THE LISP 2 PROGRAMMING LANGUAGE AND SYSTEM *

Paul W. Abrahams

Information International, Inc., New York

Jeffrey A. Barnett, Erwin Book, Donna Firth,
Stanley L. Kameny, Clark Weissman

System Development Corporation, Santa Monica, California

and

Lowell Hawkinson, Michael I. Levin, Robert A. Saunders

Information International, Inc., Los Angeles, California

INTRODUCTION

LISP 2 is a new programming language designed for use in problems that require manipulation of highly complex data structures as well as lengthy arithmetic operations. Presently implemented on the AN/FSQ–32V computer at the System Development Corporation in Santa Monica, California, LISP 2 has two components: the language itself, and the programming system in which it is embedded. The system programs that define the language are accessible to and modifiable by the user; thus the user has an unparalleled ability to shape the language to suit his own needs and to utilize parts of the system as building blocks in constructing his own programs.

While it provides these capabilities to the do-it-yourself programmer, LISP 2 also provides the complete and convenient programming facilities of a ready-made system. Typical application areas for LISP 2 include heuristic programming, algebraic manipulation, linguistic analysis and machine translation of natural and artificial languages, analysis of particle reactions in high-energy physics, artificial intelligence, pattern recognition, mathematical logic and automata theory, automatic theorem proving, game-playing, information retrieval, numerical computation, and exploration of new programming technology.

The primary source materials on LISP 2 are the LISP 2 Primer,[1] which provides an introduction to the language for those with little or no programming experience, and the LISP 2 Reference Manual,[2] which provides a complete specification of the language.

The LISP 2 programming system provides not only a compiler, but also a large collection of run-time facilities. These facilities include the library functions, a monitor for control and on-line interac-

* Produced by SDC and III in performance of contract AF 19(628)–5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency, Information Processing Techniques Office, and Subcontract 65–107.

tion, automatic storage management, and communication with the monitor system of the machine on which the system is operating.

A particularly important part of the program library is a group of programs for bootstrapping LISP 2 onto a new machine. (Bootstrapping is the standard method for creating a LISP 2 system on a new machine.) The bootstrapping capability is sufficiently powerful so that the new machine requires no resident programs other than the standard monitor system and a binary loader.

LISP 2 includes and extends the capabilities of its ancestor, LISP 1.5.[3] LISP 1.5 has been notable for its mathematical elegance and symbol-manipulating capabilities. It is unique among programming languages in the ease with which programs can be treated as data, in its "garbage collection" approach to reclaiming unused storage, and in its ability to represent programs organized as a collection of small, easily understood function definitions. Full recursion without special user provisions is a natural outgrowth of the structure of the language. However, LISP 1.5 lacks a convenient input language and efficiency in the treatment of purely arithmetic operations.

LISP 2 was designed to maintain the advantages of LISP 1.5 while remedying its deficiencies. The first major change has been the introduction of two distinct language levels: Source Language (SL) and Intermediate Language (IL). The two languages have different syntaxes but the same semantics (in the sense that for every SL program there is a computationally equivalent IL program). The syntax of SL resembles that of ALGOL 60,[4] while the syntax of IL resembles that of LISP 1.5. IL is designed to have the same structure as data, and thus to be capable of being manipulated easily by user (and system) programs. An advantage of the ALGOL-like source language is that the ALGOL algorithms can be utilized with little change.

The second major change has been the introduction of type declarations and new data types, including integer-indexed arrays and character strings. At a future time, packed data tables, which can presently be simulated through programming techniques, will be added. Type declarations are necessary to obtain efficient compiled code, particularly for arithmetic operations, but by using the default mechanisms, a programmer may omit type declarations entirely (albeit at the cost of efficiency).

The third major change has been the introduction of partial-word extraction and insertion operators. Further, an IL-level macro expansion capability has been included, which makes possible the definition of operations in terms of a basic set of open-coded primitives. These changes made it possible to write the entire system in its own language without loss of efficiency. At the same time, the compilations of user programs are more economical in time, and to some extent in space, than they would be without these facilities. Furthermore, the knowledgeable user can trade space against time through appropriate redefinition of system functions.

A fourth major change, the introduction of pattern-driven data manipulation facilities, along the lines of COMIT [5] and METEOR,[6] is still in the process of implementation. Because of the open-ended nature of LISP 2, these facilities can be added without disrupting the existing system structure. We mention this facility here, despite the fact that it does not yet exist, because it is an integral part of the over-all design of the language. Since the specifications are not final as of this writing, however, we shall not discuss them further.

To orient the reader toward the exposition of the language, we present a short example at this point. Further examples will be given later. The following program [7] is written in SL:

```
% RANDOM COMPUTES A RANDOM
  NUMBER IN THE INTERVAL (A, B)
OWN INTEGER Y;
REAL FUNCTION RANDOM(A,B);
    REAL A,B;
BEGIN Y←3125*Y;
      Y←Y\67108864;
        RETURN (Y/67108864.0 * (B–
            A)+A)
END;
```

The only significant difference between this program and the ALGOL original is the use of the reverse slash "\" to indicate the computation of the remainder. The corresponding program in IL is:

```
(DECLARE (Y OWN INTEGER))
(FUNCTION (RANDOM REAL)
  ((A REAL) (B REAL))
  (BLOCK NIL (SET Y (TIMES 3125 Y))
    (SET Y (REMAINDER Y 67108864))
    (RETURN (PLUS (TIMES (QUOTIENT
        Y 6.7108864000E+7)
      (DIFFERENCE B A )) A))))
```

The process of converting SL programs into compiled code is shown in Fig. 1. SL is first translated into IL by syntax translator. IL is then translated

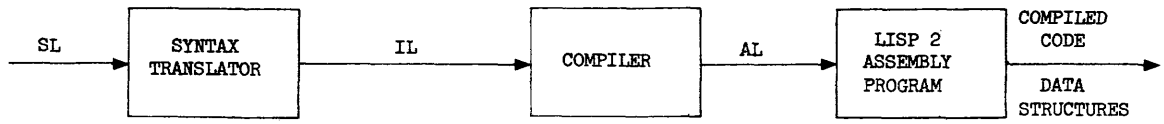

Figure 1. System organization. SL = source language; IL = intermediate language; AL = assembly language.

into assembly language by a compiler. Finally, the assembly language is translated into machine language by an assembly program. The process is entirely accessible to the user, in that he can write programs in IL or assembly language if he so chooses.

The remainder of this paper is divided into two parts, one dealing with the language and the other with the implementation. Certain aspects of the language that were intended primarily as implementation tools, e.g., open subroutines, are discussed in connection with the implementation.

In discussing the language, we shall present simultaneous discussions of the syntax of SL and IL, accompanied by discussion of the semantics of both. In this way the semantic equivalence of SL and IL will become apparent. It should be borne in mind that the primary use of SL is for programs written by people, while the primary use of IL is for programs written by machines. Thus the syntax of SL is designed for convenience in writing, while the syntax of IL is designed to reflect in its form the structure of the program that it represents.

THE LISP 2 LANGUAGE

Tokens

Tokens are the smallest units of input or output data with which LISP 2 programs ordinarily deal and are significant because of their role in defining the standard input/output conventions with regard to both programs and data. The major categories of tokens are:

1. Delimiters
2. Numbers
3. Simple strings
4. Identifiers
5. Operators

The delimiter tokens are:

( ) [ ] cr

Numbers as tokens may be either signed or unsigned in IL, but must be unsigned in SL since a preceding sign is interpreted as an operator. Some examples of unsigned numbers are:

unsigned integer	1	2	3E5	
unsigned octal	12Q	14Q6		
unsigned real	.87	12.	4.5E5	2.E–10

Signed numbers are like these, but are preceded by a sign. Other examples of tokens are:

identifier	AB	H21	GO.TO
operator	* / = ↑ >= \ + ←		

A string consists of a sequence of characters delimited at each end by "#". The character " ' " inside a string causes the character following to be entered in the string. Some examples of strings are:

#AB(C)D#
#A'#256'##
#ISN' 'T#

An identifier may be created from a string by preceding it with the escape character. This character is changeable within the system but will usually be "%." If "%" is the escape character, the following is an identifier:

%#AB(C)D#

An identifier created in this way is said to have an "unusual spelling," since, in general, such identifiers will be created only when they cannot be written in any other way unambiguously.

Data

The most general form of a LISP 2 datum is an S-expression, where the S stands for "symbolic." S-expressions are built up from atoms, which may be numbers, strings, identifiers, function specifiers, and arrays. As in LISP 1.5, the class of S-expressions is defined recursively as follows:

1. Every atom is an S-expression.
2. If e_1 and e_2 are S-expressions, then

$$(e_1 \ . \ e_2)$$

is an S-expression. Thus, for instance,

((A . B) . (C . D))

is an S-expression.

S-expressions of the form:

$(e_1 . (e_2 . \ldots . (e_n . NIL) \ldots))$

are known as lists, and can be written in the abbreviated form:

$(e_1\ e_2 \ldots e_n)$

The e_i are called the elements of the list. The two notations may be intermixed; thus

((A . 1) (B . 2) . . . (Z . 26))

is an S-expression in the form of a list, but the elements of the list are not themselves in the form of lists. The atom NIL can also be written in the form ( ), and designates the empty list.

The LISP functions CAR, CDR, and CONS are defined by:

CAR applied to $(e_1 . e_2)$ yields e_1
CDR applied to $(e_1 . e_2)$ yields e_2
CONS applied to e_1 and e_2 yields $(e_1 . e_2)$

In terms of the list notation, CAR finds the first element of a list and CDR removes the first element from a list. Thus CAR applied to the list (A B C D) yields A, and CDR applied to the same list yields the list (B C D). CDR applied to a list of one element yields the empty list ( ). The function NULL has value TRUE for the empty list ( ) (also represented as NIL) and value FALSE for anything else. The function CONS of two arguments can be used to add an element at the head of a list; thus CONS applied to the element A and the list (B C D) yields the list (A B C D). CONS is the basic operator used for constructing lists.

IL programs are written in the form of S-expressions, and therefore can be treated as data. The ability to treat programs as data in a natural way is an essential feature of LISP. SL programs can also be treated as data, because of the existence of strings; however, this is not nearly so natural as it is with IL.

Arrays are atoms because CAR and CDR are not defined for them. Constant arrays are written by enclosing their elements in brackets. For example:

[2 5 –1 4]

is a one-dimensional array of integers, and:

[[A B C] [A1 B1 C1] [A2 B2 C2] [A3 B3 C3]]

is a two-dimensional array of S-expressions.

Data Types. Although every LISP 2 datum is an S-expression, it is useful to pick out certain subsets of the set of all S-expressions and to designate these subsets by data type names. The data type names and the subsets they denote are:

BOOLEAN	Truth value data, represented by TRUE and FALSE. The empty list ( ), the atom NIL, and the Boolean value FALSE are regarded as synonymous.
INTEGER	Signed integers.
OCTAL	Another form of integer, basically regarded as unsigned, that prints in an octal output format.
REAL	Floating-point decimals.
FUNCTIONAL	LISP 2 function.
SYMBOL	The entire set of S-expressions. Strings and identifiers must be of this type.
type ARRAY	An array whose elements are of the specified type, where type is either BOOLEAN, INTEGER, OCTAL, REAL, FUNCTIONAL, or SYMBOL.

The different data types are not mutually exclusive, in that the class of data of type SYMBOL includes all other classes of data. Except for SYMBOL, all of the data classes include atomic data only.

Expressions

An expression is a designation of a datum. The datum designated by an expression is the value of the expression. The elementary components from which expressions are built up are constants, variables, and operational forms. We shall first discuss these, and then show how they are combined to form more complex expressions.

Constants. A constant is a datum appearing in a program context that denotes itself, i.e., its representation is both its name and its value. Consequently, a constant cannot change value during the execution of a program. A symbolic constant is denoted by a quoted S-expression. In SL, an S-expression is quoted by preceding it with a prime, e.g., 'ALPHA or '(L1 L2). In IL, an S-expression is quoted by preceding it with QUOTE in a list, e.g., (QUOTE ALPHA) or (QUOTE(L1 L2)). Quotation is necessary for identifiers and lists to prevent them from being interpreted as variables or operational forms.

Variables. A variable is also an elementary designation of a datum. However, the value of a variable may be changed during the execution of a program. A variable is normally denoted by a single identifier. Associated with every variable is a collection of bindings, each of which is a location containing a value. Bindings are created by declarations, which may appear in blocks, in functions, or on the supervisor level (see below). Blocks and functions are the two different kinds of program units. At execution time, a program unit may be activated either by the supervisor or by another program unit; thus there is a hierarchy of active program units.

When execution of a program unit commences, a binding is created for each variable declared by the program unit. When execution of the program unit is completed, these bindings disappear. Thus, each active program unit has a set of bindings associated with it, and the hierarchy of bindings corresponds to the hierarchy of active program units. In general, the value of a variable is the value attached to the most recently created and still existing binding of that variable. It is possible to use an assignment action to change the value associated with the current binding of a variable.

Associated with every variable is a type, a storage mode, and a transmission mode. The type of a variable restricts but does not necessarily determine the types of the data that are its values at different times. In particular, a variable whose type is SYMBOL may assume values of any type whatsoever.

There are three storage modes for variables: fluid, own, and lexical. A fluid variable can be referred to from outside the program unit that binds it, while a lexical variable cannot. Thus, fluid variables are more general but are also more prone to conflicts of names. Fluid variables are primarily used as a means of communication among separately compiled programs. An own variable is like a fluid variable except that only one binding can exist for it, and that binding must be made by a supervisor action. Own variables are designed primarily for communication with non-LISP 2 programs.

A variable may designate a datum either directly or indirectly. If the variable designates the datum directly, then it designates the actual value of the datum; if the variable designates the datum indirectly, then it designates the location in which the value is stored. This distinction is significant chiefly when a datum is being passed as an argument to a function; the transmission mode of the argument variable indicates whether a value or a location of a value is being passed. If a location is being passed, then the transmission mode is said to be locative; otherwise the transmission mode is said to be by value.

Operational Forms. An operational form is used to apply a function to its arguments, to invoke a macro transformation, to alter the flow of a program, or to locate an element of an array. An operational form in SL is written:

$$f(e_1, e_2, \ldots, e_n)$$

where f is the form operator and the e_i are its operands. In IL the operational form is written as:

$$(f\ e_1\ e_2 \ldots e_n)$$

If the form operator designates a function, then to obtain the value of the operational form, the operands are first evaluated, and then the function is applied to the values so obtained. An array is handled similarly; the subscripts are treated as arguments of a function that finds the desired element of the array.

Each function has associated with it a value type and a set of argument types. Any argument that is not of the expected type is converted to that type when the conversion is legal. The value type restricts the type of the result of the evaluation in the same way that the type of a variable restricts the values that the variable may assume.

In general, the order of evaluation of the operands of an operational form is not guaranteed. This is a departure from most other problem-oriented languages, but leads to improved compiled code. Also, with the advent of parallel processing computers it may be desirable to have several arguments evaluated simultaneously. If evaluating an operand has any side effect on the evaluation of any other operand, then the results of the evaluations will be unpredictable. However, the operator ORDER applied to an operational form will cause the operands to be evaluated in order of appearance.

Macros may be used to effect transformations of a program after it has been translated from SL to IL and before it has been compiled. When a macro name appears as a form operator, the effect at compile time is to cause the entire operational form to be replaced by a new form. The new form is calculated by a function associated with the macro; the argument of this function is the IL version of the operational form. Much of the task of compilation is

achieved through the use of macros that are invisible to the user; however, the user can also define his own macros. The use of macros is discussed further in connection with the user control facilities of the compiler.

Other Expressions. Elementary expressions (i.e., constants, variables, and operational forms) may be combined in SL by means of prefix and infix operators. Thus, all of the usual arithmetic and Boolean expressions are permitted in the usual algebraic notation. The symbolic operators CAR and CDR are also prefixes, which help to reduce the accumulation of parentheses. If a, b, and c are any expressions in SL, the relational expression:

$$a < b < c$$

and all similar forms have the same meaning in SL as they do in mathematics. Any number of relational operators can be combined in a relational expression, and different operators can be used in the same expression.

Infix and prefix operators cannot be used in IL, and must be replaced by corresponding operational forms. For example:

A * B + 3 — ALPHA ↑ 2

is written in IL as:

(PLUS (TIMES A B) 3 (MINUS (EXPT ALPHA 2)))

A similar notation is used for relational expressions. Conditional expressions in SL have the form:

IF p_1 THEN e_1 ELSE IF p_2 THEN e_2 ELSE ... IF p_n THEN e_n ELSE e_{n+1}

The final ELSE clause need not be included (unlike ALGOL). The corresponding form in IL is:

(IF p_1 e_1 p_2 e_2 . . . p_n e_n e_{n+1})

The p_i, which are Boolean expressions, are evaluated in turn from left to right until a true one is found. The value of the corresponding e_i is then used as the value of the entire expression. Conditional expressions have the useful property that evaluation proceeds only as far as necessary to determine the outcome.

A block expression is a block (see below) that appears in a context where an expression is required. A block expression is used to write a program as a sequence of statements to be executed and ultimately to produce a value. The value of a block is ordinarily specified by a RETURN statement (see below). LISP 2 differs from ALGOL in permitting a block to be an expression as well as a statement.

A CASE expression is written in the form:

CASE(s, e_1, e_2 . . . , e_n)

in SL, and in the IL form:

(CASE s e_1 e_2 . . . e_n)

where s is an integer-valued expression known as the selector. If the value of the selector s lies in the range $1 \leq s \leq n$, then the expression e_s is evaluated and is the value of the CASE expression. If $s < 1$ or $s > n$, the value is e_n.

An assignment expression is written in the form:

v ← e

in SL and in the form:

(SET v e)

in IL.

If v is a variable and e is an expression, the assignment expression has the effect of evaluating the expression e and assigning its value to v. The value of the entire expression is the value of e. Assignment expressions, like all other expressions, can be used as arguments in operational forms; in particular, they can be nested to achieve simultaneous assignment of value to several variables.

The general form of the left side of an assignment expression is a locative expression. A locative expression designates a part of a data structure or variable structure. A variable is a particular case of a locative expression. Locative expressions can be used to designate the current binding of a variable, an element of an array, part of a list structure, or particular bits of a word of memory. Thus, the two assignments:

A ← '(MARY DOE) ;
CAR A ← 'JOHN ;

will cause the value of A to become:

(JOHN DOE)

Blocks and Statements

A block may be either a block expression, a block statement, or a compound statement. All three of these are written in the same form and are evaluated in the same way. Whether a block is a block expres-

sion, a block statement, or a compound statement depends on both the context of the block and what is contained within the block.

In SL, a block is written in the form:

BEGIN d_1; d_2; . . . d_k; s_1; s_2; . . . s_n END

where the d_i are block declarations and the s_i are statements. Each block declaration specifies one or more internal parameters, which are variables that are bound while the block is active. The corresponding form in IL is:

(BLOCK(d_1 d_2 . . . d_k) s_1 s_2 . . . s_n)

A statement is an action to be taken. Any expression (other than a variable) can be used as a statement, but not every statement can be used as an expression. When an expression appears in a context where a statement is expected, the expression is evaluated, but the value is discarded. A statement may have one or more labels associated with it; these are referred to in GO statements (see below) and indicate where to transfer control. Variables can not be statements because of the conflict with labels.

When evaluation of a block begins, bindings are simultaneously created for each internal parameter specified by a block declaration. These bindings remain in existence until the evaluation of the block is completed, at which time they disappear. Each binding contains a value for the variable that it binds. The nature of the binding is specified by the block declaration that creates it. After the bindings have been made, execution of the statements in the block begins. The statements are executed in turn unless the sequence of control is altered by a GO statement or by a RETURN statement. Execution of the block is terminated either by executing a RETURN statement or by executing the last statement of the block without a transfer of control.

A block declaration in SL is in the form:

p_1 p_2 p_3 s_1, s_2, . . ., s_n

The p_i consist of a type, a storage mode, and a transmission mode (in any order). Lexical storage and transmission by value are specified by omission; if the type is omitted, a default type is used. If all p_i are empty, the symbol DECLARE must be used. Each of the s_i is either the name of a variable or in the form:

$$v \leftarrow e$$

where e is an expression giving an initial value for the variable v. If no initial value is given, a default value, depending on the type, is used. A block declaration causes all the specified variables to be internal parameters of the block and to have the properties specified by the p_i.

In IL, each declaration specifies the properties of one and only one variable; thus, in the translation from SL to IL, it is necessary to break up each declaration that declares more than one variable into a sequence of declarations (with appropriate factoring of properties). An IL declaration is in the form:

(v p_1 p_2 p_3 p_4)

where one of the properties is the initial value, if any.

The various types of statements and their effects may be summarized as follows:

1. *GO statement*—transfers control to the named statement.
2. *RETURN statement*—terminates evaluation of a block and determines the value of a block expression.
3. *Compound statement*—permits the insertion of a sequence of statements in a context where only a single statement is expected. A compound statement is in the form of a block with no declarations.
4. *Conditional statement*—selects one of several possible statements to be executed on the basis of the truth or falsity of a sequence of Boolean expressions.
5. *Simple expression*—causes the evaluation of the expression; the value is discarded.
6. *FOR statement*—causes an iteration to be performed for a sequence of values of a named variable.
7. *TRY statement*—causes control to be returned to itself if an exit condition is detected during the execution of a statement within the TRY statement.
8. *Block statement*—like a compound statement, except that internal parameters may be declared in the same manner as in a block expression.
9. *CASE statement*—selects one of several possible statements to be executed on the basis of the value of an integer-valued expression.
10. *Empty statement*—can be used to place a label; contains nothing and makes no action.

The FOR statement has some unusual features that merit further discussion. The statement:

FOR v IN x DO s

causes the statement s to be executed for each element of the list x, with v assuming the successive

elements as its value in each execution of s. If ON is used instead of IN, v first assumes as values the entire list x, then its successive terminal segments CDR x, CDDR x, etc., until the list x is exhausted. The clause:

UNLESS b

may be inserted as part of a FOR statement to inhibit execution of the statement s whenever the Boolean expression b is TRUE. The UNTIL clause of ALGOL, used in conjunction with STEP, is replaced by a relational operator and an expression; iteration continues until the variable of iteration no longer satisfies the specified relation. This approach avoids the need to recompute the sign of the increment for each iteration.

Functions

A function definition is a specification of a computational procedure; the procedure itself is a function. A function definition in SL is in the form:

t FUNCTION n $(x_1, x_2, \ldots, x_n)$; $d_1, \ldots d_k$; e

where t is the type of the value of the function, n is the name of the function, the x_i are dummy variables that stand for its arguments, the d_i are declarations governing the arguments, and e is an expression whose value is the value of the function.

The corresponding form in IL is:

(FUNCTION (n t) $(d_1\ d_2 \ldots d_k)$ e)

where a declaration is given for each argument. Thus the declarations not only give the properties of the arguments but also name them. If the value type of the function is omitted, then the name n can be written without parentheses and the default type will be used.

The argument parameters are used to denote the values of the actual arguments within the body of the function definition. The body of the function definition e is the expression that defines the value of the function. The argument declarations specify the type, transmission mode, and storage mode of the arguments.

Functional Data. A function may be used in either of two ways: as an operator or as a datum. We have already seen how functions can be used as form operators. An example of the use of a function as a datum would be the input to a numerical integration routine; the input is the function to be integrated, and the output is the integrand. An example oriented more closely to symbolic data processing would be the use of the LISP function MAPCAR, whose arguments are a list to be transformed and a transformation function. The output of MAPCAR is the transformed list. Thus

MAPCAR ('(2 5 4 9), FUNCTION ADDER
(J); INTEGER J; J+2)

would evaluate to the list:

(4 7 6 11)

Since a function is itself a datum, it can be used in any context where a datum is expected. Thus, functions can themselves be used as arguments of other functions, and functions can be values of variables. A function can be designated by its definition, by its name, or by a variable having the function as its value.

There are two contexts in which a function may be referenced—as a datum, as we have just said, and as a form operator. When a function is used as a form operator, it must be designated either by a functional variable (i.e., a variable whose values are functions) or by a function name. The effect of using a function definition as a form operator can be achieved by assigning the function definition to a functional variable (which is legitimate, since the function definition then appears in a data context) and then by using the functional variable as the form operator.

Functions of an Indefinite Number of Arguments. It is possible to define functions that expect an indefinite number of arguments. In defining such a function, there is no way to enumerate the names of the arguments; therefore an argument vector, i.e., a one-dimensional array having a single variable name v, designates the set of arguments. The length of the vector is specified by a second variable k. In the argument list, the argument vector (which must be the first argument) is designated by writing v(k) in SL and (v INDEF k) in IL. When the function is entered, the value of v is the vector of arguments, and the value of k is the length of this vector. The different elements of the argument vector can then be referred to within the body of the definition by subscripted occurrences of v.

For example, the function SUMSQUARE might be written to take the sum of the squares of its arguments. We would then define it in SL as follows:

```
REAL FUNCTION SUMSQUARE(X(I));
  BEGIN INTEGER J; REAL Y;
    FOR J←1 STEP 1 UNTIL > I DO
      Y←Y + X(J)↑2;
  RETURN Y
END
```

Here X is the argument-vector parameter and I is its length. The corresponding IL definition is:

```
(FUNCTION (SUMSQUARE REAL) ((X
    INDEF I))
  (BLOCK ((J INTEGER) (Y REAL))
    (FOR J (STEP 1 1 GR I)
    (SET Y(PLUS Y (EXPT (X J)2))))
(RETURN Y)))
```

An actual use of SUMSQUARE might look like:

```
SUMSQUARE (2, 7, 4)
```

in SL, and:

```
(SUMSQUARE 2 7 4)
```

in IL.

Sections

A section is a collection of declarations and definitions that operate as a unit. Dividing a large program into sections makes it possible to write different parts of the program independently without name conflicts. It also makes it possible for one user to refer to programs written by another user without name conflicts. A section is designated by its section name, which is an identifier. Each section is associated with a set of variables that designate the various entities defined within the section. At any given time there is a single active section, which is known as the current section; all other sections are external sections. A variable in a particular section, whether current or not, can be referred to by tailing (often called "qualifying") e.g., "JOE$SAM" refers to the variable JOE in section SAM.

The section mechanism permits parts of LISP 2 programs to be written and checked out independently. At merge time, attention need be paid only to variables used for names of common functions and communication variables. Since the system programs are in a special section, the user need not worry about name conflicts; at the same time, the system programs are accessible to the user through the tailing mechanism. Thus the user can, if he chooses, treat the system programs as an extension of his own program rather than as a black box.

Supervisor Level Operations

LISP 2 is controlled by a supervisor program that is itself named LISP and that can be called as a function. When the user starts up the LISP system, the supervisor is called immediately. The supervisor accepts commands to perform various operations. The actions taken by the supervisor in response to these commands are known as top-level operations. The following top-level operations are possible:

1. Evaluate an expression
2. Establish a current section with given name and default type
3. Create a fluid or own variable of specified type and transmission mode
4. Define a function
5. Define a dummy function (used to establish type information in certain cases)
6. Define a macro
7. Define an instruction sequence to be used in compilation
8. Define an assembly-language program
9. Declare a variable to be synonymous with another variable.

The user can specify the input and output devices to be used; the on-line typewriter is taken as the default case. After each operation the system sends any necessary output to the output device and proceeds to the next operation.

Input/Output. One of the primary design aims in LISP 2 I/O has been the maintenance of as much machine independence as possible. This is accomplished by distinguishing user interfaces from system interfaces and insulating the user from the system interfaces. This effect is achieved by creating machine-independent data aggregates called "files," and permitting the user to operate with files by means of LISP 2 functions.

To the user, a file is a source or sink for information, which is filled on output and emptied on input. A file itself is both device- and direction-independent. The relationship of a file to an external device is determined by the user at run time, when he specifies whether the file is to be an input file, an output file, or both.

To the system, a file consists of a sequence of records, represented internally as an array of type OCTAL if the file is binary, and as a string if the file

is composed of characters. (ASCII 8-bit characters are used internally throughout LISP 2.) To reduce buffer storage overhead, only one record for a given file can be in main memory at a time. String records are further structured into lines. The number of characters per line and lines per record may be specified by the user, but must be consistent with the conventions used by the external monitor system.

When a record in a file is moved from an external device into core, it is transformed into a LISP 2 string. The transformation may involve character code conversions and insertion or deletion of control characters. The transformation is governed by a collection of control words associated with the file. During output, this transformation, known as "string post-processing," is reversed.

File Activation and Deactivation. A file may be either active or inactive; an active file, in turn, may be either selected or deselected. No record is kept within LISP 2 of inactive files; however, many files may be active concurrently.

A file is activated by evaluating the function OPEN which establishes all necessary communication linkages between LISP 2 and the monitor. The file is named by an identifier that is its referent throughout its active life. The user further specifies the desired file description at this time. This description is given only once and consists of a list of file properties desired by the user, such as the unit (tape, disc, teletype, CRT, etc.), form (binary, ASCII, BCD, etc.), format (line and record sizes), and various protection and identification parameters.

Deactivation of a file is achieved by evaluating the function SHUT. SHUT breaks all the communication linkages and deletes all internal structures such as arrays, strings, and variables that were dynamically established by OPEN. The user may specify the disposition of the file, e.g., the saving of the tape or the insertion of the file in disc inventory. The external monitor is informed of such actions by LISP 2.

File Selection. At any given time, exactly one file is selected for input and one for output; all other active files are deselected. The LISP 2 reading functions all operate on the currently selected input file; the printing functions all operate on the currently selected output file. The functions INPUT and OUTPUT are used for selecting the input file and the output file, respectively.

When a new file is selected, the record, line, and column controls for the deselected (replaced) file are preserved, and the new file record, line, and column controls are reestablished. Once a file is selected, all I/O primitives act only on that file. Thus it is possible to write a LISP 2 program that is independent of form, format, and device by supplying the name of the file as an argument of the program at run time. This scheme allows a LISP program to be debugged with files generated on-line and subsequently run with bulk data from tape or disc files simply by changing the selected file.

Other I/O Functions. A variety of I/O functions are available for reading and writing binary and symbolic data. There are character-level primitives that permit testing, printing, reading, and transforming characters. Other functions allow reading and printing at the token and S-expression levels. Character mappings permit LISP 2 to communicate with restricted character-set devices.

Examples

An example is now given of a complete SL program. The example includes not only the program itself but also the control actions necessary to test it:

```
SYMBOL SECTION EXAMPLES, LISP;
% LCS FINDS THE LONGEST COMMON SEG-
%   MENT OF TWO LISTS L1 AND L2
FUNCTION LCS(L1,L2); SYMBOL L1, L2;
  BEGIN SYMBOL X, Y, BEST ← NIL; INTE-
      GER K←O, N, LX←LENGTH(L1);
    FOR X ON L1 WHILE LX > K DO
    BEGIN INTEGER LY ← LENGTH (L2);
      FOR Y ON L2 WHILE LY > K DO
      BEGIN N ← COMSEGL (X,Y);
        IF N <=K THEN GO A;
        K ← N;
        BEST ← COMSEG (X,Y);
      A: LY ← LY − 1
      END;
      LX ← LX − 1
    END;
    RETURN BEST;
  END;
% COMSEGL FINDS THE LENGTH OF THE
%   LONGEST INITIAL COMMON SEGMENT
%   OF
%   TWO LISTS X AND Y.
  INTEGER FUNCTION COMSEGL (X,Y);
    IF NULL X OR NULL Y OR CAR X /=
        CAR Y
      THEN O ELSE COMSEGL (CDR X, CDR
          Y) + 1;
```

```
% COMSEG FINDS THE LONGEST INITIAL
%  COMMON SEGMENT OF TWO LISTS X
%  AND Y
  SYMBOL FUNCTION COMSEG (X, Y);
    IF NULL X OR NULL Y OR CAR X /=
        CAR Y
      THEN NIL ELSE CAR X . COMSEG(CDR
          X, CDR Y);
% LENGTH COMPUTES THE LENGTH OF L
  INTEGER FUNCTION LENGTH (L); SYM-
      BOL L;
    BEGIN INTEGER K ← O; SYMBOL L1;
      FOR L1 IN L DO K ← K+1;
      RETURN K;
    END;
LCS ('(A B C B C D E), '(B C D A B C D E F));
    STOP
machine: (B C D E)
```

This example illustrates the use of list processing capabilities combined with integer arithmetic and iteration. The operator "<=" means "less than or equal to," and the operator "/=" means "not equal to." The LISP operators CAR, CDR, and NULL are all used as prefix operators without parentheses. The dot in the third line of COMSEG is an infix operator that stands for the LISP function CONS. The statement "FOR X ON L1" causes iteration to take place on the successive terminal segments of L1. Thus, if L1 is the list (A B C D), then iteration takes place successively on (A B C D), (B C D), (C D), and (D). The function LENGTH, defined here, is available as a system function and is redefined only as an illustration.

THE PROGRAMMING SYSTEM

System Overview

A diagram of the LISP 2 system which shows the relationship among its different components is shown in Fig. 2. Information enters the system via the I/O package in either SL or IL. The I/O package transforms the input into a stream of characters —the input to the finite state machine—which in turn generates a stream of tokens. Among other things, the finite state machine performs the task of linking up a newly received identifier with a previous copy of the same identifier. The token stream produced by the finite state machine is routed by the supervisor to either the syntax translator or to a reading program for IL, depending on whether SL or IL is expected. In either case, the result is an ex-

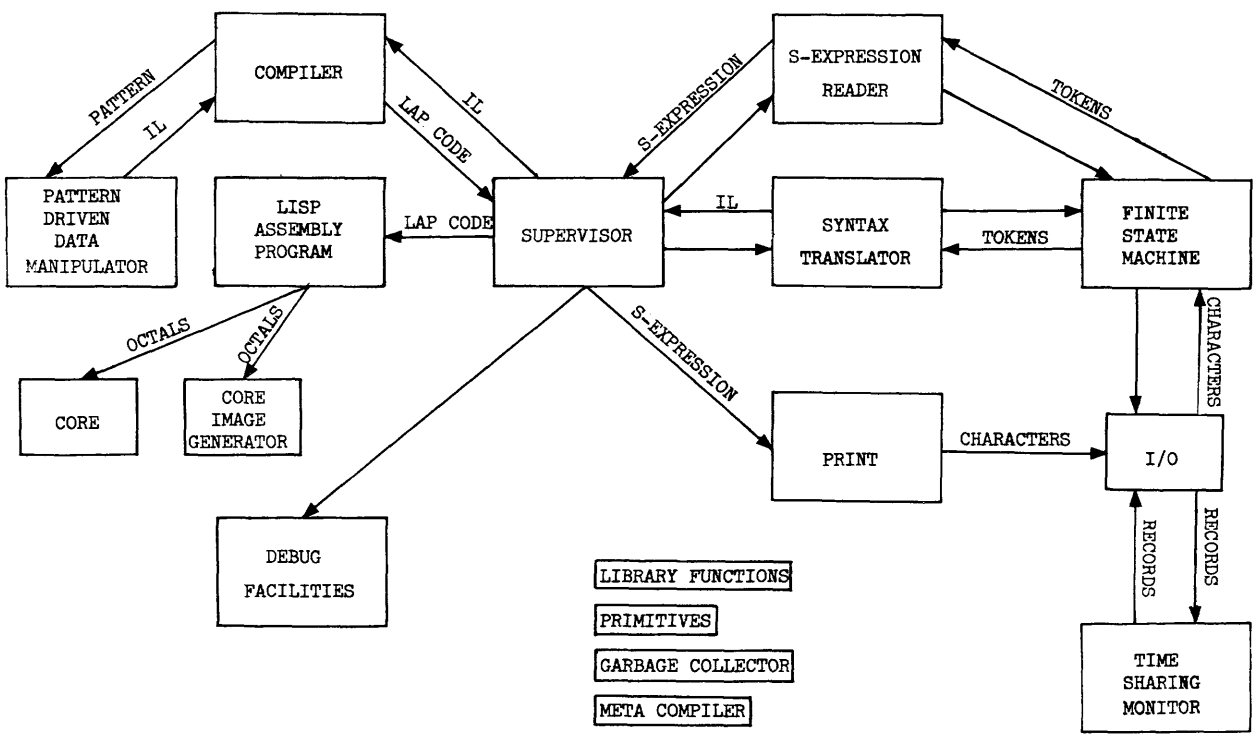

Figure 2. System components and information flow paths (unlabeled connections designate control paths).

pression in IL. The supervisor determines when compilation is to take place, and also handles processing requests.

The syntax translator takes a stream of SL tokens and transforms it into an IL expression. This expression can be returned as output, passed to the compiler, or both. The choice is made by the supervisor under the control of the user. The syntax translator consists of parsing and generating programs that are compiled from a set of syntax equations. These syntax equations define SL in terms of IL.

The compiler, which is the most complex component of the system, converts IL into input for LAP, the LISP Assembly Program, or for the core image generator. Both LAP and the core image generator accept input in assembly language (AL). If LAP is being used, then the result of assembly is a relocatable segment of code stored in an area of the machine reserved for binary program. If the core image generator is being used, then the result is a string of pairs of binary numbers, each consisting of a core location and the contents of that location, stored on a magnetic tape or other external medium. The core image generator is only used when a new system is being created.

The META compiler, the garbage collector, and the primitives are all implicitly involved in the operation of the system. The META compiler is a library program that generates a syntax translator from a set of syntax equations. The garbage collector is the program that collects dead storage when available storage has been exhausted. The primitives are the basic library functions in terms of which the entire system is written.

Memory Management

Most of the concepts of memory management used in LISP 1.5 are also used in LISP 2. Memory management in LISP 2 is based on several considerations:

1. LISP 2 data structures may vary in size by orders of magnitude at run time, and storage for such data structures must be allocated automatically.
2. Since recursion is permitted, successive generations of data structures must be retained simultaneously.
3. Programs and data structures that are no longer needed must be purged without explicit action on the part of the user.
4. Numerical data must be stored in such a way as to permit efficient numerical calculations.

LISP 2 data structures may be either variable or fixed in size. The variable data structures are arrays, strings, and symbolic expressions. Although an array, once established, does not change in size, the size of an array is frequently not known until the occasion arises to create it. In the case of list structures, the situation is even more complex; a list structure may be modified in such a way as to increase or decrease its size.

Arguments of functions and internal parameters of blocks are stored on a pushdown stack. Since all temporary storage belonging to LISP 2 functions is recorded on the pushdown stack, which is maintained by the LISP 2 system, recursion is permitted with no special user provisions. Unlike LISP 1.5, LISP 2 stores numbers directly on the pushdown stack as single cells. Therefore, it is possible to perform arithmetic without the loss of efficiency that would arise from packing and unpacking numbers referenced indirectly. Symbolic expressions, strings, and arrays, however, are accessed by means of pointers stored in the stack. The data structures thus pointed to are discarded when the function creating them has completed its execution; however, they do not disappear, but remain as garbage until the next garbage collection, the description of which follows.

In LISP 2, data structures are grouped according to their storage characteristics and a storage area is set aside for each group. The groups are:

1. Elementary symbolic entities (symbolic constants, function and variable names, etc.)
2. Compiled programs
3. List structures
4. Arrays and strings

In addition, a storage area is set aside for the pushdown stack. These storage areas are arranged in pairs, where one member of the pair grows from the bottom up and the other grows from the top down. Data storage is obtained by taking storage space from the appropriate area until that area is exhausted (which occurs when its boundary meets the boundary of the area that is paired with it). At this point, the garbage collector is invoked. Garbage collection erases all inaccessible data structures and reclaims the emptied space for new structures. For instance, if a LISP 2 function has been redefined, the program

corresponding to its old definition is inaccessible and thus is erased. During garbage collection, the different areas are compacted, relocating code and/or data structures, if necessary, so as to eliminate the gaps left by erased structures.

The different kinds of structures are stored in different areas because their requirements in terms of garbage collection are different. For instance, the elementary symbolic entities cannot be moved, but other kinds of data can be moved. Similarly, list structures consist of independent nodes, while arrays consist of blocks of different sizes.

The Syntax Translator and the META Compiler

The translation from SL to IL is performed by a syntax translator that was generated by the META compiler. The META compiler is based upon a program developed by Special Interest Group for Programming Languages of the Los Angeles Chapter of ACM.[8] The META compiler takes as input a specification of the syntax of SL, together with instructions on how each syntactic entity is to be transformed to IL. It produces an IL program that actually carries out the translation from SL to IL. The description of the syntax of SL is given in an extended version of Backus-Naur Form.[4]

The META compiler produces top-to-bottom compilers with a controlled backup feature and an interface with the finite state machine (see below). Both the controlled backup and the finite state machine are efficiency features. The controlled backup allows the designer of a language to specify in the syntax equations when the state of the machine must be saved because two or more parsings start with the same construction.

As it is possible to regenerate the syntax translator with new syntax equations at any time, the syntax and semantics of SL are not, in principle, rigidly fixed. In practice, variants on the syntax translator will be used in order to translate other languages into LISP 2 IL. These other languages, unlike SL, will normally not be semantically equivalent to IL.

Finite State Machine

The finite state machine (FSM) is a token-parsing program used by the syntax translator and the S-expression reader. Reading LISP 2 entities is expensive, not only in the original creation of the internal structures, but also in the time spent in garbage collecting when the structures are discarded. Consequently, it is desirable to avoid backup at the character level and its resulting re-creation of duplicate structures. Since backup must be used by the syntax translator, the FSM was imposed between it and the character stream to eliminate reprocessing of tokens. Having the bottom-to-top FSM interface with the top-to-bottom syntax translator eliminates a large portion of the overhead associated with reading in the LISP 2 system. The S-expression reader does not require backup, but since the FSM existed, it was convenient to use tokens for building S-expressions also.

The FSM behaves like a Turing machine. It moves from state to state as it reads characters; when a terminal state is reached, it "prints" a character from its output alphabet (tokens) and sets its state to the initial one. Parsing and manufacture of structures are done simultaneously as characters are recognized. No reprocessing of the parsed characters is ever necessary, since in a terminal state the token is already complete (except for a final action, such as combining the parts of a real number).

The LISP 2 Compiler

The LISP 2 compiler is a large, one-pass, optimizing translator whose input is a function definition in IL and whose output is an assembly-language list of instructions suitable for input to LAP. Most of the compiler is independent of the target machine, since the compilation concepts themselves are machine-independent. The declarations of all fluid variables appearing within the function are written into the output listing, since these must agree with fluid variable declarations made elsewhere. Checks are made for both format and semantic errors during compilation. The compiler consists of three major sections: the analyzer, the optimizer, and the user control functions.

Analyzer. The top-level control of the compiler resides in the analyzer, which operates recursively. Each item to be compiled is passed to the analyzer either directly or indirectly. If the item is a variable, an appropriate declaration is found and code for retrieving the variable is generated; otherwise the code for a function call is generated, a macro expansion is performed and the result compiled, or linkage to an appropriate code generator is made. A pattern-matching function has been implemented for use in the LISP 2 compiler. The patterns are written in a modified form of Backus-Naur Form (not the same

as the one used in the syntax translator). The patterns are matched to an S-expression and the value of the match is either TRUE or FALSE. The pattern-matching function checks for syntactic correctness and distinguishes among different forms at the same time.

Optimizer. Optimization of the code produced by the LISP 2 compiler is handled by many groups of routines, each responsible for certain actions. The communicative mechanisms between these various parts and the rest of the compiler will be described in some detail below.

The movers, a highly machine-dependent set of functions, produce code that alters the state of a compilation in a specified way, such as moving an object to an accumulator or converting a datum to a specific type. Embodied in the movers is a predicate capability that answers the question, "Is this move possible under these conditions (say, one machine instruction)?" The movers are used to build all address and modifier fields of generated instructions. Associated with the movers is a post-processor that rewrites the output code after the main compiler has produced it. Redundant load-store sequences and some unnecessary branches are removed from the listing. Also, certain groups of instructions are rewritten to make use of machine-specific instructions.

The arithmetic optimization package handles code generation for addition and multiplication. The algorithm that is used is a standard one, namely, first sorting the arguments by type and then by priority sequence within a particular type. The sequence depends on whether the arguments are memory or accumulator references. A single set of functions handles both multiplication and addition, with the aid of several functional arguments.

A second kind of optimization has to do with the elimination of unnecessary transfer instructions. This task is accomplished through the analysis of confluence points, i.e., places in the program at which several paths of control converge. For instance, consider the conditional expression:

$$(\text{IF } p_1\ e_1\ p_2\ e_2\ \ldots\ p_n\ e_n)$$

The appearance of this conditional expression establishes a confluence point at the end of the compiled code that represents it. After the execution of any of the e_i, control goes to this confluence point. Moreover, the confluence point is hereditary for each of the e_i, i.e., if one of the e_i is a conditional expression, then its confluence point is the same as that of the entire expression. Analogous considerations hold for conditional statements. Confluence points are also hereditary with respect to RETURN statements of blocks, i.e., the confluence point of a RETURN statement is the same as that of the block in which it appears.

When an expression is compiled, the characteristics of the value that is produced must be specified. These characteristics include type, whether it is in a special register or in an ordinary memory cell, its address modifier (direct or indirect), which registers it may be left in, whether the actual value is needed or whether the negative or reciprocal of the value is so described, etc. These characteristics are remembered by a set of state variables, which are bound for each call to the analyzer. As a statement or expression is compiled, a listing is generated and the state variables set to reflect the state of the compilation. The compiler is passive in the sense that a compilation produces only the minimum amount of code necessary to allow the result to be described by the state variables.

User Control Facilities. The user can give the compiler explicit instructions to aid in the compilation process. As in LISP 1.5, macros are an integral part of the language. Many of the facilities of the language, e.g., FOR statements, are implemented by means of system macros. When a FOR statement (in IL form) is encountered during compilation, it appears as an operational form whose operator is FOR. The compiler tests each form operator to see if a macro is defined for it. In the case of FOR, there is such a macro. The macro is invoked with the FOR statement (in the form of an S-expression) as input. The output is a block containing an equivalent iterative loop. This block is then compiled in place of the FOR statement. Macros may also be defined by the user, and no distinction is made between system macros and user macros.

Certain machine-dependent operators are particularly useful as primitives in compilation. CORE is an operator that acts like an array whose content is all of the machine memory. Therefore CORE(x) is the content of location x. BIT is an operator that specifies a certain contiguous portion of a word. There are also several operators that permit an expression to be forced to a certain type or permit a datum of one type to be used as though it were of another type. Although such mechanisms exist in most compilers, LISP 2 has made these items available through the language.

The LISP 2 Assembly Program

The LISP 2 Assembly Program, LAP, is a program that generates a code segment from a list of symbolic instructions and labels. LAP also allocates storage for variables on the pushdown stack, and insures that references to fluid and own variables are consistent among different compiled functions. LAP does more than most assemblers, in that it handles all aspects of pushdown stack mechanics; consequently, references to variables are made by naming the variable in the appropriate field of any instruction that references it. Thus, the pushdown stack need never be referenced explicitly.

LAP includes a number of system macros specifically designed for LISP 2 programming. The prologue and epilogue of a function are generated by BEGIN and RETURN respectively; CALL is used to generate a call to a LISP 2 function in the standard format. Storage allocation on the pushdown stack is performed by the BLOCK, DECLARE, and END macros; FLBIND creates any necessary bindings for fluid variables. LAP does not have a generalized macro facility; any effect that could be achieved by such a facility, however, can also be achieved by preprocessing.

The address field of an instruction may be used to allocate, refer to, or release temporary storage on the pushdown stack. The address fields TOP. and POP. are normally used with instructions of the "load" type. Both TOP. and POP. refer to the most recently allocated pushdown cell, but POP. has the additional effect of releasing that cell. PUSHA. and PUSHP. both cause a new pushdown cell to be allocated, and refer to that cell; PUSHA. and PUSHP. are normally used in instructions of the "store" type. PUSHA. is used for absolute quantities and PUSHP. for symbolic quantities, so that a map of the pushdown stack can be maintained.

To illustrate the use of assembly language, as well as the output code produced by the compiler, we give the Q32 assembly language version of the program RANDOM presented as an example earlier in the paper:

```
(LAP (FUNCTION (RANDOM REAL)
  ((A REAL) (B REAL))
  (STF TOP.)
  (BEGIN)
  (LDA Y)
  (MUL 3125 (L567.7 R S))
  (STB Y)
  (ARGS)
  (LDA Y)
  (STF PUSHA.)
  (LDA (NUMBER 67108864) S)
  (CALL (REMAINDER . LISP))
  (STF Y)
  (LDC A)
  (FAD B)
  (STF PUSHA.)
  (LDA Y)
  (FLT (ENTRY B48.))
  (FDV (NUMBER 6.7108864000E-7))
  (FMP POP.) (FAD A) GO9017 (END) (RE-
      TURN))
(((REMAINDER . LISP) FUNCTION (FUNC-
    TIONAL INTEGER INTEGER INTEGER)
  NIL) (Y  OWN INTEGER NIL)) USER)
```

ACKNOWLEDGMENTS

LISP 2 is being developed jointly by Information International, Inc., and System Development Corporation, with contractual support from the Advanced Research Projects Agency of the Department of Defense. Personnel actively participating in this program include:

Dr. Paul W. Abrahams (III)
Mr. Jeffrey A. Barnett (SDC)
Mr. Erwin Book (SDC)
Mrs. Donna Firth (SDC)
Mr. Lowell Hawkinson (III)
Dr. Stanley L. Kameny (SDC)
Mr. Michael I. Levin (III)
Mr. Robert A. Saunders (III)
Mr. Clark Weissman (SDC)

In addition, we wish to acknowledge the voluntary support and contributions received from Professor Marvin Minsky and his associates at MIT, Professor John McCarthy and his associates at Stanford University, Dr. Daniel G. Bobrow of Bolt, Beranek and Newman, and many others.

REFERENCES

1. M. Levin, "LISP 2 Primer," SDC Document TM–2710/101/00 (July 15, 1966),

2. T. Abrahams, "LISP 2 Reference Manual," SDC document in preparation.

3. M. I. Levin, *LISP 1.5 Programmers Manual,* MIT Press, Cambridge, Mass., 1962.

4. "Revised Report on the Algorithmic Language ALGOL 60," *Comm. ACM,* vol. 6, no. 1, pp. 1–17 (1963).

5. V. Yngve, *COMIT Reference Manual,* MIT Press, Cambridge, Mass., 1962.

6. D. G. Bobrow, "METEOR, a LISP Interpreter for String Transformation," in "The Programming Language LISP," Information International, Inc., Cambridge, Mass, 1964, pp. 161–90.

7. "ALGOL algorithm #266," *Comm. ACM,* vol. 8, no. 10, p. 605 (1965).

8. D. V. Schorre, "META II, a syntax-directed compiler writing language," *Proc. ACM,* p. D1.3–1 (1964).