr pointer should have a zero address and a nonzero type. This last type specifies the operation to be performed. In cdring down the list, SIMPLE evaluates each of the expressions in the car, saving the resulting values. These values are available as arguments to the operation to be performed. The operations available are:

0 = (more arguments)	4 = ATOM
1 = CAR	5 = PROGN
2 = CDR	6 = LIST
3 = CONS	7 = FUNCALL

For operations CAR, CDR, and ATOM there should be one argument; for CONS, two. No checking is performed for this. For PROGN, LIST, and FUNCALL there may be any nonzero number of arguments.

CAR, CDR, CONS, ATOM, and LIST are the standard LISP primitive operations. LIST is actually a REVERSE-LIST, because it produces a list of the arguments in reverse order (the implementation is trivial, taking advantage of the fact that the interpreter must cons up a list of the arguments anyway—but this implementation happens to produce the list in a reversed order). This matters only if the calculations of the arguments have side effects which could interfere with each other. PROGN is a standard LISP primitive which evaluates any number of arguments and returns only the last one. This is useful only when side effects are used. Even though no primitive side effects are provided in the prototype instruction set, PROGN was included to test that idea.

FUNCALL is the operation which calls a user pro-

As an example, an expression calling for the evaluation of the expression

including the definition of APPEND itself is shown in Figure 4.

- (1) pass operand to storage manager and request CAR;
- (2) retrieve result of storage operation.

- (1) pass the cdr part to storage manager;
- (2) pass the car part and request CONS;
- (3) retrieve result.

In point of fact, in the prototype processor, the storage manager includes no garbage collector. The prototype was one project of a "project set" including some two dozen separate circuits, all of which had to be fit onto a single chip together. This imposed severe area limitations which restricted the address size to eight bits and required the elimination of the microcode for the garbage collector. However, space for microcode was the *only* limitation; all the other necessary structures for garbage collection were in the design. The complexity of a full garbage collector is comparable to that of the evaluator shown above.

The diagram illustrates a linked list structure for a Lisp interpreter. The list contains nodes for SYMBOL, LIST, CALL, CONS, and other symbols, with pointers to documentation and other parts of the system. A legend defines VAR symbols: VAR -3 = X, VAR -2 = Y, VAR -1 = APPEND.

```

graph TD
    CALL --> N1
    subgraph N1 [ ]
        LIST1[LIST]
        MORE1[MORE]
    end
    MORE1 --> N2
    subgraph N2 [ ]
        S1[SYMBOL]
        L1[LIST]
    end
    L1 --> N3
    subgraph N3 [ ]
        S2[SYMBOL]
        L2[LIST]
    end
    L2 --> N4
    subgraph N4 [ ]
        S3[SYMBOL]
        S4[SYMBOL]
    end
    S4 --> NIL1[NIL]
    S3 --> N5
    subgraph N5 [ ]
        S5[SYMBOL]
        L3[LIST]
    end
    L3 --> N6
    subgraph N6 [ ]
        S6[SYMBOL]
        S7[SYMBOL]
    end
    S7 --> NIL2[NIL]
    S6 --> N7
    subgraph N7 [ ]
        P[PROC]
        F[FUNCAL]
    end
    F --> NIL3[NIL]
    P --> N8
    subgraph N8 [ ]
        I[IF]
    end
    I --> N9
    subgraph N9 [ ]
        C[CALL]
        L4[LIST]
    end
    L4 --> N10
    subgraph N10 [ ]
        V1[VAR -3]
        A[ATOM]
    end
    A --> NIL4[NIL]
    V1 --> N11
    subgraph N11 [ ]
        C2[CALL]
        M1[MORE]
    end
    M1 --> N12
    subgraph N12 [ ]
        C3[CALL]
        C4[CONS]
    end
    C4 --> NIL5[NIL]
    C3 --> N13
    subgraph N13 [ ]
        V2[VAR -3]
        C5[CAR]
    end
    C5 --> NIL6[NIL]
    V2 --> N14
    subgraph N14 [ ]
        C6[CALL]
        M2[MORE]
    end
    M2 --> N15
    subgraph N15 [ ]
        V3[VAR -2]
        M3[MORE]
    end
    M3 --> N16
    subgraph N16 [ ]
        V4[VAR -1]
        F[FUNCAL]
    end
    F --> NIL7[NIL]
    V4 --> N17
    subgraph N17 [ ]
        V5[VAR -3]
        C6[CDR]
    end
    C6 --> NIL8[NIL]
    V5 --> N18
    subgraph N18 [ ]
        V6[VAR -1]
        F2[FUNCAL]
    end
    F2 --> NIL9[NIL]
    V6 --> N19
    subgraph N19 [ ]
        V7[VAR -2]
        C7[CALL]
    end
    C7 --> N20
    subgraph N20 [ ]
        V8[VAR -3]
        C8[CALL]
    end
    C8 --> N21
    subgraph N21 [ ]
        V9[VAR -2]
        C9[CALL]
    end
    C9 --> N22
    subgraph N22 [ ]
        V10[VAR -3]
        C10[CALL]
    end
    C10 --> N23
    subgraph N23 [ ]
        V11[VAR -2]
        C11[CALL]
    end
    C11 --> N24
    subgraph N24 [ ]
        V12[VAR -3]
        C12[CALL]
    end
    C12 --> N25
    subgraph N25 [ ]
        V13[VAR -2]
        C13[CALL]
    end
    C13 --> N26
    subgraph N26 [ ]
        V14[VAR -3]
        C14[CALL]
    end
    C14 --> N27
    subgraph N27 [ ]
        V15[VAR -2]
        C15[CALL]
    end
    C15 --> N28
    subgraph N28 [ ]
        V16[VAR -3]
        C16[CALL]
    end
    C16 --> N29
    subgraph N29 [ ]
        V17[VAR -2]
        C17[CALL]
    end
    C17 --> N30
    subgraph N30 [ ]
        V18[VAR -3]
        C18[CALL]
    end
    C18 --> N31
    subgraph N31 [ ]
        V19[VAR -2]
        C19[CALL]
    end
    C19 --> N32
    subgraph N32 [ ]
        V20[VAR -3]
        C20[CALL]
    end
    C20 --> N33
    subgraph N33 [ ]
        V21[VAR -2]
        C21[CALL]
    end
    C21 --> N34
    subgraph N34 [ ]
        V22[VAR -3]
        C22[CALL]
    end
    C22 --> N35
    subgraph N35 [ ]
        V23[VAR -2]
        C23[CALL]
    end
    C23 --> N36
    subgraph N36 [ ]
        V24[VAR -3]
        C24[CALL]
    end
    C24 --> N37
    subgraph N37 [ ]
        V25[VAR -2]
        C25[CALL]
    end
    C25 --> N38
    subgraph N38 [ ]
        V26[VAR -3]
        C26[CALL]
    end
    C26 --> N39
    subgraph N39 [ ]
        V27[VAR -2]
        C27[CALL]
    end
    C27 --> N40
    subgraph N40 [ ]
        V28[VAR -3]
        C28[CALL]
    end
    C28 --> N41
    subgraph N41 [ ]
        V29[VAR -2]
        C29[CALL]
    end
    C29 --> N42
    subgraph N42 [ ]
        V30[VAR -3]
        C30[CALL]
    end
    C30 --> N43
    subgraph N43 [ ]
        V31[VAR -2]
        C31[CALL]
    end
    C31 --> N44
    subgraph N44 [ ]
        V32[VAR -3]
        C32[CALL]
    end
    C32 --> N45
    subgraph N45 [ ]
        V33[VAR -2]
        C33[CALL]
    end
    C33 --> N46
    subgraph N46 [ ]
        V34[VAR -3]
        C34[CALL]
    end
    C34 --> N47
    subgraph N47 [ ]
        V35[VAR -2]
        C35[CALL]
    end
    C35 --> N48
    subgraph N48 [ ]
        V36[VAR -3]
        C36[CALL]
    end
    C36 --> N49
    subgraph N49 [ ]
        V37[VAR -2]
        C37[CALL]
    end
    C37 --> N50
    subgraph N50 [ ]
        V38[VAR -3]
        C38[CALL]
    end
    C38 --> N51
    subgraph N51 [ ]
        V39[VAR -2]
        C39[CALL]
    end
    C39 --> N52
    subgraph N52 [ ]
        V40[VAR -3]
        C40[CALL]
    end
    C40 --> N53
    subgraph N53 [ ]
        V41[VAR -2]
        C41[CALL]
    end
    C41 --> N54
    subgraph N54 [ ]
        V42[VAR -3]
        C42[CALL]
    end
    C42 --> N55
    subgraph N55 [ ]
        V43[VAR -2]
        C43[CALL]
    end
    C43 --> N56
    subgraph N56 [ ]
        V44[VAR -3]
        C44[CALL]
    end
    C44 --> N57
    subgraph N57 [ ]
        V45[VAR -2]
        C45[CALL]
    end
    C45 --> N58
    subgraph N58 [ ]
        V46[VAR -3]
        C46[CALL]
    end
    C46 --> N59
    subgraph N59 [ ]
        V47[VAR -2]
        C47[CALL]
    end
    C47 --> N60
    subgraph N60 [ ]
        V48[VAR -3]
        C48[CALL]
    end
    C48 --> N61
    subgraph N61 [ ]
        V49[VAR -2]
        C49[CALL]
    end
    C49 --> N62
    subgraph N62 [ ]
        V50[VAR -3]
        C50[CALL]
    end
    C50 --> N63
    subgraph N63 [ ]
        V51[VAR -2]
        C51[CALL]
    end
    C51 --> N64
    subgraph N64 [ ]
        V52[VAR -3]
        C52[CALL]
    end
    C52 --> N65
    subgraph N65 [ ]
        V53[VAR -2]
        C53[CALL]
    end
    C53 --> N66
    subgraph N66 [ ]
        V54[VAR -3]
        C54[CALL]
    end
    C54 --> N67
    subgraph N67 [ ]
        V55[VAR -2]
        C55[CALL]
    end
    C55 --> N68
    subgraph N68 [ ]
        V56[VAR -3]
        C56[CALL]
    end
    C56 --> N69
    subgraph N69 [ ]
        V57[VAR -2]
        C57[CALL]
    end
    C57 --> N70
    subgraph N70 [ ]
        V58[VAR -3]
        C58[CALL]
    end
    C58 --> N71
    subgraph N71 [ ]
        V59[VAR -2]
        C59[CALL]
    end
    C59 --> N72
    subgraph N72 [ ]
        V60[VAR -3]
        C60[CALL]
    end
    C60 --> N73
    subgraph N73 [ ]
        V61[VAR -2]
        C61[CALL]
    end
    C61 --> N74
    subgraph N74 [ ]
        V62[VAR -3]
        C62[CALL]
    end
    C62 --> N75
    subgraph N75 [ ]
        V63[VAR -2]
        C63[CALL]
    end
    C63 --> N76
    subgraph N76 [ ]
        V64[VAR -3]
        C64[CALL]
    end
    C64 --> N77
    subgraph N77 [ ]
        V65[VAR -2]
        C65[CALL]
    end
    C65 --> N78
    subgraph N78 [ ]
        V66[VAR -3]
        C66[CALL]
    end
    C66 --> N79
    subgraph N79 [ ]
        V67[VAR -2]
        C67[CALL]
    end
    C67 --> N80
    subgraph N80 [ ]
        V68[VAR -3]
        C68[CALL]
    end
    C68 --> N81
    subgraph N81 [ ]
        V69[VAR -2]
        C69[CALL]
    end
    C69 --> N82
    subgraph N82 [ ]
        V70[VAR -3]
        C70[CALL]
    end
    C70 --> N83
    subgraph N83 [ ]
        V71[VAR -2]
        C71[CALL]
    end
    C71 --> N84
    subgraph N84 [ ]
        V72[VAR -3]
        C72[CALL]
    end
    C72 --> N85
    subgraph N85 [ ]
        V73[VAR -2]
        C73[CALL]
    end
    C73 --> N86
    subgraph N86 [ ]
        V74[VAR -3]
        C74[CALL]
    end
    C74 --> N87
    subgraph N87 [ ]
        V75[VAR -2]
        C75[CALL]
    end
    C75 --> N88
    subgraph N88 [ ]
        V76[VAR -3]
        C76[CALL]
    end
    C76 --> N89
    subgraph N89 [ ]
        V77[VAR -2]
        C77[CALL]
    end
    C77 --> N90
    subgraph N90 [ ]
        V78[VAR -3]
        C78[CALL]
    end
    C78 --> N91
    subgraph N91 [ ]
        V79[VAR -2]
        C79[CALL]
    end
    C79 --> N92
    subgraph N92 [ ]
        V80[VAR -3]
        C80[CALL]
    end
    C80 --> N93
    subgraph N93 [ ]
        V81[VAR -2]
        C81[CALL]
    end
    C81 --> N94
    subgraph N94 [ ]
        V82[VAR -3]
        C82[CALL]
    end
   
```

The evaluator and the storage manager are each implemented in the same way as an individual processor. Each processor has a state-machine controller and a set of registers. On each clock cycle the state-machine outputs control signals for the registers and also makes a transition to a new state.

The processors can communicate with each other by causing the E and G busses to be connected. The address and type parts of the busses can be connected separately. (Typically, the E bus might have its address part driven from the G bus and its type part driven by a constant supplied by the evaluator controller. This is used to do the TYPED-CONS operation.) The G bus can also be connected to the address/data lines for the off-chip memory system. The storage-manager controller produces additional signals (ADR and WRITE) to control

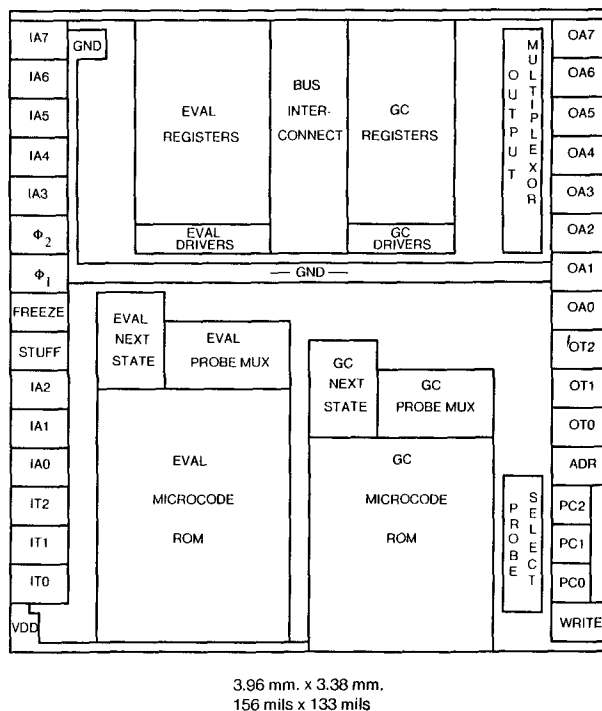
the external memory. In a similar manner, the evaluator controller produces signals which control the storage manager. (Remember that from the point of view of the evaluator, the storage manager *is* the memory interface!)

Each controller effectively has an extra "state register" which may be thought of as its "micro-PC". At each step the next state is computed by combining its current state with external signals in the following manner. Each "microinstruction" has a field explicitly specifying the next desired state, as well as bits specifying possible modifications of that state. If specified, external signals are logically OR'd into the desired state number. In the prototype evaluator these external signals are: (1) the type bits from the E bus; (2) a bit which is 1 iff the E bus type field is zero and a bit which is 1 iff the E bus address is zero. In the storage manager these signals are: (1) the four control bits from the evaluator controller; (2) a bit which is 1 iff the G bus address is zero. This is the way in which dispatching is achieved.

Once this new state is computed, it is passed through a three-way selector before entering the state register. The other two inputs to the selector are the current state and the data lines from the external memory system. In this way the selector control can "freeze" a controller in its current state by recirculating it or jam an externally supplied state into the state register (both useful for debugging operations). The "freeze" mechanism is used by the storage manager to suspend the evaluator until it is ready to process the next request. In the same way, the external memory can suspend the storage manager by asserting the external FREEZE signal, thereby causing a "wait state."

(The FREEZE signal is provided as a separate control because the dynamic logic techniques usual in NMOS were used; if one stopped the processor simply by stopping the clock, the register contents would dissipate. The clocks must keep cycling in order to "refresh" the registers. The state recirculation control allows the machine to be logically stopped despite the fact that data is still circulating internally. We discovered that this technique imposed constraints on other parts of the design: The incrementation logic is the best example. It was originally intended to design an incrementing counter register, which would increment its contents in place during a clock cycle in which an "increment" signal was asserted. If this had been done, however, and the processor was frozen during an instruction which asserted this signal, the counter would continue to count while the processor was stopped! This could have been patched by having the FREEZE signal override the increment signal, but it was deemed simpler to adopt a design strategy in which nothing at the microcode level called for any data to be read, modified, and stored back into the same place. Thus, in the actual design one must read data through modification logic and then onto the bus, to be stored in a different register; then if this operation is repeated many times because of the FREEZE signal, it makes no difference.)

Fig. 5. Physical layout of prototype LISP processor.



Each state-machine controller consists of a read-only memory (implemented as a programmed-logic-array), two half-registers (clocked inverters, one at each input and one at each output), and some random logic (e.g., for computing the next state). The controllers are driven by externally supplied two-phase nonoverlapping clock signals; on phase 1 the registers are clocked and the next state is computed, and on phase 2 the next-state signals appear and are latched.

All of the signals from the two controllers ($62 = 34 + 28$ in the prototype) are multiplexed onto twelve probe lines by six unary probe-control signals. (These signals are derived from three binary-encoded off-chip signals.) When a probe-control signal is asserted, the memory output pads (11 data pads plus the ADR signal in the prototype) are disconnected from the G bus and connected to the twelve probe lines. In this way the chip can be frozen and then all controller outputs verified (by cycling the probe-control signals through all six states). Also recall that the controller states can be jammed into the state registers from the memory input pads. This allows the controller microcode to be tested completely without depending on the registers and busses working.

Figure 5 shows the physical layout of the prototype chip. The two controllers are side by side, with the evaluator on the left and the storage manager on the right. Above each controller is the next-state logic and probe multiplexor for that controller. Above those are the register arrays, with the busses running horizontally through them. The bus connections are in the center.

The input pads are on the left edge, and the output pads on the right edge. The input pads are bussed through the evaluator's register array parallel to the E bus lines, so that they can connect to the G bus. (Unfortunately, there was no time to design tri-state pads for this project.)

10. Discussion

A perhaps mildly astonishing feature of this computer is that it contains no arithmetic-logic unit (ALU). More precisely, it does have arithmetic and logical capabilities, but the arithmetic units can only add 1, and the logical units can only test for zero. (Logicians know that this suffices to build a "three-counter machine," which is known to be as universal (and as convenient!) as a Turing Machine [24]. However, our LISP architecture is also universal and considerably more convenient.)

LISP itself is so simple that the interpreter needs no arithmetic to run interesting programs (such as computing symbolic derivatives and integrals, or pattern matching). All the LISP interpreter has to do is shuffle pointers to and from memory and occasionally dispatch on the type of a pointer. The incrementation logic is included on the chip for two reasons. In the evaluator it is used for counting down a list when looking up lexical variables in the environment; this is not really necessary, for there are alternative environment representation strategies. In the storage manager incrementation is necessary (and, in the prototype, sufficient) for imposing a total ordering on the external memory, so as to be able to enumerate all possible addresses. The only reason for adding 1 is to get to the next memory address. (One might note that the arithmetic properties of general two-argument addition are not exploited here. *Any* bijective mapping from the set of external memory addresses onto itself (i.e., a permutation function) would work just fine (but the permutation should contain only one cycle if memory is not to be wasted!). For example, subtracting 1 instead of adding, or Gray-code incrementation, would do.)

This is not to say that real LISP programs never use arithmetic. It is just that the LISP interpreter itself does not require binary arithmetic of the usual sort. (It does, however, require CONS, CAR, and CDR, which in a formal sense indeed form a kind of "number system" [18], where CONS corresponds to "add 1" and both CAR and CDR to "subtract 1." In this view, the purpose of the storage manager is to interface between two kinds of arithmetic, namely "LISP arithmetic" and Peano arithmetic. This suggests one way to perform variable lookups without indexing or even arithmetic counting: A variable reference would be encoded as a list whose *length* indicates how far back to look in the environment. The lookup procedure would cdr the reference list and the environment together; when the one ran out, the car of the other would be the desired value.)

This architecture is intended to use devices which are

addressed as memory, in the same manner used by the PDP-11, for example. We envision having a set of devices on the external memory bus which do arithmetic. One would then write operands into specific "memory locations" and then read arithmetic results from others. Such devices could be very complex processors in themselves, such as specialized array or string processors. In this way the LISP computer could serve as a convenient controller for other processors, for one thing LISP does well is to provide recursive control and environment handling without much prejudice (or expertise!) as to the data being operated upon.

Expanding on this idea, one could arrange for additional signals to the external memory system from the storage manager, such as "this data item is needed (or not needed)," which would enable external processors to accomplish their own storage management cooperatively with the LISP processor. One might imagine, for example, an APL machine which provided tremendous array processing power, controlled by a LISP interpreter specifying which operations to perform. The APL machine could manage its own array storage, using a relatively simple storage manager cued by "mark" signals from the LISP storage manager.

The possibility of additional processors aside, this architecture exhibits an interesting layered approach to machine design. One can draw boundaries at various places such that everything above the boundary is a processor which treats everything below the boundary as a memory system with certain operations. If the boundary is drawn between the evaluator and the storage manager, then everything below the boundary together constitutes a list-structure memory system. If it is drawn between the storage manager and the external memory, then everything below the boundary is the external memory. Supposing the external memory to be a cached virtual memory system, then we could draw boundaries between the cache and main memory, or between main memory and disks, and the same observation would hold. At the other end of the scale, a complex database management system could be written in LISP, and then the entire LISP chip (plus some software, perhaps in an external ROM) would constitute a memory system for a database query language interpreter. In this manner we have a layered series of processors, each of which provides a more sophisticated memory system to the processor above it in terms of the less sophisticated memory system below it.

Another way to say this is that we have a hierarchy of data abstractions, each implemented in terms of a more primitive one. Thus, the storage manager makes a finite, linear memory look "infinite" and tree-structured. A cache system makes a large, slow memory plus a small, fast memory look like a large, fast memory.

Yet another way to view this is as a hierarchy of interpreters running in virtual machines. Each layer implements a virtual machine within which the next processor up operates.

It is important to note that we may choose *any* boundary and then build everything below it in hardware and everything above it in software. Our LISP system is actually quite similar to those before it, except that we have pulled the hardware boundary much higher. One can also put different layers on different chips (as with the LISP chip and its memory). We choose to put the evaluator and the storage manager on the same chip only because (a) they fit, and (b) in the planned full-scale version, the storage manager would need too many pins as a separate chip.

Each of the layers in this architecture has much the same organization: It is divided into a controller ("state machine") and a database ("registers"). There is a reason for this. Each layer implements a memory system, and so has *state*; this state is contained in the database (which may be simply a small set of references into the next memory system down). Each layer also accepts commands from the layer above it and transforms them into commands for the layer below it; this is the task of the controller.

We have already mentioned some of the analogies between a LISP-based processor and a traditional processor. Corresponding to indexing there is component selection; corresponding to a linearly advancing program counter there is recursive tree-walk of expressions. Another analogy we might draw is to view the instruction set as consisting of variable-length instructions (whose pieces are joined by pointers rather than being arranged in sequential memory locations). Each instruction (variable reference, call to CONS, call to user function, etc.) takes a number of operands. We may loosely say that there are three *addressing modes* in this architecture: immediate data, as with a quoted constant; lookup of a named value, as in a variable reference; and recursive evaluation. In the last case, merely referring to an operand automatically calls for the execution of an entire routine to compute it!

11. Conclusions

We have presented a general design for and a specific example of a new class of hardware processors. This model is "classical" in that it exhibits the stored-program, program-as-data idea, as well as the processor/memory dichotomy which leads to the so-called "von Neumann bottleneck" [1]. It differs from the usual stored-program computer in organizing its memory differently and in using an instruction set based on this memory organization. Whereas the usual computer treats memory as a linear vector and executes a linear instruction stream, the architecture we present treats memory as linked records and executes a tree-shaped program by recursive expression evaluation.

The processor described here is not to be confused with the "LISP Machine" designed and built at MIT by

Greenblatt and Knight [9, 16, 19, 41]. The current generation of LISP Machine is built of standard Schottky TTL, and its hardware is organized as a very general-purpose microprogrammed processor of the traditional kind. It has a powerful arithmetic-logic unit and a large writable control store. Almost none of the hardware is specifically designed to handle LISP code; it is the microcode which customizes it for LISP. Finally, the LISP Machine executes a compiled order code which is of the linearly advancing PC type; the instruction set deals with a powerful stack machine. Thus, the LISP Machine may be thought of as a hybrid architecture that takes advantage of linear vector storage organization and stack organization, as well as linked-list organization. In contrast, the class of processors we present here is organized purely around linked records, especially since the instruction set is embedded in that organization. The LISP Machine is a well-engineered machine for general-purpose production use and so uses a variety of storage-management techniques as appropriate. The processor described here is instead intended as an illustration of the abstracted essence of a single technique, with as little additional context or irrelevant detail as possible.

We have designed and fabricated SCHEME-78, a prototype LISP-based processor. The actual hardware design and layout was completed by one of the authors (Steele) as a term project for a course on VLSI given at MIT by L. Conway in the fall of 1978. The prototype processor has a small but complete expression evaluator and an incomplete storage manager (everything but the garbage collector). It also has a fatal artwork error (akin to a typographical error—three small pieces of metal were accidentally omitted), which prevented any significant testing of the fabricated parts. Further information on this specific design, including a complete logic diagram, may be found in [36]; see also [35].

In 1979 we (with Holloway and Bell [14]) designed and fabricated a full-scale VLSI processor (known as SCHEME-79) having a complete garbage collector, with more built-in primitive operations, and a more complex storage representation. This processor has 32-bit words, divided into 7 type bits, 24 address bits, and 1 bit reserved for the garbage collector. It has more powerful arithmetic and logical capabilities: It can subtract 1 as well as add 1 and can test for equality rather than merely test for zero. Testing of this chip is not complete; however, it has successfully executed a few nontrivial programs and survived several garbage collections. Preliminary indications are that it interprets pure LISP code at approximately the same speed as a DEC PDP-10 model KA10 processor.

We intend to design and fabricate further versions of this architecture. The SCHEME-79 chip was designed with no regard for speed, and we estimate that careful attention to electrical considerations can improve performance severalfold. Algorithmic and data structure improvements (such as "CDR-coding" [9, 11], "dreaming" [37], and "phantom stacks" [30a]) will probably gain

another factor of four. Thus, we estimate a potential overall improvement of 10 to 20, making this interpretive architecture competitive with compiled code for more traditional computers.

A final philosophical thought: It may be worth considering kinds of "stuff" other than vectors and linked records to use for representing data. For example, in LISP we generally organize the records only into trees rather than general graphs. Other storage organizations should also be explored. The crucial idea, however, is that the instruction set should then be fit into the new storage structure in some natural and interesting way, thereby representing programs in terms of the data structures. Continuing the one example, we might look for an evaluation mechanism on general graphs rather than on trees, or on whatever other storage structure we choose. Finally, the instruction set, besides being represented in terms of the data structures, must include means for manipulating those structures. Just as the usual computer has ADD and AND, just as the LISP architecture presented here must supply CAR, CDR, and CONS, so a graph architecture must provide graph manipulation primitives, etc. Following this paradigm we may discover yet other interesting architectures and interpretation mechanisms.

Acknowledgments. We are very grateful to L. Conway for coming to MIT, teaching the techniques for NMOS design, and providing an opportunity for us to try our ideas as part of the course project chip. The text used for the course was written by C. Mead and L. Conway (a 1978 draft of [22]). Additional material was written by B. Hon and C. Sequin [15]. It should be mentioned that the course enabled a large number of students to try interesting and imaginative LSI designs as part of the project chip. This paper describes only one project of the set, but many of these student projects may have useful application in the future.

P. Penfield and J. Allen made all this possible by organizing the LSI design project at MIT and by arranging for C. Botchek and L. Conway to teach.

C. Botchek provided our first introduction to the subject and started our wheels spinning.

The course and project chip were executed with the cooperation, generosity, and active help of the Xerox Palo Alto Research Center (PARC), who provided software and design support; Micromask, who generated the masks; and Hewlett-Packard, who fabricated the wafers at their Deer Creek facility.

D. Lyon and A. Bell of Xerox PARC performed plots of the projects and assembled the projects into the final mask specifications. They were of particular direct aid to Steele in debugging this project.

G. Miranker and W. Henke maintained the plotting software used at MIT to produce intermediate plots of student projects during the design cycle, and they were also helpful in making modifications to the software to accommodate this project.

D. Antoniadis of MIT packaged and bonded the chips in 40-pin DIPs. Prof. Antoniadis was also a source of good advice concerning device physics and fabrication.

P. Deutsch and F. Corbato were kind enough to hand-carry project plots from California to Boston to help meet the project deadline.

T. Knight and J. Holloway provided useful suggestions and sound engineering advice, as usual. (In particular, Knight helped Steele to design a smaller pad to reduce the area of the project, and Holloway suggested the probe multiplexor technique for testing internal signals.)

We thank the referees for their comments and encouragement.

G. Steele's graduate studies during 1978-1980 were supported by a Fannie and John Hertz Fellowship. In the spring of 1978 his studies were supported by a National Science Foundation Graduate Fellowship.

Received 11/79; revised 5/80; accepted 7/80

References

(Note: [3] is not cited in the text.)

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM* 21, 8 (Aug. 1978), 613-641.
2. Baker, H. B., Jr. List processing in real time on a serial computer. *Comm. ACM* 21, 4 (April 1978), 280-294.
3. Berkeley, E. C., and Bobrow, D. G., Ed. *The Programming Language LISP: Its Operation and Applications*. Inform. Internat., Inc., Cambridge, Mass. 1964.
4. Bobrow, D. G., and Wegbreit, B. A model and stack implementation of multiple environments. *Comm. ACM* 16, 10 (Oct. 1973), 591-603.
5. Church, A. The Calculi of Lambda Conversion. In *Annals of Mathematics Studies*, Number 6, Princeton Univ. Press, Princeton, N.J., 1941 (reprinted by Klaus Reprint Corp., N.Y., 1965).
6. Conrad, W. R. A compactifying garbage collector for ECL's non-homogeneous heap. Tech. Rep. 2-74, Ctr. for Res. in Computing Technology, Harvard Univ. Cambridge, Mass., Feb. 1974.
7. Deutsch, L. P., and Bobrow, D. G. An efficient, incremental, automatic garbage collector. *Comm. ACM* 19, 9 (Sept. 1976), 522-526.
8. Galley, S. W., and Pfister, G. The MDL language. Programming Technology Division Document SYS.11.01, Project MAC, MIT, Cambridge, Mass., Nov. 1975.
9. Greenblatt, R. The LISP machine. Working Paper 79, MIT Artif. Intell. Lab., Cambridge, Mass., Nov. 1974.
10. Gries, D. An exercise in proving parallel programs correct. *Comm. ACM* 20, 12 (Dec. 1977), 921-930.
11. Hansen, W. J. Compact list representation: Definition, garbage collection, and system implementation. *Comm. ACM* 12, 9 (Sept. 1969), 499-507.
12. Hart, T. P., and Evans, T. G. Notes on implementing LISP for the M-460 computer. In *The Programming Language LISP: Its Operation and Applications*, E. C. Berkeley and D. G. Bobrow, Eds. Inform. Internat., Inc., Cambridge, Mass., 1964, pp. 191-203.
13. Hewitt, C. Viewing control structures as patterns of passing messages. *Artif. Intell. J.* 8, 3 (June 1977), 323-364.
14. Holloway, J., Steele, G. L., Jr., Sussman, G. J., and Bell, A. The SCHEME-79 chip. Memo 559, MIT Artif. Intell. Lab. Cambridge, Mass., Jan. 1980.
15. Hon, R., and Sequin, C. *A Guide to LSI Implementation*. Xerox PARC, Palo Alto, Calif., Sept. 1978.
16. Knight, T. The CONS microprocessor. Working Paper 80, MIT Artif. Intell. Lab., Cambridge, Mass., Nov. 1974.
17. Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.
18. Levin, M. Mathematical logic for computer scientists. TR-131, Project MAC, MIT, Cambridge, Mass., June 1974.

19. The LISP Machine Group: Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. LISP machine progress report. Memo 444, MIT Artif. Intell. Lab, Cambridge, Mass., Aug. 1977.
20. McCarthy, J., et al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1962.
21. McDermott, D.V., and Sussman, G.J. The CONNIVER reference manual. Memo 295a, MIT Artif. Intell. Lab., Cambridge, Mass., Jan. 1974.
22. Mead, C., and Conway, L. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass., 1980.
23. Minsky, M.L. A LISP garbage collector using serial secondary storage. Memo 58 (revised), MIT Artif. Intell. Lab., Cambridge, Mass., Dec. 1963.
24. Minsky, M.L. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, N.J., 1967.
25. Moon, D.A. MacLISP reference manual, revision 0. Project MAC, MIT, Cambridge, Mass., April 1974.
26. Morris, F.L. A time- and space-efficient garbage compaction algorithm. *Comm. ACM* 21, 8 (Aug. 1978), 662-665.
27. Moses, J. The function of FUNCTION in LISP. Memo 199, MIT Artif. Intell. Lab., Cambridge, Mass., June 1970.
28. Reynolds, J.C. Definitional interpreters for higher order programming languages. *Proc. ACM Nat. Conf.*, Boston, Mass., 1972, pp. 717-740.
29. Saunders, R.A. The LISP system for the Q-32 computer. In *The Programming Language LISP: Its Operation and Applications*, E.C. Berkeley and D.G. Bobrow, Eds., Inform. Internat., Inc., Cambridge, Mass., 1964, pp. 220-231.
30. Schorr, H., and Waite, W.M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM* 10, 8 (Aug. 1967), 501-506.
- 30a. Stallman, R.M. Phantom Stacks: If you look too hard, they aren't there. MIT Artif. Intell. Lab. Memo. No. 556, Cambridge, Mass., July 1980.
31. Steele, G.L., Jr. Compiler optimization based on viewing LAMBDA as Rename plus Goto. S.M. Th., MIT, Cambridge, Mass., May 1977.
32. Steele, G.L., Jr. Debunking the "expensive procedure call" myth. *Proc. ACM Nat. Conf.* Seattle, Wash., Oct. 1977, pp. 153-162, revised as Memo 443, MIT Artif. Intell. Cambridge, Mass., Oct. 1977.
33. Steele, G.L. Jr., and Sussman, G.J. The revised report on SCHEME: A dialect of LISP. Memo 452, MIT Artif. Intell. Lab., Cambridge, Mass., 1978.
34. Steele, G.L. Jr., and Sussman, G.J. The art of the interpreter; or, the modularity complex (parts zero, one, and two). Memo 453, MIT Artif. Intell. Lab., Cambridge, Mass., Jan. 1978.
35. Steele, G.L., Jr., and Sussman, G.J. Storage management in a LISP-based processor. *Proc. Caltech Conf. on Very Large-Scale Integration*, Pasadena, Calif., Jan. 1979, 227-241.
36. Steele, G.L., Jr., and Sussman, G.J. Design of LISP-based processors; or, SCHEME: A dielectric LISP; or, finite memories considered harmful; or, LAMBDA: The ultimate opcode. Memo 514, MIT Artif. Intell. Lab, Cambridge, Mass., March 1979.
37. Steele, G.L., Jr., and Sussman, G.J. The dream of a lifetime: A lazy scoping mechanism. Memo 527, MIT Artif. Intell. Lab., Cambridge, Mass., Nov. 1979.
38. Sussman, G.J., and Steele, G.L. Jr. SCHEME: An interpreter for extended lambda calculus. Memo 349, MIT Artif. Intell. Lab, Cambridge, Mass., Dec. 1975.
39. Wand, M. Continuation-based program transformation strategies. Tech. Rep. 61, Comp. Sci. Dept., Indiana Univ. Bloomington, Indiana, March 1977; *J. ACM* 27, 1 (Jan. 1980), 164-180.
40. Wegbreit, B., et al. ECL programmer's manual. Ctr. for Res. in Comptng. Technology, Harvard Univ., Cambridge, Mass., Dec. 1974.
41. Weinreb, D., and Moon, D. LISP machine manual (preliminary version). MIT Artif. Intell. Lab., Cambridge, Mass., Nov. 1978.

Systems Modeling and Performance Evaluation H.D. Schwetman
Editor

Disk Scheduling: FCFS vs. SSTF Revisited

Micha Hofri
Technion—Israel Institute of Technology

We report on a rather extensive simulation effort directed at evaluating the merits of two scheduling strategies, FCFS and SSTF, for moving-arm disks under stationary request arrival process. For First-Come-First-Served (FCFS) scheduling, analytic results for the mean waiting time are also given (in a closed form). If the objective of a schedule is to minimize the mean waiting time (or queue size) and its variance, the results seem to confirm the overall superiority of Shortest-Seek-Time-First (SSTF), particularly for medium and heavy traffic. This holds also when the input is highly correlated or addresses the cylinders nonuniformly. These results contradict some statements published in recent years. The domain of policies where SSTF is optimal is considered. The simulation methodology is described in some detail.

Key Words and Phrases: moving-arm disk, scheduling, FCFS, SSTF, correlated input, simulation, optimal schedule

CR Categories: 4.35, 6.35, 8.1, 8.3

1. Introduction

Moving-arm disks have been with us, as secondary storage devices, for a long time. They also do not seem to be on the way out. They are commonly expected to play a major role in (particularly, large to medium) computing systems for some time to come. All this is by way of explaining the great amount of effort which has been expended thus far on studies aimed at improving the methods of using these devices (see [4, 6, 8] and further references therein).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's present address: M. Hofri, Dept. of Computer Science, Technion, Israel Institute of Technology, Haifa, Israel.
© 1980 ACM 0001-0782/80/1100-0645 \$00.75.