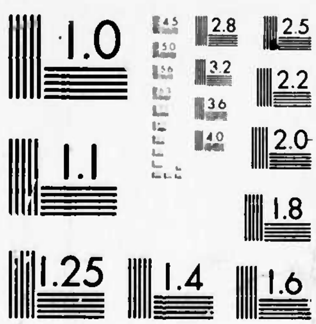
AD-A047 179 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES FOR COMPUTER AIDE--ETC(U)
SEP 77 C RIEGER, H SAMET, J ROSENBERG N00014-76-C-0477

UNCLASSIFIED TR-595 NL

1 OF 1
AD
A047179

END
DATE
FILMED
1 -78
DDC

047 1



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(12)

TR-595 September 1977

Artificial Intelligence Programming Languages for Computer Aided Manufacturing

Chuck Rieger, Hanan Samet, Jonathan Rosenberg
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT: Eight Artificial Intelligence programming languages (SAIL, LISP, MICROPLANNER, CONNIVER, MLISP, POP-2, AL and QLISP) are presented and surveyed, with examples of their use in an automated shop environment. Control structures are compared, and distinctive features of each language are highlighted. A simple programming task is used to illustrate programs in SAIL, LISP, MICROPLANNER and CONNIVER. The report assumes reader knowledge of programming concepts, but not necessarily of the languages surveyed.

This report was funded by the National Bureau of Standards, and by the Office of Naval Research.

CONTENTS

1. Introduction

This report describes some recently developed Artificial Intelligence programming languages in the context of a computer-aided manufacturing environment. The languages surveyed are SAIL, LISP, MICROPLANNER, CONNIVER, MLISP, POP-2, AL, and GLISP. These languages are distinct from languages previously used in computer-aided manufacturing environments [Leslie72] in that they provide capabilities for the development of high-level symbolic planning and supervisory control in addition to the simple numerical control of machine tools. The paper includes (1) surveys and comparisons of the distinctive features of these languages as they might be used in a computer-automated manufacturing environment, (2) a sample automated manufacturing task, and how it might be expressed as a program in each language, (3) discussions of the standardization status of each language, and (4) conclusions with emphasis on the types of features which are most desirable and applicable to the automated-shop environment.

2. SAIL

2.1. Introduction

SAIL has its origins in a merger of LEAP [Feldman69], an associative language, and a version of ALGOL 60 [Naur60]. Therefore, unlike most of the other artificial intelligence languages, it is not LISP-based. Instead, it is a general purpose compiled language with an extensive run-time library of functions. As befits its ALGOL origins, SAIL has block structure and explicitly typed, statically scoped variables. The data types available include INTEGER, REAL, STRINGS of arbitrary length, structure, pointer, LIST, SET, ITEM, and aggregates of the previous (i.e., ARRAYs).

Some of the more important features of SAIL are discussed separately below. These include the associative data base facility, the capability for usage of SAIL as a host language in a CODASYL [CODASYL71] data base management system, the control structures, and the system building facilities. Finally, a summary is presented of current standardization efforts.

2.2. Associative Data Base

SAIL contains an associative data base facility known as LEAP which is used for symbolic computations This enables the storage and retrieval of information based on partial specification of the data. Associative data is stored in the form of associations which are ordered three-tuples of ITEMs, denoted as TRIPLES. Examples of TRIPLES are:

```
FASTEN XOR NAIL EQV HAMMER;
FASTEN XOR SCREW EQV SCREWDRIVER;
FASTEN XOR BOLT EQV PLIER;
```

Associations may be conceptualized as representing a relation of the form

```
      Attribute XOR Object EQV Value
   or Attribute (Object) = Value
```

Most programming languages (e.g., LISP) provide the following associative-like mechanism:

```
Given:  Attribute,Object
Find:   Value
```

However, SAIL enables the programmer to specify any of the components of the association, and then have the LEAP interpreter search the associative store for all triples which have the same items in the specified positions. For example, the following may be used to retrieve all items that can fasten a nail:

```
FASTEN XOR NAIL
```

An ITEM is a constant and is similar to a LISP atom. Items have names and may also be typed so that data can be associated with them. An item may be declared, or created during execution from a storage pool of items by use of the function NEW. For example:

```
REAL ITEM VISE;
```

declares VISE to be an item which may have a datum of type real associated with it. The datum associated with an item is obtained by use of the function DATUM. Thus, DATUM(VISE) might be interpreted as the capacity of the vise.

In order to deal with items, the user has the capability of storing them in variables (ITEMVARs), SETs, LISTs, and associations. The distinction between SETs and LISTs is that an explicit order is associated with the latter, whereas there is no explicit order associated with the former. In addition, an item may occur more than once in a List.

Associations are ordered three-tuples of items and may themselves be considered as items and therefore participate in other associations. Triples are added to the associative store by use of a MAKE statement and erased from the associative store by use of an ERASE statement. For example, the following code could be used to detach assembly 1 from assembly 2 and attach it to assembly 3:

```
ERASE ATTACHED XOR ASSEMBLY1 EQV ASSEMBLY2;
MAKE  ATTACHED XOR ASSEMBLY1 EQV ASSEMBLY3;
```

The motivation for using an associative store is a flexible search and retrieval mechanism. Binding Booleans and Foreach statements are two methods of accomplishing these goals.

The binding Boolean expression searches the associative store for a specified triple and returns TRUE if the triple is found and FALSE otherwise. The aim of the search is to find an association which meets the constraints imposed by the specified triple. If some of the components of the triple are unknown (such components are preceded by the special item BIND), then a successful search will result in the binding of the designated component. For example:

```
IF FASTEN XOR BIND OBJECT EQV PLIER THEN PUT OBJECT IN PLIER!SET;
```

In this case the store is searched for an object that can be fastened by a PLIER and if such an object is found, it is placed in the set PLIER!SET. Note the use of the item variable OBJECT in the association. A successful search will result in this variable being bound.

The FOREACH statement is the heart of LEAP. It is similar to the FOR statement of ALGOL in that the body of the statement is executed once for each binding of the control variable. For example:

```
FOREACH X | PART XOR B747 EQV X AND DATUM(X) < 3
        DO PUT X IN B747!ORDER!SET;
```

In this case, assuming that the datum associated with each part denotes quantity at hand, the associative store is searched for all parts of a B747 of which there are less than three on hand. These parts are placed in the set B747!ORDER!SET.

2.3. <u>Data Management Facility</u>

Unlike other artificial intelligence languages, SAIL has the capability of being used with an existing data base management system (DBMS-10 [DEC]) to handle large data bases stored on external storage. An interface exists [Samet76] which allows SAIL to be used as the data manipulation language in a CODASYL based data base management system. SAIL is relatively unique in this respect in that COBOL [COBOL74] has almost been exclusively used as the data manipulation language (DML) of such systems. This situation is not surprising since examination of the data description facility of the CODASYL report reveals a very strong similarity to the data division of COBOL. Nevertheless, there have been some attempts to use FORTRAN ([Stacey74], [RAPIDATA]).

Ideally, a data manipulation language should include the following features. First, a full procedure capability which allows parameter passing, dynamic storage allocation, and recursion. Second, processing of Boolean requests should not be difficult. In a COBOL-based system this task is rather cumbersome as pointed out by [Parsons74]. In order to avoid currency problems raised by partial satisfaction of Boolean requests (the backtracking problem [Taylor76]), the user must build collections of pointers to related records. Third, there should be a capability for building an in-core data base so that operations such as set UNION and set INTERSECTION can be performed without the overhead of accessing extended storage more than once for any record.

SAIL has a mechanism, LEAP, for building associative data bases. Currently, this only works for internal memory due to implementation decisions. SAIL also has a record structure capability which enables the user to build an in-core data base. In a COBOL-based data base management system, whenever the user obtains an instance of a record type from the data base (i.e., he locates it via a FIND and fetches it via a GET), he has no convenient way of keeping it in temporary memory while obtaining another instance of this record type. Of course, he can allocate temporary storage for the various fields; however, this becomes rather unwieldy, especially when he wishes to keep track of more than two instances of a record type. Alternatively, instances of certain record types can be refetched from the data base. In fact, this is the strategy that is generally followed. However, the cost is prohibitive.

Briefly, the SAIL interface provides a SAIL record structure declaration for each record type that has been defined in the data base management system. Primitives exist for the creation and modification of such records. The dynamic storage allocation capability of SAIL enables the creation of several instances of each record type each of which is identified by an entity known as a record pointer.

As an example of the use of SAIL as a host language in a data base management system, consider the following program fragment. The task is to traverse a set named SUPPLIER owned by a WAREHOUSE record and extract an integer data item known as PARTNUM from each PART record which is a member of the set. The exact instance of the set occurrence is identified by the owner record, WAREHOUSE, having the value ELECTRICAL for the data item INDUSTRY. Since SAIL has a data structuring facility (known as a RECORD!CLASS and similar to a PL/1 [Beech70] structure) we define a data structure known as LISTX and a function to add items to the front of a LISTX structure. The data structure LISTX has two fields - ELEMENT which is of type INTEGER and NEXT which is of type RECORD!POINTER (and points to another instance of the LISTX data structure). The function ADDTOLIST has two arguments - a pointer to the head of an instance of LISTX and the integer to be added to this instance.

```
RECORD!CLASS LISTX(INTEGER ELEMENT;
                   RECORD!POINTER (LISTX) NEXT);

PROCEDURE ADDTOLIST(REFERENCE RECORD!POINTER(LISTX) HEAD;
                    INTEGER VAL);
BEGIN
    RECORD!POINTER (LISTX) TEMP;
    TEMP := NEW!ELEMENT(LISTX);
    LISTX:ELEMENT[TEMP] := VAL;
    LISTX:NEXT[TEMP] := HEAD;
    HEAD := TEMP;
    END;
```

The COBOL/DML and SAIL encodings are given below. The critical difference is the step "Add PARTNUM in PART to result list." It is not immediately obvious how the concept of a list would be implemented in COBOL.

```
COBOL Program:
            MOVE 'ELECTRICAL' TO INDUSTRY IN WAREHOUSE.
            FIND WAREHOUSE RECORD.
            IF SUPPLIER SET EMPTY GO TO NONE!SUPPLIED.
NEXT:       FIND NEXT PART RECORD OF SUPPLIER SET.
            IF ERROR-STATUS = 0307 GO TO ALL!FOUND.
            GET PART.
            Add PARTNUM in PART to result list.
            GO TO NEXT.
ALL!FOUND:

SAIL Program:
            INDUSTRY := "ELECTRICAL";
            FIND!CALC(WAREHOUSE);
            IF EMPTY!SET(SUPPLIER) GO TO NONE!SUPPLIED;
            WHILE TRUE DO BEGIN
                FIND!NEXT(PART,SUPPLIER);
                IF ERROR!STATUS = 0307 THEN DONE;
                GET(PART);
                ADDTOLIST(HEAD,PARTNUM);
                END;
```

2.4. Control Structures

In addition to the ususal control structures associated with ALGOL-like languages (e.g., FOR loops, WHILE loops, case statements, recursive procedures, etc.), SAIL has capabilities to enable parallel processing, backtracking, and coroutines. In SAIL, a process is a procedure that may be run independently of the main procedure. Thus several processes may be run concurrently. Note that the main procedure is also a process.

A process is created with a SPROUT statement as follows:

```
SPROUT(<item>,<procedure call>,<options>)
```

where <item> names the process for future reference, <procedure call> indicates what the process is to do, and <options> is used to specify attributes of the SPROUTed and current process. Unless otherwise stipulated (in <options>), a SPROUTed process begins to run as soon as it is SPROUTed and in parallel with the SPROUTing process.

Similarly, there exist primitives which result in the suspension of a process, the resumption of a process, and in the blocking of a process until a number of other processes have terminated. These tasks are accomplished by the SUSPEND, RESUME, and JOIN primitives respectively.

SUSPEND and RESUME have as their arguments single items while JOIN has a set of items as its argument. These items are the names that have been set up for the process by an appropriate SPROUT command.

For example, a procedure to tighten a bolt may be defined as follows:

```
ITEM P1,P2;
   .
   .
   .
SPROUT(P1,GRASP(HAND1,SCREWDRIVER));
```

```
SPROUT(P2,GRASP(HAND2,BOLT));
        .
        .
        .
JOIN({P1,P2});
TURN(HAND1,CLOCKWISE);
        .
        .
        .
```

Since SAIL runs on a single processor computer system, true multiprocessing is not possible. Instead, the SAIL runtime system contains a scheduler which decides which process is to run and for how long. The programmer makes use of the <options> field of the SPROUT statement to specify information which the scheduler uses to determine the next process to be run. Such information includes time quantum sizes, priority, whether or not to immediately run the SPROUTed process, etc.

A process may result in the binding of ITEMVARs by use of a MATCHING PROCEDURE which is basically a Boolean procedure. When one of the parameters is an unbound FOREACH itemvar, then upon success the parameter will be bound . The matching procedure is actually SPROUTed as a coroutine process and SUCCEED and FAIL are variants of RESUME which return values of TRUE or FALSE respectively. In addition, FAIL causes the process to terminate whereas when the matching procedure is called by the surrounding FOREACH via backup, then the procedure is resumed where it left off on the last SUCCEED.

For example, consider a box containing a number of different fasteners (nails, regular screws, bolts, nuts, tacks, etc.). The goal is to obtain Phillips screws. This can be achieved by the following MATCHING PROCEDURE which returns a different Phillips screw each time it is invoked.

```
MATCHING PROCEDURE GET!FASTENER (?ITEMVAR FASTENER,F!TYPE);
BEGIN
     FOREACH FASTENER | FASTENER IN BOX AND
                                  TYPE XOR FASTENER EQV F!TYPE
          DO SUCCEED;
     FAIL;
     END;
```

Note that FASTENER is a FOREACH ITEMVAR which upon success will be bound.

Backtracking is supported by variables of type CONTEXT. However, the programmer must specify the points to which backup is to occur (for example, recall SUCCEED). State saving and restoring is achieved by use of CONTEXT variables which act as pointers to storage areas of undefined capacity in which are stored the entities to be saved and restored. Actual state saving and restoring is accomplished by use of the primitives REMEMBER and RESTORE.

Processes may communicate with each other by use of the SAIL event mechanism. This is a message processing system which enables the programmer to classify the messages and to wait for certain events to occur. Events occur via the CAUSE construct which has as its arguments the event type, the actual notice, and instructions with respect to the disposition of the event. Similarly, there is a construct known as INTERROGATE which specifies a set of event types and instructions with respect to the disposition of the event notice associated with the designated event types. A variant of this facility has been used extensively in the implementation of the Stanford Hand Eye Project [Feldman71].

2.5. System Building Capabilities

SAIL includes many features which are designed to aid in system building. Assembly language statements may be interspersed with regular SAIL statements by use of the START!CODE and QUICK!CODE constructs. A number of different files which are to be used with the program can be specified via use of REQUIRE statements.

The statements:

```
REQUIRE "TOOLS" LOAD!MODULE;
REQUIRE "CAMLIB[1,3]" LIBRARY;
```

will cause SAIL to inform the loader that the file TOOLS.REL must be loaded. In addition, the file CAMLIB on disk area [1,3] serves as a library and is searched for needed routines.

The statement:

```
REQUIRE "HEADER.SAI" SOURCE!FILE;
```

will cause the compiler to save the state of the current input file, and scan HEADER.SAI for program text. When HEADER.SAI is exhausted, scanning of the original file resumes at a point immediately following the REQUIRE statement. This feature is particularly useful when dealing with libraries since in this case the REQUIREd file can contain EXTERNAL declarations thereby freeing the application programmer from such work and possible errors.

A rather extensive conditional compilation capability is associated with SAIL. This enables the development of large programs which can be parameterized to suit a particular application without compiling unnecessary code and thereby wasting memory for program segments which are never used. This capability is used to enahance a macro facility to include compile-time type determination; for loops, while statements, and case statements at compile-time; generation of unique symbols, and recursive macros. For example:

```
DEFINE GRASP(SIZE) = [IFCR SIZE > 1 THENC VISE
                      ELSEC PLIERS
                      ENDC];
```

results in the definition of a macro named GRASP having one formal parameter, SIZE. The result is the name of a tool that is appropriate for the size of the item that is to be grasped - i.e., a vise in case size is greater than 1 (assuming size is measured in centimeters, etc.) and pliers otherwise. For example:

```
TOOL1 := GRASP(10.0);
TOOL2 := GRASP(0.5);
```

will result in the following statements:

```
TOOL1 := VISE;
TOOL2 := PLIERS;
```

Note that the choice is made at compile-time and thus the programmer need not be concerned with the available grasping mechanisms Thus the program compilation step can be used to aid in the writing of the program. The example illustrates the importance of such a feature when certain tasks can be achieved by similar, yet not identical, means.

SAIL also provides an excellent interface with the operating system. This enables its use for real-time applications such as control of external devices. In fact, interrupts can be handled and the user has at his disposal all of the I/O capabilities that an assembly language programmer has. This enables the development of programs ranging from scanners to mechanical arm controllers. In addition to compatibility with assembly language debuggers, SAIL has a high-level breakpoint package known as BAIL [Reiser75].

2.6. Standardization

Currently, SAIL has only been implemented on the PDP-10. It runs under both the TENEX [BBNEXEC] and TOPS-10 [TOPS10] operating systems. There is an effort underway at SUMEX to develop a language similar to SAIL known as MAINSAIL [Wilcox76]. The goal of that project is to capture the features that make SAIL an attractive language (in particular the ease of interaction with the operating system) and to develop a language that is capable of being run on a large number of machines. The orientation of the project is towards mini-computers. The language is considerably different than SAIL and existing SAIL programs will have to be modified in order to be capable of compiling. An extensive run time library is being provided as is a record structuring facility. It is still uncertain whether the associative data base capability of SAIL (i.e., LEAP) will be incorporated in MAINSAIL.

3. The LISP Family of Languages

3.1. LISP

LISP ([McCarthy60], [Levin65], [Weissman67], [Siklossy76]), a list processing language developed by John McCarthy at MIT in the late 50's, is an implementation of parts of Alonzo Church's work [Church41] in the lambda calculus. McCarthy's intention was to recast the elegance of recursive function theory as a theory of computation. Thus, the first implementations of LISP relied exclusively upon recursion as the computational paradigm (i.e., no iteration), which, although elegant, resulted in a first version of LISP which was not competitive with FORTRAN as a practical programming tool. However, LISP's character has changed considerably, so that today LISP is an extremely powerful and general purpose programming language which nevertheless retains its original elegance.

The most interesting features of LISP are:

(1) The language is practically devoid of syntax; all constructions in LISP fall into two categories: atoms and compositions of atoms.

(2) Program and data are interchangeable, since they are represented in the same format. Therefore, in LISP it is possible for one function to construct another function as data, then execute it by indicating to the LISP system to regard it as code; alternatively, an existing function's code may be examined, modified or augmented by another function at run-time. In fact, a function is capable of self-modification if appropriate care is exercized.

(3) Memory allocation and management are automatic and transparent to the user, except where the user explicitly desires to influence them. With the exception of arrays, there are no space declarations to be made, freeing the programmer from the details of space allocation, and generally allowing for the unlimited growth of any given data structure. (For the most part, LISP data structures have no size or complexity constraints.) Used memory which is no longer involved in the computation is recycled automatically by a garbage collector either on demand from the user at specified points or automatically.

(4) LISP is an interpreted language. The system proper is a function of one argument, (EVAL X), such that calling EVAL with any LISP data structure as its argument causes that argument to be regarded as code and executed. However, most LISP systems include a compiler which will produce stand-alone machine code for interpreted functions. Typically, compilation provides an order of magnitude speedup which makes LISP competitive with other compiled languages, or even with well-coded assembly language. Since interpreted and compiled code may be intermixed, it is possible to retain the flexibility and power of the interpreter, while obtaining the speed required for production applications.

(5) LISP remains recursive, while also accommodating iterative algorithms via a so-called PROG feature. Both recursion and iterative programming are illustrated in subsequent sections.

(6) Because of the technique LISP uses in storing local and global variables, some very powerful context-switching can be carried out, providing a fast way to enter and exit hypothetical planning environments and to cause the

behavior of a program to vary as a function of its environmental context.

3.1.1. LISP Data Structure

LISP's data structure, called the S-expression, is simple, yet extraordinarily flexible, providing a substrate upon which a programmer may design his own complex data structures. An S-expression is either an "atom" or a "CONS node". An atom can be regarded as either a variable, a constant (a passive symbol), or both. There are no declarations in LISP; new atoms are simply admitted to the system as they are scanned at the input level, and atoms with the same name are guaranteed by the system to be unique (i.e., they have the same internal pointer, or address).

The other type of S-expression, the CONS node, provides a means of structuring atoms and other CONS nodes into hierarchical data structures. A CONS node is ordinarily implemented as a single computer word (say, 36 bits long) which contains a left pointer, called its CAR, and a right pointer, called its CDR. CONS nodes are created dynamically via the function (CONS X Y), where X and Y are any other S-expressions, or passively (as data constants) via the construction (X.Y). CONS nodes can be composed to form arbitrarily complex hierarchies, the bottommost elements of which are usually atoms (i.e., pointers to atomic S-expressions).

To illustrate, suppose we wish to represent a particular tool, say a screwdriver, in a LISP data structure. We first decide upon a name for it, say, SCREWDRIVER-1, and what characteristics of it we wish to encode. Let us suppose the characteristics are: type is Phillips, color is yellow, shaft length is 10 centimeters, and head size is 0.3 centimeter. There are many ways to encode this in LISP; the external representation of the one we adopt here is:

```
((NAME SCREWDRIVER-1)
 (TOOL-TYPE SCREWDRIVER)
 (STYLE PHILLIPS)
 (SHAFT-LENGTH 10 CM)
 (COLOR-CODING YELLOW)
 (HEAD-SIZE 0.3 CM))
```

Here, all symbols such as NAME, YELLOW, etc. are LISP atoms. (So too are the numbers; however numbers are not entirely equivalent with symbolic atoms.) The particular hierarchy we have adopted is a list of lists, where each sub-list consists of an initial atom describing that sub-list's role in the structure, and a list of the information associated with that role in the description.

This structure would be graphically represented as follows:

```
+---+        +---+        +---+        +---+        +---+        +---+
|*|*|------->|*|*|------->|*|*|------->|*|*|------->|*|*|------->|*|/|
+---+        +---+        +---+        +---+        +---+        +---+
  |            |            |            |            |            |
+---+  +---+ +---+  +---+ +---+  +---+   |          +---+  +---+   |
|*|*|->|*|/| |*|*|->|*|/| |*|*|->|*|/|   |          |*|*|->|*|/|   |
+---+  +---+ +---+  +---+ +---+  +---+   |          +---+  +---+   |
  |      |     |      |     |      |     |            |      |     |
NAME     |   TOOL-TYPE|   STYLE PHILLIPS |        COLOR-CODING|     |
  SCREWDRIVER-1  SCREWDRIVER   |----------                 YELLOW  |
                               |                     |-------------
                             +---+  +---+  +---+   +---+  +---+  +---+
                             |*|*|->|*|*|->|*|/|   |*|*|->|*|*|->|*|/|
                             +---+  +---+  +---+   +---+  +---+  +---+
                               |      |      |       |      |      |
                      SHAFT-LENGTH   10     CM       |     0.3    CM
                                                 HEAD-SIZE
```

and could be constructed passively (as a fully constant structure) via a quoted S-expression:

```
'((NAME SCREWDRIVER-1) (TOOL-TYPE SCREWDRIVER) ...)
```

or dynamically via CONS:

```
(CONS (CONS 'NAME (CONS 'SCREWDRIVER-1 NIL))
      (CONS 'TOOL-TYPE (CONS 'SCREWDRIVER NIL))
      ...
      (CONS 'HEAD-SIZE (CONS 0.3 (CONS 'CM NIL)))
)
```

Since it would be a rather harrowing experience to construct very large S-expressions dynamically in this fashion, LISP provides a spectrum of higher-level functions for constructing, modifying and accessing S-expressions. Some highlights of these will be covered briefly in a subsequent section. For our example, a more concise expression of code which would build this structure dynamically would be:

```
(LIST (LIST 'NAME 'SCREWDRIVER-1)
      (LIST 'TOOL-TYPE 'SCREWDRIVER)
      ...
      (LIST 'HEAD-SIZE 0.3 'CM)
)
```

Presumably, having defined this tool, we would want to record it as one available tool in a large supply of tools. Again, there would be numerous methods of doing this. One way would simply be to maintain a global list of all known tools in the system, and to add this entire description as a new tool on this list:

```
(SETQ NEW-TOOL '((NAME SCREWDRIVER-1) (TOOL-TYPE SCREWDRIVER) ...))
(SETQ MASTER-TOOL-LIST (CONS NEW-TOOL MASTER-TOOL-LIST))
```

(SETQ is one of LISP'S assignment statements.) Alternatively, we might wish to put only the name of the screwdriver on the master tool list, and associate all the remaining information with property DESCRIPTION on SCREWDRIVER-1's _property list_:

```
(PUT 'SCREWDRIVER-1 'DESCRIPTION
     '((TOOL-TYPE SCREWDRIVER) ... (HEAD-SIZE 0.3 CM)))
(SETQ MASTER-TOOL-LIST (CONS 'SCREWDRIVER-1 MASTER-TOOL-LIST))
```

3.1.2. Property Lists

Any LISP atom may have a property list (built up from CONS nodes). Conceptually, the property list allows the attachment of an arbitrary number of attribute-value pairs to the atom, thereby serving to describe the characteristics of the real-world entity represented by the atom. This is a powerful feature for any programming language, since it allows "micro-descriptions" of atoms which ordinarily will not be seen by the processes that manipulate the hierarchical structures in which the atom participates. These microdescriptions can be maintained and accessed by the functions PUT, GET and REMPROP in case more detail about an atom is desired.

Properties are attached to an atom via the function (PUT <atom> <attribute> <value>), looked up via (GET <atom> <attribute>), and removed via (REMPROP <atom> <attribute>). We have seen one way to associate the screwdriver information with the atom SCREWDRIVER-1 using property lists. Another, more convenient way would be to split apart all the various attributes of this atom, making each a different entry on the property list:

```
(PUT 'SCREWDRIVER-1 'TOOL-TYPE 'SCREWDRIVER)
(PUT 'SCREWDRIVER-1 'STYLE 'PHILLIPS)
...
(PUT 'SCREWDRIVER-1 'HEAD-SIZE '(0.3 CM))
```

To determine SCREWDRIVER-1's head size, we would then write: (GET 'SCREWDRIVER-1 'HEAD-SIZE). If such an attribute of SCREWDRIVER-1 exists, it will be located and returned.

3.1.3. Representative LISP Data Structure Manipulating Functions

We include here a definition and brief example of several of the more standard, high-level LISP functions that pertain to data structure creation, modification and searching.

3.1.3.1. (MEMBER X Y)

If S-expression X is a member of S-expression Y (assumed to be a list), return "TRUE", otherwise, return "FALSE".

EXAMPLE: (MEMBER 'SCREWDRIVER-1 MASTER-TOOL-LIST) returns a pointer to the atom T ("true") if SCREWDRIVER-1 is on the MASTER-TOOL-LIST, and a pointer to the atom NIL ("false") otherwise.

3.1.3.2. (ASSOC X Y)

Y is a list of lists. Y is scanned, comparing the first item of each sublist to X until a match is found, or until Y is exhausted. In case a match is found, ASSOC returns the entire sublist whose first item matched X.

EXAMPLE: (ASSOC 'HEAD-SIZE '((NAME SCREWDRIVER-1) ... (HEAD-SIZE 0.3 CM))) would return the sublist (HEAD-SIZE 0.3 CM).

3.1.3.3. (SUBST X Y Z)

X, Y and Z are arbitrary S-expressions. SUBST creates a new copy of Z, where all occurrences of Y in Z are replaced with X's.

EXAMPLE: (SUBST 0.2 0.3 '((NAME SCREWDRIVER-1) ... (HEAD-SIZE 0.3 CM))) would produce a new structure for our screwdriver, identical in all respects to the original, except that its head width would be 0.2 instead of 0.3.

3.1.3.4. (APPEND X Y)

X and Y are lists. A new list is created which is the result of appending Y onto the end of X.

EXAMPLE: (APPEND '((NAME SCREWDRIVER-1) (STYLE PHILLIPS)) '((COLOR-CODE YELLOW) (HEAD-SIZE 0.3 CM))) would produce ((NAME SCREWDRIVER-1) (STYLE PHILLIPS) (COLOR-CODE YELLOW) (HEAD-SIZE 0.3 CM))

3.1.4. LISP Data Types

In addition to atoms and CONS nodes, most LISP systems include the following other data types:

1. integer numbers
2. real numbers
3. strings
4. arrays
5. octal numbers (for bit-level manipulations)

Some versions of LISP (notably MACLISP [Moon74]) have highly developed numerical and trigonometric facilities and accompanying optimizing compilers geared to the efficient generation of "number crunching" software.

3.1.5. LISP Functions

A LISP "program" is a collection of functions. No function is syntactically declared as the "main program". Functions are generally typeless (i.e., no distinction such as "integer", "real", "string", etc. is made). However, each function may be declared so that its calling arguments are passed to it either evaluated (as in most programming languages), or unevaluated. Except for this distinction, there is no need for function-related declarations.

A function is regarded as simply another type of data. As such, one typically defines a function by assigning to some atom the function as the atom's value. Strictly speaking, the function itself is nameless, and is identified by the form:

```
(LAMBDA <argument-list> <body>)
```

When a "lambda expression" is stored as the value of an atom, we say that a function has been defined. Although the implementation details governing how a lambda expression comes to be associated with an atom vary considerably, one common format for defining a function in LISP is:

```
(DEFUN <name> <arguments> <body>)
```

DEFUN is a macro which creates the appropriate lambda expression and assigns it to the atom <name> as the function's body. A function may be annihilated or altered simply by reassigning the value of the atom which represents it. Another virtue of this separability of a function from its name is that nameless functions can be created and passed as arguments to other functions without having to bother to name them if they are needed only once.

To illustrate LISP functions, let us define a function of two arguments, (LOCATE-ALL <tool-type> <tool-list>), which, given the name of a tool type (e.g., SCREWDRIVER), and a master tool list, will search the tool list for tools of the specified type and report back a list of all tools of that type it finds. Framing this as a recursive function, we write:

```
(DEFUN LOCATE-ALL (TYPE MASTER-LIST)
  (COND ((NULL MASTER-LIST) NIL)
        ((EQUAL (GET (CAR MASTER-LIST) 'TOOL-TYPE) TYPE)
         (CONS (CAR MASTER-LIST)
               (LOCATE-ALL TYPE (CDR MASTER-LIST))))
        (T (LOCATE-ALL TYPE (CDR MASTER-LIST)))))
```

that is, if (COND) the master list is (or has been reduced to) NIL, then report back "nothing"; otherwise, if the next item on the master list (its CAR) is of the correct type (as determined by the GET), then add this tool to the list to be reported (i.e., CONS it onto the front of this list) and proceed with the search on the remainder of the list (its CDR); otherwise (T...), simply proceed, without recording the current tool.

Alternatively, we could express this algorithm in iterative form via the PROG feature:

```
(DEFUN LOCATE-ALL (TYPE MASTER-LIST)
  (PROG (RESULT)
    LOOP (COND ((NULL MASTER-LIST) (RETURN RESULT))
               ((EQUAL (GET (CAR MASTER-LIST) 'TOOL-TYPE) TYPE)
                (SETQ RESULT (CONS (CAR MASTER-LIST) RESULT))))
         (SETQ MASTER-LIST (CDR MASTER-LIST))
         (GO LOOP)))
```

i.e., enter a PROG (akin to an ALGOL begin-end block), defining one temporary local variable, RESULT; then, while the master-list remains non-nil, repeatedly examine its next item, collecting those with the correct type on the RESULT list (via SETQ, LISP's "assignment statement"), scanning to the next tool on the master list (SETQ MASTER-LIST (CDR MASTER-LIST)).

3.1.6. The PROG Feature

As just illustrated, LISP accommodates iteratively-phrased algorithms via a construction called a "PROG". A PROG has the form:

(PROG <local-variables> <statement-1> ... <statement-n>)

As a PROG is entered, the local variables (if any) are allocated for the scope of the PROG, and each is initialized to NIL. Next, the statements which comprise the PROG's body are sequentially executed (evaluated) until execution either "falls off the bottom" of the PROG (an implicit exit from the PROG), or until a GO or RETURN is encountered. Statements which are atoms are interpreted as labels within a PROG, and are ignored during sequential execution. When a GO is encountered, a branch to the specified label occurs, and sequential execution proceeds from that point.

Since a PROG introduces some temporary variables which must be reclaimed as the PROG is exited, there must be some way of informing LISP that a PROG is about to be exited. The function RETURN is used for this purpose, informing the system that a PROG is being exited, and specifying what value the PROG is to return to the calling environment.

PROG's may be nested and may appear at any point in a LISP program. The PROG construction will typically result in a more efficient implementation of an algorithm than the corresponding recursive implementation. Although some feel that PROG makes LISP "impure", it is in reality the feature which is probably most responsible for LISP's present widespread acceptance in both the AI community and elsewhere.

3.1.7. LISP Macros

Most LISP implementations support two types of macros: compile-time macros and scanner macros. A compile-time macro is nothing more than a function which, when evaluated, computes not a final result, but another S-expression which, when evaluated, will compute a final result. Thus, when a macro is encountered by the LISP interpreter, a double evaluation is performed (the first to compute the intermediate form, the second to run the intermediate form). When LISP functions are compiled into actual machine code, the compiler recognizes macros and evaluates them once to obtain the intermediate form which it then compiles. This technique is a very general and powerful implementation of the macro concept.

Most LISP scanners are quite modular, in the sense that they can be conditioned to initiate an arbitrary computation upon encountering a given character in the input stream. For example, in Wisconsin LISP [Norman69], there exists a facility called (READMAC <char> <function>), which conditions the scanner to call <function> (no arguments) whenever <char> is detected in the input stream. <function> is free to perform any computation, and whatever <function> returns is spliced into the scanner's input stream. This style of table-driven scanner makes it possible to superimpose additional syntax on LISP input, even to the point where LISP can model another language's syntax (by redefining delimiters, etc.). MLISP [Smith70] is an example of this.

3.1.8. Variable Scoping

LISP variable values are derived as a function of the run-time environment rather than as a function of lexical environment. As a program executes, there are two times at which new variables are introduced, or "bound": (1) at function entry time (these are the names of the function's arguments that are mentioned in the LAMBDA expression), and (2) at PROG entry time (i.e., the PROG's temporary variables). Variables are "unbound" at the corresponding exit times: when a function returns or when a PROG is exited.

At the "top-level" of LISP (when no function is currently executing), any variables which receive values are thought of as "global" to the system. Therefore, at any given moment during execution, there will be a pool of global atoms plus all the atoms introduced via LAMBDA or PROG on the current sequence of function calls. All these variables and their associated values ("bindings") are recorded on a structure called the "association list" (A-LIST), a user-accessible list of CONS nodes. All variable lookups consult this list, from most recent to least recent. Since this list is dynamically maintained at run-time, the question of what variables are and are not bound (i.e., are on the A-LIST) is exclusively determined by the dynamic calling environment, rather than the lexical scope of variables at the time functions were defined. This means that "free" variables (ones which have no binding at the current level) will assume a value at run-time which is dependent upon their definitions in functions farther up the calling hierarchy. In this manner, one function "peeks into", or borrows the variables of another.

By changing the system's A-LIST pointer while inside a function, that function's entire environment can be altered. For this reason, LISP is a very powerful tool wherever hypothetical reasoning (involving switches to altered contexts) is necessary. Most other languages either lack such an ability, or make it difficult to carry out. In LISP, context switching and "taking snapshots" of contexts to which execution is to be returned are very natural operations.

3.1.9. LISP I/O

Traditionally, input/output has been LISP's weakest link. Most systems define at least the following I/O-related functions:

(READ)	read an S-expression
(READCH)	read an individual character
(PRINT X)	print S-expression X, skipping to a new line
(PRIN1 X)	print S-expression X on the current output line
(TERPRI)	skip to beginning of new line on output

While these functions provide adequate formatting control, most LISPs are deficient in file-handling operations. (INTERLISP [Teitelman74] is the exception, with more highly developed interfaces to the TENEX virtual operating system). We regard this deficiency as more of a historical accident than as an inherent problem of LISP (since adding these features is simply a matter of writing the code). In fact, there are efforts underway for improved multiple-file interaction and random access facilities both at MIT (MACLISP) and at Maryland (Wisconsin LISP).

3.1.10. Garbage Collection

Since LISP data structures can grow in unrestricted ways, a crucial part of any LISP system is a conceptually asynchronous process called the "garbage collector". The role of this process is periodically to take control, mark parts of storage that are still referenced by the ongoing computation, then reclaim all storage that is not so referenced (garbage). Garbage collection is an unavoidable overhead of any system with no declarations, and in which data structures can grow in unrestricted ways.

One potential disadvantage of garbage collection is that, once the system runs out of free storage, a garbage collection _must_ occur.

Since a garbage collect causes current computing activity to be suspended, if LISP is controlling a real-time process, disastrous consequencs can accrue. Such problems can normally be avoided by forcing the system into a premature garbage collect prior to entering real-time critical sections of computation. Alternatively, there is growing interest in truly asynchronous (parallel) garbage collection techniques which could obviate the problem altogether (see [Dijkstra75] for instance).

3.1.11. LISP as a Self-Contained System

LISP interpreters are typically implemented in assembly language. After this basic facility has been brought up, most other supporting software can be written in LISP itself. Typical software includes

(1) A compiler which will generate (potentially quite good) machine code for LAMBDA expressions (i.e., functions) and PROGs. Typically, the LISP compiler will be written in interpreted LISP, then used to compile itself. The compiled version is subsequently used as the LISP system compiler.

(2) A debug package which will permit the tracing and interactive development of functions. Typically, functions (together with their calling arguments) can be traced at entry time, and (together with their returned values) at return time. Most LISPs will also accommodate the tracing of variables (i.e., inform the user whenever a traced variable's value is about to be changed). The debugging potentials of LISP are essentially unlimited (the INTERLISP system is the most advanced to date), and are responsible (in part) for LISP's reputation as one of the best languages for the efficient and rapid development of complex software. In particular, there is no time-consuming interaction with system compilers, loaders and linkers to be contended with; a program can be developed and put into production within the confines of the LISP system itself.

(3) An S-expression editor (or system editor interface) which makes possible the convenient editing of S-expressions and maintenance of files.

3.2. MICROPLANNER

While LISP is generally accepted as the standard for computing in AI, it does not supply the user with any a-priori conceptions about intelligence. LISP is simply the blank tablet onto which the user must write his theory of intelligence or control. Not surprisingly, this resulted in numerous reinventions of the wheel in areas like database organization, problem solving, hypothetical reasoning, and language understanding. Most reinventions were at a fairly low level, but occurred often enough to warrant some investigations into some of the undercurrents of AI programming techniques.

MICROPLANNER [Sussman, Winograd, Charniak 71] is the outcropping of some of these undercurrents, particularly where automatic problem solving is concerned. MICROPLANNER was written in 1970-71 as a small-scale implementation of ideas originally proposed by Hewitt in 1969 [Hewitt69]. The intent of the language was and is to provide some automatic mechanisms of database organization, context, and heuristic search.

MICROPLANNER is implemented entirely in LISP. Because of this, its syntax is essentially LISP's syntax, and while in the MICROPLANNER environment, the user has full access to all of LISP. To distinguish MICROPLANNER (hereafter abbreviated MP) functions from pure LISP functions, the convention is to prefix all MP functions (there are about 50 of them) with "TH" (standing, we presume, for "theorem", a key notion in MP).

The most salient features of MP are these:

(1) Computation in MP is induced by pattern, rather than by calling functions by their names. In this style of computation (often called "pattern-directed invocation"), whenever a goal requires solution, a pattern describing the goal is posted to the entire system. "Entire system" normally means a large population of problem-solving experts with patterns which advertise each one's expertise. Whenever a need is posted, the system searches through the database of experts looking for those whose advertised patterns match the need. Each expert so located is then tried in turn until one succeeds, or until all have failed. This is a radically different computing paradigm from the standard paradigm of "name calling", since it makes for a very modular system where the requestor needn't know any experts by name; problems are solved by anonymous experts in the population at large.

(2) MP automatically maintains a context-sensitive database of both factual assertions and the experts just mentioned. The factual database is a collection of highly indexed n-tuples, expressed as LISP S-expressions. Any one n-tuple ("assertion"), or collection of n-tuples can be "associatively" accessed by presenting the lookup routines with a pattern containing zero or more variables. Only those facts that are deemed active in the current "context", regardless of whether they physically exist in the memory, will be located.

(3) MP does all the bookkeeping required for depth-first, nondeterministic programming. That is, anytime there is a decision of any sort in MP, the system makes a choice (either arbitrarily, or under the control of user-specified heuristics), records the alternatives for possible future reference, and then proceeds. If a failure ever causes a "backup" to that decision point, the system automatically discards the current (failing) choice, selects the next alternative, and then attempts to proceed again. In the backup process, all computations performed between the initial (bad) choice and the failure point are undone (a record of all changes to the database is maintained), and the system picks up from the decision point as though nothing had ever gone wrong. Thus, MP can be said to maintain, at least implicitly, an entire goal tree (search tree) for each problem it attempts to solve. As we will suggest later, there are both advantages and disadvantages to such automatic control.

These are the three main contributions of MP. In the following sections we highlight and illustrate some of the specific features of this problem solving language.

3.2.1. The MICROPLANNER Database

Conceptually, the MP database is divided into two segments: facts and theorems. Theorems are further classified into three categories: "antecedent" theorems, "erasing" theorems, and "consequent" theorems. Theorems are discussed in section 3.2.2.

Both facts and theorems are entered into the database via the function THASSERT; an item is deleted from the database via the function THERASE. Facts are fully-constant LISP n-tuples. Thus, to represent our screwdriver in MP, we might augment the database as follows:

```
(THASSERT (TOOL-TYPE SCREWDRIVER-1 SCREWDRIVER))
(THASSERT (STYLE SCREWDRIVER-1 PHILLIPS))
...
(THASSERT (HEAD-SIZE SCREWDRIVER-1 0.3 CM))
```

Database lookups and fetches are accomplished via the function THGOAL. Therefore, if at some point in a MP program, we required a knowledge of SCREWDRIVER-1's head width, we could write a fetch pattern of the form:

```
(THGOAL (HEAD-SIZE SCREWDRIVER-1 (THV X) (THV Y)))
```

For our example, this would respond with "success" (i.e., a fact which matched this template was located in the database, and it would produce the side effects of binding the MP variables X and Y to 0.3 and CM, respectively. The THV form is used in MP to signal references to variables (all else is implicitly constant).

Every fact and theorem in the MP database has a context marking. Whenever a fact or theorem is THASSERTed, if such a fact is not already physically present in the database, it is created and then marked as also being logically present. If the THASSERTed fact is present physically, but marked as logically not present, its logical status is changed to "present". If the fact is already logically and physically present, THASSERT does nothing, but reports a "failure" to store a new copy of the fact. THERASE exerts opposite effects on facts in the database: it causes a fact to be logically masked, either by changing the fact's logical context marking, or by actually physically deleting the fact (i.e., if the fact is being THERASEd at the level at which it was originally THASSERTed).

Context markings allow MP to keep track of the history of the logical status of each fact and theorem. This enables the system to back up to prior context levels, thereby restoring the database to the corresponding prior state. Thus, although there are mechanisms for making permanent database changes (e.g., after some segment of MP code is confident that what it has done is absolutely correct), normally (except at the top level), THASSERT's and THERASE's are not permanent; instead, they normally exist only for the duration of some stretch of planning or hypothetical reasoning.

3.2.2. MICROPLANNER Theorems

All reasoning (in fact, all computation) in MP is carried out by THANTE, THERASING, and THCONSE "theorems" which are called by pattern rather than by name. The three types of theorem are indistinguishable in internal form, except with regard to the type of event to which each responds. A THANTE theorem is triggered by the THASSERTion into the factual database of any pattern which matches its invocation pattern. A THERASING theorem is triggered by the THERASEure from the database of any factual pattern which matches its invocation pattern. In the sense that these two classes of theorems respond spontaneously (not in response to any particular request), they represent a general interrupt capability. A THCONSE theorem responds to THGOAL requests whose goal patterns match its invocation pattern.

Because of this last interaction between THGOAL's and THCONSE, a THGOAL can amount to considerably more than a simple database fetch. In MP, when a THGOAL is issued, the system first attempts to locate the desired goal directly as a fact in the database. If this fails, and the THGOAL request has indicated that it is permissible to do so, MP will begin searching for THCONSE theorems whose invocation patterns match the desired goal. If any are found, each is executed in turn until one reports success (in which case the THGOAL is satisfied), or until all THCONSE theorems have failed (in which case the THGOAL fails). It is in this manner that more complex knowledge (i.e., theorems, problem solving techniques, etc.) can be automatically brought to bear on some goal if that goal is not already explicitly present in the factual database.

The forms of these three MP theorem types are:

```
(THANTE <optional-name> <variables> <invocation-pattern> <body>)

(THERASING <optional-name> <variables> <invocation-pattern> <body>)

(THCONSE <optional-name> <variables> <invocation-pattern> <body>)
```

As a brief illustration of the uses of each of these, suppose we wish to implement the following three capabilities in MP: (a) whenever a new screwdriver is defined to the system, automatically cause its name to be added to the master tool list; (b) whenever a screwdriver is deleted from the system, automatically remove its name from the master tool list, and also remove all its accompanying information; (c) whenever, during some assembly task, a THGOAL of the form: (SCREW-IN <some screw> <some threaded hole>) is announced, automatically search for, and return the name of an appropriate screwdriver for the task (based on the screw's style and head size). Task (a) will be modeled as a MP THANTE theorem, part (b) by a THERASING theorem, and part (c) by a THCONSE theorem as follows:

```
(THANTE (X) (TOOL-TYPE (THV X) SCREWDRIVER)
   (SETQ MASTER-TOOL-LIST (CONS (THV X) MASTER-TOOL-LIST)))

(THERASING (X) (TOOL-TYPE (THV X) SCREWDRIVER)
  (THPROG (ST CC ... HS HSU)
     (SETQ MASTER-TOOL-LIST (DELETE (THV X) MASTER-TOOL-LIST))
     (THAND (THGOAL (STYLE (THV X) (THV ST)))
            (THERASE (STYLE (THV X) (THV ST))))
     (THAND (THGOAL (COLOR-CODE (THV X) (THV CC)))
            (THERASE (COLOR-CODE (THV X) (THV CC))))
     ...
     (THAND (THGOAL (HEAD-SIZE (THV X) (THV HS) (THV HSU)))
            (THERASE (HEAD-SIZE (THV X) (THV HS) (THV HSU))))))

(THCONSE (SCREW HOLE) (SCREW-IN (THV SCREW) (THV HOLE))
  (THPROG (ST HS HSU DRIVER DST DHS DHSU)
    (THGOAL (STYLE (THV SCREW) (THV ST)))
    (THGOAL (HEAD-SIZE (THV HOLE) (THV HS) (THV HSU)))
    (THGOAL (TOOL-TYPE (THV DRIVER) SCREWDRIVER))
    (THAND (THGOAL (STYLE (THV DRIVER) (THV DST)))
           (EQUAL (THV DST) (THV ST)))
    (THAND (THGOAL (HEAD-SIZE (THV DRIVER) (THV DHS) (THV DHSU)))
           (EQUAL (THV DHS) (THV HS)))
    (THRETURN (THV DRIVER))))
```

3.2.3. <u>Heuristic Guidance of Theorem Application</u>

It is possible, by including special indicators in THGOAL, THASSERT and THERASE calls, to influence the order in which theorems are applied, or in fact to indicate whether or not they should be applied at all. Specifically, a THGOAL (similar remarks apply to THASSERT and THERASE) with <u>no</u> indicators will fail unless the requested goal can be satisfied exclusively by database fetches (no theorems will be applied). (This is the form we have been using for illustration purposes.) If there is an indicator present, it has either the form of a "filter" or a specific "recommendation list" of theorems (referenced by name). When a filter is included in a THGOAL request, only those theorems whose properties pass the filtering test (theorems can possess property lists) will be candidates for application. If the indicator has the form of a specific recommendation list, all theorems on that list will be applied first (in order) before any other theorems from the general theorem base are attempted. Both forms allow the programmer to insert limited heuristic influences. Also, since one MP theorem can create or modify another MP theorem, the filter facility provides a setting in which a collection of theorems themselves can evolve into a more structured configuration on the basis of past experience (e.g., who in the past has proven to be the most reliable expert). Although filtering and recommendations are a step in the right direction, as we will discuss later, CONNIVER provides a more flexible environment in which to encode heuristic knowledge.

3.2.4. Searching and Backup in MP

Search and backup in MP can occur for two reasons: (1) some THCONSE theorem which was run to accomplish a THGOAL fails, and another theorem must be invoked (restoring the environment to the state at which the first theorem took over), or (2) some object to which the system has committed itself is discovered to be inappropriate, giving rise to the need of locating another candidate object and retrying. The THGOAL-THCONSE mechanism underlie the selection and backup where theorems are concerned, but object selection is handled differently, via the THPROG MP construction.

In the previous THCONSE example, the goal was to locate some screwdriver which satisfied some set of features (in that case, the correct STYLE and HEAD-SIZE). This was accomplished by a THPROG which "conjectures" that such an object, say X, exists, then proceeds to determine whether or not this conjecture is true. In the example above, the THPROG searched for a screwdriver of type and size which matched the type and size of the particular screw which was to be inserted. For the sake of illustration, suppose the screw was of type Phillips of head size 0.3. Then, the THPROG in the example above would have performed essentially the same search as the following, more specific, THPROG:

```
(THPROG (X)
    (THGOAL (TOOL-TYPE (THV X) SCREWDRIVER))
    (THGOAL (STYLE (THV X) PHILLIPS))
    (THGOAL (HEAD-SIZE (THV X) 0.3))
    (THRETURN (THV X)))
```

i.e., introduce an initially uncommitted variable, X, to represent the object being searched for. First, obtain a candidate for X by finding an object which is of TOOL-TYPE SCREWDRIVER (the first THGOAL does this). At that point, X will be tentatively bound to the first such an object found. Continue with this candidate until either all THGOALs have been satisfied (in which case, the candidate is a success), or until some THGOAL fails (in which case, the system must back up and choose another candidate). Since some objects may pass the first THGOAL, or even two, but not all three, the system must automatically keep track of what object it is currently considering, and what other objects remain to be tested. This is the source of backups which are propagated because of bad object selections.

To keep track of theorem and object selection backups, MP maintains a decision tree, THTREE, which is essentially a record of every decision made, and what to do in case the decision leads to a failure. The strength of THTREE is, of course, that it frees the programmer from having to worry about failures: if there is a solution, it will eventually be found by an exhaustive search. The fatal weakness of THTREE is that it imposes an often undesirable depth-first ordering on the search (i.e., one subgoal must be solved in its entirety before any other subgoals can be attacked). This makes it difficult, if not impossible, to fabricate complexly intertwined solutions, since subgoals cannot communicate laterally in the tree. The MP organization is also quite awkward in its backup technique because of the depth-first organization of THTREE. Often, one small failure will cause an entire branch of THTREE to be undone, when in fact most of it was correct. It would be more desirable to be able to discard only the bad part of the tree, retaining the parts which are correct, so that wholesale resynthesis of large parts of the THTREE does not have to occur. Unfortunately, this is, again, very difficult, if not impossible to do in MP. CONNIVER has a better control structure in these respects.

3.2.5. Other Representative MP Capabilities

To complete our description of MICROPLANNER, we include two representatives of the other functions available in this language, together with a brief example of each.

3.2.5.1. (THFIND <mode> <variables> <skel> <body>)

THFIND provides a way of finding all objects in the system which satisfy a certain set of criteria. A THFIND is essentially a THPROG which is made to fail artificially after each successful location of an object which satisfies the criteria. <mode> indicates how many objects are to be located (e.g., "ALL", "(AT-LEAST <count>)",...); <variables> serve the same role as THPROG variables; <skel> specifies what form to return as each object is found; <body> contains the THGOAL's, etc. which define the criteria. THFIND returns either a failure (in case <mode> number of objects could not be found), or a list of <skel>'s, each <skel> corresponding to one successful object thus found.

```
EXAMPLE: (THFIND ALL (X) (THV X)
              (THGOAL (TOOL-TYPE (THV X) SCREWDRIVER))
              (THGOAL (STYLE (THV X) PHILLIPS))
```

would return a list of all tools which were Phillips screwdrivers.

3.2.5.2. (THMESSAGE <variables> <pattern> <body>)

As subgoals are descended into (i.e. "on the way down" the goal tree), THMESSAGE statements have no effect. They are essentially "hooks" which will intercept failures beneath them in the goal tree as such failures propagate back up to the THMESSAGE via a (THFAIL THMESSAGE <pattern>). Upon being backed up to by a THFAIL, any THMESSAGE whose pattern matches the THFAIL pattern will take control (its <body> will be executed). Thus, the THMESSAGE-THFAIL combination provides a way of anticipating possible problems without actually checking for them beforehand. If all goes well beneath the THMESSAGE, it will never run; however, if someone gets into trouble beneath the THMESSAGE (in some way the THMESSAGE is prepared for), the THMESSAGE can correct the problem and then cause the part of the tree beneath it to be reattempted.

```
EXAMPLE:  ...       (anticipate difficulty in inserting a screw)
        (THMESSAGE (X Y) ((THV X) WILL NOT TURN IN (THV Y))
          (THGOAL (LUBRICATE (THV X)))         (attempt a remedy)
          (THGOAL (SCREW-IN (THV X) (THV Y))))  (retry)
        ...
        ...       (attempt to insert some screw in some hole)
        ...       (report a failure back up to the THMESSAGE)
        (THFAIL THMESSAGE ((THV SCREW) WILL NOT TURN IN
                                          (THV HOLE)))
        ...
```

would anticipate, detect, report, and correct a problem, then retry.

3.3. CONNIVER

The most recent stage in the evolution of the LISP family of languages was the result of McDermott's and Sussman's development of a language called CONNIVER [McDermott, Sussman 73]. CONNIVER's development was principally motivated by the control structure deficiencies of MP, as suggested in the earlier discussion of THREE. Although there were some improvements in the database and pattern-directed invocation control (e.g., the pattern matcher is more sophisticated), the most significant feature of CONNIVER is its ability to maintain numerous computations in states of suspended animation, then to switch among them, working on many subgoals or alternate strategies in unison rather than one at a time. In such an environment, partial computations need not be undone simply because some small aspect of the problem solving has gone awry.

CONNIVER is less a programming language than it is a collection of ideas about control structure. (The language apparently has never been used for more than one or two significant programming tasks [Fahlman73]). Because of this, our discussion will omit most references to syntax, and highlight only the aspects of CONNIVER's control structure which are unusual or unique to it.

3.3.1. Frames, Au-revoir and Adieu

In a conventional programming language (MP included), one function calls another function either by name or pattern and waits until the called function returns control. In a conventional language, once a function returns, that copy of it dies; the function may be called anew, but the new call will cause a new "copy" of the function to begin. No memory of a function's current status can be preserved across call-return sequences. This type of control is usually carried out under the control of push-down stacks which record calling arguments and return addresses; calling a function causes stacks to be pushed, while returning from a function causes stacks to be popped, annihilating all control information.

In CONNIVER, things are quite a bit different. To call a function in CONNIVER is to create a so-called "frame" for the called function, rather than to push information onto a central stack. A function's frame will contain all the information needed to characterize the function at any moment (e.g., from what A-LIST it derives values for its free variables, to whom it is to return when it has finished, etc.). There are two important features of a frame. First, it is a user-accessible LISP data structure. This means that a function may alter its own or another function's frame in arbitrary ways, causing free variables to be looked up on some other function's A-LIST, or causing the identity of the function to which control is to be returned to be altered. Second, because there is no central stack which is chronologically pushed and popped at function entry/exit, execution control is free to meander from one function to the next without permanently closing any function. Thus, at any moment, there can be numerous suspended functions which may be resumed at the point at which they last relinquished control, or in fact, at an arbitrary labeled point within them.

As one might expect, this ability makes the context marking technique for items in the database more complex than in MP. In particular, since control may eventually be returned to any suspended function (the system in general has no way of knowing whether or not it actually will be), every fact in the database must have markings which specify for every suspended function, F, whether or not that fact is supposed to be logically present while F is running. To accomplish this type of marking, the MP context scheme was generalized from a stack-like arrangement to a tree of contexts. Basically, every fact lives on some branch of the tree, and functions have access to limbs of the tree. Although there is considerable overhead, the system manages to mask and unmask facts in the database in synchrony with the meandering of execution control from one function to the next.

To distinguish the permanent return of a function from the case where a function merely relinquishes control, reserving the option to continue, CONNIVER defines two methods of returning: ADIEU (final, permanent return) and AU-REVOIR (suspension). One very important application of the AU-REVOIR feature is in the (often costly) generation of alternatives. Rather than calling a function (such as THFIND in MP) to generate all possible candidates before any detailed filtering tests are applied (a procedure which may waste an inordinate amount of time in the initial collecting phase), in CONNIVER it is possible to call a "generator" function which will locate and return candidates one at a time, suspending itself across calls. This makes for a more intimate form of interaction between the generating and testing functions than is possible in MP, and can lead to more efficient searches because of this intimacy. To facilitate the use of generators, CONNIVER has some rather elaborate machinery for maintaining "possibilities lists", including a function, TRY-NEXT, which controls the extraction of possibilities from such lists.

Computation in CONNIVER is similar in most other regards to computation in MP. The counterparts of THANTE, THERASING and THCONSE theorems are, respectively, IF-ADDED, IF-REMOVED and IF-NEEDED "methods". Except for differences in syntax, and a more general pattern-directed invocation scheme, these three functions are the same as the MP versions. CONNIVER counterparts of MP's database and goal-statement functions, THASSERT, THERASE and THGOAL are, respectively, ADD, REMOVE and FETCH.

3.4. Efficiency of the LISP Language Family

Being an interpreted language, LISP is slower than, say, FORTRAN, by between one and two orders of magnitude. However, compiled LISP can be competitive with a good FORTRAN compiler. We feel that LISP provides the best of both worlds, in the sense that the interpreter provides for easy program development and debugging, while the LISP compiler can transform debugged code into production-level efficiency.

MICROPLANNER and CONNIVER, on the other hand, are inherently less efficient, primarily because of the control structures they superimpose on LISP. The fatal flaw with MP is its backup system, which can be extremely slow; compilation will not typically remedy the problem. CONNIVER is slow for similar reasons; however, in addition to data structures, processes must also be garbage collected, and an elaborate context tree must be maintained. Although these two languages contain many noteworthy features, we feel that neither (as currently implemented) is appropriate for production applications.

3.5. Standardization of the LISP Language Family

There are LISP systems for the following machines: PDP-10, PDP-11, UNIVAC 1106, 1108, 1110, CDC 6500, 6600, IBM 360, 370, SIGMA 5, and others. Being a relatively easy language to implement, we would anticipate no significant development problems for any machine, including microcomputers. Since LISP's syntax is nearly non-existent, there is exactly one dialect. Although there are minor differences in the semantics of how functions are defined, and how variables' values are accessed, such "incompatibilities" can normally be ameliorated in about one day's worth of macro-writing. Because of this, LISP can be characterized as a language which is fairly standard and transportable. Finally, most LISP systems have an accompanying compiler, usually written in LISP itself.

4. Related Languages

4.1. AL

AL is a high-level programming system for specification of manipulatory tasks, developed at Stanford Artificial Intelligence Laboratory [Finkel74]. It is a SAIL-like language and includes large runtime support for controlling devices.

Trajectory calculation is a crucial feature of manipulatory control. AL contains a wide range of primitives to support efficient trajectory calculations. As much computation as possible is done at compile-time and calculations are modified at run-time only as necessary.

Besides a dimensionless scalar data type (i.e., REAL), AL recognizes and manipulates TIME, MASS and ANGLE SCALARs, dimensionless and typed VECTORs, ROT (rotation), FRAME (coordinate system), PLANE (region separator) and TRANS (transformation) data types. Proper composition of variables of these types gives a simple means of performing calculations of any type of movement.

Also included are PL/1-like ON-conditions, which allow monitoring of the outside world, and concurrent processes.

Example:

```
PLANE p1;
  .
  .           { statements initializing p1 }
  .
SEARCH yellow                       { SEARCH is a primitive which causes
                                      a hand to move over a specified
                                      area.  yellow is a hand }
   ACROSS p1                        { hand moves across plane }
   WITH INCREMENT = 3*CM            { every 3 cm }
   REPEATING
      BEGIN                         { do at every iteration }
         FRAME set;
         set _ yellow;              { yellow is also coord system of hand }
         MOVE yellow XOR - Z*CM
                                    { move hand 1 cm down from current
                                      position along Z-axis }
            ON FORCE(Z) > 3000*DYNES
               DO TERMINATE;        { keep in touch with real world }
         MOVE yellow TO set DIRECTLY;  { move the hand back to where
                                          it was in a straight line }
      END;
  .
  .
  .
```

4.2. MLISP

MLISP (meta-LISP) is a high-level list-processing language developed at Stanford University [Smith70]. MLISP programs are translated into LISP programs which are then executed or compiled. The MLISP translator itself is written in LISP.

MLISP is an attempt to improve the readability of LISP programs as well as alleviate some inconveniences in the control structure of LISP (e.g., no explicit iterative construct). Since run-time errors are only detected by the LISP system (when actually executing the program), users frequently find themselves debugging the translated LISP code. This somewhat defeats the purpose of any high-level language.

All LISP functions are recognized and translated in MLISP, but the Cambridge prefix notation of LISP has been replaced by standard infix and prexix function notation. Instead of (PLUS X Y) one may write X + Y, and (FOO 'A B C) becomes FOO('A, B, C).

MLISP also provides a powerful set of iterative statements and a large number of "vector operators." Vector operators are used to apply standard operators in a straightforward manner to lists. Thus, in MLISP, <1, 2, 3> +@ <6, 5, 4> yields <7, 7, 7>. +@ is the vector addition operator and <A, B, C> is equivalent to (LIST A B C) in LISP.

Example:

Given a list of the form <obj1, obj2, ..., objn>, this function will return a list of the form <<obj1, holder1>, ..., <objn, holdern>> where holderi is either PLIERS, VISE or NOTHING accordingly as needed to hold the object. % ...% is an MLISP comment.

```
EXPR HOLD-LIST(OBJ-LIST);           % EXPR starts a regular func     %
   BEGIN
      NEW S;                        % local declaration              %
      RETURN                        % RETURN is a unary operator     %
         FOR NEW OBJ IN OBJ-LIST
          COLLECT
                                    % OBJ is local to the FOR loop.  %
                                    % OBJ will be bound in turn      %
                                    % to each element of OBJ-LIST.   %
                                    % COLLECT indicates that the     %
                                    % result of each iteration is    %
                                    % to be APPENDed to the previous %
                                    % result and this whole list     %
                                    % returned as the result of      %
                                    % the FOR.                       %
           IF (S GET(OBJ, 'SIZE)) LEQUAL 5
              THEN
                 <<OBJ, 'PLIERS>>
              ELSE
                 IF S LEQUAL 10
                    THEN
                       <<OBJ, 'VISE>>
                    ELSE
                       <<OBJ, 'NOTHING>>
   END;
```

4.3. POP-2

POP-2 is a conversational language designed by R. M. Burstall and R. J. Popplestone at the University of Edinburgh [Burstall71].

POP-2 features an Algol-like syntax and draws heavily from LISP. Integers, reals, LISP-like lists and atoms (called 'names'), function constants (lambda expressions), records, arrays, extensible data types, and run-time macros are supported. A unique feature of the POP-2 system is the heavy use of a system stack, which the user may easily control to enhance the efficiency of programs.

A full complement of list-manipulation, numeric and storage-management functions are available.

Example:

Suppose we wish to obtain a list of all machinery not currently functioning. A useful function would be,

```
COMMENT  sublist returns a list of all elements of argument list xl
         which satisfy argument predicate p ;

FUNCTION sublist xl p;           { arguments are xl and p }
   VARS x;                       { declaration of local, no type }
   IF null(xl) THEN nil          { just like LISP }
      ELSE hd(xl) -> x;          { hd(a) = (car a) }
           IF p(x)
              THEN x::sublist(tl(xl), p)
                             { tl(a) = (cdr a), x::l = (cons x l) }
              ELSE sublist(tl(xl), p)
           CLOSE
   CLOSE
END;
```

A call might then look like,

```
sublist(machine-list,
             LAMBDA m; not(functioning(m)) END);
```

which might return,

```
[punch-press1 drill-press2 unit10]
```

which is a POP-2 list.

4.4. QLISP

QLISP is an extended version of QA4 (a PLANNER-like LISP derivative) [Rulifson 1973] embedded in the sophisticated INTERLISP system. QLISP supports a wide variety of data types designed to aid in the flexible handling of large data bases. Among the data types supported are "TUPLE," "BAG" and "CLASS." A TUPLE is essentially a LISP list that can be retrieved associatively (see below). A BAG is a multiset, an unordered collection of (possibly duplicated) elements. Bags have been found to be useful for describing certain commutative associative relations. A CLASS is an unordered collection of non-duplicated elements (i.e., basically a set).

Arbitrary expressions may be stored in the system data base and manipulated associatively. The QLISP pattern matcher is used to retrieve expressions in a flexible manner. The system function MATCHQQ may be used to invoke the pattern matcher explicitly, as in:

```
(MATCHQQ (<-X <-Y) (A B))
```

which causes X to be bound to A and Y to B ("<-" indicates this "need for a binding"). The patterns to MATCHQQ may be arbitrarily complex, as in:

```
(MATCHQQ (A (<-X <-Y)) (<-X (A (B C))))
```

in which X is bound to A and Y to (B C).

QLISP expressions are represented uniquely in the data base, unlike LISP where only atoms are unique. To distinguish between "identical" expressions, "properties" may be associated with any expression by QPUT.

```
(QPUT (UNION (A B)) EQUIV (UNION (B C)))
```

The above puts the expression (UNION (B C)) under the property EQUIV for the expression (UNION A B).

QLISP provides facilities for backtracking and pattern-directed invocation of functions, as illustrated by:

```
(QLAMBDA (FRIENDS JOE (CLASS <-F <-S <-<-REST))
    (IS (FATHER $S $F))
    BACKTRACK)
```

This function will find an occurrence of a CLASS denoting FRIENDS of JOE. F and S will be bound to the first two elements of the CLASS and REST will be bound to the remainder of the CLASS (indicated by "<-<-"). If S is a father of F, then the function succeeds. ("$" causes the current binding of its argument to be used.) BACKTRACK causes re-invocation of the function with new bindings for S, F and REST until the function succeeds or there are no untried bindings.

The user may collect teams of functions to be invoked under desired circumstances. Many QLISP data base manipulation functions may have optional arguments which denote a team of routines to be used to perform antecedent-type functions (as in PLANNER).

QLISP provides a general context and generator mechanism similar to that of CONNIVER. Also provided is a smooth, readily accessible interface to the underlying INTERLISP system which aids in the development and maintenance of large systems.

Future plans for QLISP include multiprocessing primitives, semantic criteria for pattern matching (as opposed to the current syntactic information), and the ability for the pattern matcher to return more information than a simple match or fail.

5. Examples

5.1. Introduction

A common example will be used to illustrate the distinguishing features of SAIL, LISP, MICROPLANNER and CONNIVER. With only minor variations the program segments use the same algorithm. The program-segments appear out of context and are not meant to indicate the most eficient (or preferred) implementation of the problem in each language, but merely to illustrate the languages' major attributes.

Problem statement:

Given two distinct assemblies (say A1 and A2), attempt to unscrew A1 from A2, and indicate success or failure accordingly. The "world" of the example is assumed to include:

(1) Two hands, LEFT and RIGHT, capable of moving, grasping, twisting and sensing force and motion.

(2) A fixed number (possibly zero) of PLIERS

(3) A fixed number (possibly zero) of VISEs

(4) A fixed number of "assemblies"

For each PLIERS and VISE, the data base contains an assertion of the form, "PLIERS (VISE) # n is at location (X, Y, Z) and is of capacity C cm." In addition, for each assembly the data base contains an assertion of the form, "assembly A is at location (X, Y, Z) and is of size S cm." As we shall see, the languages are distinguished in part by the methods each uses to represent such knowledge.

Each example assumes the existence of the routines described below in ALGOL-like notation.

ATTACHED(A1, A2) - TRUE if and only if the assembly represented by A1 (hereafter referred to as A1) is attached to the assembly represented by A2 (referred to as A2). The routine has no side effects.

MOVE(HAND, LOCATION) - Moves HAND (LEFT or RIGHT) to LOCATION (but see PLANNER's description of MOVE).

TWIST(HAND, DIRECTION) - Twists HAND (LEFT or RIGHT) in the given DIRECTION (CLOCKWISE or COUNTER-CLOCKWISE). The DIRECTION is oriented looking down the length of the arm. Except for SAIL, all programs assume a routine called TWIST-BOTH, which causes both hands to twist at once.

GRASP(HAND, OBJECT) - Causes HAND (LEFT or RIGHT) to grasp OBJECT, which must be within some fixed range of HAND (i.e., the hand must MOVE to the OBJECT first).

ATTEMPT(OBJ1, OBJ2, A1, A2) - Attempts to do the actual unscrewing of assembly A1 from A2 using objects OBJ1 and OBJ2 (which, in our examples, are either VISEs or PLIERS). ATTEMPT returns TRUE if and only if the attempt is successful.

Each program applies the following sequence to solve the problem:

(1) Attempt to unscrew the assemblies using the hands. This entails obtaining the location of the assemblies, moving the hands to their respective locations, grasping, and then twisting.

(2) If the objects are no longer attached, then return "success."

(3) At this point, it is assumed that the hands weren't strong enough. It is proposeo to try two pairs of PLIERS next. A search ensues for a suitable set of available PLIERS (i.e., large nough to hold the assemblies). If one set of PLIERS fails, the search is continued for another set, with the hope that the differences among PLIERS (grip, size, etc.) will eventually lead to success.

(4) An attempt to use PLIERS has failed. Try to solve the problem by holding one of the assemblies in a VISE. Perform a search for an appropriate VISE. This search proceeds in a fashion similar to that in (3).

(5) All attempts have failed. Output an appropriate message and return "failure".

5.2. SAIL

5.2.1. Sample Program

```
INTEGER PROCEDURE BIGENOUGH(ITEMVAR HOLDER, HOLDEE);

        " RETURN TRUE IFF OBJECT HOLDER IS LARGE
          ENOUGH TO HOLD OBJECT HOLDEE "

BEGIN

        INTEGER ITEMVAR C, S;

        C _ COP(CAPACITY XOR HOLDER);
        S _ COP(SIZE XOR HOLDEE);
        RETURN(DATUM(C) GEQ DATUM(S))

END;



INTEGER PROCEDURE UNSCREW(ITEMVAR A1, A2);

   " ATTEMPT TO DISASSEMBLE ASSEMBLY A1 FROM A2, BY UNSCREWING "

BEGIN

  DEFINE RUNME = 1;

  ITEMVAR V1, PL1, PL2, P1, P2;

  INTEGER FLAG;

  IF NOT ATTACHED(A1, A2) THEN RETURN(1);  " DON'T BOTHER "

  MOVE(LEFT, LOCATION XOR A1); MOVE(RIGHT, LOCATION XOR A2);
  GRASP(LEFT, A1);  GRASP(RIGHT, A2);

  " GET BOTH HANDS TWISTING AT ONCE "

  SPROUT(P1, TWIST(LEFT, COUNTER!CLOCKWISE), RUNME);
  SPROUT(P2, TWIST(RIGHT, COUNTER!CLOCKWISE), RUNME);
  JOIN({P1, P2});
  IF NOT ATTACHED(A1, A2) THEN RETURN(1);

  " HANDS NOT STRONG ENOUGH, TRY PLIERS "

  FOREACH PL1, PL2 |
          ISA XOR PL1 EQV PLIERS AND (BIGENOUGH(PL1, A1))
      AND ISA XOR PL2 EQV PLIERS AND (PL1 NEQ PL2)
      AND (BIGENOUGH(PL2, A2)) AND (ATTEMPT(PL1, PL2, A1, A2))
  DO RETURN(1);

  " EITHER THERE WEREN'T ANY PLIERS LARGE ENOUGH,
    OR THE PLIERS WEREN'T STRONG ENOUGH.  TRY A
    VISE ON ONE SIDE "

  FOREACH V1, PL1 |
          ISA XOR V1 EQV VISE AND (BIGENOUGH(V1, A1))
      AND ISA XOR PL1 EQV PLIERS AND (BIGENOUGH(PL1, A2))
      AND (ATTEMPT(V1, PL1, A1, A2))
  DO RETURN(1);

  " ALL ATTEMPTS FAILED "
```

```

    OUTSTR("CAN'T UNSCREW " & CVIS(A1, FLAG) & " & "
            & CVIS(A2, FLAG) & ('15 & '12));
    RETURN(0)

  END;
```

5.2.2. Commentary

2. In SAIL, FALSE = 0, TRUE <> 0. BIGENOUGH is a BOOLEAN procedure.

9. C and S are items whose DATUM is assumed to be of INTEGER type.

11. COP(<set>) returns the first item of <set>. We are assuming that there exists only one triple of the form CAPACITY XOR <object> EQV <capacity> for each <object>.

13. C and S are necessary because DATUM(COP(<set>)) is illegal. SAIL must know at compile-time what the type of a DATUM is. GEQ is a numeric test for greater than or equal.

20. UNSCREW is a BOOLEAN procedure which returns TRUE (non-zero) if it succeeds in unscrewing the objects.

26. This is a macro definition. Whenever RUNME is encountered by the SAIL compiler, it will be replaced by the constant 1. (See 39. for its use.)

39. SPROUT is a SAIL function which causes activation of its second argument (a procedure/function call) as a process. The first argument is an item whose DATUM will be set by SPROUT to contain information about the SPROUTed process (see 41. for its use). The third argument to SPROUT determines the status of the current and the created process. RUNME (bit 35 set) indicates that the current and new processes are to be run in parallel by the SAIL scheduler.

47. BOOLEAN tests in a FOREACH must be enclosed in parentheses.

48. Notice (PL1 NEQ PL2) to insure that two distinct pairs of pliers are found.

50. If the body of the FOREACH is entered, then all went well and we return success.

64. CVIS is a SAIL function which will return a character string 'name' associated with an item. FLAG is set by CVIS to indicate the presence of an error.

5.3. LISP

5.3.1. Sample Program

```
(DEFUN UNSCREW (A1 A2)

  ?  ATTEMPT DISASSEMBLY OF OBJECT A1 FROM A2, BY UNSCREWING

(PROG (PL1 PL2 V1 IN)

      (COND [(NOT (ATTACHED A1 A2)) (RETURN T)])

      (MOVE 'LEFT (GET A1 'LOCATION))
      (MOVE 'RIGHT (GET A2 'LOCATION))
      (GRASP 'LEFT A1) (GRASP 'RIGHT A2)
      (TWIST-BOTH 'COUNTER-CLOCKWISE)
      (COND [(NOT (ATTACHED A1 A2)) (RETURN T)])

    ?  HANDS NOT STRONG ENOUGH, TRY PLIERS

      (COND [(FOREACH PL1 IN PLIERS-LIST (BIGENOUGH PL1 A1)
                      PL2 IN PLIERS-LIST (AND (NOT (EQ PL1 PL2))
                                              (BIGENOUGH PL2 A2))
               DO (ATTEMPT PL1 PL2 A1 A2))
             (RETURN T)]

    ?  PLIERS NOT LARGE ENOUGH OR NOT STRONG ENOUGH.
    ?  TRY A VISE ON 1 SIDE

            [(FOREACH V1 IN VISE-LIST (BIGENOUGH V1 A1)
                      PL1 IN PLIERS-LIST (BIGENOUGH PL1 A2)
               DO (ATTEMPT V1 PL1 A1 A2))
             (RETURN T)]

      ?  ALL ATTEMPTS FAILED

            [T (PRIN1 "CAN'T UNSCREW ") (PRIN1 A1)
               (PRIN1 " & ") (PRIN1 A2) (TERPRI)
                 (RETURN NIL)])
))




(DEFUN BIGENOUGH (HOLDER HOLDEE)

  ?  RETURN T IFF OBJECT HOLDER IS LARGE ENOUGH TO
  ?  HOLD OBJECT HOLDEE

  (NOT (LESSP (GET HOLDER 'CAPACITY)
              (GET HOLDEE 'SIZE)))
)





(DEFSPEC FOREACH (LAMBDA (OBJ1 IN1 LIST1 PRED1
                          OBJ2 IN2 LIST2 PRED2
                          DO TRY)

  ?  MIMIC SAIL FOREACH IN SIMPLE CASE

(PROG (TEMP1 TEMP2)
```

```
        (SETQ TEMP1 (EVAL LIST1))
LOOP1
        (COND [(NULL TEMP1) (RETURN NIL)])    ? RAN OUT
        (SET OBJ1 (CAR TEMP1))
        (SETQ TEMP1 (CDR TEMP1))
        (COND [(NOT (EVAL PRED1)) (GO LOOP1)])   ? FAILED 1ST TEST
        (SETQ TEMP2 (EVAL LIST2))
LOOP2
        (COND [(NULL TEMP2) (GO LOOP1)])
        (SET OBJ2 (CAR TEMP2))
        (SETQ TEMP2 (CDR TEMP2))
        (COND [(NOT (EVAL PRED2)) (GO LOOP2)]
              [(EVAL TRY) (RETURN T)]             ? IT WORKED
              [T (GO LOOP2)]])
)))




(DEFMAC FOREACH (LAMBDA (OBJ1 IN1 LIST1 PRED1
                         OBJ2 IN2 LIST2 PRED2
                         DO TRY)

  ?  MACRO VERSION OF FOREACH

(LIST 'PROG '(L1 L2)
        (LIST 'SETQ 'L1 LIST1)
'LOOP1
       '(COND [(NULL L1) (RETURN NIL)])
        (LIST 'SETQ OBJ1 '(CAR L1))
       '(SETQ L1 (CDR L1))
        (LIST 'COND (LIST (LIST 'NOT PRED1) '(GO LOOP1)))
        (LIST 'SETQ 'L2 LIST2)
'LOOP2
       '(COND [(NULL L2) (GO LOOP1)])
        (LIST 'SETQ OBJ2 '(CAR L2))
       '(SETQ L2 (CDR L2))
        (LIST 'COND (LIST (LIST 'NOT PRED2) '(GO LOOP2))
                    (LIST TRY '(RETURN T))
                    '(T (GO LOOP2))))
))
```

5.3.2. Commentary

2. UNSCREW is the main function. It returns T if and only if disassembly was successful.

13. Unlike SAIL, LISP does not support concurrency. We thus assume a primitive function to get both hands twisting.

18. FOREACH is an iterative special form which mimics a simple SAIL FOREACH. FOREACH will try pairs of pliers until the given predicates succeed or it runs out of pliers (and returns NIL). Note that the arguments to a special form need not be quoted.

19. Check to insure that distinct pairs of pliers are found.

34. PRIN1 is a LISP function which loads its argument into the stream output buffer.

35. TERPRI is a LISP function which dumps the output buffer.

47. Return T if capacity >= size.

55. DEFSPEC defines a special form (sometimes called a FEXPR). A special form is identical to a LISP function except that its arguments are passed unevaluated.

63. EVAL is necessary since the argument was passed unevaluated.

66. Note the use of SET rather than SETQ. OBJ1 needs to be evaluated to get the intended atom (SET evaluates its first argument, SETQ does not).

68. Note the use of EVAL (see 63.).

72. Note the use of SET (see 66.).

82. This is an alternative macro version of FOREACH. It expands into a PROG which is similar in nature to the special form FOREACH. Note the absence of SET or EVAL.

5.4. PLANNER (MICROPLANNER)

5.4.1. Sample Program

```
(THCONSE UNSCREW (A1 A2)
   (UNSCREW (THV A1) (THV A2))

 ?  ATTEMPT DISASSEMBLY OF OBJECT A1 FROM A2, BY UNSCREWING

(THOR
     (THNOT (ATTACHED (THV A1) (THV A2)))
     (THAND
          (THGOAL (MOVE LEFT (THV A1)) (THTBF THTRUE))
          (THGOAL (MOVE RIGHT (THV A2)) (THTBF THTRUE))
          (GRASP 'LEFT (THV A1)) (GRASP 'RIGHT (THV A2))
          (TWIST-BOTH 'COUNTER-CLOCKWISE)
          (THNOT (ATTACHED (THV A1) (THV A2)))
     )

   ?  HANDS NOT STRONG ENOUGH, TRY PLIERS

     (THPROG (PL1 PL2)
          (THGOAL (ISA (THV PL1) PLIERS) (THTBF THTRUE))
          (THGOAL (BIGENOUGH (THV PL1) (THV A1)) (THNODB)
                            (THUSE BIGENOUGH) (THTBF THTRUE))
          (THGOAL (ISA (THV PL2) PLIERS) (THTBF THTRUE))
          (THNOT (EQ (THV PL1) (THV PL2)))
          (THGOAL (BIGENOUGH (THV PL2) (THV A2)) (THNODB)
                            (THUSE BIGENOUGH) (THTBF THTRUE))
          (ATTEMPT (THV PL1) (THV PL2) (THV A1) (THV A2))
     )

   ?  NO PLIERS LARGE ENOUGH, OR NO PLIERS STRONG ENOUGH.
   ?  TRY A VISE ON 1 SIDE

     (THPROG (V1 PL)
          (THGOAL (ISA (THV V1) VISE) (THTBF THTRUE))
          (THGOAL (BIGENOUGH (THV V1) (THV A1)) (THNODB)
                            (THUSE BIGENOUGH) (THTBF THTRUE))
          (THGOAL (ISA (THV PL) PLIERS) (THTBF THTRUE))
          (THGOAL (BIGENOUGH (THV PL) (THV A2)) (THNODB)
                            (THUSE BIGENOUGH) (THTBF THTRUE))
          (ATTEMPT (THV V1) (THV PL) (THV A1) (THV A2))
     )

   ?  NOTHING WORKED, JUST FAIL

     (THNOT (THDO
        (PRIN1 "CAN'T UNSCREW ") (PRIN1 (THV A1))
        (PRIN1 "    ") (PRIN1 (THV A2)) (TERPRI)
     ))
     (THFAIL THEOREM)
))



(THCONSE BIGENOUGH (HOLDER HOLDEE C S)
   (BIGENOUGH (THV HOLDER) (THV HOLDEE))

 ?  SUCCEEDS ONLY IF OBJECT HOLDER IS LARGE ENOUGH TO HOLD
 ?  OBJECT HOLDEE

(THGOAL (CAPACITY (THV HOLDER) (THV C)) (THTBF THTRUE))
(THGOAL (SIZE (THV HOLDEE) (THV S)) (THTBF THTRUE))
```

```
(THCOND [(NOT (LESSP (THV C) (THV S)))
               (THSUCCEED)]
        [T (THFAIL THEOREM)])
)
```

5.4.2. <u>Commentary</u>

2. Defines and asserts a consequent theorem with name UNSCREW.

3. This is the pattern on which to invoke this theorem if needed (e.g., (UNSCREW ASSEMBLY1 ASSEMBLY2)).

7. THOR sequentially executes each of its arguments until one succeeds, and then the THOR succeeds. The THOR is used here to prevent undesired backup.

8. (THNOT p) is defined as (COND [p (THFAIL)] [T (THSUCCEED)]) .

9. THAND succeeds if and only if all of its arguments succeed. Unlike THOR, backup may occur among the arguments of a THAND.

10. Attempt to move the left hand to object A1. There may be several experts (theorems) on moving hands, PLANNER will try as many as it needs. (THTBF THTRUE) is a theorem base "filter" which is satisfied by every theorem.

19. THPROG behaves in a similar manner to THAND except that local variables may be declared.

20. Attempt to find a pair of pliers.

21. See if the pair of pliers is large enough. (THNODB) indicates to PLANNER not to bother searching the data base. (THUSE <theorem>) indicates to try <theorem> first.

24. Make sure that we have two distinct pairs of pliers.

45. THDO executes its arguments and then succeeds. However, at this point we know that we have failed, and THNOT is used to generate a failure from THDO. This is necessary because PRIN1 returns its first argument as its result, which (being non-NIL) would cause the THOR to succeed.

49. Generate explicit failure of the theorem.

5.5. CONNIVER

5.5.1. Sample Program

```
(CDEFUN UNSCREW (A1 A2)

  ?  ATTEMPT TO DISASSEMBLE A1 FROM A2, BY UNSCREWING

           "AUX" (LOC1 LOC2 GEN1 GEN2 V1 PL1 PL2)

       (COND [(NOT (ATTACHED A1 A2)) (RETURN T)])

       (PRESENT '(LOCATION !,A1 !>LOC1))
       (PRESENT '(LOCATION !,A2 !>LOC2))
       (MOVE 'LEFT LOC1) (MOVE 'RIGHT LOC2)
       (GRASP 'LEFT A1) (GRASP 'RIGHT A2)
       (COND [(NOT (ATTACHED A1 A2)) (RETURN T)])

       ?  HANDS NOT STRONG ENOUGH, TRY PLIERS

       (CSETQ GEN1 !"((*POSSIBILITIES) *IGNORE
             (*GENERATOR (NEXT-OBJ 'PLIERS '(BIGENOUGH $ A1)))))
:PLOOP1
       (CSETQ PL1 (TRY-NEXT GEN1 '(GO 'TRY-VISE)))
       (CSETQ GEN2 !"((*POSSIBILITIES) *IGNORE
                   (*GENERATOR (NEXT-OBJ 'PLIERS
                               '(AND (NOT (EQ PL1 $))
                                     (BIGENOUGH $ A2))))))
:PLOOP2
       (CSETQ PL2 (TRY-NEXT GEN2 '(GO 'PLOOP1)))
       (COND [(ATTEMPT PL1 PL2 A1 A2) (RETURN T)]
             [T (GO 'PLOOP2)]])

       ?  NO PLIERS LARGE ENOUGH, OR PLIERS NOT STRONG
       ?  ENOUGH.  TRY A VISE ON ONE SIDE.

:TRY-VISE
       (CSETQ GEN1 !"((*POSSIBILITIES) *IGNORE
                (*GENERATOR (NEXT-OBJ 'VISE '(BIGENOUGH $ A1)))))
:VLOOP
       (CSETQ V1 (TRY-NEXT GEN1 '(GO 'NO-CAN-DO)))
       (CSETQ GEN2 !"((*POSSIBILITIES) *IGNORE
             (*GENERATOR (NEXT-OBJ 'PLIERS '(BIGENOUGH $ A2)))))
:PLOOP3
       (CSETQ PL1 (TRY-NEXT GEN2 '(GO 'VLOOP)))
       (COND [(ATTEMPT V1 PL1 A1 A2) (RETURN T)]
             [T (GO 'PLOOP3)]])

       ?  ALL ATTEMPTS FAILED

:NO-CAN-DO
       (PRIN1 "CAN'T UNSCREW ") (PRIN1 A1)
       (PRIN1 "   ") (PRIN1 A2) (TERPRI)
       (RETURN NIL)
)



(CDEFUN BIGENOUGH (HOLDER HOLDEE)

  ?  RETURN T IFF OBJECT HOLDER IS LARGE
  ?  ENOUGH TO HOLD OBJECT HOLDEE

           "AUX" (C S)
```

```
(PRESENT '(CAPACITY !,HOLDER !>C))
(PRESENT '(SIZE !,HOLDEE !>S))
(NOT (LESSP C S))
)



(CDEFUN NEXT-OBJ (TYPE PRED)

  ?  GENERATOR TO RETURN NEXT OBJECT OF 'TYPE'
  ?  WHICH SATISFIES 'PRED'

                "AUX" (OBJ TEMP)

        (CSETQ TEMP (FETCH '(ISA !>OBJ !,TYPE)))
:LOOP
        (TRY-NEXT TEMP '(ADIEU))
        (COND [(CVAL (SUBST OBJ 'S PRED))
                  (NOTE OBJ)
                  (AU-REVOIR)])
        (GO 'LOOP)
)
```

5.5.2. Commentary

2. CDEFUN defines a function to CONNIVER.

6. "AUX" <list> defines local variables.

10. PRESENT is a CONNIVER function which searches the data base for an item which matches its pattern argument. If one is found, PRESENT sets the indicated variables (marked with !< or !>) and returns the item. !,A1 indicates the current CONNIVER value of A1. !>LOC1 indicates that LOC1 is to be bound if possible.

13. GEN1 is being assigned a TRY-NEXT possibilities list. !" tells CONNIVER to do a "skeleton expansion" of the following list (which is necessary to CONNIVER's internals). The (*POSSIBILITIES) and *IGNORE are syntatic markers to TRY-NEXT whose function we can ignore. (*GENERATOR <func-call>) indicates to TRY-NEXT to use <func-call> to generate additional possibilities if needed.

19. NEXT-OBJ will continue to generate objects of type PLIERS which satisfy the predicate (2nd argument). It will generate one PLIERS at a time. (BIGENOUGH $ A1) is a skeleton predicate which NEXT-OBJ will use to screen each possibility. The current candidate is substituted for $ before the predicate is CVALuated (CONNIVER's form of EVALuation).

21. When GEN1 contains no more possibilities, TRY-NEXT will execute (GO 'TRY-VISE). Unlike LISP, GO evaluates its argument here.

24. Check to insure that two distinct pairs of pliers will be found.

64. See 10.

66. RETURN is not necessary since the value of a CONNIVER function is the last expression evaluated.

72. Define the generator, NEXT-OBJ. Note that NEXT-OBJ looks like a regular function to CONNIVER until it is called.

79. FETCH is a CONNIVER primitive which returns a possibilities list of all items in the data base which match its pattern argument. !>OBJ indicates that OBJ should be bound by TRY-NEXT to each possibility in turn.

81. TRY-NEXT binds OBJ from the possibilities list TEMP and removes the current possibility. If there is no current possibility, (ADIEU) is evaluated which causes termination of the generator.

82. The desired predicate is CVALuated after substituting the current object into the skeleton. (SUBST A B C) is a LISP function which returns a list which is the result of substituting A for every occurrence of B in list C.

83. (NOTE OBJ) is a CONNIVER function which places the current value of OBJ onto the current possibilities list.

84. (AU-REVOIR) returns control from NEXT-OBJ but leaves the generator in a suspended state. When TRY-NEXT returns control to NEXT-OBJ, execution will resume at (GO 'LOOP).

6. Conclusions

Either SAIL or LISP could provide an excellent basis for real-time planning and execution control of a large automated shop. However, each language possesses features which facilitate certain types of operations. In particular, SAIL is generically better at the low level control of I/O devices, and has more extensive abilities for interacting with the operating system (especially where file manipulations are concerned). LISP, on the other hand, is more flexible at the higher planning levels and where system development and debugging are concerned.

We envision an "ideal" system as one which merges all the desirable features of these two language classes. Such a merger would incorporate LISP's program and data structure format, augmented where necessary to accommodate SAIL-like file operations, and possibly LEAP. SAIL features would be implanted in this environment, and, at the implementor's discretion, an ALGOL-like syntax (such as MLISP) could be grafted onto the front of the system to make it more tractable.

In addition, such a merger should take care to preserve the following desirable features of SAIL and LISP:

(1) Data structures should accommodate complex symbolic information as well as primitive types. As in LISP, data structures should be free to grow in unrestricted ways, and storage declarations should be optional to the user.

(2) Program and data should, as in LISP, be in the same format. Such a representation underlies (a) a strong macro facility, (b) rapid editing, modification and debugging of programs, and (c) self-modifying and self-extending systems. The last capability, for example, enables the system, given the description of a new type of tool, automatically to synthesize the programs for controlling the tool from a library of sub-functions.

(3) Strong I/O and file manipulation facilities, as are found in SAIL, must be included. A good random-access file system is imperative for even moderately large databases. The system should have both high and low level control over input and output formatting which provides control down to the bit level of the machine.

(4) A highly-developed interrupt subsystem would be desirable. With the merger of SAIL's bit-wise interrupt control, and LISP's symbolic capabilities, such a system as is described in [Rieger 76] could be efficiently implemented. This would serve as the network protocol for a large collection of highly autonomous processes where the synthesis and control of many parallel events is important.

(5) For software development and debugging, an interpreter should exist for the language. Nevertheless, the language should be have a compiler for production usage. LISP currently satisfies these requirements.

(6) The system should provide for a large, context-sensitive, associative database. This would involve some new engineering to coordinate a MP-like database with an efficient random-access file system. [McDermott75a] surveys some ideas on this topic.

(7) There should be some degree of automatic problem-solving control which includes a CONNIVER-like context-switching and process-suspending mechanism. Accommodations should be made for SAIL-like parallel process control, and emphasis should be placed on inter-process communications protocols. Most of the ideas already exist in CONNIVER and SAIL, but they need to be synthesized into a unified system.

7. Bibliography

[Baumgart72] Baumgart, B. G. "Micro-Planner Alternate Reference Manual," Stanford AI Lab Operating Note No. 67, April 1972.

[BBNEXEC] Bolt, Beranek and Newman. "TENEX Executive Manual," Cambridge, Massachusetts, April 1973.

[Beech70] Beech, D. "A Structured View of PL/1," ACM Computing Surveys, March 1970, pp. 33-64.

[Bobrow74] Bobrow, D. G. and Raphael, B. "New Programming Languages for Artificial Intelligence," ACM Computing Surveys, September 1974, pp. 153-174.

[Burstall71] Burstall, R. M., Collins, J. S. and Popplestone, R. J. Programming in POP-2, The Round Table and Edinburgh University Press, 1971.

[Church41] Church, A. The Calculi of Lambda Conversion, Princeton University Press, Princeton, New Jersey, 1941.

[COBOL74] COBOL. "American National Standard Programming Language COBOL," X3.32 - 1974, American National Standards Institute, Inc., New York, 1974.

[CODASYL71] CODASYL Data Base Task Group. "April 1971 Report," ACM, New York, 1971.

[DEC] DEC. "DEC System-10 Data Base Management System Programmer's Procedures Manual," Document DEC-10-APPMA-B-D, Maynard, Massachusetts.

[Dijkstra75] Dijkstra, E.W.D., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M. "On-the-fly Garbage Collection: An Exercise in Cooperation," Burroughs, Plataanstraat 5, NL-4565 NUENEN, The Netherlands, EWD496-0.

[Fahlman73] Fahlman, S. "A Planning System for Robot Construction Tasks," MIT AI Memo 283, 1973.

[Feldman69] Feldman, J. A. and Rovner, P. D. "An ALGOL-Based Associative Language," Communications of the ACM, August 1969, pp. 439-449.

[Feldman71] Feldman, J. A. and Sproull, R. F. "System Support for the Stanford Hand-Eye System," Second International Joint Conference on Artificial Intelligence, London, September 1-3, 1971.

[Finkel74] Finkel, R., Taylor, R., Bolles, R., Paul, R. and Feldman, J. "AL, A Programming System for Automation," Stanford Artificial Intelligence Laboratory, Memo AIM-243, November 1974.

[Hewitt69] Hewitt, C., "PLANNER: A Language for Proving Theorems in Robots," Proc. IJCAI-1, 1969

[Leslie72] Leslie, W.H.P. (editor). Numerical Control Programming Languages, North-Holland Publishing Company, London, 1972.

[Levin65] Levin, M.I. "LISP 1.5 Programmer's Manual," The M.I.T. Press, Cambridge, Massachusetts, 1965.

[McCarthy60] McCarthy, J. "Recursive Functions of Symbolic Expressions and their Computation by Machine," Communications of the ACM, April 1960, pp. 184-195.

[McDermott72] McDermott, D. V. and Sussman, G. J. "The Conniver Reference Manual," AI Memo No. 259, MIT Project MAC, May 1972.

[Moon74] Moon, D.A. "MACLISP Reference Manual," Project MAC - Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

[Naur60] Naur, P. (Editor). "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, May 1960, pp. 299-314.

[Norman69] Norman, E. "LISP," University of Wisconsin Computing Center, Madison, Wisconsin, April 1969.

[Parsons74] Parsons, F. G., Dale, A. G. and Yurkanan, C. V. "Data Manipulation Language Requirements for Database Management Systems," Computer Journal, May 1974, pp. 99-103.

[RAPIDATA] RAPIDATA Corporation. "A FORTRAN DML Implementation for DBMS-10," Fairfield, New Jersey.

[Reiser75] Reiser, J. F. "BAIL--A debugger for SAIL," Stanford Artificial Intelligence Laboratory, Memo AIM-270, October 1975.

[Reiser76] Reiser, J. F. (Editor). "SAIL," Stanford Artificial Intelligence Laboratory, Memo AIM-289, August 1976.

[Rieger76] Rieger, C.J. "Spontaneous Computation in Cognitive Models," Department of Computer Science, University of Maryland, TR-459, July 1976.

[Samet76] Samet, H. "The SAIL Data Base Management System," Computer Science Department, University of Maryland, College Park, Maryland, Unpublished, 1976.

[Siklossy76] Siklossy, L. Let's Talk LISP, Prentice-Hall, Inc., 1976.

[Smith70] Smith, D. C. "MLISP," Stanford Artificial Intelligence Project, Memo AIM-135, 1970.

[Stacey74] Stacey, G. M. "A FORTRAN Interface to the CODASYL Database Task Group Specifications," Computer Journal, May 1974, pp. 124-129.

[Sussman72] Sussman, G., Winograd, T., and Charniak, E., "MICROPLANNER Reference Manual," M.I.T. AI-TR-203a, 1971

[Taylor76] Taylor, R. W. and Frank, R. L. "CODASYL Data-Base Management Systems," ACM Computing Surveys, March 1976, pp. 67-105.

[Teitelman74] Teitelman, W. "INTERLISP Reference Manual," XEROX Palo Alto Research Center, Palo Alto, California, 1974.

[TOPS10] DEC. "DECSYSTEM-10 Operating Systems Command Manual," DEC-10-OSCMA-A-D, Digital Equipment Corporation, Maynard, Massachusetts, May 1974

[Weissman67] Weissman, C. "LISP 1.5 Primer," Dickinson Publishing Company, 1967.

[Wilcox76] Wilcox, C. R. "MAINSAIL Language Manual," SUMEX, Stanford University, May 1976.

8. Summary Chart

	Flow of Control	Data Types	Declarations	Scope	I/O Facilities	Development Aids	Linkage to Other Processors	General Comments
LISP	sequential, labels	S-expressions (atoms of type real, arbitrary precision integer, string, symbol, array)	implicit for globals, explicit for locals, typeless	dynamic	frequently meager file interaction, [illegible]	structure editor, run-time macros, breakpoints	possible (done at interpreter level)	extremely consistent, flexible, general purpose, usually interpreted (can be compiled for efficiency), program and data are indistinguishable
PLANNER	pattern-directed, sequential, labels, backtracking	S-expressions (same as LISP)	typeless	dynamic	paltry	similar to LISP including monitoring of GOALS and data base activity	same as LISP	written in LISP, non-procedural, special purpose, interpreted, data base contains arbitrary n-tuples
CONNIVER	pattern-directed, sequential, suspended processes, labels, frames	S-expressions (same as LISP)	typeless	dynamic, frame	paltry	similar to LISP including a frame debugger	same as LISP	written in LISP, non-procedural, complex control and data structures, special purpose, interpreted
SAIL	processes, coroutines, sequential, FOREACH, iteration, backtracking, labels	integer, real, string, array, structure, pointer, list, set, itemvar, context, procedure, item	statically typed declarations except for run-time items	static, block-structured	excellent - from low level to high level	BAID, BAIL, macros, conditional compilation, REQUIRE feature	easy, can use libraries, in-line assembly language statements	compiled, general purpose, extensive run-time library, associative data base consisting of triples, procedural, backtracking, network data base interface, compile-time facility, control of external devices, interprogram communication
POP-2	sequential, labels, iteration	real, integer, string, symbol, list, record, array, pair	explicit, typeless	dynamic	similar to LISP but has file I/O	similar to LISP	same as LISP	interpreted, not as consistent as LISP, general purpose, stack based, capability of returning more than one result (without using a list as in LISP), data and program are almost identical
QLISP	sequential, labels, iteration	S-expressions (same as LISP)	typeless	dynamic	same as LISP	same as LISP but user must debug in LISP and not in QLISP	same as LISP	translates into LISP, ALGOL-like syntax, readable

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Artificial Intelligence Programming Languages for Computer Aided Manufacturing		5. TYPE OF REPORT & PERIOD COVERED (9) Technical rept.,
		6. PERFORMING ORG. REPORT NUMBER (14) TR-595
7. AUTHOR(s) (10) Chuck Rieger, Hanan Samet, Jonathan Rosenberg		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-76C-0477
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Dept. Univ. of Maryland College Park, Md. 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Informations Systems Branch Office of Naval Research Wash., D.C. 20305		12. REPORT DATE (11) Sept 1977
		13. NUMBER OF PAGES 47 (12) 50p.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence
Programming Languages
Computer Aided Manufacturing
Systems Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Eight Artificial Intelligence programming languages (SAIL, LISP, MICROPLANNER, CONNIVER, MLISP, POP-2, AL and QLISP) are presented and surveyed, with examples of their use in an automated shop environment. Control structures are compared, and distinctive features of each language are highlighted. A simple programming task is used to illustrate programs in SAIL, LISP, MICROPLANNER and CONNIVER. The report assumes reader knowledge of programming concepts, but not necessarily of the languages surveyed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 409022

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Off of Naval Research
Branch Office, Boston
495 Summer St.
Boston, Mass. 02210

New York Area Office
715 Broadway-5th Floor
New York, N.Y. 10003

Mr. E. H. Gleissner
Naval Ship R+D Center
Computation and Math Department
Code 18
Bethesda, Maryland 20084

Capt. Grace M. Hopper
NAICOM/MIS Planning Branch
OP-916D
Off, Chf. of Naval Op.
Washington, D.C. 20350

Mr. Kin B. Thompson
Technical Director
Information Systems Div. OP-91T
Off., Chf. of Naval Op.
Washington, D.C. 20375

Naval Research Lab.
Technical Info. Division
Code 2627
Washington, D.C. 20375

Dr. A.L. Slafkosky
Scientific Advisor
Commandant, USMC
Code RD-1
Washington, D.C. 20380

National Security Agcy.
Attn: Dr. Maar
Fort Meade, Maryland 20755

Off. of Naval Research
Code 1021P
Arlington, Va. 22217

Asst. Chief for Tech.
ONR Dept. of Navy
Code 200
Arlington, Va. 22217

Off. of Naval Research
Information Sys. Program
Code 437
Arlington, Va. 22217

Off. of Naval Research
Code 455
Arlington, Va. 22217

Off. of Naval Research
Code 458
Arlington, Va. 22217

Defense Documentn. Cent.
Cameron Station
Alexandria, Va. 22314

Off. of Naval Research
Branch Office, Chicago
536 South Clark St.
Chicago, Ill. 60605

Off. of Naval Research
Branch Off., Pasadena
1030 East Green St.
Pasadena, Calif. 91106

Naval Electron. Lab. Ctr.
Adv. Software Tech. Div.
Code 5200
San Diego, Calif. 92152

DATE
ILME