Programming in an Interactive Environment: the "LISP" Experience

ERIK SANDEWALL

Informatics Laboratory, Linköping University, S-58183, Linköping, Sweden

LISP systems have been used for highly interactive programming for more than a decade. During that time, special properties of the LISP language (such as program/data equivalence) have enabled a certain style of interactive programming to develop, characterized by powerful interactive support for the programmer, nonstandard program structures, and nonstandard program development methods. The paper summarizes the LISP style of interactive programming for readers outside the LISP community, describes those properties of LISP systems that were essential for the development of this style, and discusses some current and not yet resolved issues.

Keywords and Phrases: interactive programming, LISP, programming methods, program structure, programming systems.

CR Categories: 3.69, 3.80, 4.0

INTRODUCTION

Why do some programming systems have to be large and complex?

In recent years there has been a trend towards simple designs in computer science research. The widespread revulsion against OS/360 in the academic community led to a quest for primitive concepts in operating systems and for very simple systems, which have been successfully developed by, for example, Brinch Hansen [1]. Similarly, reaction against large and messy programming languages encouraged the development and adoption of languages that minimized the number of facilities and features, notably PASCAL [2]. I believe that the great attraction of very simple programming languages such as BASIC and very simple database systems such as MUMPS [3] in the world of practical computing are other examples of the same trend towards simplicity.

Despite the above, the present paper is concerned with programming systems which by necessity have to be large and complex and which are very hard to structure well because we know so little about their design. Such systems are of interest for a combination of two reasons.

First, there is a long list of things that one wants a programming system, particularly if it is interactive, to do for the programmer. ("Programming system", is used to mean an integrated piece of software which is used to support program development, including but not restricted to a compiler.) The reader can easily generate his own list of functions, but here are some possibilities:

- Administration of program modules and of different generations of the same module (when errors are corrected and/or the scope of the program is extended);
- Administration of test examples and their correct results (including side effects), so that the relevant tests are performed automatically or semiautomatically when sections of the pro-

CONTENTS

gram are changed, and a report is made to the user if a discrepancy has been observed;

- Administration of formal and informal documentation of program segments, and automatic generation of formal documentation from programs;
- Interdialect translation;
- Checking of compatibility between parts of programs;
- Translation from general-purpose or specialized higher-level languages to the chosen base language ("preprocessors"), with appropriate support for compile-time and run-time error diagnostics in the base language, comments, etc.;
- Support for a given programming methodology. For example, if top-down programming is to be encouraged, then it is natural to let the interactive programming system maintain successive decomposition steps, and mutual references between an abstract step and its decomposition;
- Support of the interactive session. For example, a *history facility* [4] allows the user to refer back to previous commands to the system, edit them, and re-execute them. An *undo facility* [4] allows the programmer to back up and undo effects of previously performed incorrect commands, thus salvaging the data-structure environment that was interactively created during the interactive debugging session;
- Specialized editing, performed with an editor that understands at least the syntax of the chosen programming language, and which therefore allows the user to refer to natural entities in this language and to give fairly high-level instructions as to where additions to the program are to be inserted;
- Optimizing programs which transform a program into an equivalent but more efficient one;
- Uniform insertion programs, which in a given program systematically insert additional statements, for example for producing trace printouts or for counting how often locations in the program are visited during execution.

Second, and this is the crucial point, if these functions are performed by separate and independent programs, a considerable duplication of effort will result. Syntax analysis has to be performed not only by a compiler or interpreter, but also by specialized editors, optimizing programs, uniform insertion programs, documentation generators (such as cross-indexers), and so on. Analysis of the relationships between modules (calling structure, data-flow structure, etc.) is needed for generation of documentation, administration of test examples, compatibility controls, and program optimization. Since the results of an execution count may be used to tell an optimizer where it should spend its efforts, programs for these two tasks should be able to communicate. Also, some of the above facilities, such as the *undo* facility, are only possible if they are integrated into the programming system. For these reasons, it is natural to try to integrate facilities such as the above into one coherent programming system, which is capable of performing them all in an economic and systematic fashion.

I believe that the development of integrated, interactive programming systems, and the methodology for such systems, is the major research issue for programming systems and programming methodology today. It is significant for programming methodology, since every detailed recommendation on how to write programs is also a recommendation on how to design an interactive programming system that supports the methodology. In the area of research on programming systems, this is relatively unexplored territory waiting to be considered now that other problems such as compiler design for conventional languages seems to be fairly well understood.

The task of designing interactive programming systems is hard because there is no way to avoid complexity in such systems. Because of all the dependencies between different parts of an interactive programming system, it is hard to break up the design into distinct subproblems. The only applicable research method is to accumulate experience by implementing a system, synthesize the experience, think for a while, and start over.

Such systems have been built and used for the programming language LISP. I believe that the time is now ripe for a synthesis and discussion of the experience that has accumulated in this context. The present paper is intended to serve such a purpose.

1. BACKGROUND

Requirements on a Programming Language for Integrated, Interactive Programming Systems

In a research project to design and build an integrated, interactive programming system as an implementation experiment, one early decision must be which programming language the system will support. This language must satisfy minimal criteria, as follows.

Bootstrapping. An obvious choice is to implement the system itself in the language it supports; then one needs to work only with a single language, and the system supports its own development.

Incrementality. To achieve real interaction, the basic cycle of the programming system should be to read an expression from the user, execute it, and print out the result while preserving global side effects in its database. The expression may of course contain such things as calls to procedures.

Procedure-orientation. For obvious reasons, the language chosen should be procedure-oriented.

Internal representation of programs. Since most of the operations listed in the previous section are operations on programs, the language should make it as easy as possible to operate on programs. Therefore, there should be a predefined, system-wide internal representation of programs which reflects their structure in as pure a form as possible, for example as a tree structure. This structure should be a data structure *in* the programming language, so that user-written programs may inspect the structure and generate new programs. As a consequence, the kernel of the programming system must contain the following programs:

- a *parser* which transforms the user's programs to the internal representation;
- a *program-printer* that performs the reverse operation;
- an *interpreter* for programs in the internal representation; and/or
- a *compiler* that transforms the internal data-structure representation to

machine language for the host machine.

Notice that the tasks performed by a compiler for a conventional language are divided among the parser and the compiler in this architecture.

Full checking capability. All possible input from the programmer/user must result in rational responses from the system.

Declaration-free kernel. The contents and use of declarations is one of the important issues that one wants to experiment with in a research system, and therefore should not be frozen in the system kernel. Instead, the internal form of programs should be declaration-free, and one of the services of the developing programming system should be to account for declarations as input by the user.

Data structures and database. The system must minimally have data structures that are able to represent programs as tree structures, and a database facility where one can conveniently store and retrieve properties of items that appear in the program or in descriptions of the program. (Variable names and procedure names are simple examples of such items.) A relational database where program items are allowed to appear as arguments of relations, so that facts about them can be stored, is one way to satisfy the database requirement.

In addition to these minimally necessary data structures, the system will of course contain additional data-structuring facilities which may be desirable for the intended experimental applications of the system.

Defined input/output for data structures. In order to test a procedure interactively, one wants to be able to type in a call to the procedure and obtain back the value. Since the arguments and/or the value may be data structures, I/O for data structures must be defined in the system. Since programs are internally stored as data structures, this I/O may also be used as parser and program-printer.

Handles and interactive control. The actions taken by the system in specific situations should be controllable by the user in such a way that a user-defined procedure (a "handle") can be inserted instead of the original procedure provided by the system. For example, such handles are useful for the operation applied to expressions input by the user, and reactions to errors and exceptional conditions during the execution of a procedure.

Also, the system must allow for an assortment of different control signals that may be typed-in by the user at arbitrary times to control the ongoing computation. The "killer" interrupt, which terminates the interactive session and returns to the operating system, is exactly what the user does *not* want. The response to control signals should also be user-controllable through handles.

Additional properties may of course be required or desired, but the above list will do for our purpose. A quick check of existing languages against this list shows that:

- **Most conventional languages** (FORTRAN, PL/I, SIMULA, PASCAL, etc) have insufficient data structure facilities (especially I/O), are not incremental, have no internal representation as above;
- **BASIC** has no data structures;
- **LISP, SNOBOL, and APL** satisfy the requirements, in different ways with respect to data structures and internal representation of programs.

At least for LISP (and I believe also for SNOBOL and APL) the match to the requirements is quite accidental: the language was developed for other purposes, and most texts about the language describe it in a different way. It is not my intention here to argue that LISP should necessarily be chosen; a project to build an integrated programming system might also choose a less-known language, or design its own. My purpose is simply to explain the experience accumulated in the LISP community with integrated interactive programming systems.

The LISP Users' Community

Before a discussion of the accumulated experience from implementation and use of LISP systems, a few observations about the community of LISP users are in order.

The following account is simplified and slightly caricatured to make its point as concisely as possible.

LISP is used almost entirely as a research tool. It is the dominant language for artificial intelligence research, and a major implementation language for formula manipulation systems. Typical applications are semantic understanding systems, program analysis and generation, and theorem proving. The average LISP user writes a program as a programming experiment, i.e., in order to develop the understanding of some task, rather than in expectation of production use of the program. The act of developing the program, not the act of running it (even for test data), constitutes the experiment. As a consequence, the program is likely to be large and complex, to undergo drastic revisions while it is being developed, and to be thrown away before it has been "completed" by conventional programming standards since it will already have served its purpose before then.

The environment in which the program is written is specialized for this style of research. The programmer works in a large group with relatively good computing facilities. He expects to have a terminal at his desk and to be able to use it continuously as a tool. More importantly, his group will have been one of the first to be able to use computers in this fashion.

One consequence of this scenario (which as I said is simplified and to some extent exaggerated) is that a considerable experience with interactive programming has developed in the LISP community, both with respect to programming system facilities that support the user and with respect to programming style and program structure. Another consequence is that this know-how about interactive programming has not been properly exposed: the researchers have tended to consider the principles embodied in the program as the principal result of their work, and the craft of programming as a trivial aspect. They have probably been less motivated to discuss programming methodology than average programmers are, since the style is one of throw-away programming. To the extent that attention has focused on programming systems and programming methodology, these have been viewed as potential applications of artifical intelligence techniques. The emphasis has therefore been on longer-range goals and relatively utopian systems. One example is provided by Winograd [5].

An additional reason for the reluctance in the LISP community to discuss programming style may be a reaction to the debate about **goto**, which raged in the community in the early sixties. Programs without **goto** are written in LISP using recursive procedures and/or standard procedures with open procedural arguments, and a so-called "PROG feature" enables a restricted form of **goto** locally in a block. The controversy was resolved by a general agreement that the matter was not as important as it first seemed; this has discouraged subsequent discussion of other aspects of programming style.

Current Dialects and Implementations

Most implementations of LISP systems were accomplished in a research group that wanted to use the system, with strong feedback from users to implementers. As a consequence, the language has changed over time to satisfy new user needs, and several dialects have appeared.

The original variant of LISP was called **LISP 1.5** and is described in the *LISP 1.5 Programmer's Manual* [6], a document which is still a standard reference but has long since become obsolete. Besides the original implementation for IBM 7090, there have been implementations for CDC 3600, IBM 360/370, and the UNIVAC 1100 series. There is also an implementation written in FORTRAN [7].

The **LISP 1.6** dialect has been developed for DEC-10 under TOPS-10 (the manufacturer-provided operating system). This dialect again split into two branches, one at Stanford that preserves the name LISP 1.6, and one at MIT called **MACLISP**. The latter exists both under TOPS-10 and under the local operating system ITS (Incompatible Timesharing System).

The **INTERLISP** dialect was originally developed at Bolt, Beranek and Newman, Inc. (BBN) under the name **BBN-LISP**. After a part of the implementing group moved to Xerox Palo Alto Research Center and the responsibility for the system became shared, it changed to the new and more neutral name. The original implementation for INTERLISP was for DEC-10 under the TENEX operating system, also developed by BBN. It has been adapted for use under the TOPS-20 operating system. Additional implementations exist for IBM 360 and 370, and are being developed for other computers.

These three dialects differ in several important ways:

- Mechanisms for variable binding are different. All LISP systems use late binding, also called dynamic binding, but the three major types of systems use different mechanisms for storing the variable-value binding pairs. This is a fairly technical issue and not of interest here, but it is one of the more important points of incompatibility between programs in different dialects.
- Mechanisms for nontrivial input/output, file handling, etc. are different in the different dialects.
- INTERLISP has a very large repertoire of facilities that support the user in his work with his own program: MACLISP has fewer such facilities and organizes them differently, namely as separate progams rather than as part of an integrated programming system. This will be discussed further below.

Additional dialects and implementations exist, although a complete listing does not seem to be available. LISP for CDC 6600 builds on LISP 1.5 but has a number of special facilities, such as three-pointer cells. Variants of LISP 1.5 have been implemented on 16-bit PDP machines. The implementation of INTERLISP for IBM 360 has been transferred to computers from other manufacturers (Siemens, ICL) with compatible instruction sets.

The INTERLISP variant relies heavily on the special facilities provided by the TENEX and TOPS-20 operating systems. For the benefit of DEC-10 users running TOPS-10, the LISP 1.6 system has been modified and extended with a subset of INTERLISP's user support facilities. The work was done by a group at the University of California at Irvine, and the resulting system is called UCI-LISP.

Special-purpose processors running single-user LISP systems have been developed at Xerox Palo Alto Research Center, at the MIT Artificial Intelligence Laboratory, and at Bolt, Beranek, and Newman Inc. (BBN). All of these are experimental systems, and documentation is scant or entirely unavailable. The BBN machine is described in [8].

It is quite easy to write a simple LISP interpreter, and a large number of more or less complete systems have been developed, including multiple implementations for the same computer series. The list given here is therefore by no means complete. It is intended to give some idea of the range of existing implementations, and to characterize the two systems that will be used in the discussion that follows, namely MACLISP and INTERLISP.

A Simple LISP Application

The present paper is intended to discuss issues and principles of interactive programming systems that arose in the experience with LISP systems, rather than the details of LISP itself; it should be possible to read the paper without previous knowledge of the programming language LISP. For concreteness, however, the paper also presents specific LISP programs or sessions which illustrate some of the issues. These examples will be helpful for readers who have previously learned the LISP language, for example from one of the current textbooks [9, 10, 11, 12], but may be skipped by readers who do not know LISP. These examples are printed in smaller type than the rest of the text. The examples use the INTERLISP dialect, but should be equally intelligible to readers who are used to other dialects. (Of course, another dialect would have served as well for the examples.)

The following simple application will be used for all the examples. It has been chosen to illustrate some of the points in the section on requirements above. The task is to write a system which administers the calendars of meetings, appointments, etc. for a number of users. It should provide services such as the following:

- update and print-out each user's personal calendar;
- determine a suitable time for a meeting between two people, or a meeting of a committee, send a message to the intended participants, and update their calendars.

Later extensions might include these services:

- allow a user who receives a "computer mail" message about a seminar or other open meeting to transfer the message to the appropriate position in the calendar;
- reschedule appointments when a higher-priority or less easily movable appointment is suggested.

The set of services should be extensible in response to suggestions from the users. I will attempt to illustrate how the problem may be approached and the first stage of the program development, but certainly not the full solution.

In working out this application we bypass the problem of how the system can be made to communicate with several users at once, and assume that all calendars go into the same database. Obviously our program will emphasize simplicity rather than economy of operation. It will be referred to in the balance of the paper as the *demo program*.

In the first step of program development, the programmer selects a representation of data in the database and makes a first guess about which procedures he will need to write. These decisions are of course based on the given list of required or suggested facilities in the final system. The creative activity of making those decisions will also be bypassed here. Once the data structure has been chosen, the writing of the procedures is usually a straightforward task.

In the chosen data structure for the demo program, each user is represented as an atom, e.g., LARSSON. The user's calendar is represented as a property called CALEND, which might look as follows:

```
((  (MON JAN 12)
    ((9 15) (10 00) SEE ANDERSON)
    ((10 45) (11 00) SEE LUNDSTROM)
    ((13 00) (15 00) ATTEND
       X-COMMITTEE-MEETING))
 (  (TUE JAN 13)
    ((10 30) (11 30) ATTEND
       (INFORMATION ABOUT NEW
       PRODUCTS))
    . . . )
    . . . )
```

In general, it is a list of day-plans, where each day-plan consists of a date followed by a list of appointments. Each date is a list of day-of-week, month, and day. Each appointment is a list of starting time, ending time, verb, and object. Each starting time and ending time is a list of hour (on a 24-hour clock, for simplicity) and minute.

For convenience, we define selector functions which retrieve the components of some of these structures, e.g.,

```
(DEFINEQ
(HOURS
  [LAMBDA (TIME)
    (CAR TIME])
(MINUTES
  [LAMBDA (TIME)
    (CADR TIME])
(FROM
  [LAMBDA (APP)
    (CAR APP])
(TO
  [LAMBDA (APP)
    (CADR APP]))¹
```

We also define some other elementary functions, such as the obvious next-day function:

```
(DEFINEQ
(NEXTDAY
  [LAMBDA (DATE)
    (CONS (GETPROP (CAR DATE)
                   (QUOTE NEXTWEEKDAY))
          (COND
            ((EQ (CADDR DATE)
                 (GETPROP (CADR DATE)
```

[1] Procedure definitions are reproduced here as printed by INTERLISP's indentation printing program, *pp*. A left square bracket is equivalent to a left parenthesis. A right square bracket is equivalent to one or more right parentheses, sufficiently many to match back to the corresponding left bracket or to the beginning of the expression.

```
                    (LIST (GETPROP (CADR DATE)
                              (QUOTE NEXTMONTH))
                          1))
                   (T (LIST (CADR DATE)
                            (ADD1 (CADDR DATE]))
```

This definition assumes that the standard data about the succession of months and weekdays, and the length of months, are stored as properties. This information is input interactively by typing in:

```
(PUTPROP 'JAN 'NEXTMONTH 'FEB)
(PUTPROP 'FEB 'NEXTMONTH 'MAR)
. . .
(PUTPROP 'JAN 'NRDAYS 31)
. . .
(PUTPROP 'MON 'NEXTWEEKDAY 'TUE)
. . .
```

When we have done this in our interactive session, we can check out the procedure definition by typing in, for example,

```
(NEXTDAY 'MON 'JAN 10))
```

and getting back

```
(TUE JAN 11)
```

To check the exception case in the conditional, we type in

```
(NEXTDAY '(MON 'JAN 31))
```

and get back

```
(TUE FEB 1)
```

We are then fairly satisfied with the procedure.

A procedure to determine a shared free time period for a new appointment should run down successive days and for each day seek a time that is available for all participants. This may be done by first forming the set of free slots or "holes" for each person, and then forming intersections (in the obvious sense) of such sets of holes for different people. The set of holes is formed by the function *holesin* (*st, apl, et*), where *st* is the starting time (e.g., (9 00) = 9 a.m.), *apl* is a list of appointments during the day, and *et* is the ending time (e.g., (17 00) = 5 p.m.). The definition is obvious:

```
(DEFINEQ
(HOLESIN
  [LAMBDA (ST APL ET)
    (COND
      ((NULL APL)
       (LIST (LIST ST ET)))
      (T (CONS (LIST ST (FROM (CAR APL)))
               (HOLESIN (TO (CAR APL))
                        (CDR APL)
                        ET]))
```

Thus the value is a list of free slots, each indicated as a list of starting time and ending time. The given definition will include zero-length periods in the list, and might be trimmed to avoid that.

The procedure is immediately tested with an example. We set the global variable *apl1* to an appointment list, which we intend to use for testing purposes:

```
12_(SETQ APL1 '(
      ((9 15) (10 0) SEE ANDERSON)
      ((10 45) (11 0) SEE LUNDSTROM)
      ((13 0) (15 0) ATTEND
         X-COMMITTEE-MEETING)
      ))
(((9 15) (10 0) SEE ANDERSON)
((10 45) (11 0) SEE LUNDSTROM)
((13 0) (15 0) ATTEND X-COMMITTEE-MEETING))
```

Here "12_" is typed out by the system and serves to number the interaction (for future reference) as well as for a prompt. Then follows the expression we typed in, and the value that the system returned. We then use *apl1* in a test of *holesin:*

```
13_(HOLESIN '(9 0) APL1 '(17 0))
(((9 0) (9 15)) ((10 0) (10 45)) ((11 0) (13 0))
((15 0) (17 0)))
```

Thus, there are free slots from 9:00 to 9:15, from 10:00 to 10:45, etc.

Next, to form the list of common holes in two such lists, we define

```
(DEFINEQ
(COMMONHOLES
  [LAMBDA (L M)
    (COND
      ((NULL L)
       NIL)
      ((NULL M)
       NIL)
      ((BEFORETIME (FROM (CAR M))
                   (FROM (CAR L)))
       (COMMONHOLES M L))
      ((BEFORETIME (TO (CAR L))
                   (FROM (CAR M)))
       (COMMONHOLES (CDR L)
                    M))
      [(BEFORETIME (TO (CAR M))
                   (TO (CAR L)))
       (CONS (CAR M)
             (COMMONHOLES L (CDR M]
      (T (CONS (LIST (FROM (CAR M))
                     (TO (CAR L)))
               (COMMONHOLES (CDR L)
                            M]))
```

The definition is obvious. It arranges that the first hole in *l* will start before the first hole in *m*, and then checks the three possible cases: the first element in *m* entirely after the first element in *l*, entirely included, or partially included. It proceeds appropriately for each case.

The definition of *commonholes* assumes a "<" predicate on time specifications, with the obvious definition

```
(DEFINEQ
(BEFORETIME
  [LAMBDA (V U)
    (OR (LESSP (HOURS V)
               (HOURS U))
        (AND (EQ (HOURS V)
                 (HOURS U))
             (LESSP (MINUTES V)
                    (MINUTES U]))
```

Again the definition is immediately tested:

```
36_(SETQ H1 (HOLESIN '(9 0) APL1 '(17 0)))
(((9 0) (9 15)) ((10 0) (10 45)) ((11 0) (13 0))
((15 0) (17 0)))

37_(SETQ H2 '(
     ((8 45) (10 30))
     ((10 40) (11 30))
     ((11 50) (12 20))
     ((12 40) (15 30)) ))
(((8 45) (10 30)) ((10 40) (11 30)) ((11 50) (12 20))
((12 40) (15 30)))

38_(COMMONHOLES H1 H2)
(((9 0) (9 15)) ((10 0) (10 30)) ((10 40) (10 45))
((11 0) (11 30)) ((11 50) (12 20)) ((12 40) (13 0))
((15 0) (15 30)))
```

Finally, to find which periods of duration $\geq d$ minutes are free for both persons *p1* and *p2* on day *date*, we define (using the iterative statement of INTERLISP):

```
(DEFINEQ
(COMMONTIME
  [LAMBDA (P1 P2 DATE D)
    (for V in (COMMONHOLES
        (HOLESIN STARTDAY
            [CDR (SASSOC DATE
                     (GETPROP P1
                        (QUOTE CALEND]
            ENDDAY)
        (HOLESIN STARTDAY
            [CDR (SASSOC DATE
                     (GETPROP P2
                        (QUOTE CALEND]
            ENDDAY))
            collect V when (GREATERP
            (DURATION V) D]))
```

This definition looks up the calendars for the given date, determines the remaining free time for each, combines them using *commonholes*, and selects a subset of the resulting list, consisting of those elements which have the required duration. The definition uses two auxiliary functions. The function *sassoc* is like the standard LISP *assoc*, but does the comparison with *equal* rather than *eq* and is defined with INTERLISP's iterative statement as

```
(DEFINEQ
(SASSOC
  [LAMBDA (X L)
    (for V in L do (RETURN V)
        when (EQUAL X (CAR V]))
```

The function *duration* which converts a time-interval into a number of minutes is defined as

```
(DEFINEQ
(DURATION
  [LAMBDA (P)
    (PLUS [TIMES 60 (DIFFERENCE
                (HOURS (TO P)) (HOURS (FROM P]
              (DIFFERENCE (MINUTES (TO P))
                (MINUTES (FROM P]))
```

Both of these are of course trivial.

The global variables *startday* and *endday*, intended to specify the beginning and the end of the working day, are defined through:

```
45_(SETQ STARTDAY '(9 0))
(9 0)
46_(SETQ ENDDAY '(17 0))
(17 0)
```

To test *commontime*, we design simple calendars as CALEND properties of LARSSON and SVENSSON, and then find a common time for them. We also make simple tests of *duration* and *sassoc*:

```
48_(DURATION '((8 45) (9 15)))
30
49_(DURATION '((10 30) (12 50)))
140

50_(PUTPROP 'LARSSON 'CALEND
        (LIST (CONS '(MON JAN 10) APL1)))
(((MON JAN 10) ((9 15) (10 0) SEE ANDERSON)
((10 45) (11 0) SEE LUNDSTROM)
((13 0) (15 0) ATTEND X-COMMITTEE-MEETING)))

51_(PUTPROP 'SVENSSON 'CALEND
    '(  ((FRI JAN 7)
    ((10 30) (11 30) SEE GUSTAFSSON)
    ((14 15) (16 15) ATTEND (INFORMATION
        ABOUT NEW CAR-POOL RULES)))
       ((MON JAN 10)
        ((9 0) (10 30) SEE PERSSON)
        ((13 30) (14 45) VISIT LIBRARY))))
```

(printout of value omitted)

```
 64_(SASSOC '(A B) '((A AA) (B BB) ((A B) AABB)
     (C D)))
 ((A B) AABB)
 77-(COMMONTIME 'LARSSON 'SVENSSON
     '(MON JAN 10) 90)
(((11 0) (13 0)) ((15 0) (17 0)))
```

Thus, among the common free slots for Larsson and Svensson on January 10, only those between 11:00 and 13:00, and between 15:00 and 17:00, were 90 minutes or longer.

In this fashion the program development continues. The example may raise questions regarding the notational aspects of the programming language, and regarding the programming methodology. These issues will be treated at length later on in the paper, with reference to this example. Let us simply note here that the example has illustrated some key characteristics of the language:

- There are two unusual data structures in LISP, namely the properties of atoms which allow one to store binary relations in the data base, and the list structures which allow one to form composite expressions such as, in the example, the calendar for a number of days.
- The data base (stored on property-lists) is used not only for the data of the program, but also for constants of the program such as the NEXT-MONTH and similar properties.
- LISP atoms are different from named scalars in PASCAL since they are dynamic. New atoms are created all the time during the interactive session. (In PASCAL, scalars have to be declared in the program, and I/O of named scalars is not defined in the standard language.)
- Each procedure definition is a list structure. It can therefore be processed like any other data in the system. The question of readability and syntactic sweetening in programs will be discussed later in this paper.
- To define a procedure in the system, one uses a procedure-defining procedure (*defineq* in our example) which takes as argument a list of a procedure name and its intended definition. Procedure-defining procedures have all the properties of other procedures; in particular, they may be called from other procedures, which is the basic requirement for a program-generation facility.
- The normal routine in program development is to first structure one's data, then write the corresponding procedures (which are often obvious when the data structure is given), and then immediately to run a number of test examples for the procedure. This routine is possible because list structures have a textual representation in terms of sequences of characters, as well as an implementation in terms of records and pointers in the programming system. Therefore, I/O for them is defined.
- The argument/value relationship ("input-output characteristics") is more useful for describing what a procedure does than comments at arbitrary places in the program.

2. RESIDENTIAL AND SOURCE-FILE SYSTEMS

The description of the demo program omitted a discussion of debugging and program correction. The approach to this issue is different in different LISP systems. We will discuss it here with special reference to two systems, INTERLISP and MACLISP.

Interactive program development consists alternatingly of *tests* of procedures (or other program parts) and *editing* to correct errors discovered during the tests and/or to extend the program. INTERLISP systems, unlike many other interactive programming systems, support both of these operations. The user talks exclusively to the INTERLISP system during the interactive session. Procedures are stored in internal form as data structures in the system, and editing of procedures is done by rearranging those data structures. We shall refer to this architecture as a *resi-*

dential interactive system, since the primary copy of the program (the copy that is changed during editing operations) resides in the programming system itself. The executing system (containing procedures in internal form, test data, etc) can also be preserved *between* runs as a dump of the virtual memory, and will be referred to as an *incarnation* of the system.

The MACLISP implementation, on the other hand, assumes the existence of a separate text editor. Programs are maintained as text files, which can be read ("loaded") by the LISP system. When a program is to be changed, the user leaves the LISP system, uses a text editor that operates on the program file, returns to the LISP system, and reloads that file. We refer to this as a *source-file system*. This is of course also the standard way of implementing and using languages other than LISP.

The LISP 1.6 implementation represents an intermediate stage. It contains a resident editor ("ALVINE") but one which is not as well-developed as the one in INTERLISP, and many users prefer to use a general-purpose text editor, thus running in source-file mode.

The primary source for an evaluation of residential vs source-file systems should of course be user reaction. Unfortunately, this source of verdicts is not particularly reliable: as usual, users of the various systems tend to prefer the one with which they are familiar. Also, it is hard to isolate the issue of source-file vs residential systems from the many marginal circumstances that influence the convenience of the respective systems. The purpose of the present section is to discuss those circumstances.

Parallel Jobs

When a source-file system is used, the user must be able to maintain the programming system and the editor as parallel jobs, and to switch between them using a minimal number of keystrokes. Such service is provided by MACLISP through ITS and at Stanford through their terminal system, but is not available to users at other locations who run LISP 1.6 or MACLISP under TOPS-10.

Use of Text Files in a Residential System

INTERLISP contains a facility (**"makefile"**) whereby procedure definitions in internal form can be printed on a text file in an input-compatible format, i.e., those files can be read into the system again.

Example. In the demo example, if the user decides to name the file CALEND, he might define the contents of the file through

```
(SETQ CALENDCOMS '(
    (FNS HOURS MINUTES FROM TO
         NEXTDAY HOLESIN COMMONHOLES
         BEFORETIME COMMONTIME SASSOC
         DURATION)
    (PROP (NEXTMONTH NRDAYS) JAN
         FEB ...)
    (PROP (NEXTWEEKDAY) MON TUE ...)
  ))
```

Here the global value of the variable *calendcoms* is set to a list of three commands, the first of which specifies which procedure definitions are to be included in the file. The second command specifies a number of objects, namely the "names" of months, whose NEXTMONTH and NRDAYS properties are to be printed on the file, and similarly for the third command. Then doing

```
(MAKEFILE 'CALEND)
```

will cause the file to be created. Similarly, to store the calendars of a number of persons according to the storage conventions of the demo program, one might type

```
(SETQ CURCALENDCOMS '(
    (PROP (CALEND) LARSSON SVENSSON ...)
  ))
```

(remember that the calendar was stored as the CALEND property of the owner) followed by

```
(MAKEFILE 'CURCALEND)
```

The contents of these files are loaded back by typing

```
(LOAD 'CALEND)
(LOAD 'CURCALEND)
```

The *makefile* facility might seem redundant, since the philosophy of residential systems implies that one should be able to use the same incarnation during the entire lifetime of the program, but is useful for several reasons:

1. *Transfer of programs*. An auxiliary program that has been developed in one environment might be needed in another. A text file of procedure definitions can be printed from one incarnation and loaded into the other.
2. *Back-up*. A system incarnation is relatively vulnerable to system failures and to mistakes by the user. Text files of all procedures (and related data) are useful as a back-up.
3. *Complete garbage collection*. Printing out all procedures and all necessary data as text files, and reloading them into a fresh incarnation, serves as a strict form of garbage collection.
4. *Copy for the user*. Some work with a program is best done by having a paper copy of the whole program to read and annotate. Saved text files can be listed and used for this purpose.

The usefulness of the second and third features depends in each instance on the reliability of the system, in the broad sense, and on the thoroughness of the garbage collector. The fourth feature has been debated: several INTERLISP users feel that copy for the user should be produced by a printout program which is separate from the file generator. This would make it possible to make the printout nicer-looking (since the reloadability restriction would no longer apply), and would also permit more compact reloadable text files, since legibility for users could be ignored almost entirely in such files.

The file generated by *makefile* merely contains a sequence of forms, i.e., expressions of the same kind as those that are input interactively be the user. The function *load* reads the file using the same routine used for reading user input (except that some interactive features are suppressed). Corresponding to the FNS command in *calendcoms*, the file will contain calls to the procedure-defining procedure *defineq*; corresponding to PROP commands, it will also contain calls to a procedure that stores property values given as arguments. Using other commands, one can cause the file to contain calls to an arbitrary set of other procedures, for example for doing a computation to initialize global variables or the database. Such files are widely used in LISP programming regardless of dialect.

Size of Text Files

A large LISP program consists of a large number of procedures. It is common practice to organize it as a small number of text files, with many procedure definitions or other operations in the same file. For every file, there may be a parallel file containing the compiled versions of the same procedures. An alternative method would be to let every procedure definition be an individual file.

The usual practice has several disadvantages.

1. In residential systems it is a cause of inefficiency: if one or a few procedures have been changed, then the whole file containing those definitions must be regenerated. The operation of "pretty-printing" a file of procedure definitions (i.e., printing with meaningful indentation) is fairly expensive in processor time. Similarly, the whole file may have to be recompiled after an edit, even if compilation is done independently for each procedure, which again is a waste.

These have been major problems in the INTERLISP environment. The attempted solution has been to add extra intelligence (= a kludge?) to the pretty-printer, so that it will copy the old pretty-printed file character-by-character into a new file, and only substitute freshly generated output for those procedures where a change has been made. A similar addition has been made to the compiler. While these improvements on the INTERLISP system level provide the illusion that each actual file consists of many small files, it would clearly be more elegant and economical to design this facility into the operating system in the first place (particularly since very small files are potentially very useful for other purposes as well, e.g., for the user's archive of computer mail).

2. In source-file systems, the recom-

mended routine of text-editing the source file and reloading it into the programming system does not work well. For example, if the file also contains initialization operations as discussed under "nontrivial use of text files" above, they may cancel the test data that have been laboriously built up during the interactive session. Also, if some of the other procedures in the file have been temporarily changed in their internal versions, for example to obtain trace printouts, the changes will be overwritten. In such cases, one would like to reload just one procedure definition rather than the whole file.

In MACLISP, a facility has recently been introduced which attempts to solve the problem through a LISP procedure for editing procedures, which pretty-prints its argument as a temporary file, calls the text-editor and causes it to operate on that file, allows the user to edit, returns to the LISP system, and reloads the temporary file, all automatically. This, however, is a sensitive and error-prone process since the source files also have to be maintained, and does not seem to be in widespread use yet. Again, the problem would be resolved if each procedure definition were its own text file.

The ability to handle a large number of very small files is a nontrivial requirement on future operating systems (or to be precise, file management systems) intended for use in interactive program development. This requirement has several implications:

- There must be economical storage and representation of such small files. In LISP an average procedure definition may be a few hundred characters, and definitions of less than a hundred characters occur for many programmers (as exemplified by the demo program).
- File catalogs must have enough space and the right organization to handle the resulting very large number of files.
- File naming conventions must enable the user to operate conveniently with large numbers of files. It is no longer sufficient to let each file have one single name, even if it is mnemonic. Instead, tree-structured or property-oriented naming is necessary, together with utility functions that operate on groups of files.
- Since several programs in a residential programming system need to maintain and make use of knowledge *about* procedures, the interface between the file catalog and the database maintained by the programming system becomes critical. User-written programs must be able to retrieve information, and even store their own information in the catalog, which then generalizes into a "system database," and/or enable user programs to maintain their own "shadow catalogs" whose contents are a superset of the system catalog. Even in the second case, the supporting program that maintains the shadow catalog must be able to inspect the main system-supported catalog, and in some way keep up-to-date with it.

Handling of Comments

Different LISP implementations have developed different conventions for comments. In source-file system, comments are ignored by the read routine, and exist only in the text files. In residential systems, comments must clearly be preserved in the internal representation of procedures, so that they will be available when definitions are printed-out to the user or on a file. Thus if comments are a significant percentage of the total text they become a memory problem. Even if virtual memory is used, they contribute to fragmentation if they are stored in the same pages as surrounding code (assuming of course that the code is executed much more often than it is edited). On the other hand, if comments are located on separate pages, one may have to make "comment text" a separate data type and/or treat it separately in the general-purpose I/O for data structures to make it possible to use it for procedure definitions.

Many users prefer to have one relatively large main comment for each procedure which specifies the intended purpose and typical behavior of the procedure, and only very sparse additional comments. It is not at all clear that such comments should be integrated into the program code, since one may often want to print out those descriptive comments together with other documentation (such as cross-reference information) but without the code. Internally, it would be better to store the definition and the main comment of a procedure in two separate locations in the database, although of course one should be able to move easily from one to the other. In the text-file representation, it would be tempting to store the definition and the main comment as two separate files, if small files as discussed are made available.

Current Editing Methods

The question of residential versus source file systems is strongly interrelated with the question of editors. Existing LISP programming environments offer a range of different types.

MACLISP under ITS (the local operating system at MIT) coexists with an editor (a variant of the standard DEC text-editor, TECO) which permits display editing ("the real-time feature"), i.e., the user who runs from a display terminal may display the text that is being edited, and use single-keystroke control codes to move a cursor around the screen and perform changes (insertions, deletions) one character at a time. This is an extremely convenient and natural form of editing, but as noted above it is not easy to integrate such an editor into the programming system.

The INTERLISP editor, which is part of the INTERLISP system and therefore written in LISP, is a procedure for editing arbitrary list structures (= tree structures), but of course primarily intended for list structures that represent programs. The central routine is called with a list structure as argument, and receives user commands that operate on this tree. Elementary commands move a cursor up and down the tree, perform simple changes, and print out the "current" part of the tree (from the cursor and a few levels down). A wide repertoire of higher-level commands is available, for example, commands for searching occurrences of certain patterns and for substitution by pattern, all the time in terms of tree structures rather than text. This editor was developed for an environment of hard-copy terminals, and although it can obviously be used on display terminals as well it does not make full use of their possibilities.

Example of the INTERLISP editor. In the demo program, suppose we want to correct the procedure *holesin* so that it suppresses zero-length periods in the list that it returns as value. This is easily done by introducing two additional branches in the conditional expression in the definition of *holesin*, so that the expression comes to read:

```
(COND
  ((EQUAL ST ET) NIL)
  ((NULL APL) . . . )
  ((EQUAL ST (FROM (CAR APL)))
    (HOLESIN (TO (CAR APL)) (CDR APL) ET))
  (T . . . ))
```

where the dots indicate the two old branches in the conditional. This update would be performed using the following edit commands (commands in capital letters, comments in small letters):

```
(EDITF 'HOLESIN)
F COND
(-3 ((EQUAL ST (FROM (CAR APL)))
  (HOLESIN (TO (CAR APL)) (CDR APL) ET]
     insert new element before the third
     element of the current expression
(-2 ((EQUAL ST ET) NIL]
     insert new element before the second
     element of the current expression
PP
     pretty-print the current expression
     to check that it came out right
     (optional)
OK
     return from the editor
```

Both of these editors have higher-level capabilities, for example for defining and using macros. The INTERLISP editor may also call arbitrary LISP programs, which is a very powerful feature. Even if the language for writing macros in TECO has the power of a programming language, it is an advantage to be able to communicate between code that is executed in the editor and the database that the integrated programming system maintains for describing the program under development.

The choice between the residential INTERLISP editor and the display-oriented editing used in the MACLISP environment (as well as by some INTERLISP users) differs for different people, even after careful discussion and comparison of the alternatives, and may therefore be explained as a matter of personal taste. However, these two alternatives do not exhaust the full range of possible choices. We can distinguish three dimensions of choice, as detailed in Table I.

For each of these dimensions, Choice B is clearly better than Choice A, all other things being equal. A list-structure editor communicates with the user in terms of the real structures of interest, rather than their incidental textual representation. Also, everybody who has used a screen-oriented editor will agree to its superiority.

For the third dimension, size of increment, there is a possible intermediate case, namely the procedure-level edit in which the main programming system "knows" that a certain procedure is being edited, but does not know the details of how it is changed. The new MACLISP editor described above is on this level. Choice B is superior to this intermediate choice, which again is superior to Choice A, with respect to the possibilities they offer for bookkeeping when a procedure has been edited. For example, if the edit is intended to be definite rather than temporary, then information which has previously been computed from the old definition becomes obsolete and must be deleted and possibly replaced at once. If the procedure had been compiled, it now has to be recompiled. If the procedure-call structure or other similar formal documentation had been extracted from the program, that analysis must be reperformed or, better still, amended in those parts which need to be changed. If code had been temporarily inserted in the old definition, for example for trace printouts or statistics gathering, that code should probably be added to the edited procedure as well. (Editing should sometimes be done on a clean version of the procedure, rather than a version where these temporary changes have been made, so that the edit is retained when the insertion is removed, and so that program files will be correct when they are generated). The list could be continued; the general observation is that the editor is a crucial part of a residential programming system, and has to communicate with almost everything else. It is therefore very important to organize the total system so that such communication is helped rather than hindered. Both in principle and in practice, this is done much better when the programming system has access to more details of the editor's operations, i.e., the increments are small.

The trivial form of an editor is the one

TABLE I. POSSIBLE CHOICES OF EDITING CAPABILITIES

Issue	*Choice A*	*Choice B*
Objects on which the editor operates	Text	List structures
Preferred medium	Hard copy (i.e., a "print" command is used to display the current state of the edited item)	Screen (i.e., editor dynamically maintains a picture of the item that is being edited)
Size of increment in communication between editor and rest of programming system	File (i.e., editor is used to edit a file, which is then loaded into the programming system)	Single edit operation (i.e., each individual edit command is available to the rest of the programming system) (this corresponds to a "residential editor")

represented by the left-hand column, i.e., a hard-copy-oriented text editor that is applied to program files. The INTERLISP editor and the editing facilities of the MACLISP environment represent improvements in different dimensions. For the future, one would of course want a system that implements the advanced, Choice B facilities in all three dimensions.

Theoretically, the simplest way of accomplishing that is to write the entire screen-oriented editor, including input and output of individual characters, in LISP. Unfortunately this does not seem to be realistic, except in a personal-computer environment, because the run-time characteristics of a large LISP system makes it hard to implement low-level routines for continuously updating a screen. The dedicated single-user LISP machine at MIT, described above, however, has a screen editor written in LISP.

Procedure-Level Awareness of Editing

A possible compromise is to maintain the display editor as a separate program, but enable the LISP system to call it after having stored a procedure definition as an intermediate file. This approach could be used in a residential system. It would be easier to do there than it was in the source-file MACLISP environment, since the automatic maintenance of source files would no longer be necessary. This approach represents Choice B along the dimension of the medium, and the intermediate choice along the awareness-level dimension. Some disadvantages remain: this approach enables the programming system to know *which* procedures have been edited, but not easily *which parts* of procedures have been edited. It does not apply for certain types of data structures that do not print well, such as common sublists and circular structures. Nor does it apply when one is really interested in editing the contents of the "relational" database (property-list information) rather than list structures. Finally, the powerful ability to call arbitrary LISP code from editor commands is likely to be lost.

A variation of the same idea is to locate the display-oriented editor in the terminal communication interface of the operating system, and to let it operate on a buffer containing the immediately past history of the conversation between user and computer.[2] Besides trivial uses, for example for scrolling back to the previous interaction, the buffer would be used for editing of procedures in a residential system as follows: when the user asks to edit the definition of a certain procedure, the system prints out the definition on the user terminal. The user edits it, changes its status from "output" to "input", and sends it back to the computing system as a new definition of the procedure. This solution has the same disadvantages as the previous variation, but has the possible advantage that the editor in the terminal communication interface can be used for a number of other purposes as well.

Screen Supervisor

Another possible approach is to let the structure editor itself be integrated in the LISP system, but to let it communicate with a screen supervisor routine which is responsible for receiving commands (including cursor movements in some sense) and for updating the screen, and which is set up so that it can provide sufficiently fast response. One may want to locate it in a processor other than the one that runs the LISP system. In this architecture, the resident LISP editor and the screen supervisor routine have to communicate for each edit operation made by the user. One would expect systems of this kind to consume a lot of processor time, which, however, should not be a big problem in the future. But some hard design problems must also be solved for this approach, and it should be regarded as an interesting research topic and not a simple implementation job. An experimental system has recently been presented by Teitelman [13].

Among these alternative implementation strategies, the approach using a screen supervisor routine implies struc-

[2] Such a facility has been proposed at the Stanford Artificial Intelligence Project (the "page editor").

ture editing, whereas the approach of using a display editor which communicates with the LISP system in increments of one procedure at a time can be implemented more straightforwardly if one assumes text editing rather than structure editing. However, the use of a text editor enables one to use the same editor for several programming languages and for ordinary text, as opposed to having a separate editor for each purpose. In some computing environments this is a significant advantage.

Efficiency Considerations

Since a residential system may be expected to provide more services to the user, one would presumably be willing to pay for this by a certain increase in resource requirements, namely processor time and/or memory requirements. Even if one figures on computer costs only, a system which makes it possible to develop the same program in a smaller number of terminal hours because it has better debugging facilities may clearly be acceptable even if its costs per terminal hour are higher.

It is, however, very hard to quantify the variables in this tradeoff. There is considerable controversy regarding the relative resource consumption of MACLISP and INTERLISP for the DEC-10. Also, these systems differ in ways which are significant for resource consumption, or measurements of resource consumption, but which are not directly related to the issue of residential versus source-file systems; for example, INTERLISP is more complete in its checking of possible user errors. Some other difficulties are related to the issue at hand, but would go away if the system were built right. As noted earlier, printing text files of program definitions is quite costly; this problem would be greatly reduced if each procedure were its own file, as proposed above. Additional considerations are introduced by system facilities that are expensive in resources (time or space) and unwanted by many users, but hard to remove from the integrated system. With all these incidental circumstances, it is very hard to tell how costly the residential design as such has to be. Furthermore, if the present optimistic expectations about the advent of personal computers for LISP materialize, the question might lose its interest as the residential design becomes affordable.

Complexity Considerations

A related problem in residential systems is their complexity. There are so many things that one would like to do in a residential system with an integrated set of facilities for the user, and which in principle one could clearly do. Unfortunately too many of these features interrelate in too many ways, and it becomes very hard to control the resulting complexity. This leads to unpredictability, unreliability, unhabitability, undocumentability, and other unpleasant properties. As always, the user, who was the intended beneficiary of the system, instead becomes the victim.

Summary

The residential system design offers many possibilities to the system designer, and should in principle allow very attractive systems. But in designing such systems, one must be careful not to sacrifice objectives which are of basic importance to users, such as an editing regime which is as convenient as possible; convenient handling of text files; convenient storage of comments; efficiency; and control of complexity.

3. SUPERIMPOSED LANGUAGES

One of the consequences of the program/data equivalence in LISP is that it is useful as a high-level implementation language for new languages that the user invents in the course of his work. This is clearly a useful property in experiments with programming systems. It has been used in a number of different ways.

Providing a Surface Language

The usual syntax for writing LISP programs as parenthesized expressions ("S-expressions") was illustrated in the demo example. It is widely known and sometimes abhorred by people that do not use the language. This notation came up in the first place as a corollary of two basic design choices in LISP, namely that 1) programs should be represented as data structures; and 2) I/O for all data structures should be defined.

Thus in the basic programming system one does not have to implement a special program for reading source programs and convert them to internal representation as list structures: it is possible to rely on the general-purpose I/O, and it is clearly advantageous to do so in a minimal implementation. However, if one should prefer some other notation for programs, one may of course implement an alternative read program which converts that notation to the internal representation.

Example. Suppose we want to use an ALGOL-like notation, with the following infix operators for list structures:

l.hd	the head (first element of) the list *l*
l.tl	the tail of the list *l*, i.e. the remainder after the head had been removed
l o m	the list obtained from the list *m* by adding one element, *l*, at the front
$\langle l_1, l_2, \ldots l_m\rangle$	the list with the l_i as successive elements

Then the procedure *holesin* in the demo program would be written as

```
holesin (st, apl, et) = =
  if apl = NIL then ⟨⟨st, et⟩⟩
  else ⟨st, from (apl.hd)⟩ o
        holesin (to (apl.hd), apl.tl, et)
```

instead of the S-expression representation:

```
(HOLESIN
  [LAMBDA (ST APL ET)
    (COND
      ((NULL APL)
        (LIST (LIST ST ET)))
      (T (CONS (LIST ST (FROM (CAR APL)))
               (HOLESIN (TO (CAR APL))
                        (CDR APL)
                        ET])
```

A LISP program for reading the ALGOL-like notation would rely on low-level input primitives: the LISP system kernel contains procedures both for low-level I/O (which read or write one character or one atom at a time) and high-level I/O (which read or write a list structure as a parenthesized expression in one single stroke). The program reader itself must of course be entered into the system in S-expression form, as in the second example above.

Notice, then, that such a program is *not* a translator from the ALGOL-like notation to the S-expression representation above. There are three representations involved: the list structure (tree structure) used inside the computer, the textual S-expression representation, and the textual ALGOL-like representation. The general-purpose input routine translates from the second to the first representation; the alternative input routine for programs, from the third to the first representation.

We shall use the term *surface language* for a notation for programs that has to be read by a special program reader, e.g., the ALGOL-like notation in the example. The issue of such surface languages is bound to come up whenever one organizes a programming system according to the criteria in the section on requirements above, and it may therefore be worthwhile to summarize the LISP experience with them.

In the early stages of LISP's history, everyone seems to have assumed that surface languages would appear. After all, John McCarthy, who originated LISP, was also a member of the committee that defined ALGOL 60. The name LISP 1.5 seems to indicate that it was an intermediate solution to be used until LISP 2 (which would use such a surface language) appeared. The LISP 2 project started out ambitiously, but reportedly caught the PL/I disease (proliferation of features) and did not enjoy the PL/I antidote (money), and therefore was never completed.

Since that time, a considerable number of readers for surface languages with the same intended purpose have been written [14, 15, 16]. Users have been reluctant to adopt these systems; most users of LISP systems prefer not to use them. Some reasons are:

● The representation obtained from

standard high-level I/O, although forbidding at first, is quite easy to work with when one gets used to it.

- The major problem when writing S-expressions, namely matching of parentheses, has been solved with the introduction of "super-parentheses" such as the right bracket in the example above.
- The legibility problem has been solved by systematic indentation ("pretty-printing") to indicate depth of parentheses, equivalent to depth in the tree structure.
- It is very difficult to write a good surface-language system. Writing the translator itself is the trivial part; what is harder is to make sure that error diagnoses and other communications with the user will come out right. One must allow mixed representations such as those needed to support program-generating programs.
- In integrated systems, these problems become much harder: one must arrange for correct printout of the surface language from the internal representation, which means that either the transformation must be reversible or the source form must be preserved. One must also modify the editor to allow the user to edit in the surface representation. It turns out that the surface language processor tends to interrelate with almost every part of the integrated programming system, i.e., there is a serious structuring problem.

The moral of the story is that if a surface language is to be used, then it should be designed into the system from the start. ECL [17] has taken this approach; it is only described to the users in terms of its surface language, although the internal structure of its programming system is very similar to a LISP system. But one should also be aware that the decision to use a surface language will place a heavy burden on all other facilities that one wants to build into the system. Further, it isolates the user from the internal representation of programs, and discourages him from using it for his own purposes.

Viewing Control Primitives as Procedures

If one decides not to have a special reader for procedure definitions, one must look into others ways of improving the local readability of procedure definitions. Let us discuss first the issue of control primitives, where several approaches have been tried.

In its original conception [6], LISP was a very simple programming language with conditional expressions, and function calls allowing recursion as the only control primitives. While these primitives are theoretically sufficient, they force the programmer to introduce and name a new procedure almost every time a loop is to be performed. This is clearly inconvenient. However, other control primitives can easily be accomodated within the syntax of LISP's standard input/output, i.e., nested tuples.

The obvious notation for an iteration statement and for local *goto's* in a block are exemplified by the following two equivalent procedures for printing n and n^2 where n ranges from 1 to m:

```
(DEFINEQ
[PRINTSQUARES1 (LAMBDA (M)
  (FOR N FROM 1 TO M DO (PROGN
        (PRIN1 N)
        (TAB 10)
        (PRINT (TIMES N N ]
[PRINTSQUARES2 (LAMBDA (M) (PROG (N)
        (SETQ N 0)
 LOP (SETQ N (ADD1 N))
        (COND ((GREATERP N M) (RETURN)))
        (PRIN1 N)
        (TAB 10)
        (PRINT (TIMES N N))
        (GO LOP])
```

The *for* expression and *prog* expression in these examples satisfy the data-structure syntax, and also satisfy a basic assumption of the LISP interpreter, namely that in every list which is to be evaluated (every "form"), the first element must be a function name or operator which specifies how the rest of the form is to be treated. (Infix operators are harder to handle and will be discussed below). The *prog* primitive was implemented on the machine-code level in early stages of LISP's development, which established a *de facto* standard. The *for* primitive was not standardized. One can guess that the following reasons contributed:

1) It is easy for each user to define a *for* primitive similar to the one above, using the existing primitives of the language.
2) Many variations of *for* statements are possible, for example involving loops over the members of a list, and there are different ways of collecting a value of the expression—one may form a list of the value returned in each cycle of the loop, add those values if they are numbers, form their union if they are sets, or perform almost any other binary operation.
3) The prevailing ethics of LISP programming in its early days encouraged the use of so-called mapping functions. For the example above, one would define a mapping function that might be called *mapint*, and use it as follows:

```
(DEFINEQ
(PRINTSQUARES3 (LAMBDA (M)
  (MAPINT 1 M (FUNCTION (LAMBDA (N) (PROGN
          (PRIN1 N)
          (TAB 10)
          (PRINT (TIMES N N]
```

Here *mapint* would be defined as (in LISP 1.5)

```
(DEFINEQ
(MAPINT (LAMBDA (FROM TO FN)
      (COND ((GREATERP FROM TO) NIL)
            (T (PROG2 (FN FROM)
                      (MAPINT (ADD1 FROM)
                        TO FN]
```

which means it calls the procedural argument *fn* for integer arguments ranging from *from* to *to* in steps of 1. Mapping functions are nice and pure in theory but not so convenient in practice, since each user tends to build up a wildly growing fauna of different mapping functions. In a *for* operator, it is easier to accomodate different facilities through different choices of keywords such as *from* and *to* in the example. INTERLISP [4] contains a rich *for* statement in its program library; it allows one to write, for example, the following expression for forming a list of those members of a given list *k* whose position number is a member of a given list *p*:

```
(FOR X IN K AS I FROM 1 COLLECT X WHEN
    (MEMBER I P))
```

where *x* and *i* are the two iteration variables and run in parallel.

Mapping functions and the *for* operator have one thing in common: they are higher-level constructs which allow the user to express program control conveniently, and which have been defined from other and more basic primitives in the LISP system. Additional control structures such as a *case* construct (called *selectq* in INTERLISP) have also been developed in some LISP systems and by individual users. This development can be understood by viewing LISP as a very-high-level implementation language, and as an extensible language with facilities for defining new control structures.

If interpretation of the LISP program is sufficient, then then the implementation of constructs such as *for* and *selectq* only requires system primitives which exist in LISP 1.5 and all subsequent systems, namely the so-called FEXPR's (or NLAMBDA's) and also the *prog* facility. In order to compile them as well, one needs either a macro facility or handles on the compiler. (Handles will be described in a later section.) In fact, a macro facility may be viewed simply as a handle on the compiler. Current LISP systems are usually equipped with a macro facility for exactly this purpose.

Infix Operations

Perhaps the most striking difference between the basic LISP notation and conventional programming languages is the absence of infix operators. The interpreter assumes as input a data structure where the procedure name or operator is the first element of every sublist. Such data structures print as shown in the following example: $a*b+c-d$ become

```
(PLUS (TIMES A B) (DIFFERENCE C D))
```

Several methods are possible for allowing the user a more natural notation:

1) Introduce a special operator that indicates infix expressions, so that one would write the above example as

   ```
   (| A TIMES B PLUS C MINUS D)
   ```

   The | operator would then be defined in the same way as *for* and *selectq*, discussed in the previous section.
2) Modify the LISP interpreter to allow, for example,

   ```
   (A TIMES B PLUS C MINUS D)
   ```

3) Modify the I/O conventions so that the printed expressions are infix, but the internal representation as datastructures is prefix like before.
4) Rely on the exception handling mechanism in the interpreter. The user is allowed to type, for example,

   ```
   A*B+C−D
   ```

   which is read as one single atom by the input routines. When the interpreter encounters this "variable" an error occurs because the "variable" does not have a value. The error machinery will then inspect the name of the "variable," reconstruct its intended meaning, and convert to the basic LISP notation.

The first method does not look as well as the others, but is easy to implement and use and can be implemented by any user who wants an infix-notation facility. The second approach has been used in QLISP [19], but not in LISP for several reasons including compatibility. The third approach would very likely cause confusion for users, if provided as an add-on feature. However, Tholerus [18] has developed this idea in a systematic way; his proposal should be considered seriously in designing future systems. The fourth method is used by the CLISP facility in INTERLISP [4]. It has several advantages, such as allowing a natural notation and being transparent to users who do not use the facility (at least theoretically). However, the CLISP facility, which in its entirety contains several other mechanisms besides this one, is controversial among INTERLISP users: some like it, some do not. It is an open question whether this is due to remaining initial difficulties in the current implementation, or whether it is inappropriate to locate the infix/prefix transformation in the exception-handling machinery.

Since numerical computations are usually of minor importance in LISP applications, the availability of infix operators for arithmetic functions is not very important. Infixes for Boolean connectives and for common operations on lists, such as *cons*, *append*, *member*, *union*, and many similar functions are more significant. However, a possible objection is that infixes seem to do best in relatively small expressions, and may do more harm than good in complex expressions with many levels of application of functions, which are common in LISP. Consider for example the following translation into CLISP (in the INTERLISP system) of the function *nextday* as defined in our calendar application above:

```
(NEXTDAY
  [LAMBDA (DATE)
    ⟨(GETPROP DATE:1 'NEXTWEEKDAY) !
          (if DATE:3=(GETPROP DATE:2
                                'NRDAYS)
             then ⟨(GETPROP DATE:2
                                'NEXTMONTH)
                        1⟩
          else (LIST DATE:2 DATE:3+1)))⟩])
```

Here the exclamation sign is used as an infix operator for *cons*. It is not at all clear that readability has increased compared to the original definition.

Declarations

The term "declarations" is used here for information about entities in a program (variables, procedure names, etc.) which is stated once and then used during the execution of the program and/or in an analysis of the program such as a compiler. With this definition, declarations in an incremental system often take on a different appearance than in a conventional compiling system. In the conventional system, declarations are always statements in the program that is fed to the compiler. In an incremental system, where one has a database which may be updated and used by successive expressions that are evaluated by the system, it is possible to input declarations separately, to store them in the database, and to use them either statically when procedures are input or compiled, or dynamically when they are interpreted. This approach also makes it possible to extend the use of the declarations gradually, and thus obtin a more and more complete description of the program.

The first use of declarations in conventional languages such as ALGOL was to distinguish integer, real, and Boolean variables. This usage is not necessary in interpreted LISP, since the distinction is made in the data and determined dynamically during execution. The first use of declarations in LISP was instead to provide the compiler with information about free variables, i.e., variables that are declared in one procedure and used in another procedure deeper in recursion.

When LISP is to be compiled on today's conventional computers with typeless data, however, the compiler can make good use of typing information for numerical data (integer, real, etc.). Most LISP systems do this in the same fashion as assembler languages, i.e., by having a separate real plus, integer plus, etc., in addition to the general-purpose plus. The specialized functions are recommended for frequently executed parts of the program. Surface-language systems such as REDUCE [16] also allow conventional, ALGOL-like type declarations for variables, and do the same type checking, type conversion, and selection of operation as a conventional compiler.

The approach used by the current LISP systems in this area may seem shockingly primitive, but one must remember that numerical calculations play a very minor part in the artificial intelligence applications in which the systems are most used. REDUCE, on the other hand, is used mainly for formula manipulation, where heavy calculations do occur and where there is a fairly broad assortment of numerical or pseudonumerical types: real, integer, complex, rational, polynomial, etc.

Especially for artificial-intelligence applications of LISP, declarations of data structures are potentially more useful. Relatively little has been done to implement such facilities, and some users argue that they are not needed.

Example. Let us first illustrate with an example what data-structure can do in the LISP context. The data structure for the personal calendar in the demo program could be described as per Display 1, according to a design proposal for an extension to REDUCE [19].

The definition should be self-explanatory since the data structure has been defined informally above. The final LIST in the definition of TIME is a constructor function which is defined by default in the other declarations. With these definitions, the user can declare variables to have one of these types, for example

```
DECLARE FR:FROM;
```

and to select and change their components using a functional notation, for example

DISPLAY 1.

```
RECORD CALENDAR = LISTOF DAYPLAN;
RECORD DAYPLAN = STRUCT(DATE:DATELIST, APPS: LISTOF APPOINTMENT);
RECORD DATELIST = STRUCT(DAYOFWEEK:DAYNAME, MONTH:MONTHNAME, DAY:INTEGER);
RECORD DAYNAME = (MON, TUE, WED, THU, FRI, SAT, SUN);
RECORD MONTHNAME = analogous
RECORD APPOINTMENT = STRUCT(FROM:TIME, TO:TIME, VERB:ATOM, OBJECT:SYMBOLIC);
RECORD TIME = STRUCT(HOURS:INTEGER, MINUTES:INTEGER):LIST;
```

```
HOURS(FR).
```

This machinery obviously serves two major purposes, namely to make programs more easily legible, and to make type checking possible. Since REDUCE is a surface language to LISP's interpretable data structures, the type checking and choice of operation for each selector is done when a procedure is read into the system. The checking is done individually for each procedure that is input, and global declarations may be stored permanently in the database.

The arguments for such a use of declarations are the same as in other languages, and are well known. However, some LISP users oppose it on the same grounds as they resist separate readers for programs, namely that declarations used in this way interact unfavorably with other parts of the programming system and tend to insulate the program from the programmer. It is then often argued that 1) The kinds of bugs which can be diagnosed by type checks can also be quickly found and corrected in interactive program development; 2) Bugs in symbol-processing programs have more visible effects, and are therefore easier to find, than bugs in programs for, let us say, numerical computation or simulation; and 3) The obligation to declare the data types of variables in conformity with one of the proposed systems does more harm than good because it restrains the programmer.

These statements are of course contested, but their validity may be a matter of personal programming style.

Even if one does not need declarations for type checking, there is still the question of program legibility. Many LISP programmers solve this problem in the manner indicated in the demo program, i.e., by defining low-level procedures such as *hours* and *from* which select components of structures, and then use them rather than the elementary combinations of *car* and *cdr*. This practice can obviously be automated. This has been done in the so-called record package in INTERLISP, where some of the declarations in the above example for REDUCE could be written as

```
(RECORD DAYPLAN (DATE · APPOINTMENTLIST))
(RECORD DATE (DAYOFWEEK MONTH DAY))
(RECORD APPOINTMENT (FROM TO VERB
  OBJECT))
(RECORD TIME (HOURS MINUTES))
```

The facility in INTERLISP does not do any type checking, and only serves the purposes of legibility and data independence. It is weaker than the proposed facility in REDUCE in some ways; for example, it does not allow one to declare APPOINTMENTLIST to be a list of APPOINTMENTs in the example. It is stronger in some other ways, particularly since it allows records to be implemented in various ways such as lists, arrays, hash-arrays, property-lists, etc., and therefore achieves data independence over a wider range of realizations.

Still other uses of declarations are for documentation purposes and as support for utility programs that perform service operations on parts of the database, e.g., data entry, tabular presentation, and merges of structured data from different sources. The design of such facilities has been treated in [20] and [21].

These different experiments with the use of declarations in LISP systems have one thing in common, namely that they all work by extensions to the basic LISP system. The approach is similar to the one used for control primitives, as discussed above: the basic system is used as an implementation tool and for its extensibility. It does not contain the desired facilities, but it makes it easy to implement the facilities of one's choice, and to experiment with changes in these facilities. In the case of declarations, the program/data equivalence and the database during the interactive session are the significant properties of the language that make this approach possible.

The extensibility properties which make it possible to implement subsystems for control structures and high-level data structures are also generally available for the users of these subsystems. For example, the *for* construct in INTERLISP allows the user to add more keywords to the existing ones (*do, collect, while,* etc.) and to define how the new keywords are to be used. Similarly, the proposed new type facility in REDUCE stores coercion rules as properties of data-type names or, more abstractly, as a network with names of data types as nodes. This network is available to the user, who can extend it dynamically.

The language extensions mentioned here, such as REDUCE and the CLISP facility in INTERLISP, are by no means the only ones. Because of the ease of implementing simple facilities of this kind, many LISP users implement and use their own.

A conclusion from this experience is that extensibility in programming languages may not be particularly interesting as a goal in itself. If the language is designed on certain basic premises such as incrementality and program/data equivalence, extensibility is obtained automatically.

Very-High-Level Languages

The period 1970–75 saw a number of attempts to create very-high-level languages based on LISP. PLANNER [22], QA4 [23], CONNIVER [24], QLISP [25], and POPLER [26] are the best known ones. An early forerunner was LISP A [27]. These systems have been reviewed in [28].

The basic feature of these systems is that they integrate a number of ideas and facilities which could be described and discussed separately, but which enhance each other and therefore should be implemented together. They include:

- A common mechanism for invocation of programs and data. A procedure call and an instruction to check for the existence of a relationship in the data base should use the same syntax, and should be interchangeable.
- Pattern-directed invocation. Each invocation should be an expression containing constants and variables, which calls for the values of the variables to be computed through access to the database to find entries that match the pattern, and/or through calls to procedures whose "names" or description indicate that they can be useful for filling in blanks in such patterns.
- Nondeterminism, meaning that the computation may split in such a fashion that it continues independently in each of the branches. Pattern-directed invocation may result in nondeterminism, since one pattern may be matched in several ways and since several procedures or data items may be relevant for the same pattern.

The implementation of at least some of these systems was quite modular. It consisted of a pattern-matcher, a database handler, a nondeterministic executive, etc., with common conventions and mutual procedure calls as the common glue. In retrospect, it appears that the glue was not as strong as it first seemed.

Each of the modules could be built in many different ways, and the particular combination of choices implicit in any one system found very few satisfied users. Also, the resulting systems were big and clumsy, and users were reluctant to pay the price in computer resources of using the entire systems. Except for a few striking early demonstrations, for example by Winograd [29], these systems do not seem to have been used much. The present trend is to make use of, and continue to develop, each module separately, and to use them as individual tools for various applications.

Higher-level languages of still another type built on the LISP system, namely embedded languages, are closely related to questions of program structure, and are discussed separately further on.

4. PROGRAMMING METHODOLOGY

The LISP community has barely participated at all in the last few years' debate on structured programming and other aspects of programming methodology. However, some program development methods are explicitly encouraged by properties which are now classical in the LISP language and existing LISP systems.

For bottom-up program development, it

is desirable to be able to test procedures as they are developed. LISP systems support this practice through the incremental implementation and the predefined I/O for data structures. This was illustrated in the demo program.

The complementary method of top-down programming (usually carried out through stepwise refinement) also suggests some reasonable properties for interactive programming systems, for example:

- If the definition of the procedure P contains a call to the procedure Q, but Q is not actually called for a certain argument vector **x** to P, then the programming system should be able to operate and to compute $P(\mathbf{x})$ even if Q has not yet been defined.
- If P calls Q as in the previous case and Q is undefined, but computation of $P(\mathbf{y})$ actually leadsto a call to Q, than the programming system should make a "soft landing." In other words, it should not print an error message and abort, but rather preserve the current environment and allow the programmer to inspect the situation, decide on a suitable value that Q could have returned, type it into the programming system, and make the computation continue.

The first of these possibilities is of course available in LISP, as in most interpretive systems. My own experience as well as reports from other programmers indicates that this method is standard practice.

The second method, i.e., interaction as a substitute for an undefined procedure, is supported by INTERLISP systems (and also, but not as smoothly) by MACLISP.

Example. The following sample console session illustrates what happens in our calendar application if the function *beforetime* (which is intended to check whether one hour-minute combination precedes another one) has not been defined when the function *commonholes* is tested. User input is in italics.

```
34_(COMMONHOLES H1 H2)
u.d.f.
(BEFORETIME broken)
35:IN?
COMMONHOLES: (BEFORETIME (FROM (CAR
M)) (FROM (CAR L)))
35: (FROM (CAR M))
(8 45)
36: (FROM (CAR L))
(9 0)
37:RETURN T
BEFORETIME = T
```

The system has typed out *u.d.f.* (for "undefined function") and the name of the offending function. The user typed the command IN? to find out where the problem occurred. He then typed in the arguments of the function in the offending position, to find out their current values. With this, he decided the value that the offending expressions should have, and allowed the computation to continue.

The next time the problem occurs, he decides that he had better define *beforetime* after all. He does so in the ordinary way, and then proceeds as follows:

```
41:GO
 BEFORETIME = NIL
 (((9 0) (9 15)) ((10 0) (10 30))
 ((10 40) (10 45)) ((11 0) (11 30))
 ((11 50) (12 20)) (12 40) (13 0))
 ((15 0) (15 30)))
```

The computation proceeds to the final answer.

The example is slightly artificial, since normally INTERLISP would not bother with an interaction when the error occurs at such a shallow level of recursion.

The common programming practice, however, is not to leave procedures undefined intentionally, but instead to make a simple definition at first, and later substitute a more elaborate one. The original definition may be a real dummy, for example it may simply return a constant value, but more often it is a viable although simplified case of the intended definite procedure. For example, input/output of object data is often done in the form of S-expressions (i.e., using high-level kernel I/O) in early stages of program development, and nicer-looking I/O is substituted later on. This enables the programmer to focus first on the essential aspects of the problem.

Example. This practice was illustrated in the calendar example, where we showed how calendars were output using the standard I/O for data structures. The resulting notation is obviously not acceptable for the final user, but is acceptable for small test cases during program development. As a first step towards improved readability, one could try the system's standard pretty-printer (for printing using indentation), which would print Svensson's calendar in the example above as

```
(((FRI JAN 7)
 ((10 30)
  (11 30)
  SEE GUSTAFSSON)
 ((14 15)
  (16 15)
  ATTEND
  (INFORMATION ABOUT
          NEW CAR-POOL RULES)))
 ((MON JAN 10)
 ((9 0)
  (10 30)
  SEE PERSSON)
 ((13 30)
  (14 45)
  VISIT LIBRARY)))
```

As this clearly did not work very well, one could define a special printing function for calendars as

```
(DEFINEQ
(PRINCAL
  [LAMBDA (P)
  (for DAY in (GETPROP P (QUOTE CALEND))
      do (PROGN (PRINT (CAR DAY))
                   (for ENTRY in (CDR DAY)
                   do (PROG2 (TAB 6)
                                (PRINT ENTRY)))
                   (TERPRI]))
```

This results in the output shown by Display 2.

This example serves to illustrate two different principles. First, we notice that a few lines of programming produced a reasonably legible printout, and infer that 15 or 20 more lines of straightforward code should be sufficient for producing a listing that is entirely satisfactory to the final user, with 10:30 instead of (10 30) etc. The programmer can therefore confidently assign low priority to communication of data from the program: it is something which almost takes care of itself, with just a little help. An analogous situation holds for input of data to the program. In a conventional programming language the programmer would have to write and debug input/output routines for data before even starting to test and debug any other code.

The example also illustrates a method of program development which may be characterized as *structured growth*: an initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules. The process continues recursively as each module is rewritten. The principle applies not only to input/output routines, but also to the flexibility of the data handled by the program, the sophistication of deduction, the number and versatilty of the services provided by the system, etc. The growth can occur both "horizontally," through the addition of more facilities, and "vertically" through a deepening of existing facilities and making them more powerful in some sense.

That LISP users tend to prefer structured growth rather than stepwise refinement is not an effect of the programming system, since both methods are supported. I believe, however, that it is a natural consequence of the interactive development method, since programs in early stages of growth can be executed and programs in early stages of refinement cannot. Nor has there been much discussion about the matter, at least not in terms of abstract programming methodology. (The term structured growth has not been used before). Structured growth seems to be simply a

DISPLAY 2.

```
95_(PRINCAL 'SVENSSON)
(FRI JAN 7)
  ((10 30) (11 30) SEE GUSTAFSSON)
  ((14 15) (16 15) ATTEND (INFORMATION ABOUT NEW CAR-POOL RULES))
  (MON JAN 10)
  ((9 0) (10 30) SEE PERSSON)
  ((13 30) (14 45) VISIT LIBRARY)
```

technique that people tend to use spontaneously if they are given a free choice in an open environment.

An obvious objection to this programming method is that it may encourage "featurism," i.e., large programs with many different facilities rather than concise programs that embody a small number of powerful principles. A possible answer is that a computer program that approximates an aspect of intelligent human behavior very often has to account specifically for many different kinds of situations and thus that one really needs a programming technique for building feature-rich systems, whereas attempts to find simple principles are doomed anyway. Incidentally, the techniques for obtaining feature-rich systems may also be important for designing programs with which untrained humans find it comfortable to interact.

From the moralistic view of programming methodology, one might of course still argue that the method of structured growth is a bad habit, since it encourages one to make changes in programs, and changes are a known source of errors. The answer would be that even if some kinds of program changes are dangerous and/or bad, that does not prove that all of them are.

Stepwise refinement and structured growth are both variants of top-down programming, since they encourage one to decide on the overall structure of the program first, and then proceed to decisions about its parts. Much could be done in the programming system to further support top-down work in addition to what is done by current LISP systems, for example:

- Keep track of "open statements" or "expansion points." When users type in a piece of code which they know has to be refined or rewritten later, it should be possible to tell this to the system and ask later what work remains to be done.
- Distinguish different types of edit operations. Given an initial approximation to a program, some changes will be made to correct bugs, and some to extend the program. It would be desirable to treat these two kinds of changes separately, so that several layers of refinement or growth could be maintained in parallel. It is a common observation that early versions of a program are useful introductions to later, full-grown versions. Similarly, the value of (idealized) early-refinement steps as documentation has been emphasized in [30]. The problem, of course, is that the distinction between the different types of edits is not as clear-cut as one would wish.

5. PROGRAM STRUCTURE

The debate about structured programming in the world of conventional programming languages has been very much concerned with the proper syntax and semantics for control primitives, such as iteration statements (*for* loops), selection statements (*case* statements), and others. In one sense these questions are about the local structure in the program text. However, they also affect the global structure of programs because of the principle of hierarchical design of programs: the top-level structure of a program is a procedure or block which refers to subprocedures and/or subblocks recursively, but all entities on all levels are constructed in the same language. This view has been expressed by, for example, Dahl [31].

The feeling about these issues in the LISP community is quite different. Local smoothness of notation is often considered uninteresting, for example when the S-expression notation is used for programs, or when mapping functions are used rather than a *for* operator (see the section on superimposed languages above).

This attitude combines with a nonhierarchical view of program structure. Many LISP programmers seem to have a two-level model of their programs: on the higher level, a program is a collection of procedures, which are related through their invocation structure (procedure-call structure); on the lower level, each procedure has a definition. The higher-level structure is the important one for understanding a program: it is supported by documentation which specifies the behavior of each procedure on its data, by documentation tools for automatic extraction of the invocation structure, by the advice facility (see below), etc. Questions of control primitives clearly do not arise on this level. While they do arise when one descends into the interior of the black box that is a procedure, one tries to stay away from this as much as possible, and when one does work with the definition of a procedure one fully expects to face difficulties of many kinds, all trivial and uninteresting. Incidentally, the same two-level view of programs seems to be common among users of APL.

Program structure in LISP programs does not end with the procedure-call model, however. The following are some other principles of program structure that are prevalent in the programming practices of the community.

Programmable Systems

It is common practice in computer science to think of "elementary algorithms," i.e. subroutines which can be specified, analyzed, and programmed in closed form, and then serve as building blocks for larger programs. This concept may have been inherited from numerical analysis or from engineering. Knuth's works [32] are a systematic exposition of such algorithms.

In LISP programming, such algorithms often serve as mainframes rather than building-blocks for programs. The algorithm is written in open-ended form, i.e., some of its operations are marked as "user-defined" or as "handles." When the algorithm is to be used for a specific application, appropriate code is attached to each handle.

From a formal programming-language point of view, one might argue that such use of elementary algorithms does not alter their status as procedures since the handles may be easily implemented as procedural arguments. But the real issue is a different one: the view of the algorithm in the program structure is changed. Instead of being a small building block, which for many purposes can be viewed as a black box, it becomes a major way of determining the structure of the program. There are also other consequences. Conventional analyses of mathematical properties of algorithms become irrelevant, because of the arbitrary code that is attached to the handles. Also, in the practice of LISP programming, it is often more convenient to store the attached procedures in the database of the programming system, and let the general algorithm look them up as needed, rather than use a large number of procedural arguments. This matter is discussed in detail in [21].

A few examples of programmable systems involve:

Parsing of a programming language whose syntax can be described by a context-free grammar (if the borderline between syntax and semantics is chosen so as to make this possible). This is a classic problem, and a number of algorithms for this task have been developed and studied extensively. In a LISP-style language, it is more natural to write a parser which, for each reserved word and infix operator, looks up an associated piece of code and executes it. In practice it is natural to separate priority degrees as passive parameters outside the procedures, and let them determine when and how the operator-associated procedures are to be called. Thus handles may contain both passive data (parameters) and active data (procedures). Systems of this kind have been proposed independently by Pratt [34] and Tholerus [33].

Static analysis ("compile-time analysis") of conventional programming languages. Knuth [35] has proposed a scheme for such analysis that uses attributes associated with nodes in the syntactic parse tree. Nordström [36] has modified the method and allowed arbitrary procedures to be associated with productions and called in a more selective way, and thereby made systematic use of the handles on the basic algorithm. The resulting program structuring method is reminiscent of one proposed by Hoare [37] and results in a very well-structured program.

Analysis of natural language. Woods's Augmented Transitional Network Parser [38] is a widely used program that makes systematic use of programmability. Its kernel algorithm is that of a finite-state automaton, which has been made programmable by allowing handles in every state transformation. This also eliminates the restriction on the power of finite-state acceptors.

The LISP interpreter itself, like other routines in most modern LISP systems, is highly programmable and has a large number of handles. Thus, for example:

- The three steps in the system's *basic loop* (read–evaluate–print) are just initial assignments to handles, and may be changed as the user desires;
- *Exception conditions*, for example for undefined variables, cause a call to a procedure that the user may define;
- *Pretty-printers* are usually programmable so that they can be fitted to the user's needs.

Procedural Embedding

The classical view of a computer program, i.e., that a program is composed of a number of elementary algorithms and operates on a body of data, has failed in the LISP-using community not only through the emergence of programmable systems, but also through the doctrine of procedural embedding of knowledge. A short review of history may be in order here.

In the middle 1960's, it was suggested that certain artificial intelligence systems should be organized as theorem provers that would contain most of their knowledge as "data," i.e., as expressions in a logical calculus. The system would also contain a theorem-proving program which would make deductions from the available data, answer questions, etc. The advent of the resolution method in 1965 as well as some demonstrations of its use increased the enthusiasm for this view.

Strong objections to this view surfaced around 1970. It was argued that a logical calculus was not an appropriate vehicle for the kind of information that has to reside in an artificial intelligence system, and that major parts of the information in the computing system must be represented procedurally, i.e. as executable code, rather than as inert data. While it is not possible to portray this long controversy here, let us note that the current fashion among the remaining theorem-prover enthusiasts is to view a theorem-prover as an interpreter for a very-high-level language, which in a sense marks the triumph of the procedural school.

The rejection of theorem-proving represents the rejection of an algorithm which for a while seemed to be of basic importance, and even a rejection of the use of closed algorithms as the top level of a system. With resolution, one could make a mathematical analysis of variants of the resolution algorithm and hope to learn something about the behavior of the system being constructed. With the procedural approach, one is encouraged to think of large systems simply as programs, where closed algorithms appear only at the lowest levels. Mathematical analysis of the algorithms can then at best say something about the efficiency of the system, but not about the limits of its competence.

One might have expected this development to focus attention on issues of program structure and programming methodology, but in fact the questions of design of very-high-level programming languages came to dominate instead.

Embedded Languages

Another practice of program structuring, also related to programmable systems, is to organize the program around a highly specialized "language" that describes the application. For example, the INTERLISP *makefile* facility, which is used to generate text-format files of procedure definitions and initialization data, is organized as an executive which interprets a file descriptor for the desired file. The file descriptor contains information about which procedures are to be printed on the file, in which format they are to be printed, what other operations are to be performed when the file is generated and when it is loaded, and so on. This has already been illustrated in the example in the section on Residential and Source-File Systems above. The descriptor may initially have been a catalog or a parameter, but as the *makefile* package underwent structured growth, the descriptor acquired all the characteristics of a programming language.

The point lies not in the significance of this particular example, but in the general observation that the programming language has encouraged the use of such embedded languages. They are of course internally represented as data structures (making them easy to interpret) and externally as standard printouts of those data structures (which makes them sufficiently easy to read, and which makes I/O of expressions in the embedded language trivial). Since programs and data are interchangeable, it is possible for an expression in the embedded language, which of course is data to its interpreter and therefore to the LISP system, to contain calls to procedures that are written in LISP.

The programming method of using embedded languages is very widespread among LISP users, although many such languages are so small that they are not advertised as such. If one wants to enable programmers to mold the programming system to fit their needs, then use of embedded languages in this sense is probably a more viable approach than use of extensible languages.

The use of specialized languages for specific tasks in a class of applications represents a nonconventional method of decomposing complex problems. Instead of hierarchical decomposition, where a large problem is decomposed into small problems of the same kind as the top-level problem (for example, all subproblems are subprograms), we instead decompose the problem into two parts: 1) design an appropriate language for expressing solutions to problems similar to the given one; and 2) express the solution to the problem in that language. This method of decomposition has a definite advantage in situations where the given task may be redefined at a later stage. However, for implementing such specialized languages in a convenient fashion, one needs a very-high-level implementation language. When it allows the implementation of embedded languages, LISP serves exactly this purpose.

Production Systems

Embedded languages that started out as simple auxiliary devices sometimes grow into full-fledged systems. The very-high-level languages, as discussed above, may have followed this path. Another example is production systems, which have recently attracted attention as a nice way of structuring many types of problems (see, for example, [39]). Production systems have been used successfully in the MYCIN project [40].

Data-Driven Systems

There is one underlying method that makes possible programmable systems, embedded languages, and several other practices, namely what we have called *data-driven programming* [20, 21]. It is similar to indirect jumps in machine language, and can be described as follows: in conventional high-level languages, each procedure has a name, and one procedure calls another if the definition of the former explicitly contains the name of the latter. A data-driven call is one where the calling procedure accepts "input" data (for example, input from the user, or arguments to the procedure), looks up a procedure which in the database of the programming system has been associated with these data, and calls that procedure. This is not possible in conventional high-level languages, but is possible in LISP because of the program/data equivalence.

Example. In the domain of the demo program, suppose we want to define a procedure *nextweekday* which is a refinement of *nextday*. Remember that *nextday* computes the next day in the calendar after a given day, e.g., *nextday* of (FRI JAN 14) will be (SAT JAN 15). The new procedure *nextweekday* will skip Saturdays, Sundays, holidays, etc. Because of the variety of different reasons why a certain day may not be a working day ("weekday"), we decide to associate with each day-of-week (such as SAT) a procedure which takes a day as argument, and returns T if this day is an acceptable weekday as far as it knows, and NIL otherwise. We proceed similarly for the names of the months. In both cases, these procedures that implement local expertise are stored on the property-list of the day-of-week and the month, under the indicator OK-AS-WEEKDAY. Then *nextweekday* can be defined as shown in Display 3. In this case, using INTERLISP's iteration statement, *d* will range over *nextday* (*date*), *nextday* (*nextday*(*date*)), etc., until a *d* is found which satisfies the two criteria. Then the knowledge that Saturdays are never acceptable weekdays can be embedded as

```
(PUTPROP 'SAT 'OK-AS-WEEKDAY
  '(LAMBDA (D) NIL))
```

The knowledge that Mondays are acceptable except in August is embedded as

```
(PUTPROP 'MON 'OK-AS-WEEKDAY
  '(LAMBDA (D) (NOT (EQ (CADR D) 'AUG]
```

The knowledge that Christmas day is the only nonacceptable day in December except the ones accounted for by the days of week is represented as

```
(PUTPROP 'DEC 'OK-AS-WEEKDAY
  '(LAMBDA (D) (NOT (EQ (CADDR D) 25]
```

In general, the method described through this example makes it possible to organize knowledge as many small procedural chunks, and to associate these chunks of knowledge with *data items* (namely names of months and names of days of week) which occur in the application and which are therefore automatically understood by the programmer. This turns out to be a very common programming technique in the LISP community. One somewhat larger example is analyzed in detail in [21]. This technique strongly facilitates the writing of interpreters for embedded languages, and in programmable systems is an often preferred alternative to having procedural arguments. Also, an indirect or data-driven procedure call is equivalent to a **case** statement (at least for single-step indirectness and a static database), but is much more convenient to work with interactively since it is particularly easy to add additional case branches by adding more entries to the database (see the discussion above about structured growth), and since the data base of data-driven procedures may be presented in different ways to the user at different times. Nordström's method for organizing a program according to a syntax for the data structures of its application differs from Hoare's method discussed above in exactly this respect.

Although the LISP system kernel makes data-driven programming possible, the higher-level mechanisms in current LISP systems do not encourage it. One consequence of data-driven programming is that procedures are often not characterized by a single mnemonic name, but instead by the position in the database where they are located. However, many user-supporting programs in the INTERLISP system which operate on procedures assume that each procedure has an individual name (an atom) which is used for storing additional information about the procedure. This assumption is used in so many places that it is probably too late to change it in the present system.

One way of using data-driven programming is described by Aiello [41].

Low-Level Program Generation

All programmers, in any programming language, sometimes run into situations

DISPLAY 3.

```
(DEFINEQ
(NEXTWEEKDAY
  [LAMBDA (DATE)
    (FOR D_(NEXTDAY DATE) BY (NEXTDAY D)
       DO (IF (AND (APPLY* (GETPROP (CAR D) 'OK-AS-WEEKDAY)
                           D)
                   (APPLY* (GETPROP (CADR D) 'OK-AS-WEEKDAY)
                           D))
              THEN (RETURN D]))
```

Display 4.

```
(DEFINEQ
(OK-MONTH (M L) (PUTPROP M 'OK-AS-WEEKDAY
  (FUNCTION (LAMBDA (DATE) (NOT (MEMB (CADDR DATE) L]
```

Display 5.

```
(DEFINEQ
(OK-MONTH (M L) (PUTPROP M 'OK-AS-WEEKDAY
  (SUBST L 'L '(LAMBDA (DATE) (NOT (MEMB (CADDR DATE)
                                         (QUOTE L]
```

where they have to prepare nontrivial amounts of program in a routine fashion. Procedures and macros are well-known devices for handling such situations. The program/data equivalence in Lisp makes possible another and more powerful method to handle such situations, namely the use of procedure generators.

Usually, the concept of a program-generating program is surrounded by a mystique and a feeling that it is something that may arrive in the distant future. This is certainly appropriate with respect to programs that generate a whole program from its specifications. But a more pragmatic approach to program generation is evident in the Lisp community, namely the approach where the programmer identifies regularities in the structure of the program that he/she is writing or going to write, and immediately writes a small program generator that handles that regularity. Winograd mentions this technique in [42].

Example. In the implementation of *nextweekday* described in the previous example, we may wish to generate the procedure definitions for some of the months automatically. Suppose many months are characterized by a number of fixed dates during the month which are not acceptable weeks, i.e., their OK-AS-WEEKDAY properties have the form

```
(LAMBDA (DATE) (NOT (MEMB (CADDR DATE)
  '( . . . )]
```

where . . . indicates a list of integers. We could then define a procedure *ok-month* (m, l) where m is the name of the month, and l is the list of integers, in either of the two ways, shown in Display 4 and Display 5 respectively.

The first variant is the most elegant one, but assumes that the operator *function* returns a closure where the current value of l is preserved. Such closures or "funarg-expressions" were defined in the original Lisp 1.5, were ignored in some later implementations, but have gradually found their way back into the language. The second variant constructs the appropriate definition for each case by performing a substitution in a schema for the definition.

This procedure may then be called as, for example,

```
(OK-MONTH 'JAN '(1))
```

assuming that New Year's day is the only nonstandard holiday in January.

Advising and Insertive Programming

When low-level program generation is performed, one often wants to arrange matters so that several expressions or commands (usually input by the user) will make successive amendments to the definition of a procedure, for example so that each expression adds one more branch to a **selectq** (= **case**) statement. This is of course easy if the definitions are so regular that the program generator can correctly determine the correct place in which to make the amendment. This practice of insertive programming is discussed and illustrated in [21].

Example. In the data-driven OK-AS-WEEKDAY procedures used in the previous example, every such procedure will contain certain criteria for days that are not allowed. If we standardize the form of the procedures to be

```
(LAMBDA (DATE) (NOT (OR . . . . . . . . . )))
```

then it is easy to write a procedure *holidayrule* (m,x) which takes a month or day-of-week m and an expression x, where x is supposed to be the criterion for a holiday, and which adds x to the OK-AS-WEEKDAY property of m. The definition would

DISPLAY 6.

```
(DEFINEQ
(HOLIDAYRULE (LAMBDA (M X) (PROG (D)
  (SETQ D (GETPROP M 'OK-AS-WEEKDAY))
  (COND [(NULL D) (PUTPROP M 'OK-AS-WEEKDAY
             (SUBST X 'X '(LAMBDA (DATE) (NOT (OR X]
         (T (NCONC1 (CADR (CADDR D)) X]
```

go as shown in Display 6. It can then be called as, for example,

```
(HOLIDAYRULE 'JAN '(EQ (CADDR DATE) 1))
```

or (to declare the third Friday in September a holiday):

```
(HOLIDAYRULE 'SEP
      '(AND (EQ (CAR DATE) 'FRI)
            (BETWEEN 15 (CADDR DATE) 21)))
```

Here *between* is defined so that it is true here when 15 ≤ *caddr(date)* ≤21. In this fashion, separate holiday rules can be input independently, to gradually build up a procedure.

A similar although more complex facility is *advising*, which is meant to allow the user to make changes to existing programs without knowing their exact structure. In the simplest case, the user-programmers are supposed to know the system as a procedure-call structure, i.e., they know the names, intended purpose, and procedure-call structure of the procedures but these remain black boxes, and the users have no knowledge of the "inside" of the procedures. The advise facility in INTERLISP then allows the user to reroute outgoing or incoming procedure calls for a procedure, and to associate additional (side) effects with them.

In practice, advising is implemented through insertive programming, since every advised procedure is wrapped into an extra, outermost **begin-end** block ("PROG expression") whose structure is known to the advising routine. Of course, this machinery is supposedly invisible to the user.

Example. We wish to "tell" the procedure *nextday* in the demo application about the rule for February 29 during leap years and ordinary years. Assuming that the NRDAYS property of FEB is 29, we can do this by advising *nextday* to step to March 1 if its proposed value is February 29, and the current year is not a leap year. This is done in INTERLISP as:

```
(ADVISE 'NEXTDAY 'AFTER
  '(IF (AND (EQ (CADR !VALUE) 'FEB)
            (EQ (CADDR !VALUE) 29)
            (NOT (LEAPYEAR)))
       THEN (SETQ !VALUE (LIST (CAR !VALUE)
         'MAR 1]
```

Technically, the advising scheme is very elegant. Jim Goodwin has suggested [43] that (like data-driven procedures) it is an example of a machine-code-level facility that has been made available in a high-level form: advising is high-level patching. The crucial question about its usefulness is whether the user's actual request, the thing that really is to be done, can be translated into the right operation on the procedure-call structure (or whatever other model of the program the advising package supports). This is partly a question of how the program has been organized, and partly of how it has been documented.

In summary, the LISP programming culture offers a variety of unorthodox programming methods and program structures. Some, such as the structured growth method of program development, are really consequences of the interactivity and incrementality in the system. Others, such as data-driven programming, program generation, and the use of embedded languages, are consequences of the equivalence between programs and data, which LISP at present shares with only a few other research-oriented languages.

6. SESSION SUPPORT

One of the significant improvements in residential programming systems involves the possibility of letting the system support the interactive session, for example "remember" what the user has done before, and enable him/her to redo or undo those actions by convenient commands. This line of research has been pioneered by Teitelman in the INTERLISP system [44]. I shall only add here a few short notes on some nontrivial issues that arise in this context.

History – Structure or Text?

The INTERLISP system preserves the recent session history, i.e., a sequence of input expressions and corresponding output, in internal (list-structure) form. Some simple uses of this history are for scrolling and for the REDO command, whereby the user can reexecute a previously entered command, possibly after having edited it.

An alternative way of approaching the same objective might be to use a display-oriented editor in the terminal communication interface, as described in the section on alternative editing methods above. In addition to its previously described uses, such an editor could also copy previous input to a later place in the conversation buffer, edit it, and send it as fresh input to the programming system. The potential advantage to the user would be that one could manage with a smaller repertoire of commands, since the ordinary editing commands would also be sufficient for redoing and similar operations. On the other hand, there is the disadvantage, particularly for sophisticated users, that the preservation and use of history interfaces less well with the programming system per se.

In choosing between these two approaches, one must also decide whether one prefers to regard the session history as a structure (in which case the INTERLISP approach is better) or as a text file (in which case the new approach would seem more appropriate).

Why choose between structure history and text history, why not have both? This suggestion is likely to be resisted by programming-language purists, who wish to have a minimal number of orthogonal features in a programming system, and to be welcomed by systems engineers who wish to give the user everything that can possibly be of use. The only way to resolve that issue is to define what operations are to be performed by the structure history and the text history, respectively.

Cancellation of Features

One of the most impressive features in the INTERLISP system is the DWIM (Do-What-I-Mean) facility, which is invoked when the basic system detects an error and which attempts to guess what the user might have intended. When this facility is presented to new users, it is not uncommon for them to use it for a trivial typing error that could easily be corrected using the character-delete key. However, the user relies on DWIM for the correction, which at periods of peak computer load may take considerable time. Moreover, a user who does not know how the DWIM facility works might be reluctant to hit an interrupt key for fear of making things even worse. For this reason, the actual DWIM program has been set up so that supposedly it can be safely interrupted at any point. The information on how to interrupt it is an essential part of the instructions to the user.

Two general morals may be drawn from this example:

1) When you provide users with luxurious features, always make sure that they are able to, and know how to, cancel them at times when they cannot afford their use;
2) This affects decisions on the budget for computer equipment. As computer systems become more and more heavily loaded, more of the advanced features in interactive programming systems are cancelled. If the intended purpose of the research was to develop and make experimental use of those facilities, then the real productivity of the research decreases rapidly.

Program Changes

Many of the advanced features in residential programming systems become possible because definitions of procedures can be changed dynamically under program control, and many different parts of the system will want to perform changes, temporarily or permanently, in the users' procedures. More often than not, the old definition has to be stored so that it can be reinstated if the user so desires.

In this situation, the relationship between the different changes becomes precarious. If the user first performs operation A, then performs operation B, and then undoes A, should then the original definition (before A was performed) be reinstated, or should B be performed on it also? There is no simple answer—what would be better depends on the choice of A and B. But in the existing INTERLISP system, where each program-changing operation has its own recovery mechanism, there have been amusing examples of unintended and counter-intuitive results as the different recovery operations interacted with each other. Perhaps the conclusion is that a uniform system for maintenance of old versions and updates of procedures that can be used by all definition-changing utilities should be introduced early in the design of the system.

These examples of open issues in the design of session-support systems show that, although impressive systems already exist, there is room for additional work—not merely to extend or rewrite existing software, but also to reconsider and analyze questions that involve principles.

7. POTENTIAL APPLICATIONS

As described in this paper, programming in LISP is characterized by the peculiar architecture of the programming system and the programming techniques made possible by that architecture. At present, LISP is mostly used for two types of applications:

1) Experimental programming in artificial intelligence research, where programs are developed for the purpose of better understanding certain complex programming tasks;
2) Implementation of formula manipulation systems.

I believe that the same technology would also be useful for a number of other applications, particularly:

- For pilot implementation of data processing applications. A pilot system may be used to give the end user a hands-on impression of possible facilities in a proposed new data processing system. The pilot implementation does not need to run efficiently or handle large volumes of data, but it requires a programming language in which it can be implemented quickly and modified easily to accomodate changes proposed by the end user when he tries the system. The interactive character and the database facilities in LISP systems meet these requirements.
- For personal databases, i.e., for databases which are used in an office-system or word-processing environment, and which are constructed and used by one or a few individuals for their personal use.
- In computer science research on programming languages and databases. LISP used as a very-high-level implementation language may make it possible to experiment with proposed new concepts in languages and database systems, instead of just thinking about them.

In all three cases, the suggestion is that one should use a programming system which incorporates the significant properties of LISP systems, as discussed in this paper. It is of course immaterial whether one uses a dialect of LISP itself or a newly invented language.

CONCLUSION

This paper has given an overview of existing programming methodology in the LISP user's environment, emphasizing methods for interactive program development. The major conclusions are:

1) The "residential" design of programming systems, whereby all facilities for the user are integrated into one system with which the user communicates during the entire interactive session, offers great possibilities for user convenience. At the same time, a number of basic user demands may conflict with the attempt to build such a residential system.

2) The design of a residential interactive programming system is interrelated with the design of the *surrounding runtime environment* in at least two important respects: the role of the file system (especially the requirement for a very large number of very small files, and a file directory which can serve as or communicate with a database for user-written programs), and the proper place of text editing and other terminal support in the overall system architecture (inside the programming system, or as a separate module).

3) A residential interactive programming system will need to contain a large number of modules that support various facilities. These modules interact in many ways, which causes the design of the programming system to be a *very hard structuring problem*. Two modules have been identified which tend to interact with almost everything else, namely the editor and the surface-language analyzer (the latter being optional).

4) The observed behavior of LISP users indicates that top-down programming can be done not only using stepwise refinement, but often in a better way through what has been called here *structured growth*, which is a relatively disciplined way of *changing* one's programs.

5) Several programming methods in a LISP environment can be summarized as involving the use of *superimposed languages*, be they ALGOL-like surface languages, very-high-level languages, or embedded languages for very specialized purposes. Of these, embedded languages are believed to be the most viable.

6) A technique which is useful in several ways is what we have called here *data-driven procedure calls*, which resembles indirect jumps in machine languages; they are possible in LISP but are incompletely supported by higher-level facilities in current LISP systems. Nevertheless, the technique is frequently used in practice and has become accepted habit because of its power.

The experience and practices of the LISP programming community may contribute new and unorthodox input to the general discussion about programming systems and programming methodology.

ACKNOWLEDGMENTS

This paper is based on discussions with a large number of devoted LISP programmers; their contribution is appreciated. Successive generations of the manuscript have been enhanced by valuable comments and suggestions from Jim Goodwin, Warren Teitelman, Peter Denning, and an anonymous reviewer, for which I am very grateful.

REFERENCES

[1] BRINCH HANSEN, P. "The Solo operating system," *Softw. Pract. Exper.* 6, 2 (1976), 141–206.

[2] WIRTH, N. "The programming language PASCAL," *Acta Inf.* 1, (1971), 25–68.

[3] *Introduction to MUMPS-11 Language*, DEC-11-MMLTA-C-D, Digital Equipment Corp., Maynard, Mass., 1976.

[4] TEITELMAN, W. *INTERLISP reference manual*, Xerox-Palo Alto Research Center, Palo Alto, Calif., 1974.

[5] WINOGRAD, T. "Breaking the complexity barrier," unpublished memo, Stanford Artificial Intelligence Laboratory, Stanford, Calif., 1974.

[6] McCarthy, J. et al. *LISP 1.5 programmer's manual*, MIT Press, Cambridge, Mass., 1962.

[7] NORDSTRÖM, M. et al. *LISP F1–a Fortran implementation of LISP 1.5*, Computer Sciences Dept., Uppsala Univ., Sweden, 1970.

[8] ASH, W. et al. *Intelligent on-line assistant and tutor system*, BBN Report No. 3607, Bolt, Beranek, and Newman, Inc., Cambridge, Mass., 1977.

[9] WEISSMAN, C. *LISP 1.5 primer*, Dickenson Publishing Co., Belmont, Calif., 1967.

[10] HARALDSON, A. *LISP–details*, Computer Sciences Dept., Uppsala Univ., Sweden, 1975.

[11] ALLEN, J. *The anatomy of LISP*, McGraw-Hill, in press.

[12] WINSTON, P. *Artificial intelligence*, Addison-Wesley Publ. Co., Reading, Mass., 1977.

[13] TEITELMAN, W. "A display oriented programmer's assistant," in *Proc. Fifth Int. Jt. Conf. Artificial Intelligence*, Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, 1977, pp. 905–915.

[14] HENNEMAN, W. "An auxiliary language for more natural expression," in *The programming language LISP, its operation and applications*, E.C. Berkely and D.G. Bobrow (Eds.), MIT Press, Cambridge, Mass., 1964.

[15] SMITH, CANFIELD D. *MLISP*, Stanford Artificial Intelligence Laboratory, Stanford, Calif., 1970.

[16] HEARN, A. *REDUCE user's manual*, Memo 50, Stanford Artificial Intelligence Laboratory, Stanford, Calif., 1968.

[17] WEGBREIT, B. et al. *ECL programmer's manual*, Harvard Univ., Cambridge, Mass., 1972.

[18] THOLERUS, T. *REC–a recursive programming language with visible control stack*, Computer Sciences Dept., Uppsala Univ., Sweden, 1975.

[19] GRISS, M. "The definition and use of datastructures in REDUCE," in *SYMSAC 76 Proc. 1976 ACM Symp. Symbolic and Algebraic Computation*, R.D. Jenks (Ed.), ACM, N.Y., pp. 53-59.

[20] SANDEWALL, E. "Ideas about management of LISP data bases," in *Proc. Fourth Int. Jt Conf. Artificial Intelligence*, Artificial Intelligence Laboratory, Cambridge, Mass., 1975, pp. 585-592.

[21] SANDEWALL, E. "Some observations on conceptual programming," in *Machine intelligence 8*, E.W. Elcock and D. Michie (Eds.), John Wiley & Sons, N.Y., 1977, pp 223-265.

[22] HEWITT, C. *Planner: A language for manipulating models and proving theorems in a robot*, Artificial Intelligence Memo 168, MIT, Cambridge, Mass., 1970.

[23] RULIFSON, J. F. et al. *QA4, a procedural calculus for intuitive reasoning*, Stanford Research Inst., Menlo Park, Calif., 1972.

[24] MCDERMOTT, V. et al. *The Conniver reference manual*, Artificial Intelligence Memo 259, MIT, Cambridge, Mass., 1972.

[25] SACERDOTI, E. et al. *QLISP: A language for the interactive development of complex systems*, Stanford Research Inst., Menlo Park, Calif., 1976.

[26] DAVIES, D. et al. *Popler 1.5 reference manual*, Univ. of Edinburgh, Edinburgh, Scotland, 1973.

[27] SANDEWALL, E. *LISP A:* "A LISP-like system for incremental computing," in *Proc. AFIPS 1968 Spring Jt. Computer Conf.*, Vol. 32, Thompson Book Co., Washington, D.C., pp. 375-384.

[28] BOBROW D.; AND RAPHAEL, B. *New programming languages for A. I. research*, Stanford Research Inst., Menlo Park, Calif., 1973.

[29] WINOGRAD, T. "Procedures as a representation for data in a computer program for understanding natural language," PhD Thesis, MIT, Cambridge, Mass., 1971.

[30] BERRY, D. "Structured documentation," *SIGPLAN Newsletter* (Nov. 1975).

[31] DAHL, O.-J.; AND HOARE, C. A. R. "Hierarchical program structures," in *Structured programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press Inc., N.Y., 1972.

[32] KNUTH, D. *The art of computer programming, Vol. 1: fundamental algorithms*, Addison-Wesley, Publ. Co., Reading, Mass., 1968, and following volumes.

[33] NORDSTRÖM, M.; AND THOLERUS, T. *A parsing technique applied to the programming language REDUCE*, Computer Sciences Dept., Uppsala Univ., Sweden, 1974.

[34] PRATT, V. "Top down operator precedence," in *ACM Symp. Principles of Programming Languages*, ACM, N.Y., 1973.

[35] KNUTH, D. "Semantics of context-free languages," *Math. Syst. Theory J.* (1968), 127-145.

[36] NORDSTRÖM, M. "A method for defining formal semantics of programming languages applied to Simula," Dissertation, Uppsala Univ., Sweden, 1976.

[37] HOARE, C. A. R. *Recursive data structures*, STAN-CS-73-400, Computer Science Dept., Stanford Univ., Stanford Calif., 1973.

[38] WOODS, W. "Transition network grammars for natural language analysis," *Commun. ACM* **13**, 10 (1970), 591-606.

[39] GIBBONS, G. Letter in *ACM Forum, Commun. ACM* **19**, 2 (1976), 105-106.

[40] SHORTLIFFE, D. A. "An artificial intelligence program to advise physicians regarding antimicrobial therapy," *Comput. Biomed. Res.* **6** (1973), 544-560.

[41] AIELLO, L. et al. "Recursive data types in LISP: A case study in type driven programming," in *Proc. 2nd Int. Symp. Programming*, Institut de Programmation, Paris, 1976.

[42] WINOGRAD, T. *Five lectures on artificial intelligence*. STAN-CS-459, Computer Science Dept., Stanford Univ., Stanford Calif, 1974.

[43] Personal communication.

[44] TEITELMAN, W. "Toward a programming laboratory," in *Proc. First Int. Jt. Conf. Artificial Intelligence*, 1969.

RECEIVED NOVEMBER 3, 1976; FINAL REVISION ACCEPTED OCTOBER 25, 1977.