



STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

August 1973

NEW PROGRAMMING LANGUAGES FOR AI RESEARCH

By

Daniel G. Bobrow
Xerox Palo Alto Research Center
Palo Alto, California 94304

Bertram Raphael
Stanford Research Institute
Menlo Park, California 94025

Paper presented at Third International Joint Conference
on Artificial Intelligence, Stanford University, Stanford,
California, August 20-23, 1974.

Artificial Intelligence Center

Technical Note 82

I INTRODUCTION

Most programming languages are universal in the sense that any algorithm that can be expressed by a program in one language can also be expressed in any of the other languages. However, the set of unique facilities provided by a language makes some types of programs easier to write in that language than in any other. Indeed, the main reason for introducing new features into a programming language is to automate procedures that the user needs and would otherwise have to code explicitly; such features reduce the housekeeping details that distract the user from the algorithms in which he is really interested. Therefore, underlying the design of any programming language is a set of assumptions about the types of programs that users of that language will be writing.

Historically the needs of the artificial intelligence (AI) research community have stimulated new developments in programming systems. The first high-level list-processing primitives were developed by Gelernter for a geometry theorem prover (Gelernter 1959); the first general string-manipulation system was developed by Yngve for computational linguistics research; the first wide uses of conditional expressions and recursion were at least partly due to John McCarthy's AI interests.

For more than a decade, the list processing and symbol manipulation languages--such as COMIT, IPL, LISP, SLIP (Bobrow 1964)--have been the basis for almost all AI achievements. Although the effectiveness of research with these languages has improved dramatically due primarily to greatly expanded memory sizes and new interactive debugging facilities, the languages themselves have remained remarkably stable. In recent years,

however, new directions for emphasis in AI research--such as studies of representation of knowledge, robotics, and automatic programming--have led to a widely felt need for certain rather novel features to be embedded into programming languages; and some languages containing several of these features have recently been implemented. The purpose of this paper is to give an overview of the nature of these new programming features and the present state of their availability in the new languages.

II LANGUAGES COVERED

The languages to be discussed in this paper are generally in an early stage of development: they are inadequately documented; neither the designs nor the implementations are fully debugged; they have been used at most for only a few significant application programs; and they are so dependent upon local operating systems and environments that they are extremely difficult to export. However, the common threads of new ideas running through these languages appear so basic and useful that some of the languages have already received widespread publicity, and once the ideas stabilize, the successors to these systems are likely to provide the basic tools for AI research for years to come.

In this paper we shall not attempt to identify and describe specific, completely-defined languages, because such descriptions would rapidly become obsolete in view of today's level of activity in the system design area. Instead, we shall devote the next section (Section III) of the paper to discussing the new features present in many of the emerging systems. Section IV will then compare how these features are being handled in each of four evolving families of languages being developed at different major AI research centers:

- (1) SAIL, a language developed at the Stanford AI Project (Swinehart 1971);
- (2) PLANNER and CONNIVER, systems being developed at the MIT AI laboratory (Sussman 1970; Baumgart 1972; McDermott 1972);

- (3) the QLISP language being developed as an extension of INTERLISP (formerly called BBN-LISP) which is a joint continuing product of Xerox PARC and BBN (Reboh 1973; Teitelman 1973);
- (4) the POPLER extension of POP-2 from the University of Edinburgh (Davies 1973).

These four sets of languages each have long, independent histories.

SAIL is a marriage of LEAP (Feldman 1969), an associative retrieval formalism, and a version of ALGOL 60. It has been in use at Stanford since 1969. Recent improvements, stimulated by the needs of AI researchers, have been primarily focused on adding more powerful and flexible control mechanisms.

The PLANNER concept was developed by Hewitt at MIT starting in 1967 (Hewitt 1971, 1972), and Sussman and Winograd built a first implementation, MICRO-PLANNER, which contained a subset of PLANNER features. These projects established the basis of the currently popular concept of procedural representation of knowledge. CONNIVER is a recent attempt by Sussman at MIT to remedy some observed shortcomings in the practical use of PLANNER, while preserving its good ideas.

QLISP was a successor to QA4, which was developed primarily by Rulifson (Rulifson 1968, 1972) at SRI. These languages evolved from years of question-answering and theorem-proving research. Although strongly influenced by PLANNER, QA4 was intended to be a more uniform and complete formalism. QLISP is a current attempt to make QA4 features more accessible by merging them into an established, widely-available LISP system INTERLISP. INTERLISP has provided a general

control structure framework at the systems level along with many user interaction facilities.

POPLER is largely based on the ideas in PLANNER, but is implemented and embedded in POP-2. POP-2 provides a compiler oriented stack machine with an extensible data type facility. Although POPLER1.5 has been available only since spring of 1973, POP-2 has been in use for AI research at the University of Edinburgh for a number of years.

III SPECIAL FEATURES COMMON TO THE NEW LANGUAGES

The following paragraphs discuss some of the special features of the new languages and why they are desirable. These special features deal with:

- Data types and memory management
- Control structures, including pseudo-parallelism, conditional interrupts ("demons"), alternative contexts, and backtracking
- Pattern matching, used for both data retrieval and program control
- Automatic deductive mechanisms

In order to present the principal features of these languages in a reasonable amount of space, the following discussion is necessarily oversimplified. Many features of each language will not be discussed, and we may take some liberty with syntax to make short examples readable out of context. The reader should refer to the appropriate reference manuals for more accurate and complete presentations of the ideas outlined below.

A. Data Types

The earliest programming languages permitted the user to manipulate only numbers, either as scalars or arrays. The major contribution of the symbol manipulation languages was the introduction of symbolic data types, such as lists, trees, and strings. A few languages, including SNOBOL4 (Griswold 1968), permit the user to define additional types of data structures; but such data extension facilities are not widely used, probably because the user then has the burden of providing all the basic operators needed to work with his new data type, and even if he does he

will wind up with a unique program that is unusually difficult for others to read and understand.

Lists, trees, and strings were adequate building blocks as long as AI researchers were groping for representations and algorithms to handle idealized "toy" problems. Recently, as the emphasis has shifted to larger, more complex, more realistic problem domains, a greater variety and richness in data types has become desirable. In particular, in addition to lists, trees, and strings, one would like to be able to use content-retrievable ordered triples (or n-tuples), unordered sets, and formal statements of a logical formalism or a programming language, as basic data types. These are operationally different types, but still basically problem independent, e.g., an unordered set, not a personnel record. For each data type, the programming language should provide a set of operations or functions needed to create items of that type, to perform basic manipulations on that type (e.g., LISP's car, cdr, and cons for binary trees, union and intersection for sets, etc.), and when appropriate to transform one type into another. All of the new languages provide important new data types and a framework for their use.

The symbol-manipulation languages, like most other programming languages, left the programmer with full responsibility for creating, indexing, and accessing any data files he wished to use. Recently, AI researchers have recognized the need for large, relatively permanent information files that must be maintained and accessed in an efficient manner. In some cases these files must be divided into sections that are each available only to certain programs under certain conditions. The new

languages provide built-in, automatic mechanisms for handling such data files in a convenient way.

B. Control Structures

In the earlier AI languages the flow of control among procedures or functions, the primitive units of program that we shall here call access modules, was strictly hierarchical. An access module is any unit in which is introduced new bindings, that is, associations between variable names and values. In hierarchical control, every module was expected to complete its work (perhaps calling other modules) and then return control to its parent, the module that activated it. Recursive control was permitted, i.e., a module could call itself or one of its ancestors in the control hierarchy as a subroutine; however, each such recursive call to a module could be thought of as creating a separate instance or activation of the module to be used at a new level so that the strict hierarchy was maintained. Moreover, once an activation had been exited, it disappeared and could not be "continued" again. The bindings in that module were lost. Reinitiating a module caused a new activation of the module to be created and run from its beginning.

A generalization in some LISP systems was the so-called "funarg" mechanism, which allowed a set of variables and their current values (bindings) to be passed from one part of the control tree to another independent of the continued existence of the defining context. This feature allowed use of free variables in the definition of a functional argument which would not conflict with use of the same variable in the program that called the functional argument, and also preservation of

variable values between calls. However, the basic relation between access modules was still strictly hierarchical. Special purpose coroutines existed in the IPL-V "generator" feature, which allowed reentry to certain predefined routines, usually for scanning data bases incrementally.

Much more flexible control structure is a major contribution of the newer AI languages. The basic control innovation that is now being made available is the ability to save a module and its context in a state of suspended animation. An active access module can relinquish control not only by returning control to its parent and vanishing, as in a hierarchy, but also by giving control to another module that is in such a suspended state. The suspended module is poised to continue execution from right where it left off. The "resumer" can save his own state, but no other module is obligated to "return" control to such a suspended module in order to complete a computation. Thus in addition to moving up and down hierarchical trees by initiating and terminating execution of access modules, flow of control may now also wander among the access modules by suspending and resuming their executions in any order, unconstrained by the tree structure of their inherent control relationships. Each activation will have a unique caller, i.e., the module that initiated (and thereby created) it, but it may also be reached from (and transfer control to) any number of other modules.

In order to make this flexible control flow possible, the implementations of the new languages must provide for appropriate bookkeeping. Bobrow and Wegbreit (1973b) define a general model for control that has been used as the basis for such implementations including those in three of the languages described below (CONNIVER, INTERLISP and

POPLER). This model defines the set of information or frame that must be associated with every activation of an access module, to make possible its suspension and reactivation in a meaningful way. This information includes:

- (1) A binding link that specifies where to find values (bindings) of variables local to this activation (such as LISP prog and lambda variables).
- (2) An access link that specifies the environment in which to find values for free variables (not specified in the local environment, and found in LISP, for example, by tracing up the normal hierarchical chain of control).
- (3) A control link that specifies which module activation is to continue processing if the current module terminates at a "normal" exit (such as a conventional return statement).
- (4) The process state in the module, which specifies where and how to continue a previously suspended operation. This includes current temporaries, and the current "program counter".

A point to note about a frame of an access module is that it has in the frame itself all the information necessary to continue running the activation, e.g., the continuation point and values of temporary variables of a module at the time it calls another access module are stored in the caller. Because independent returns to a frame may require distinct continuation points and temporary storage, a separate copy of this part of the frame must be made for each independent successor, although the bindings need not be copied.

The control structure induced by this model is a tree of modules, with control passing among any of the suspended modules in the structure. If only one process is active at a time, we call it a coroutine regime. If processing can be thought of as going on simultaneously in several modules, we call it a multiprocessing regime. Multiprocessing is usually done by scheduling through time-quantum interrupts at the system level, or time allocation in the language interpreter. It can also be effectively achieved in a coroutine regime by cooperation, e.g., by having each active process frequently resume an executive module to request further resource allocation.

Backtracking is a special coroutine regime in which instances of modules are saved at decision points, and restored in a last-in-first-out sequence when subsequent modules "fail" (a special form of termination). In addition to restoring the control environment (automatic on resuming processes), some alternate philosophies have developed on restoration of the data state--sometimes called the data context--which existed at the time the decision point was saved (both for variable bindings and general data bases). Some systems normally "undo" all data changes (e.g., MICRO-PLANNER) some only bindings (e.g., LISP70 (Tessler 1973)), and some provide a programmable facility for undoing any specified changes (Teitelman 1969). Backtracking as a search mechanism follows a strictly depth-first search on a decision tree, and this formerly popular control mechanism has been criticized (Sussman 1972) for its inefficiency in many situations.

Another addition to the AI jargon in the control domain is the "demon", which is a module that is activated when certain conditions become true. Demons are usually implemented by having data-accessing functions check for

"sensitive data", evaluating the current monitoring condition involving the sensitive data "touched", and having those functions transfer control to the demon module when appropriate.

C. Pattern Matching

Pattern matching was first used extensively in the string manipulation languages (COMIT, SNOBOL), in order to permit substrings to be identified by their contents rather than by their addresses. This kind of data specification is extremely useful in current AI applications that frequently require large symbolic data stores, so some form of pattern-directed data retrieval has been included in all the new languages.

Basically, pattern-matching facilities allow the comparison of a given template with a set of data items, where the template may have a number of variables. If the template matches one of the data items, a side effect of the match is the setting of values to these variables. A template may match more than one item, and it may match an item in more than one way, so some pattern-matching facilities allow the user to specify whether he wishes all possible matches to be found at once, or one match at a time (on request) after each preceding one is processed.

In addition to searching through a data base, pattern matching may be used as part of the control mechanism to select the next subroutine. Here each subroutine contains a template as part of its definition, and the subroutine can be executed only if its template matches its actual argument. A common use for this mechanism is in goal-directed problem solving systems. Suppose each subroutine is capable of producing as its output a certain, unique form of data structure; then a template describing

that form is included in the definition of the subroutine. Now when the top-level program wishes to achieve a "goal"--e.g., produce a certain data structure--it need merely call for the execution of any subroutine that matches the goal (without knowing which particular subroutine will step forth). Such use of pattern-directed function invocation (originated in PLANNER (Hewitt 1971)) permits "knowledge" to be distributed throughout the programs of a complex system, permitting more flexible modification and growth of AI programs than would be allowed by the more conventional top-down hierarchical control.

D. Deductive Mechanisms

The extent to which a programmer may specify what he wants accomplished without detailing how it is to be done is one way of defining the "level" or "power" of a programming language. For example, the lowest level computer languages, assembly codes, require an explicit statement for each wired-in instruction to be executed. Algebraic languages, such as FORTRAN, permit the user to describe a desired result by a combination of mathematical relations, e.g., $x=a+b-c$, and leave to the compiler decisions about the order in which elementary commands are performed. LISP programs may be recursive, in which case the system has added responsibility for stack manipulation--and possibly translating the recursion into an iteration.

The new languages go a step further. They permit the programming system to carry out certain activities, including modifying the data base and deciding which subroutines to run next, using only constraints and guidelines the programmer sets up for each programmed activity. For

example, the programmer can request a result, and the procedures which "match" the request will be tried by the system using a problem solving mechanism working within the pre-established guidelines. The process of constructing a problem-solving program then becomes a matter of developing and modifying guidelines, and specifying matching criteria, rather than developing procedural algorithms. We call the semi-automatic search and data-construction features of these languages deductive because they bear some resemblance to so-called "theorem-proving programs" that attempt to deduce desired logical expressions (theorems) from previously-specified expressions (axioms). The mechanisms built into these languages allow expression of many strategies for proving different types of theorems, and/or solving complex problems.

IV DIFFERENCES BETWEEN THE NEW LANGUAGES

A. Data Types and Storage Mechanisms

In this section we describe the novel data types of the various languages and how they are formed, stored, accessed, and manipulated. Of course, these languages also have the usual arithmetic data types and individual variable and array storage, which are generally unexceptional and need no further discussion here.

1. SAIL. SAIL is an ALGOL-like language which contains a symbolic data system based upon an associative storage mechanism (originally called LEAP (Feldman 1969)). The basic element of this system is the item, a unique data element which may be named by an identifier and referenced by an item-variable. Two kinds of structures may be formed from items:

- Sets of items
- Associations, which are ordered triples of items.

An association of three items is usually denoted

$$\text{item1} \boxtimes \text{item2} = \text{item3}$$

where item1, item2, and item3 are called (appropriately enough) the first, second, and third elements of the association, respectively. They are also occasionally called the attribute, object, and value of the association in suggestion of the semantics usually given associations, e.g.

$$\text{COLOR} \boxtimes \text{APPLE} = \text{RED}$$

is a typical association.

An association may itself be designated to be an item included as an element of higher-order associations. However, the use of such nested associations appears to be somewhat awkward.

Appropriate functions exist for creating and deleting associations, and for inserting and removing items from sets. In addition, the FOREACH statement conveniently specifies iteration through sets, e.g.

FOREACH X SUCH THAT X IN set DO statement
causes statement to be executed repeatedly for X bound to each element of set in turn.

The most important feature of associations is that they are automatically stored in a permanent data structure that may be accessed in an associative manner. In particular, the set referenced in a FOREACH statement can be implicitly defined by referencing the association store; e.g.

FOREACH X SUCH THAT COLOR \bowtie X = RED DO statement
will cause all known red things to be processed by statement, even though no such explicit set had been created. FOREACH statements also allow looping through corresponding pairs, triples... In addition, the special symbol ANY can be used as a "don't care" comparator; thus the specification COLOR \bowtie X = ANY defines the set of all objects X that have any color attribute in the association memory.

The sets and associations in SAIL provide a convenient, efficient mechanism for accessing certain symbolic data structures. However, sets and triples are not necessarily as convenient as list structures for many applications. For this reason a new data type, list, has recently been added to SAIL. It is not yet clear how cleanly lists can be merged into

the system; it will be interesting to observe whether major SAIL users switch from triples to lists as their primary data representation. A major problem with these lists is that the only legal elements of these lists are items.

2. PLANNER/CONNIVER. Full PLANNER (as defined by (Hewitt 1972)) is an evolving theoretical framework of programming languages, and therefore is not an appropriate candidate for this review of existing systems, though a number of its ideas will be discussed. The PLANNER system described here is the MICRO-PLANNER implementation defined in (Sussman 1970).

MICRO-PLANNER and CONNIVER are implemented in LISP and allow access to LISP constructs. Thus all LISP data structures are available when needed-- although care must be taken to distinguish between LISP and PLANNER values of similar variables.

PLANNER makes available to the programmer two semantically different types of data items: assertions and theorems. An assertion is an ordered n-tuple represented by a LISP list of non-numeric objects called items. The range of possible formats for assertions is wide, except that by convention an assertion usually represents a true fact. For example, the following are possible assertions:

(SMELLY GARLIC)

(SMALL-PRIMES (1 2 3 5 7 11))

(COLOR APPLE RED)

(THIS IS AN ASSERTION)

Assertions may be created, erased, and associatively accessed in a manner similar to SAIL associations.

Theorems in PLANNER are the procedures or subroutines of the language. Each theorem contains as part of its definition information about when it should be invoked and what kind of effect it is expected to have upon the data base of assertions. Thus theorems are procedural, describing actions to be carried out, as opposed to the declarative assertions. However, theorems as well as assertions may be dynamically created or modified by running programs (e.g., by the execution of other theorems).

An explicit data-base context mechanism is an important independent CONNIVER and QA4 innovation. Instead of a single global base to which assertions (items) may be added or removed, CONNIVER has a tree of such data bases. Any modification of the data base occurs only in that data associated with the current node of the context tree, and data contexts may easily be changed under program control. For example, a game situation can conveniently be represented by a context tree of board position where the highest context is the initial position and each lower context is derived from its parent by making just the changes required for the current move. This structure permits consideration of alternative situations merely by switching contexts, rather than making extensive changes to the data base. (Of course, at some point these changes must actually be performed, or else the data base must be multiply generated and stored; but CONNIVER transfers the responsibility for this bookkeeping from the programmer to the implementation.)

3. QLISP/INTERLISP. QLISP has attempted to provide a full range of data types, in order to give the programmer considerable freedom in choosing a representation for his problem domain. The major data types are: tuple, class, bag and vector. Built-in functions provide for forming, combining and testing data of the different types. All QLISP data structures are stored in a permanent data base called the discrimination net. Insertion into the net converts any element into a canonical form so that every possible expression has a unique representation and location in the net. Because expressions in the net have unique locations, they can be given permanent property lists just like LISP atoms. These properties are used in the deductive process described later.

Tuples (short for "n-tuple") and vectors are simply LISP lists with any number of elements, but are made unique by the discrimination net search. A tuple and vector differ only in their evaluation rules.

A class is viewed as unordered, and repeated elements are ignored; e.g., the following are equivalent:

```
(CLASS ONION (TUPLE MILK EGGS) POTATO ONION)
```

```
(CLASS (TUPLE MILK EGGS) ONION POTATO (TUPLE MILK EGGS) )
```

Internally, classes are put into lexicographic order with duplications removed. A bag is an unordered tuple, or equivalently, a set that may have repeated elements; e.g., (BAG A B B) is equivalent to (BAG B A B) but different from (BAG A B). Bags are particularly useful when known as the argument of certain operators. For example, if we say that the argument of PLUS is a bag, then we need not assert that PLUS is associative and symmetric, because the system already knows this as properties of bags.

4. POPLER. The data type mechanism of POP-2 allows creation of new data structures with fields of arbitrary size and interpretation. It provides a straightforward way of defining the constructor, accessor and updater functions of any new type, and automatic storage management of the data elements. For example we can define a record of type "person" containing three components, a list item (of indicated width \emptyset for full pointer size), a 7 bit field for age, and a 1 bit field for sex:

```
recordfns("person", [0 7 1]) →sex →age →name →unpackper →makeper;
```

Note that this defines and names five new functions for creating (makeper) and accessing records of this type.

The POPLER extension of POP-2 also provides an "associative" memory for items of certain kinds modelled on CONNIVER. The data base is interrogated by using a retrieval pattern, and items which match (instantiate) that pattern are retrieved. "List constants" and "procedures" (corresponding to PLANNER assertions and theorems) are stored in separate data bases. Within each data base, items are stored in a particular context. A context is a tree-like data structure used to control the scope within which any item is present. As in CONNIVER, this permits consideration of alternative situations merely by switching contexts rather than changing a global data base. In order to facilitate access to items, POPLER provides an index to all items asserted in any state. This ensures one unique copy is used by all the states. The use of a global index introduces one problem in garbage collection. Some states may become inaccessible, and therefore so should some items; but the index still holds a pointer to the items. POPLER allows the user to get around this problem somewhat by specifically "unindexing" items.

B. Control Structures

1. SAIL. As a compiler based system, SAIL (Feldman 1972) insists that points in the program at which branches in the control structure can take place be known at compile time. Thus the flow of control for the complete computation is constrained to stay within a branching structure defined in advance; e.g., no process can decide during execution to create a process that branches from above itself in the hierarchical control chain. Only the current control context can be split.

A new process is created by the command

SPROUT (item, procedure)

where item names the new process for future reference, and procedure specifies the program that the new process is to execute. A SPROUTed process is assumed to begin running immediately and runs in parallel with the process that contained the SPROUT instruction. Since true parallelism is not possible on a single-processor computer, the SAIL runtime system includes a scheduler that supervises the multiprocessing regime by deciding which process is to be executed at a given instant. Processes can contain instructions to suspend or terminate themselves or other processes, and can specify priorities and time quanta for running. A coroutine regime is facilitated by a RESUME construct which suspends the current process and reactivates a named suspended process. Multiprocess time coordination is done by JOIN(set) which suspends the process calling JOIN until all processes in set have terminated. Backtracking within a process is possible, but the programmer must explicitly identify places to which backtracking is possible, and then specify which variables are to have

their values restored. No mechanism is available for one process to evaluate a particular expression in the binding context of another process.

Additional communication among processes is facilitated by a kind of mail or "notice" service based upon a message queuing system. Any process may place messages for other processes into various queues; processes may themselves be placed on other queues, while waiting for an appropriate message. One of the jobs of the multi-processing scheduler is to "deliver the mail" whenever possible. Demons can be implemented by means of this message-queuing mechanism, although no special ones are standardly available.

2. PLANNER/CONNIVER. Both MICRO-PLANNER and CONNIVER are implemented as interpretive languages written in a LISP system which permits only a standard recursive call control structure (plus the possibility of machine-language subroutines); therefore, the flexible control features of the new languages must be designed into their interpreters and are not available to LISP functions. These features essentially provide modules with the binding, access, control, and state-saving information required by the general control model described earlier (Sec. IIIB).

In MICRO-PLANNER, the general control capabilities are not directly available to the user; the system provides the user with establishment of backtrack points (by pattern-matching or GOAL statements), and the automatic restoration of appropriate contexts whenever backtracking takes place. Furthermore, backtracking can be invoked either explicitly (by a FAIL statement), or spontaneously when a process runs out of things to do.

In CONNIVER, every time a non-atomic expression is evaluated by the interpreter, a new frame is created. In addition, all the components of every frame--bound variables, access link, control link, and so on--are available to the programmer. Thus, by explicitly referencing and modifying frames a CONNIVER programmer may create any kind of control regime he wishes.

The FRAME command produces a pointer to the current frame, so that it may be modified or executed again later. The TAG command is similar to FRAME, but also specifies a starting location within the frame rather than the entire frame. For example, after execution of the CONNIVER version of the program

```
(PROG (X)
      (SETQ X 50)
      (SETQ G2 (TAG A))
      (SETQ G1 (FRAME))
      (PRINT 'FOO)
      A (PRINT 'FIE) )
```

the global variables G1 and G2 would both point to a frame within this prog. The subsequent command (CONTINUE G1) would start right after the (SETQ G1(FRAME)) and print both FOO and FIE; (CONTINUE G2) restarts at A and would print FIE only.

Frames may also be used to specify a binding context. The CEVAL command requests evaluation relative to a specified context. Thus after the above PROG had been run, the following code:

```
.
.
.
(SETQ X 17)
(PRINT (CEVAL 'X G1))
.
.
.
```

would print 50.

In addition to the basic frame-manipulation commands, CONNIVER also offers the programmer some higher-level control constructs. For example, the AU-REVOIR command for exiting from a function first constructs a continuation point using FRAME so that the exited function may be resumed at a future time. This facilitates the construction of co-routines. Additional special functions and message-passing conventions are provided for facilitating use of generators, a special kind of co-routine process which can generate requested data items one or a few at a time, and when more are requested, resume processing in their original contexts. The IF-ADDED and IF-REMOVED processes are demons which are activated when items are added to or removed from the data base. Although the backtracking regime of PLANNER would be easy to program in CONNIVER, it would conflict with a major motivation of CONNIVER (Sussman 1972) to allow the flexibility of multiple processes with possibly independent data base contexts. Decision points and failure mechanisms are not provided in the basic systems because it was felt that this encouraged poor programming practices.

3. INTERLISP/QLISP. QLISP is primarily a subroutine package, plus some syntactic extensions embedded in INTERLISP (Teitelman 1973). Unlike CONNIVER with MIT LISP and POPLER with POP-2, QLISP and INTERLISP procedures operate in the same control world. Since QLISP has no separate interpreter, its control structure is completely dependent upon that of INTERLISP--which, at the time of this writing, is simply the recursive call structure of LISP. However, a general frame-oriented control structure is

currently being implemented in INTERLISP using a "spaghetti stack" technique (Bobrow & Wegbreit 1973a) which has the property that for ordinary recursive function calls it costs very little more than the usual stack storage allocation mechanism. This system is almost operational and will be combined with QLISP in the near future.

The INTERLISP frame will contain the usual binding, access, and control links and a continuation point (current state), as described earlier, plus some other fields for additional features. Functions will exist that enable the programmer to locate existing frames by name or by following along access or control chains, creating a new process using any existing frame as above, and constructing arbitrary control structure trees of new frames. Multiprocessing is done by explicit passing of control among processes, or to a user-programmed scheduler.

An extremely general relative evaluation function will permit independent specification of both access and control environments before evaluating a specified expression. The effects of both the CONTINUE and the CEVAL commands of CONNIVER and the relative stack evaluation of BBN-LISP can be obtained as special cases of this new INTERLISP capability.

Another feature of the INTERLISP frame is the exit-function. In any system that implements flexible control structures, when a module makes a normal return to its parent, certain bookkeeping operations must be performed by the system during the actual transfer. INTERLISP provides a place in the frame for a user function to be specified for execution at this time. This exit function may be specified at run time by a different module. Thus, for example, a module can insert an exit function in the

module three above it in the control chain which causes a breakpoint to the user just before that higher module returns to its parent.

The present control structure of QLISP is rather restrictive, because the new INTERLISP features mentioned above have not been available to build upon. In particular, only "recursive" backtracking is possible; that is, one can only backtrack to a higher point in a depth-first control tree. This means that once a QLISP expression exits with a value, that expression cannot be re-entered as a generator to produce another value. However, as Sussman (Sussman 1972) pointed out, most sequential backtracking programs can be rewritten into nested recursive tests. QLISP provides, as a temporary expedient, a recursive backtracking version BIS of its basic associative retrieval program IS. IS takes a pattern as its argument, and tries to find an instance of that pattern in the data base. BIS takes as an additional argument a test for any expression found. If a proposed expression is rejected, BIS attempts to find a different instantiation of its pattern argument. For example, the following program will search the data base for something that John owns which is colored red: (The pattern-matching operations are explained further in Section C.)

```
(BIS (OWNS JOHN + X)
```

```
  (IF (IS (COLOR $X RED)) THEN (PRINT $X)
      ELSE (FAIL)))
```

After the spaghetti stack and associated control operations are added to INTERLISP, the QLISP IS function will probably be modified to create its own backtrack point, so that the above code could be replaced by (IS (OWNS JOHN + X)) followed by the above IF statement, without needing an enclosing BIS operator.

Demons, in current QLISP, are set up as groups of functions called teams that may be associated with any net storage or retrieval command. This gives the programmer the flexibility needed to design either an efficient system, in which he carefully selects the appropriate times to trigger each demon, or a more carefree system, in which he calls for all demons at every opportunity.

4. POPLER/POP-2. POPLER1.5 follows the PLANNER philosophy in terms of making a failure mechanism and backtracking an important part of the control facility. It uses the Bobrow and Wegbreit frame structure model and allows general multiprocessing to be programmed with primitives similar to the ones described for CONNIVER and INTERLISP. The POPLER interpreter does the time-sharing quantum management. Data base demons are modelled directly on PLANNER.

Its special additional fields for the module frame are an updateable frame data item which can be accessed by the user, a frame type which specifies certain continuation properties of the procedure, and an action list which is used for the backtrack control scheme. The action list contains failure actions which are executed when backtracking occurs, and exit actions which are executed when a POPLER function returns via its control link. The latter provide the same facility as the exit function of INTERLISP. The extended control facilities are only available in POPLER, and not in the underlying POP-2.

C. Pattern Matching

In this section we shall describe the principal automatic pattern-matching and variable-binding operations of the new languages.

1. SAIL. Following normal ALGOL conventions, variables in SAIL must be declared with their types. Item variables or itemvars, represented by identifiers, name locations that may have SAIL items as their contents. These contents (also referred to as the values of the itemvars) are frequently determined by a search and match operation invoked by a FOREACH statement. For example, if X and Y are itemvars, the statement

```
FOREACH X,Y SUCH THAT father  $\boxtimes$  X = Y DO . . .
```

will cause the template

```
father  $\boxtimes$  ___ = ___
```

to be matched against all triples in the data base that begin with "father", call the second and third elements of each such triple X and Y, respectively, and execute the program specified after the DO for each such pair. Thus, patterns per se, as data structures, do not exist in SAIL. Rather, the program syntax simultaneously specifies several patterns and uses them to retrieve desired items from the data base.

2. PLANNER/CONNIVER. PLANNER, CONNIVER, QA4, and QLISP, like LISP, do not have declarations for variables. In LISP all identifiers in argument positions are assumed variables unless explicitly "quoted". In pattern matching context, however, it is much more convenient to operate in "inverse quote mode"; that is, to assume all identifiers are constants

unless marked by a prefix to be a variable. The specific prefix used identifies the type of binding the variable may take.

PLANNER has three types of pattern variables: `?`, `$?` and `$←`. The pattern `?` matches anything. The basic distinction between the prefixes `$?` and `$←` is that `$?X` insists on preserving a previously assigned value for `X`, if any, whereas `$←X` permits the value of `X` to be changed. For example, if we let `←` be the assignment operator, after

$$\$←X \leftarrow A,$$

the operation `$?X ← B` will cause a failure error because `X` is already bound to `A`.

In present implementations, pattern matching in PLANNER can only instantiate variables at the top level of the data list structure. This does not seem to be a serious constraint, primarily because patterns are only matched against assertions, and PLANNER assertions rarely have more than one level of structure.

CONNIVER uses pattern matching in much the same way as PLANNER--to fetch items from the data base, or to identify applicable programs by their patterns. The pattern matching algorithm is kept simple by requiring the programmer to identify the role of each variable in a pattern by means of a prefix. Several prefixes are used:

`?X` permits `X` to be assigned any value

`!X` restricts `X` to be assigned to an expression that contains no variables

`,X` requires that a previously-assigned value of `X` be substituted into the pattern before the match begins

@exp specifies that exp, which may be any LISP expression, is to be evaluated by the LISP interpreter before the match begins.

The CONNIVER pattern matcher may be used on arbitrary LISP data and may contain variables at any level. For example, the pattern

```
((FREDS ?X) . ?REST)
```

matches both

```
((FREDS FATHER) WHISTLES) and
```

```
((FREDS GONE) HE SAID),
```

generating association lists

```
((X FATHER) (REST (WHISTLES))) and
```

```
((X GONE) (REST (HE SAID))).
```

3. QLISP/INTERLISP. Pattern matching plays a much more important role in QLISP than it does in the previously-discussed languages. Patterns are used here not only to access the data base and to select appropriate functions (by means of goal statements or other demon constructs), but also as a basic method for operating upon complex data structures.

QLISP variables come in three varieties and two modes, all identified by prefixes. The varieties are +, ?, and \$:

+X permits X to be assigned any value.

?X permits X to be assigned a value if it has none before, but does not permit a preassigned value to be changed.

\$X references a preassigned value of X, that must exist.

QLISP functions resemble LISP functions but, instead of a list of bound variables to associate with actual arguments, the lambda expression begins with a pattern to be matched against the actual argument. (QLISP functions

have only one argument, but this can be an n-tuple.) Pattern extraction eliminates the need for possibly confusing chains of cars and cdrs. For example, suppose we want a program to transform a list structure of three elements in the following way:

$$(A (B C)) \rightarrow ((C B) A).$$

The LISP function to do this would be:

$$(LAMBDA (X) (LIST (LIST (CADADR X) (CAADR X)) (CAR X))).$$

In QLISP it would be much more transparent:

$$(QLAMBDA (TUPLE + X (TUPLE +Y +Z)) (TUPLE (TUPLE $Z $Y) $X)).$$

Moreover, if the actual data did not have the appropriate form, e.g., if we tried to run these programs on the lists

$$(A B C) \text{ or}$$
$$(A (B C) (D E)),$$

the LISP program would generate an error at some lower level that might be difficult to diagnose, or (in the second example) it would calculate a meaningless result that would cause some future program to run into trouble; the QLISP program immediately reports that its argument does not have the anticipated structure.

There are two modes of variables: individual variables, which we have been discussing thus far; and segment variables, denoted by the prefixes ++, ??, and \$\$, which match any number of elements of a class, bag, or tuple.

Now we can see how the pattern-matching technique for labeling substructures of an expression is particularly useful for QLISP structures of mixed data type. Suppose we wish to find an expression that plays some special role in an arbitrary set of algebraic expressions such as

{ 17, a-b, 5+c+d+e, c+b+d, d-a}.

This set could be represented in QLISP by

```
(CLASS 17
  (TUPLE DIFF A B)
  (TUPLE PLUS (BAG 5 C D E))
  (TUPLE PLUS (BAG C B D))
  (TUPLE DIFF D A)).
```

Now let us pose the question, "If any number is subtracted from something in one expression and added to something in another, tell me what it is added to". When matched against the above set, the following pattern

```
(CLASS (TUPLE DIFF +V1 +V2)
  (TUPLE PLUS (BAG +V2 + +X))
  + +V3)
```

will cause the variable X to be bound to the answer,

\$X =(BAG C D).

4. POPLER. Pattern matching in POPLER is used for the same purposes as in PLANNER/CONNIVER. Pattern variables in POPLER have four types, two modes, and restrictions. The restrictions include a data type restriction, and user programmable tests. The types are as indicated by the prefix forms below, where we have taken the liberty of substituting the pound sign (#) for the Sterling pound sign.

##X matches only the current value of X.

#*X will assign a matched item to a variable as long as the restrictions are satisfied.

#:X will tentatively assign the value, but sets up a failure action to restore the old value in case of later failure back.

#>X behaves like #:X if the variable is unassigned, but will only match the value of X (as does ##X) if it has an assigned value.

In matching list structure elements, individual variables can also have a segment mode, and match an interior segment of a list. The segment mode forms of the above types are prefixed with ###, #**, #::, and #>>.

POPLER patterns are very general, with variables at arbitrary levels in a list, and a stock of standard pattern "actors" (Hewitt 1972) which help specify the pattern. These include an actor which tests whether a specified list is contained in the target, one which will check property lists, and combiners to allow alternatives and conjunctions. New actors are easy to add.

D. Deductive Mechanisms

In this section we shall describe the principal automatic search, deduction or decision-making facilities for the new languages.

1. SAIL. SAIL does not have any explicit deduction mechanism. However, complex semi-automatic search procedures that implement certain deductive principles can easily be programmed with the aid of a device called a "matching procedure". A matching procedure is a boolean procedure that may contain unbound pattern variables as arguments. The matching procedure is called from a FOREACH enumeration statement. It returns either by succeeding and returning values for the previously unbound parameters, or failing, which causes the FOREACH to terminate.

For example, suppose we wish to execute some program hum for every known part of a human being. If the data base has associations such as

```

part ⊗ human = hand
part ⊗ human = foot
part ⊗ hand = finger
part ⊗ finger = fingernail
part ⊗ foot = toe

```

The statement

```
FOREACH X SUCH THAT part ⊗ human = X DO hum
```

would only run hum on hand and foot. However, the following use of a recursive matching procedure partof would run hum on all parts of the human, because partof specifies the desired transitivity of the part relation:

```
FOREACH X SUCH THAT partof (human,X) DO hum
```

Here is a definition of partof, with comments enclosed in quotes:

```

MATCHING PROCEDURE partof(itemvar a; ?itemvar b);
  "The question mark indicates a possibly unbound parameter"
BEGIN
  FOREACH b SUCH THAT part ⊗ a = b DO
    BEGIN SUCCEED; "pass back as first answer each value of b found by
      direct memory look-up"
      q ← b;
      FOREACH b such that partof(q,b) DO SUCCEED;
        "Recursive call for transitivity; whenever any b is
          found, it is passed back to the caller."
    END; "Outer FOREACH now iterates."
  FAIL; "No more possible answers."
END;

```

2. PLANNER/CONNIVER. The key to the deductive mechanism of PLANNER is the theorem, an expression containing as major elements a target pattern we shall call P and a program Q. There are three categories of theorems: consequent, antecedent, and erase. These categories differ primarily in the ways they are invoked.

The most important category with respect to deduction is the consequent theorem, which usually has the logical form

Q implies P ;

that is, "if program Q were successfully executed then the assertion matched by pattern P would be proven". Frequently the "program" Q itself merely requests that an assertion be proven, so that the consequent theorem sets up an automatic backward-chaining mechanism for searching the data base.

These searches are initiated by the goal statement. For example, suppose some program wishes to determine whether a finger is part of a person, when the data base contains the assertions (PART ARM PERSON), (PART HAND ARM) and (PART FINGER HAND). The program statement

(GOAL (PART FINGER PERSON))

would first look directly for the assertion (PART FINGER PERSON) in the data base, but not find it; then the goal mechanism would look for a consequent theorem whose pattern matches the assertion of the goal. If the theorem

```
(CONSEQUENT
  (PART $?X $?Z)           [Pattern P]
  (GOAL (PART $?X $?Y))    }
  (GOAL (PART $?Y $?Z))    } [Program Q]
```

is stored in theorem memory, then by matching (PART \$?X \$?Z) against the goal (PART FINGER PERSON) the theorem would "run", i.e., attempt to prove, two new instantiated goal statements:

(GOAL (PART FINGER \$?Y)) and
(GOAL (PART \$?Y PERSON)).

Upon matching the first goal against the data base, Y is instantiated as HAND; the second goal can then be satisfied by another use (a recursive call) of the same consequent theorem.

The above example shows how facts implicitly present in the combined data base and theorem memory can be deduced when needed. An alternative approach to making needed facts accessible is to deduce them at the earliest opportunity and store them explicitly for future possible use. This approach is possible in PLANNER by using antecedent theorems.

Whenever anything is asserted, i.e., added to the data base, all antecedent theorems are checked against the new assertion. If the P part matches the assertion, the Q part is immediately executed.

Suppose we have the following theorem:

```
(ANTECEDENT
  (PART $?X $?Y)                [Pattern P]
  (GOAL (PART $?Y $?Z))          }
  (ASSERT (PART $?X $?Z))        } [Program Q]
```

Now, continuing the above example, if some program executes

```
(ASSERT (PART FINGERNAIL FINGER))
```

then P of the above theorem matches, so the two-statement Q is automatically instantiated and executed. First (GOAL PART FINGER \$?Z) is proven from the data base by setting Z to HAND, and then (ASSERT (PART FINGERNAIL HAND)) is executed. This latter assertion again invokes the same antecedent theorem. Eventually

```
(PART FINGERNAIL ARM) and
(PART FINGERNAIL PERSON)
```

are also added to the data base, eliminating the need for deducing these facts (with consequence theorems) if they are ever needed in the future.

Of course, antecedent theorems must be used judiciously to avoid cluttering up the data base with many relatively useless facts.

The third PLANNER theorem type is erase: Its Q part is executed whenever P matches a fact that is being erased (deleted) from the data base. Just as antecedent theorems, which are triggered by assertions, are usually used to assert additional derived facts, erase theorems, which are triggered by erasures, are usually used to erase additional dependent facts in order to clean up the data base.

CONNIVER takes the view that PLANNER theorems and their associated search or data-manipulation activities are too automatic. Instead of offering, for example, a GOAL mechanism that searches through alternative derivations (by means of consequent theorems) until a final proof is found, CONNIVER gives the programmer mechanisms for designing his own search algorithms. These mechanisms can be used to construct algorithms similar to the ones built into PLANNER, if desired, but they also permit much more flexible communication and dynamic modification of the search procedures.

CONNIVER data items each reside in their own named data contexts; each major element of a CONNIVER search algorithm also resides in its own control context. The PLANNER concept of pattern-directed program invocation--embodied in the P and Q elements of every PLANNER theorem--has been carried over into CONNIVER. However, the process of matching the pattern P, to generate data items and to bind variables, takes place in a control context independent from the program Q that makes use of those bindings. The two processes may be interleaved in any manner desired by the programmer, who is given convenient handles for coordinating and communicating between such processes. Thus, although there are no built-in

deductive mechanisms quite like PLANNER's theorems, CONNIVER makes it easy for a programmer to devise his own, more tightly controlled, deductive procedures.

3. QLISP/INTERLISP. The QLISP goal statement is quite similar to the goal statement of PLANNER; it causes first a search through the data store for an item matching the argument of the goal, and then, if that search is unsuccessful, the goal statement invokes the execution of appropriate functions (programs) whose target pattern matches the goal. Every QLISP function has as part of its definition a "bound variable expression" that contains a pattern that is matched against its argument before the function may be invoked. This pattern filters out inappropriate arguments, and it binds variables within the function definition to appropriate elements of the selected arguments. As in PLANNER, QLISP functions also maintain appropriate control information to backtrack automatically through alternative matches until a goal is successfully completed, although backtracking between functions is dependent on the completion of the INTERLISP control structure implementation. (By QLISP functions we mean the new QLAMBDA form which has been added to the repertoire of function types available in INTERLISP.)

In addition, the QLISP goal statement has several features that are oriented somewhat differently from the comparable PLANNER statement:

- a) The goal statement consists of two distinct operations in sequence: a data base search, followed by the pattern-directed execution of programs. These two operations, called is and cases, respectively, are commonly used independently.

- b) SAIL normally only stores true propositions (assertions) in its main permanent data base. Although any expression may be stored in the PLANNER data base, the built-in GOAL search mechanism assumes they all represent assertions. Although PLANNER assertions may have property lists, these are not used by the system in any standard way. QLISP encourages the programmer to put in the net any complex structures he wishes; each has a unique occurrence, and a property list. In following goal chains, QLISP system functions know that assertions are just those net expressions that have on their property lists a MODELVALUE attribute with truth value T (true) or NIL (false).
- c) The QLISP demon mechanism is implemented by requiring the user to specify teams of demon functions (perhaps none) as part of every data storage or retrieval operation. By specializing these data operations, e.g., one for changes to the robot world, another for logical propositions, the team mechanism allows tighter specification of relevant demons. Standard teams for ASSERT and DELETE could implement the PLANNER antecedent and erase theorems.
- d) Like CONNIVER, QLISP has both data and control contexts which may be created or modified by the programmer. Goal statements may be executed with respect to any specified contexts; therefore data-base changes need not be undone, as in PLANNER, with erase and antecedent theorems. As the applicable scope of an expression changes, CONNIVER permits moving of expressions to other contexts. QLISP will be able to perform a similar function.

4. POPLER. The POPLER deductive mechanism is modeled after PLANNER and CONNIVER, with procedures which can be called by "pattern directed invocation". A target-pattern, representing what is to be done, is used to select a procedure whose procedure pattern matches the target-pattern. There are four types associated with different classes of targets: achieve, infer, assert or erase. Assert procedures are the "antecedent theorems" of PLANNER, invoked when appropriate data items are added to the data base; erase procedures operate when items are removed. Achieve and infer procedures correspond to two different PLANNER uses of "consequent theorems"; check, is something now the case, versus, goal, make something be the case. Inferring does not allow the use of operators which are defined to change the world. To use Davies example (1973 p.7.1), in a chess playing program we should certainly want to distinguish between the target statements:

- 1) achieve([I am checkmated]);
- 2) infer([I am checkmated]);

The former would attempt to lose the game, while the latter only checks whether the game is already lost.

In addition to direct invocation of relevant procedures with backtrack control, POPLER allows the CONNIVER-like construction of a possibilities list independent of the invocation of the procedures. Possibilities can be procedure items or generators, and can be pruned by any function with access to this possibilities list. POPLER also allows user association of recommendation lists and filters to help modify the straightforward (default) depth first search.

V CONCLUSIONS

We have described in this paper several new programming languages for AI research. We shall now review some of the principal features and present status of these languages:

(1) SAIL. This language is one of the most stable, debugged, and heavily used of the languages surveyed. It runs on a PDP-10 under the DEC 10-50 monitor. Its ALGOL base provides full algebraic capability, with well-tested I/O and interface to assembly-language subroutines. The debugging features remain an extension of an assembly code debugger. Swinehart (1973) has implemented a display package for a system built on top of SAIL which allows informative exploration of a control structure tree. The associative memory is a single, permanent, top-level structure; no convenient way exists for partitioning access to the associations on the basis of control context, subject matter, etc., except by explicit programming. Major storage management, including erasure of abandoned data, is the programmer's (rather than the system's) responsibility; so is the specification of which variables to save or restore upon backtracking. Fairly elaborate process control and communication features have recently been added to the language, and it seems likely that the language will continue to be modified in an evolutionary manner to respond to the needs of its users.

(2) PLANNER. MICRO-PLANNER is an implementation of a subset of Hewitt's PLANNER ideas (Hewitt 1972). MICRO-PLANNER was written in LISP under the

MIT ITS system that runs only on the PDP-10 at MIT, but MICRO-PLANNER has been transferred to other LISP systems for experimental use. PLANNER introduced the important coupled concepts of pattern-directed program selection and procedural representation of knowledge. Extensive automatic depth-first search and backtrack control is a debatable feature of PLANNER: this automatic control structure permits the programmer to describe his algorithms in a piecemeal declarative fashion without worrying about sequential program flow; but it can lead to highly inefficient thrashing in the absence of suitable constraints, and the right constraints are frequently awkward to express. The natural conventions for using the data store assume that it holds only elementary propositions that are presumed to be true; there are no built-in checks for consistency, and there is no convenient way to make use of the knowledge that a given proposition is false. PLANNER has received much publicity, partly because of the outstanding research for which it is being used at the MIT AI Laboratory. Its future probably depends upon the efficiency of its new implementations and the experiences of a growing community of users.

(3) CONNIVER. This new system grew out of the collective experience of MICRO-PLANNER users. It too is implemented in LISP under ITS. The philosophy of CONNIVER is to return a much greater degree of control--and responsibility--to the user than was permitted by PLANNER. As a result, CONNIVER is a system in which it is possible for skilled programmers to design efficient algorithms that involve the kind of complex interacting processes needed in current AI research. Since CONNIVER does not have PLANNER-like conventions to structure the semantics of its data and

programs--assertions, theorems, goals, etc.--it may be substantially more difficult for a new user to learn. On the other hand, the inefficiencies of blind backtracking may make PLANNER also an impractical language except in the hands of an expert who learns the subtleties of its more complex control options.

(4) QLISP/INTERLISP. Much of the QLISP philosophy, and large chunks of its actual code, were taken directly from QA4, an experimental language implemented at SRI more than two years ago. The usefulness of QLISP's new data types and pattern-matching facilities were tested in QA4 in problem-solving and automatic-programming research. The major shortcomings of QA4--such as slow execution and lack of debugging tools and utility functions--have been overcome by embedding QLISP directly into INTERLISP. All the well-established capabilities of INTERLISP debugging aids, user file structures, and so on, as well as all of basic LISP, are automatically available. Although some of the control structure operations available through the QA4 interpreter are not present in QLISP, the new control features of INTERLISP will soon make the combined INTERLISP/QLISP system one of the most flexible systems available. QLISP is still under active development and a first version of QLISP is now being completed at SRI; preliminary versions have been available for experimental use for several months. A new version of the pattern matcher that gains generality by using a unification-like algorithm is being added and substantial changes may take place when the new control structure implementation for INTERLISP is complete. QLISP will eventually contain an efficient implementation of

most of the desirable language features we have discussed, and will be available to the large, interested community of TENEX system users.

(6) POPLER. POPLER 1.5 is a programming language implemented in and an extension of POP-2, a system developed at the University of Edinburgh for application to Artificial Intelligence programming. POP-2 has been used for several years on the ICL 4130 in Edinburgh; in the past year a PDP-10 implementation has been completed (for the 10/50 monitor), and others have been started. Recently a new interactive editor has been completed, and the system is well documented and friendly to users. POP-2 is a simple programming language with good data structure facilities: built-in words, arrays, strings, lists and records. A "garbage collector" automatically controls storage for the programmer.

POPLER 1.5 was completed in spring 1973, and has not yet been used for any major projects. It looks like it will provide most of the facilities of a PLANNER like system. It has a sophisticated control structure which is "visible" to the programmer and program. Programs can be compiled into POP-2 or kept in data structures and interpreted by a special evaluator. The language has general facilities for pattern matching, pattern invocation of procedures, and pattern directed retrieval from a context structured data base. The principal problem with POPLER stems from the fact that it is built on top of POP-2 and is not integral with the system. Another problem is that it currently works only in Edinburgh, and the folklore of the system is in very few hands.

The languages we discussed above are the principal systems currently in use, or likely to be in use in the near future, at the largest AI laboratories. Among them, they represent the major new directions in the development of AI software tools. Several other experimental language systems have some features in common with the systems we have surveyed. We did not include such other systems in detail in this paper because either the system was purely experimental and unlikely to see extensive use, the features of AI interest are only incidental to the main functions of the language, or we were not sufficiently familiar with the system to treat it accurately here. Nevertheless, the reader may be interested in learning more about at least the following relevant systems:

- (1) ABSET (Elcock 1971), a programming language based on sets, developed at the University of Aberdeen;
- (2) ECL, an extensible language system developed (Wegbreit 1972) at Harvard University for work in automatic programming. Similar to INTERLISP internally, it has particularly good handling of extended data types. It combines a pleasant source language, an interpreter for list structure representation of programs, and several levels of optimizing compiler. It has been working for about a year.
- (3) LISP-70 (Tesler 1973), a compiler based LISP system designed to be a "production language" for AI, that is, it biases language features toward efficient implementation. The full system is not yet up, but a prototype MLISP2 (Smith 1973) used some of the pattern matching and automatic rule maintenance for some natural

language processing work. LISP-70 looks as if it will be an interesting language when it is finally available.

- (4) SMALLTALK (Kay 1973) and PLANNER73 (Hewitt 1973), both embody an interesting idea which extends the SIMULA (Ichbiah 1971) notion of classes, items that have both internal data and procedures. A user program can obtain an "answer" from an instance of a class by giving it the name of its request without knowing whether the requested information is data or is procedurally determined. Alan Kay has extended the idea by making such classes be the basis for a distributed interpreter in SMALLTALK, where each symbol interrogates its environment and context to determine how to respond. Hewitt has called a version of such extended classes actors, and has studied some of the theoretical implications of using actors (with built-in properties such as intention and resource allocation) as the basis for a programming formalism and language based on his earlier PLANNER work. Although SMALLTALK and PLANNER73 are both not yet publicly available, the ideas may provide an interesting basis for thinking about programs. The major danger seems to be that too much may have been collapsed into one concept, with a resultant loss of clarity.

The development of software tools for AI research is currently an extremely active area, and the systems emerging from this activity are still in a great state of flux. However, one can begin to see the features fitting together. For example, current AI research interests in representation of knowledge demands the availability of permanent associative memories and complex symbolic structures formed from new data

types such as sets. These structures almost force the use of pattern-based analysis and retrieval methods. Pattern matching applied to such data, in turn, is highly likely to be ambiguous and thus suggests backtrack control. The simultaneous availability of all these features results in extremely parsimonious descriptions of current AI algorithms. Within the next few years we expect that one or more successors of QLISP, PLANNER/CONNIVER, POPLER, and similar developmental efforts will stabilize and become the basis for the major AI results of the next decade. Such successors will embody not only the new features described here, but will face up to the programming process as an entity (as described in Bobrow 1972), and provide the programmer the tools necessary to facilitate all his interactions in building a working system (e.g., editing, debugging, optimization, documentation, etc.). Another challenge will be to seek coherence in the Babel, and enough agreement on the forms of programs to allow successive researchers to stand on the shoulders of their predecessors, not their toes.

BIBLIOGRAPHY

- Baumgart, B.G. MICRO-PLANNER ALTERNATE REFERENCE MANUAL. Stanford AI Lab Operating Note No. 67, April 1972.
- Bobrow, D.G. REQUIREMENTS FOR ADVANCED PROGRAMMING LANGUAGES FOR LIST PROCESSING APPLICATIONS. Communications of the ACM. Volume 15, Number 7, pp. 618-627, July 1973.
- Bobrow, Daniel G. and Raphael, Bertram. A COMPARISON OF LIST-PROCESSING COMPUTER LANGUAGES. Communications of the ACM. Volume 7, Number 4, April 1964.
- Bobrow, Daniel G. and Wegbreit, Ben. A MODEL AND STACK IMPLEMENTATION OF MULTIPLE ENVIRONMENTS. Communications of the ACM, Volume 16, Number 10, October 1973.
- Bobrow, Daniel G. and Wegbreit, Ben. A MODEL FOR CONTROL STRUCTURES FOR ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES. Proceedings of IJCAI, Stanford, California, August 1973.
- Burstall, R.M., Collins, J.S., and Poppleston, R.J. PROGRAMMING IN POP-2. Edinburgh University Press, 1971.
- Davies, D. Julian M. POPLER 1.5 REFERENCE MANUAL. University of Edinburgh, TPU Report No. 1, May 1973.
- Elcock, E.W. et al. ABSET, A PROGRAMMING LANGUAGE BASED ON SETS: motivation and examples. Machine Intelligence 6, Edinburgh University Press, 1971.
- Feldman, J.A. et al. RECENT DEVELOPMENTS IN SAIL--An ALGOL-based language for artificial intelligence. FJCC, 1972.
- Feldman, J.A. and Rovner, P.D. AN ALGOL-BASED ASSOCIATIVE LANGUAGE. Communications of the ACM. Volume 12, Number 8, pp. 439-449, August 1969.
- Gelernter, H. and Rochester, N. REALIZATION OF A GEOMETRY THEOREM-PROVING MACHINE. Proc. International Conference on Information Processing. Paris, Unesco House, 1959.
- Griswold, R.E. et al. THE SNOBOL4 PROGRAMMING LANGUAGE. Prentice-Hall, 1968.
- Hewitt, C. PROCEDURAL EMBEDDING OF KNOWLEDGE IN PLANNER. Proceedings of IJCAI, London, September 1971.

- Hewitt, C. DESCRIPTION AND THEORETICAL ANALYSIS (USING SCHEMATA) OF PLANNER: A language for proving theorems and manipulating models in a robot. AI Memo No. 251, MIT Project MAC, April 1972.
- Hewitt, C. et al. A UNIVERSAL MODULAR ACTOR FORMALISM FOR ARTIFICIAL INTELLIGENCE. Proceedings of IJCAI, Stanford, California, August 1973.
- Ichbiah, J.D. and Morse, S.P. GENERAL CONCEPTS OF THE SIMULA 67 PROGRAMMING LANGUAGE. Compagnie Internationale pour le Informatique, Paris, September 1971.
- Kay, A. et al. SMALLTALK, NOTEBOOK. Xerox Palo Alto Research Center, 1973.
- McDermott, Drew V. and Sussman, Gerald Jay. THE CONNIVER REFERENCE MANUAL. AI Memo No. 259, MIT Project MAC, May 1972.
- Reboh, Rene and Sacerdoti, Earl. A PRELIMINARY QLISP MANUAL. SRI AI Center Technical Note 81, August 1973.
- Rulifson, J.F., Waldinger, R.J., and Dirksen, J.A. QA4, A LANGUAGE FOR WRITING PROBLEM-SOLVING PROGRAMS. Proceedings IFIP Congress, 1968.
- Rulifson, J.F. et al. QA4: A PROCEDURAL CALCULUS FOR INTUITIVE REASONING. SRI AI Center Technical Note 73, November 1973.
- Smith, D.C. and Enea, H.J. MLISP2. Stanford AI Lab Memo AIM-195, May 1973.
- Sussman, Gerald Jay and McDermott, Drew Vincent. WHY CONNIVING IS BETTER THAN PLANNING. AI Memo No. 255A, MIT Project MAC, April 1972.
- Sussman, G.J. and Winograd, T. MICRO-PLANNER REFERENCE MANUAL. AI Memo No. 203, MIT Project MAC, July 1970.
- Swinehart, D. A MULTIPLE-PROCESS APPROACH TO INTERACTIVE PROGRAMMING SYSTEMS. PhD Thesis, Stanford University, 1973.
- Swinehart, D. and Sproull, B. SAIL. Stanford AI Project Operating Note No. 57.2, January 1971.
- Teitelman, W. TOWARD A PROGRAMMING LABORATORY. Proceedings IJCAI, Washington, D.C., 1969.
- Teitelman, W., Bobrow, D., and Hartley, A. INTERLISP REFERENCE MANUAL. Xerox Palo Alto Research Center, 1973.
- Tesler, L.G. et al. THE LISP70 PATTERN MATCHING SYSTEM. Proceedings of IJCAI, Stanford, California, August 1973.
- Wegbreit, Ben et al. ECL PROGRAMMER'S MANUAL. Harvard University, September 1972.