DATA AND CONTROL STRUCTURES REQUIRED FOR ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES

Cheron Rae Vail

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Data and Control Structures
Required for Artificial Intelligence
Programming Languages

by

Cheron Rae Vail

December 1980

Thesis Advisor: D. R. Smith

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): DATA AND CONTROL STRUCTURES REQUIRED FOR ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; Dec 1980 / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): CHERON RAE VAIL	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: December 1980 / 13. NUMBER OF PAGES: 167
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School, Monterey, California 93940	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
Data structures
Control structures
Programming languages
LISP

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

This thesis investigates the requirements demanded of programming languages employed in Artificial Intelligence (AI) research. The primary focus is on the data structures and control strictures implemented in a variety of AI languages, past and present. An appendix contains a discussion of the trend toward increased declarative capability in the AI languages. Another

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. appendix presents a review of the design characteristics and major accomplishments of several artificial intelligence applications systems.

Approved for public release; distribution unlimited.

Data and Control Structures Required for Artificial Intelligence Programming Languages

by

Cheron Rae Vail
A.B., Humboldt State University, 1974

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1980

ABSTRACT

This thesis investigates the requirements demanded of programming languages employed in Artificial Intelligence (AI) research. The primary focus is on the data structures and control structures implemented in a variety of AI languages, past and present. An appendix contains a discussion of the trend toward increased declarative capability in the AI languages. Another appendix presents a review of the design characteristics and major accomplishments of several AI applications systems.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

A large amount of gratitude is due CDR Michael O'Bar for all his personal encouragement and administrative effort originally devoted toward my attendance at NPS, the final result of which is this thesis. Dr. Douglas Smith, my thesis advisor, is greatly appreciated for his determination that this be a good thesis, and his flexibility and unperturbable patience with my irregular schedule of effort toward its completion. A special thanks to Dr. Bruce MacLennan, second reader, for his valuable advice and attention to detail and agreeing to join late in the process when time was critical.

I. INTRODUCTION

> It may well be that ultimately, the field of AI will in large part be concerned with the development of superpowerful computing languages. In this light, the best way to measure AI progress is to look at AI languages [38].

The sophistication of most computing tasks in Artificial Intelligence (AI) can be informally measured by the amount of additional time and memory the designer claims necessary to perfect operation. Of course, time is a constraint that will never really be eliminated - our lives are only so long. The memory constraint may be slightly relieved as more is continually packed into smaller packages. Consequently, as Nilsson suggests, emphasis must be placed on the development of "superpowerful computing languages" that will, in effect take better advantage of the memory and time available than past or even current programming languages.

The fundamental problem of any complex AI computing task is how to efficiently access, search, modify, or even create large stores of knowledge. Thus, the development of more powerful languages must include the development of more powerful data representation and program control techniques than are currently available. In most cases, a refinement of

current techniques, such as pattern matching, or an implementation of a current theory, e.g., Minsky's frame concept, is what is needed to advance the state of the art. Whether this can be done by extending the capabilities of current AI languages or by starting from scratch to create entirely new ones that are based on new points of view, still remains to be determined.

This thesis examines several data representation techniques and program control structures that have been developed and implemented in programming languages to specifically satisfy the needs of AI research, past and present. The particular languages that were selected to demonstrate the implementations of concepts in the two areas mentioned above are listed below. Most are popular general purpose AI languages that account for most of the programming applications to date. The last three are fairly new and still in their acceptance phase.

> IPL (1956 - A.Newell, H.Simon) Information Processing Language. Introduced the concept of list processing. First implemented on a JOHNNIAC, it went through 5 iterations in development. The final version, IPL-V, was implemented on an IBM 704 by late 1959. The language was widely used over the next 6 or 7 years on a variety of machines including an IBM 650, 709/7090, 1620, UNIVAC 1105 and 1107, Control Data 1604 and 620, Burrougns 220, Philco 2000, and AN/FSQ-32 [48].

COMIT (1957 - Yngve) A string processing language. Designed and programmed at MIT as a joint project of the Mechanical Translation Group of the Research Laboratory of Electronics and the Computation Center. It was intended to provide the professional linguist with a programming system in which he could easily write programs needed to carry out his research. Development began on an IBM 704 and shifted to the 709/90. An improved version, COMIT II, appeared in 1968 on the IBM 7090/94 and was implemented on the System/360 by 1975 [49].

LISP (1959 - J.McCarthy) List Processing language. The most recent and widely version of this language is LISP 1.5. It is a machine independent, theoretically oriented language designed for problems requiring list processing, recursive or hierarchical control, and symbol manipulation. It is was originally implemented on machines such as the IBM 1620, 709/90/94. DEC PDP-1, SDS 940, and has been implemented on a variety of machines since [48].

LEAP (1969 - J.Feldman, P.Rovner) Language for the Expression of Associative Procedures. It is based on ALGOL 60 and contains set theoretic and associative operations and data types. It has been implemented on a DEC PDP-10 and PDP-11, as part of the SAIL language [18].

SAIL (1971 - D.Swinehart, B.Sproul) A heavily used language based on AIGOL employing list data structure and containing the set theoretic, associative operations and data types of LEAP. It has been implemented on a PDP-10 and PDP-11 [4].

PLANNER (1971 - C.Hewitt) A goal-directed language for proving theorems and manipulating models for a robot. Facilities include creation and dismissal of goals, provision for recommendations on which theorems to use in trying to draw conclusions from an assertion, and a powerful deductive system [49]. The language introduced the important coupled concepts of pattern-directed procedure invocation and the procedural representation of knowledge [4]. PLANNER has never been fully implemented. A subset of the language called Micro-PLANNER has been implemented in LISP on a DEC PDP-10 at at MIT [49].

CONNIVER (1972 - G.Sussman, D.McDermott) A language with LISP-like syntax. Its development was motivated by the desire to eliminate the defects in Micro-PLANNER [49]. Much of the control which is automatic in PLANNER is returned to the user in CONNIVER. The extra responsibility also makes this language more difficult to learn. It is implemented on a DEC PDP-10 at MIT [4].

INTERLISP (1971 - W.Teitelman) Interactive LISP. As an extension of LISP 1.5 it adds debugging aids, user file structure, and other features not previously available, such as arrays, strings, and user-definable data types [54]. It has been implemented on a variety of machines, primarily DEC PDP-10's [49].

QLISP (1973 - R.Reboh, E.Sacerdoti) Formerly the QA4 programming language, an experimental language written in LISP and implemented at the Stanford Research Institute around 1973. QLISP offers data types and pattern matching facilities that cause it to resemble PLANNER in philosophy and detail [49]. By embedding QLISP into INTERLISP additional features of fast execution, debugging aids, and utility functions are acquired. The result is the QLISP/INTERLISP system, one of the most flexible AI programming systems currently available. This system has been implemented on a DEC PDP-10 and an IBM System 360/370 [4].

POPLER (1973 - J.Davies) Based on the POP-2 low level language designed at the University of Edinburgh for application to AI programming. The current version of POPLER, POPLER 1.5, provides most the facilities of a PLANNER-like system. It is currently implemented on a DEC PDP-10 at Edinburgh [4].

ABSET (1971 - E.Elcock, J.Foster, P.Gray, J.McGregor, A.Murray) An interactive programming language based on sets developed at the University of Aberdeen. Its invention is an attempt to provide a programming environment in which it is possible to take or defer decisions about a program. Logically separable decisions can be taken separably and in any order [15].

PROLOG (1972 - A.Colmerauer, H.Kanovi, R.Pasero, P.Roussel) Programming in Logic. A PLANNER-like language founded on symtolic logic-computational mechanisms and embodying procedural interpretation of deduction. Instead of functions it uses relations, i.e., ordered sets of clauses each of the form pattern:-body [36]. It is a pattern matching process operating on general record structures [62].

TELOS (1977 - L.Travis, M.Honda, R.LeBlanc, S.Zeigler) An extension of PASCAL with additional data and control abstraction mechanisms to suit it to AI programming requirements. It does not include a list data type but provides a data encapsulation mechanism for the user to define this and a variety of other types as well [59].

The next chapter examines the fundamental programming requirements as imposed by the nature of the research and as implemented in the earliest AI languages.

Chapters Three and Four consider data representation and program control structures and concepts available or employed in current established AI languages as well as the newer languages not so well established. Chapter Five contains a chart summarizing the data and control features of the languages described above.

Following that, Appendix A offers a discussion of the trend of the newer languages to be progressively more nonprocedural. Appendix B contains the general designs and distinguishing characteristics of 15 AI applications systems.

II. BACKGROUND

Artificial Intelligence is the "branch of computer science devoted to programming computers to carry out tasks that if carried out by human beings would require intelligence [23]." This is a broad definition but indicative of the comprehensiveness of the research dedicated to that purpose. The areas of current investigation include robotics, game-playing, computer understanding of natural language, computers as knowledgeable experts in a variety of scientific fields, computers as commonsense problem solvers, computers as mechanical assistants for menial tasks, and as educational assistants. Many systems have been designed to accomplish a variety of tasks in these areas (see Appendix B). All are prodigious undertakings, but few are fully developed or utilized outside of the laboratory environment in which they were created. Generally, those that aren't experimental are of a special purpose nature. Each of these however have advanced the state of the art toward the goal of AI as identified by Graham, above, or that by Boden: to use "computer programming to cast light on the principles of intelligence in general and human thought in particular." Thus, scientists are employed in either trying to discover how the human mind works by computer modelling of

information processing behavior demonstrated in game-playing, natural language understanding, and general problem solving, or they are interested in developing systems to assist or replace humans in tasks requiring high visual concentration, a great deal of knowledge in a specific field, a large amount of physical strength, or complicated and time-consuming computation.

Implementation of the systems to achieve these ends will most likely continue to be realized in software. It becomes evident that the development of programming languages to facilitate AI research must occupy a place of high priority. "Theoretical and practical advance" in AI is dependent upon it [5].

A programming language for AI interests invariably has more demanded of it than one used for general purpose computing. Tasks in AI involve an enormous amount of database updating, searching, and manipulation in order to discover or deduce the solutions from a large store of knowledge pertinent to the problem at hand. For example, consider a system like INTERNIST, created by H. Pople in 1975 to "provide cognitive support in the formation and solution of difficult clinical problems in internal medicine [40]." Its knowledge is represented in two element types. Either something is a disease entity or a manifestation of some disease. There are about 400 of the former and well over 2000 of the latter (see Appendix B). Given that each

disease entity is associated with a list of manifestations, a method must be derived for entering the knowledge base at a reasonable location in order to begin a sensible search. In INTERNIST this is assisted by the fact that the database is partitioned semantically around organ systems. Once a start point is found, backward chaining on a conceptual tree representation of the knowledge could take place as actual symptoms are matched to the system's manifestations. Additional input would at times be requested, and "backing up" would most likely occur whenever there is insufficient certainty in the matching process. At such a point return to an earlier instance where there was an untried but feasible alternative would occur and the search would head in a related direction still trying to accurately pin down the correct disease. An efficient programming language designed to anticipate such needs as backward chaining, backtracking, pattern-matching, and context saving becomes desirable, if not necessary, for most AI problems, of which INTERNIST is typical.

INTERNIST is exemplary of another basic characteristic of AI programming tasks, that is the manipulation of non-numeric, or symbolic, data. Such data includes strings of characters that one would find in research with natural language systems or mathematical expressions like those found in theorem-provers. In general, the sort of work a computer would be required to perform for AI includes

algebraic formula manipulation, computational linguistics, information retrieval, or automatic decision making, all of which turn it into a symbol processor rather than a number processor[40].

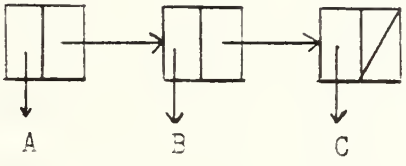
Languages dedicated to non-numeric computation are known as symbol manipulation languages. Furthermore, within this category there are two types, list processors and string processors. These are not necessarily mutually exclusive classifications. String and list structure differ primarily in the way they are stored internally in the computer. A "string is a sequence of elementary items, usually alphabetic characters" that are generally, though not necessarily by definition, tightly packed in sequential memory for the sake of efficiency [44]. A list, on the other hand, may be a sequence of items which are elementary (atoms) or lists themselves. Associated with each item is the sequencing information needed to locate the next item, usually not in an adjacent location [44]. Lists require more memory than strings because of the additional locations required for the pointers directing the sequencing. In general, lists are easier to modify; insertions and deletions can be made by changing a few pointers rather than by having to move large amounts of data.

List Processing:

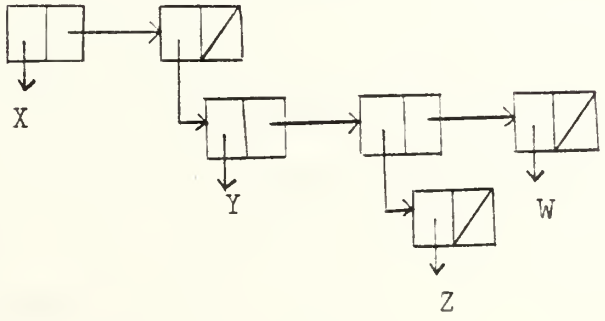
A list processing language is one whose primary data type is a list. A list is a sequence of elements which are either atomic symbols or lists. For example, (A B C) is a list where A, B, and C are simple units (atoms) that stand only for themselves. (X(Y(Z)W)) is a list where X and W are atomic but (Y(Z)) and (Z) are lists themselves.

A word has two parts, traditionally called the "address" and the "decrement". The address holds a pointer to the location of the property list, in LISP, holding descriptive information for the atom of interest, or to the head of another list. The decrement may point to the next list item or an atom although usually the next list item. The two previous examples would possess the following internal arrangement:

(A B C)=



(X(Y(Z)W))=



The basic operations applicable to lists are retrieval of the first element and retrieval of the list that results from removing the first element, i.e., of accessing either the address or decrement part. The basic property of a data element is that of being atomic or itself a list. The basic relation between data elements is that of being identical or not[11].

A valuable quality of list structure is that it divides the data being represented into "major components which may be accessed independently, and which may themselves be hierarchically structured [44]."

An important advantage of a list processing language is the ability and ease it provides to allocate storage dynamically. Memory space for data structures need not be preassigned. Storage for each structure is allocated only as needed, and it is almost never in sequential locations. Since memory reassignment, as well as assignment, is dynamic, there must be a store of cells available for use and mechanisms for obtaining "new" cells from, and returning unneeded cells to, that store [4]. The problem that arises is in keeping track known as the "erasure problem," is in keeping track of used, unused, and "erased" memory cells. One method of handling the problem is for the programmer to reclaim cells as they are no longer needed by linking them to a 'free storage list.' Another method is that of "garbage collection" (see definition, p. 64).

List processing languages advantageously provide for recursive definitions of routines. Recursion is dependent on dynamic storage allocation for efficiency since when a function is defined in terms of itself the programmer usually does not know in advance how deeply nested the process will go before natural termination.

The list structure concept was first introduced by Newell, Simon, and Shaw as early as 1956. Newell and Simon had designed a system, the LOGIC THEORIST (LT), which was tasked with proving theorems in propositional calculus, in particular the theorems in Principia Mathematica by Whitehead and Russell. Their objective had been to simulate a theory of human problem solving on a computer. IT was highly restricted in application but it did demonstrate that a machine "could perform tasks heretofore considered intelligent, creative and uniquely human [35]." The language they created in which to implement their theorem prover was IPL. The LOGIC THEORIST program consumed vast amounts of memory so there was no possibility of allocating storage permanently for any particular function. Faced with limited computer memory, they devised the list structure. It turns out, as well, that the representation of data in lists lends itself nicely to the simulation of human thinking processes.

Like most languages available in 1956, IPL was primitive. Programs written in it closely resembled machine language programs (see figure 2a). Storage allocation, including retrieval of abandoned list cells for re-use, was entirely the responsibility of the programmer[43]. It used a sequential control scheme with some provision for two-way branching and subroutine calling. There was no distinction made between main program and subroutine. Any routine could call any other. Translation terminated only after the highest level routine terminated [43]. The programmer was responsible for parameter passing. Recursion was easily achieved since any routine could execute itself as a subroutine. It was possible to call a routine with a name that was supplied as input data, or to construct arbitrary lists at run-time in the proper format for a routine and then execute that routine. Furthermore, since a program could also manipulate existing routines, an IPL program was self-modifiable.

In 1959, not long after the appearance of IPL, J. McCarthy introduced his list processing language, LISP. His intention was to make the ideas inherent in IPL (and H. Gelernter's plane geometry theorem proving program) cleaner, more elegant, and more powerful[35]. LISP was more readable than IPL. Storage allocation and deallocation were made a system responsibility via a garbage collection algorithm.

```
R1      J60         (skip first element)
    70  J8          (terminate if done)
    40  H0          (save current place)
        R1          (reverse rest of list recursively)
    12  H0          (get current element)
        J65         (insert at end)
        J68   0     (delete from top and stop)
```

An IPL program to reverse the order of elements in a list (reprinted from [43]).

Figure 2a

An additional and still valuable quality of LISP is the capability to represent instructions as well as data internally in a list structure. Thus, program and data are indistinguishable. A program may in fact be data for itself or another program. This allows programs to use their "their list-processing abilities on themselves to modify themselves [5]."

McCarthy strove for, and to a great extent achieved, a quality still unequaled in its comprehensiveness by most other languages: that of mathematical neatness. His goal was to "allow proofs of properties of programs using ordinary mathematical methods [34]." He was able to do this by basing the language on the lambda calculus.

LISP programs clearly demonstrate another advantage of the use of recursion, namely clarity and simplicity of the program text for iterative procedures. The figures below provide a simple illustration of this fact. Figure 2b contains LISP code for a routine to reverse the elements of

an arbitrary list. Figure 2c is a PL/I-80 routine to perform a similar task, i.e., reverse the characters in an arbitrary sentence. Eventhough the PL/I-80 routine need not worry about individual elements that might themselves be sentences it is still a great deal longer.

Other 'list processing' languages include PLANNER, CONNIVER, PROLOG, INTERLISP, and CLISP.

```
(REVLIST (LAMBDA (LIST)
         (COND (NULL LIST ())
         (T (APPEND (REVLIST (CDR LIST))
                    (CONS (CAR LIST ()))))
```

LISP
(Reprinted from Siklossy, 1976)

Figure 2b

```
reverse:
    proc options (main);
    dcl
       sentence ptr,
       1 wordnode based (sentence),
         2 word char(30) varying,
         2 next ptr;

       do while ('1'b);
       call read();
       if sentence = null then
           stop;
       call write();
       end;
   read:
       proc;
       dcl
           newword char(30) varying,
           newnode ptr;
       sentence = null;
       put skip list('What's up? ');
           do while ('1'b);
           get list(newword);
           if newword = '.' then
               return;
           allocate wordnode set (newword);
           newnode->next = sentence;
           sentence = newnode;
           word = newword;
           end;
       end read;

  write:
       proc;
       dcl
           p ptr;
       put skip list('Actually. ');
           do while (sentence = null);
           put list(word);
           p = sentence;
           free p->wordnode;
           end;
       put list('.');
       put skip;
       end write;

end reverse;
```

PL/I-80
(Reprinted from the PL/I-Applications Guide-1980)

Figure 2c

String Processing:

The basic idea of a string processing language is that a program consists of an ordered finite set of transformation rules, an idea inspired by Markov algorithms. Each of these has a left-hand-side describing the composition of a string of characters, and a right-hand-side specifying a transformation to be applied to a string that it to matches the pattern described by the left-hand-side[12]. Rules are applied in order of their appearance in the text, in so far as is appropriate. At every step the first applicable rule is used and the process is repeated until no applicable rule exists, or until an explicit stopping condition is given. A very simple example of this process is shown below.

Transformation rules:

1. *b / ab
2. bb / ba
3. aa / ab
4. a* / *

Given: bbabaab Produce: abababab (i.e. a sequence of alternating a's and b's, beginning with a and ending with b)

Application order of rules:	new string:
1.	abbbabaab
2.	abababaab
3.	abababbab
2.	ababababa
4.	abababab
done.	

A string may be thought of as a vector, array, or anything that specifically represents an ordered sequence of elements. However, the number of elements in a string is not predetermined and may often vary dynamically [3]. Whereas a list is a particular way of representing information in a computer, a string is one of the types of information that may be represented [48]. In other words, a string is a list whose elements can not be lists.

The basic operations performed on strings include searching for patterns and transforming them into different patterns. The basic units of a data string, called items, may be either individual alphanumeric characters, special characters, or pre-specified strings of characters [48].

Strings have been used primarily and conveniently in the representation of text material, such as sentences in a natural language or any arbitrary sequence of characters from some particular data area.

String processing was first featured as a fundamental data structure in the COMIT programming language introduced about 1958. Developed by a group of researchers working under the direction of Yngve at MIT, it was designed in order to "provide the professional linguist with a programming system in which he could easily write the programs needed for his research[48]."

COMIT is primarily a sequentially controlled language. The programmer defines a sequence of "rules" that are ordered in the program on some arbitrary priority. Each rule specifies a desired input that if found directs the action in the rule's output specification to be taken. There is provision for branching, looping, closed subroutines, and recursive subroutines. The programmer is responsible for keeping track of parameter passing.

A COMIT program is not very readable (see figure 2d), although much of its early popularity was due to the actual ease of writing and using the language. COMIT has served as the model for the type of facilities needed for string manipulation, and virtually every language which has included features of this kind has based them to a large degree, in spirit as well as notation, on COMIT.

```
INSERT    $Ø+K/.Ø          (go if K=Ø)          -
*         -=               (at least 1 blank)   LOOP
*         $1=Ø             (no blank)           -
LOOP      $1+$+-=1+2+3+-   //*Ø4 2 3 4          TESTB
*         $=$Ø+1           //*A4 1              LOOP
TESTB     $Ø+K+-=          //*Ø4 3 (remove      TESTB
                           all leading blanks)
*         $Ø+K/.ØØ=Ø/.Ø1                       LOOP
*         K-$=             //*A4 1              -
LASTLINE  $=-+1+*          //*WAM1 2 3          END
END
```

COMIT routine to intersperse blanks in a line of text

Figure 2d

Application:

The choice of a list or string processing language will usually depend on the nature of the programming problem. "Strings are more useful for linear comparisons, lists for structural access [44]." In natural language analysis of translation, written strings of characters have been found to represent spoken strings of sound in the most natural way, since language normally occurs in linear sequences. Lists are most appropriate for tasks wherein some data hierarchy is known or suspected to exist.

III. DATA STRUCTURES

The earliest programming languages manipulated only scalars and arrays since computers were most useful for numerical problems. Consequently, not until the introduction of symbol manipulation languages allowing computation on nonnumeric data items, did the field of Artificial Intelligence, for which the computer is the major tool of research, come into existence. In fact AI finally received attention from the computing industry as a serious area of research with the development of the first programming languages designed for its needs. These languages were IPL, LISP, and COMIT. The first two are list processing languages and the third is a string processing language. Out of the three LISP acquired the most popularity in and out of the AI arena.

It is extremely important, no matter what the field of application, that data be represented in the most appropriately direct and simple manner possible. In AI this becomes critical as most programming tasks bump the upper threshold of the computers available memory early in the processing. Thus a programmer never wants to allocate more storage than is absolutely necessary at any point in the processing. A major breakthrough in this respect was made with the invention of the list data structure which makes

convenient the dynamic allocation and deallocation of memory. List structure is also a perfect venicle for recursion (see definition, p. 93), now a capability often taken for granted in most languages. The list structure alone is not sufficient for all types of AI programming tasks. For example, the HEARSAY-II speech understanding system is implemented in SAIL wnerein this system may take advantage of the set data structure and association data type.

Before continuing, a subtle distinction between data type and data structure must be drawn. Fundamentally, a data type is an interpretation applied to a string of bits [65]. It may be structured or scalar. A scalar data type includes real, integer, double precision, complex, logical, character, pointer, and label. Structured data types include arrays, sets, records, lists, etc. That is, they are objects made up of elementary data types. For instance, an array is a set of index-value pairs. An array is usually assigned consecutive memory locations, but not necessarily. For each index (usually an integer) which is defined there is an associated value (usually numeric). The structured data type becomes a data structure wnen it is associated with a set of well-defined operations that may be performed on that data type; to create, delete, access, or modify it. Using the example of an array again, it can be created by naming or declaring its size before it is referenced. A value may be

retrieved by specifying the index associated with that value. It may be stored similarly. Data structure operations may be considered as a "definition" of a data type [24]. Furthermore, a "data type specification" (sometimes referred to as an abstract data type) is a representation-independent formal definition of each operation of a data type [24].

In the past, programming languages left the responsibility of creating, indexing, and accessing data files to the programmer. However, the newer AI languages provide built-in automatic mechanisms for handling large, relatively permanent information files conveniently. The exploitation of associative memory for structure storage and subsequent information retrieval (a pattern matching operation) is one example. Most of the new languages are list processors. Besides the facilities for dynamic storage allocation, they provide automatic garbage collection (see definition, p. 64).

The programming languages developed for AI have been the means by which some of the more novel inventions in data representation have been introduced. The remainder of this chapter will focus on the descriptions and implementations of the most interesting and important of these. In particular, they are lists and strings, tuples, bags, and classes, sets, encapsulated data types, semantic networks, property lists, associations, and contexts and frames. Some miscellaneous terms and concepts are also defined.

A. LISTS AND STRINGS

The notions of list and string structures were introduced in the preceding chapter. In general lists are considered to be the fundamental data type for AI languages. The concept was originally put forth with the introduction of IPL in 1957. Languages which feature list data types and structures since that time include: LISP, QLISP, INTERLISP, PLANNER, CONNIVER, LEAP, SAIL, and POPLER. It is interesting to note that INTERLISP and QLISP are extensions of LISP. Also that PLANNER and CONNIVER are translated into LISP before being compiled.

The string data structure provides a means to manipulate character data. COMIT (1957) was the first language to utilize this concept. Since then only SNOBOL remains as an efficient and popular string-oriented language.

B. TUPLE, BAG, AND CLASS

These three structures are similar in nature as they all represent some collection of items. All three are provided as standard features in QLISP [45], whereas in TRIOS they may be constructed via a special data type specification mechanism.

TUPLE. A tuple is similar to a list but may be accessed associatively. A tuple may be asserted (placed in the data base), deleted (removed from the data base), or retrieved from the data base. For example, consider the tuple:

(AT NPS MONTEREY)

The statement ASSERT(AT NPS MONTEREY) would store that tuple in the data base as a true fact by placing the attribute MODELVALUE with the value T on the property list for the tuple. The statement IS(AT <-thing NPS) would search the data base trying to find a value previously asserted in a 3-tuple between the items AT and NPS. The search would be done associatively. For example, if the tuple (AT CIVILIANS NPS) were stored in the data base with a T value for the MODELVALUE attribute in its property list then the value 'CIVILIANS' would be returned for the pattern variable '<-thing.'

BAG. A bag is a "multi-set, an unordered collection of elements with possible duplication" [47]. Thus, (BAG A A B C) is equivalent to (BAG A C B A) but not equivalent to (BAG A B C).

Bags are useful for "describing the argument lists of associative commutative relations" [Sacerdoti, et al, 1976]. Suppose the relation PLUS were defined for a bag-type argument, then the expressions PLUS (A A B C) and PLUS(A C B A), which are both internally represented with (PLUS(BAG A A B C)), would be equivalent, by definition [47].

CLASS. A class, on the other hand, is an unordered collection of elements wherein repeated elements are allowed but the internal representation ignores the duplications[4].

Classes, bags, and tuples may be used together in a statement to separate different types of elements. For example, consider the following statement and subsequent request:

On the kitchen table there is a cup, coffeepot, newspaper, and toast. In the living room there is a newspaper on a chair, a TV, and a lamp.

What is on the kitchen table that is also in another room?

The two statements could be represented more formally as:

```
(CLASS(TUPLE ON
       ((BAG cup coffeepot newspaper toast) kitchentable))
      (TUPLE IN (BAG (TUPLE ON newspaper chair) tv lamp)
       livingroom)).
```

One could at this point set up a template using special pattern variables to specify a matching operation to deduce the information requested above. This could in fact be done with the list representation:

```
(CLASS (TUPLE ON (BAG <-X <-<-U kitchentable))
       (TUPLE IN (BAG <-X <-<-V <-Y))
```

where the variables prefixed with arrows are unbound pattern variables that will be bound when a match is found. In this

example they receive the bindings:

U=(BAG cup coffeepot toast)

X=newspaper

Y=livingroom

V=(BAG (TUPLE ON chair) tv lamp)

where "newspaper (X) in the livingroom (Y) is the answer to the request.

C. SET

Basically, a set type represents a finite unordered collection of items (of the same type) containing at most one occurrence of an item [18]. Thus, a set is also a class. Although, a class may contain a collection of items of varying types whereas a set may not. Some languages (e.g., LEAP) do require an ordering property. Sets form the basis for the ABSET language [16]. LEAP and SAIL also provide built-in functions for application to user-defined set data. The basic operations on sets are union (conjoining two sets), intersection (finding common elements in two sets), test for membership, and insertion and deletion of individual items.

The set concept can be a powerful one. It provides clarity and a good foundation for ways of achieving repetition. Repetition formulas such as "this is true for all members of this set" are used in place of "go to" statements or recursion. Since ABSET is more strongly

set-oriented than any of the other languages a closer look at some of its features is warranted.

In ABSET, sets can be defined in a general way, for example, as all those items satisfying a certain predicate. The domain and range of every function (all functions must be total) are specified in terms of previously defined sets. For example if B is a new set being introduced it may be defined in terms of a preexisting set and an equivalence relation defined for that set. Some set A may have several relations defined for it. If A is the set of all fractions we could create the equivalence set TWO-THIRDS which contains all fractions that reduce to 2/3. B might then be simply defined as the set equal to A according to the equivalence relation TWO-THIRDS.

The set structure is built on the following functions:

IN (the usual set-membership predicate, yields a boolean value)

EQUINSET (gives a equivalence relation over a set referred to earlier)

SITEM (gives an item of a set, if empty gives an arbitrary item)

SREM (gives a set which is the same as the original, but excluding the item specified in the SREM expression)

CUPF (a function that outputs the union of two sets)

POWF (a function that gives the set of functions from one set to another)

UPTOF (a function that outputs an ordered set)

THOSEF (a function that outputs a set of all items of a given set which satisfy a given predicate)

Thus, if set1=(a b q z),then SREM(set1,c) yields the set (a, b z); IN(set1,p) yields a false value. If SET2=(7,2.9,28,1.9,51,C,R,V), SET3=Booleans. and SET4=Integers then

CUPF (SET3 SET4)= (T,F,1,2,3,...), and

THOSEOF SET2 IN SET4= (7,28,51).

The motivation for the development of ABSET was to distinguish between "an ordering of decisions and an ordering of evaluation[16]." 1977]. The desire was to create a language that does not force the programmer to early commitment of decisions he or she would rather prefer to postpone. For example, after indicating an object is an array, it should be possible to defer declaration of its size, or if it is a procedure to defer specification of the details of the corresponding algorithm. The set concept was considered a natural vehicle for achieving these goals.

D. ENCAPSULATED DATA TYPE

Encapsulation is the process of defining a data structure from a user-specified data type along with a specification of the type's structuring method and the set of procedures that determine the primitive operations [39]. An example of the encapsulation of a binary tree data type (borrowed from [24]) is shown in figure 3a.

In the TELOS language a "capsule data type generator" is provided to assist in constructing "capsules (encapsulated data types)." TELOS does not directly provide the user with lists, classes, bags, trees, or graphs. However, all of these may be easily constructed with the use of the capsule generator.

The capsule structure provides localization of representation of detail, that is, scope or context within which the specified type is applicable. This is a powerful data abstraction mechanism that combined with TELOS's control abstraction mechanisms is capable of establishing a simple and direct means of implementing high-level theoretical concepts necessary for the development of a general theory of intelligence [39].

```
type Binarytree [item]
  declare EMPTYTREE() -> Binarytree
          MAKE(Binarytree,item,Binarytree) -> Binarytree
          ISEMPTYTREE(Binarytree) -> Boolean
          LEFT(Binarytree) -> Binarytree
          DATA(Binarytree) -> item U {UNDEFINED}
          RIGHT(Binarytree) -> Binarytree
          ISIN(binarytree,item) -> Boolean;
  for all l,r   Binarytree, d,e   item let
          ISEMPTYTREE(EMPTYTREE)=true
          ISEMPTYTREE(MAKE(l,d,r))=false
          LEFT(EMPTYTREE)=EMPTYTREE
          LEFT(MAKE(l,d,r))=l
          DATA(EMPTYTREE)=UNDEFINED
          DATA(MAKE(l,d,r))=d
          RIGHT(EMPTYTREE)=EMPTYTREE
          RIGHT(MAKE(l,d,r))=r
          ISIN(EMPTYTREE,e)=false
          ISIN(MAKE(l,d,r),e)=
               if d=e
                    then true
                    else ISIN(l,e) or ISIN(r,e)
    end
end Binarytree
```

Using the encapsulation above a simple program may be written to construct a binary tree and extract the left subtree:

```
TREE1=EMPTYTREE()
TREE2=EMPTYTREE()
TREE3=EMPTYTREE()

TREE1=MAKE(TREE1,A,TREE2)
TREE2=MAKE(TREE2,B,TREE3)
TREE0=MAKE(TREE1,C,TREE2)

LEFT(TREE0)
```

The implementation independent way of evaluating an expression involving the operators of the data structure would yield:

```
LEFT(MAKE(MAKE(EMPTYTREE(),A,EMPTYTREE()),C,
         (MAKE(EMPTYTREE(),B,EMPTYTREE()))).
```

This reduces to
MAKE(EMPTYTREE(),A,EMPTYTREE())
which is equivalent to A)

Encapsulation of a Binary tree.
Figure 3a

E. PROPERTY LIST

A property list is a structure used in list processing languages and is associated with an atom. It is a list made up of attribute-value pairs. The size of the list is not fixed and may shrink or grow during computation [1] via special functions for inserting and deleting attribute-value pairs. In LISP, these are 'putprop' and 'get' respectively [63].

Example of a property table representing a property-list:

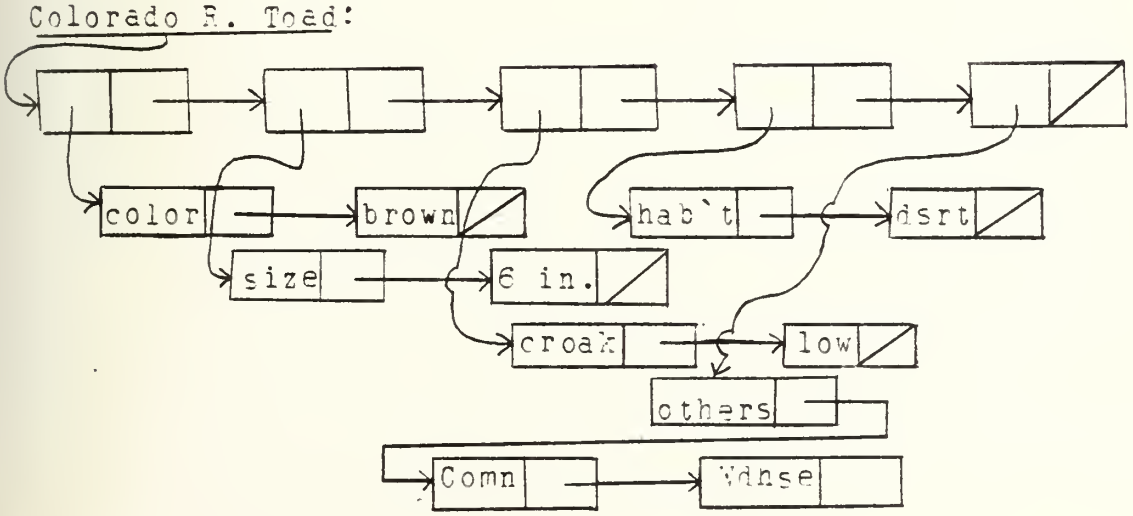
Attribute:	Value:
Color	Dk. brown
Ave. Size	6 inches
Croak	Low
Habitat	Desert
Other Members	Common Toad
Same Family	Woodhouse Toad

Thought of as data structure abstractions, property lists are symbol tables, as the example above illustrates. The name entries are the properties and the value-entries are the property values [1]. An attribute may also name a

relationship between the described object and other objects, where the value identifies the other party or parties to the relationship (see last entry in example above) [44]. In a data base, equivalent expressions will have the same property list [47].

Property lists were first introduced to programming languages in IPL. They are basic structures for any list processing language. No atom has meaning without some specification of its characteristics in some context. By the same token, no property list is accessible unless it has explicit association to some atom.

In LISP, property lists are incorporated into the language features that is, functions for manipulating them exist as built-in operators [23]. The internal representation of the property list shown above would appear as follows (demonstrating that a p-list is a list of associated pairs):



The LISP functions that are used to manipulate property lists include[63]:

GET(X Y) which searches a list X for an attribute which matches Y. If such an attribute is found then the next list element, i.e., the attribute's value, is returned. Otherwise the value of GET is NIL.

PUT(X Y Z) puts on a property list of literal atom X the attribute Y followed by value Z. Any previous of Y is replaced by Z. The value returned for PUT is Y.

PROP(X Y FN) searches list X for an attribute equal to Y. If one is found, the value returned for PROP is the rest of the list beginning immediately after that attribute. Otherwise, the value is FN(), where FN is a function of no arguments.

REMPROP(X Y) removes all occurrences of the attribute Y and its value from property list X. The value returned is NIL.

F. SEMANTIC NETWORKS

A semantic network (net) is a graph defined by objects and their property lists (list of attribute-value pairs describing the object). The object and the values are uniquely represented by nodes while attributes label arcs [23]. A program can traverse the graph in a semantically ordered fashion by starting at the specified node of interest and following arcs sequentially to nearby related nodes. This is an appealing notion in that it seems analogous to the way the human brain will jump from one related idea to another.

Most of the work on semantic memory representation has been done with respect to natural language understanding. The original intent of such research is to represent semantic knowledge in a computer in the way in which humans would store such knowledge in the brain. A data base partitioned into a semantic net provides inferencing capability. In other words, a program can ferret information from the data base that is not explicitly represented. The value of this quality is in not having to clutter the data base with noncritical facts that are derivable from other facts already present.

A semantic net scheme devised by Quillian (see figure 3b) segments the data base into "planes." A plane is a collection of concepts (nodes) connected by directed arcs (pointers) which together represent a certain semantic

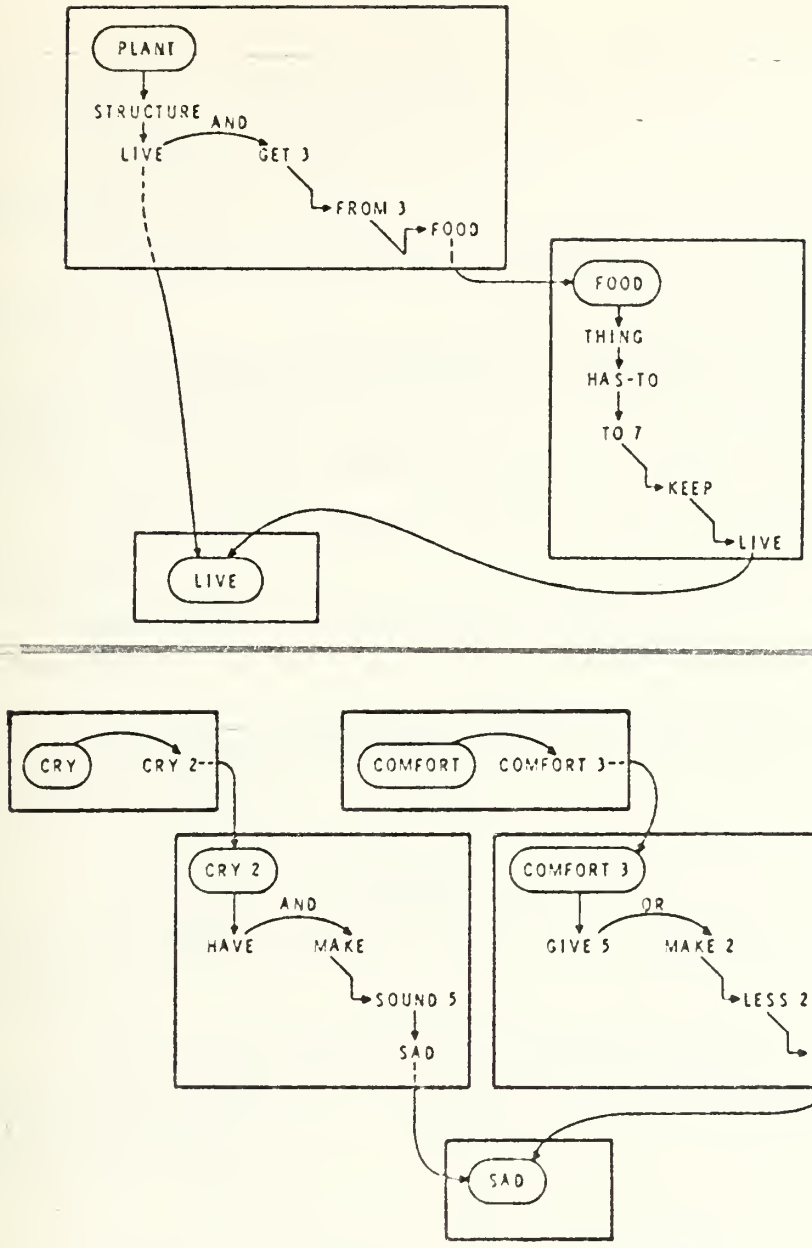
"topic"; a distinct piece of knowledge. Pointers are used extensively to connect related nodes within planes (solid arrows) as well as between planes (dashed arrows). Pointers between planes point from node to plane, rather than node to node. In figure 3b there are four topics, three concerning various meanings of the word, plant; one having to do with a particular meaning of 'food.' Additional knowledge (planes) may be added to the net. Nodes may be connected explicitly (within the plane) and implicitly (between planes) by initiating a pattern search throughout the data base network. Information may be retrieved from the net by requesting that one or more relationships, if they exist, be found between two topics. The program would start at one node of interest, follow arcs to related nodes, then follow from those to still more distantly related nodes and so on to retrieve a related concept.

Semantic nets have also been used in visual scene analysis [64]. Figure 3e is a picture of a simple arch and an initial representation of it in a semantic net. As the program is exposed to other examples and varieties of arches, and structures that are not exactly arches but similar, this simple net may be refined and expanded to account for various arrangements by adding nodes and arcs specifying possible additional parts and relationships (as in figure 3f). The final expansion also contains all information learned along the way, e.g., in the example, A

is not merely supported by B and C as was assumed in the beginning. It was discovered that A must in fact be suported by B and C. Later a new structure may be described and a determination made by the program as to whether it might be a sort of arch based on a pattern match attempt between the network representation and the actual components of the object. Furthermore, questions may be asked about the results of the matching process.

As mentioned earlier, a valuable quality of semantic networks is that they allow inferencing of information not explicitly stated in the data base. This capability can be demostrated by means of a dialogue that occurred between Raphael and his semantic information processing program, SIR [41]. Figure 3c shows the actual dialogue and figure 3d depicts the semantic representation where the nodes are objects or nouns and the arcs are their asserted relationships. The information being sought is the number of John's fingers All that is explicit in the network is that John has two arms, each arm has one hand, and each hand has five fingers. Once the necessary relations have been established, as the diagram shows, the program calls a procedure that is invoked whenever the task is to "reply as to how many elements of the set x are parts of the individually," where here x=fingers and y=John. The answer is computed by the procedure and the program reports it to be "10" [41].

Although semantic networks are not actually featured as data structures in any of the AI languages considered, it has become an important scheme for data base organization. This is true for the reason just cited as well as the fact that they are relatively easily implemented with the new list processing languages.



The inferencing capability of Quillian's nets may demonstrated in the example as different planes are shown to have intersections (live, for plant and food; sad, for cry and comfort). The claim is that the distance from the planes to their intersections is directly proportional to the relatedness of the concepts.

Semantic Network - Quillian
Figure 3b

(***. A Boy Is a Person)
(I Understand)

(***. John Is a Boy)
(I Understand)

(***. A Finger Is Part of a Hand)
(I Understand)

(***. How Many Fingers Does John Have ?)
(The above Sentence Is Ambiguous ** But I Assume (Has)
 Means (Has As Parts))
(I Don't Know Whetner Finger Is Part Of John)

(***. There Is One Hand On Each Arm)
(I Understand)

(***. There Are Two Arms On A Person)
(I Understand)

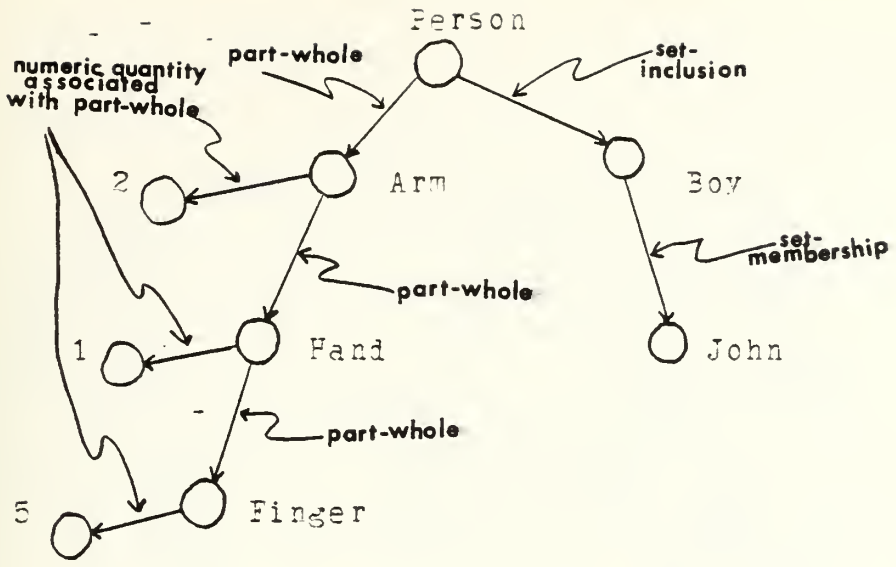
(***. How ManyFingers Does John Have ?)
(The Above Sentence Is Ambiguous ** But I Assume (Has)
 Means (Has As Parts))
((How Many Finger Per Hand ?))

(***. A Hand Has 5 Fingers)
(The Above SEntence Is Ambiguous ** But I Assume (Has)
 Means (Has As Parts))
(I Understand)

(***. How Many Fingers Does John Fave ?)
(The Above Sentence Is Ambiguous ** But I Assume (Has)
 Means (Has As Parts))
(The Answer Is 10)

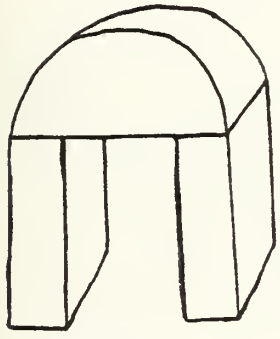
Semantic Network - Raphael.

Figure 3c

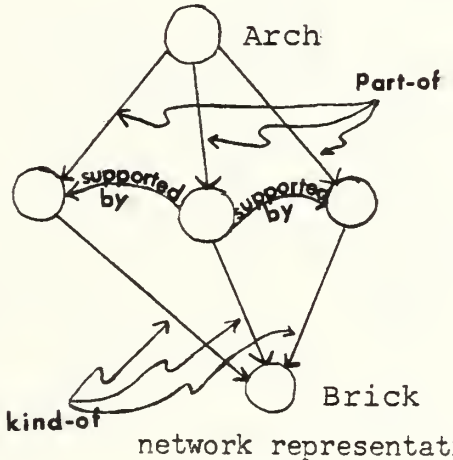


Semantic Net picture.

FIGURE 3d.



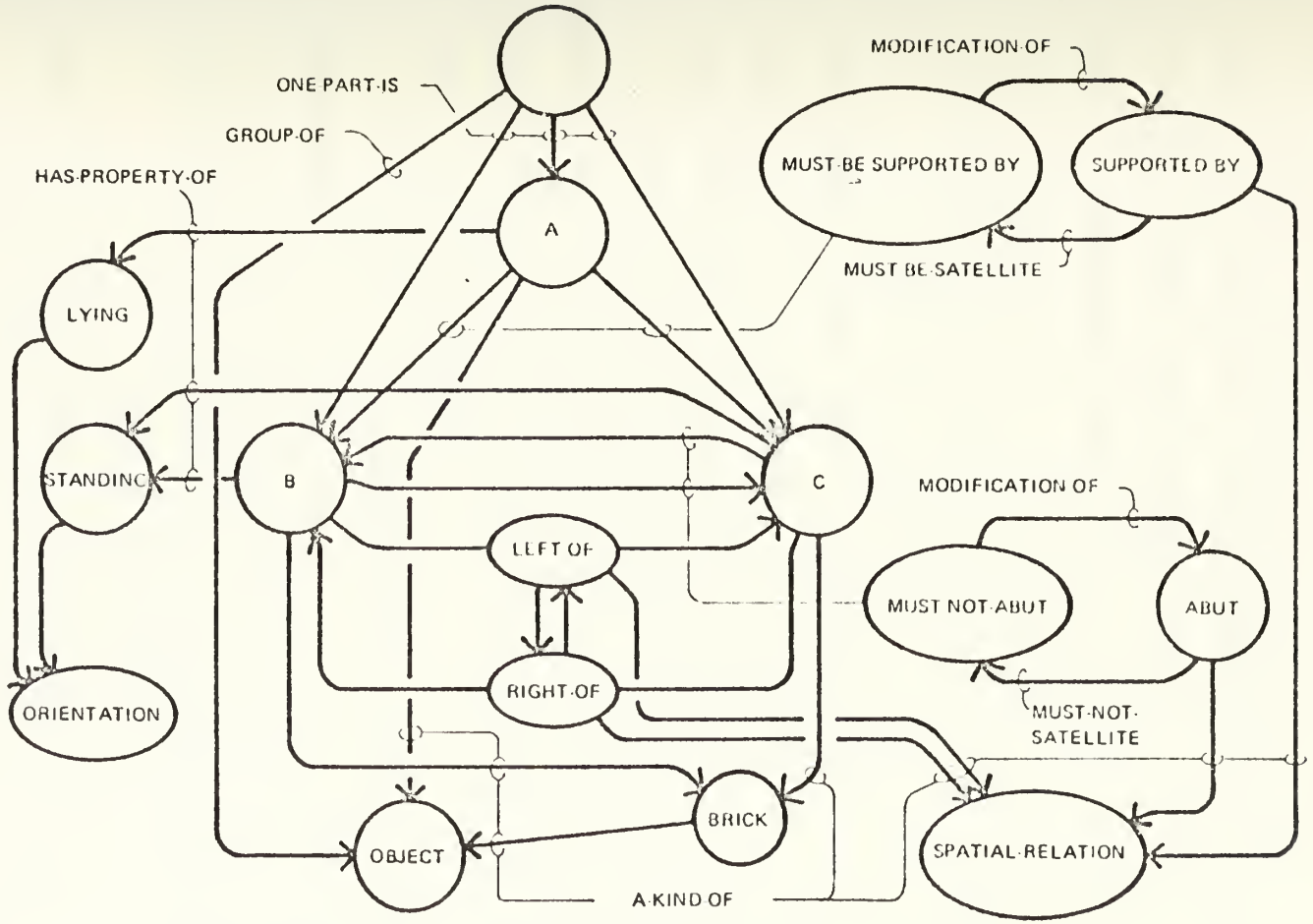
example of an arch



network representation

Semantic Net - Winston.

Figure 3e



Source: From Winston, ed., *The Psychology of Computer Vision*, Figure 5.62, p. 198.

Semantic Net-Refined (Winston)

Figure 3f

G. ASSOCIATION AND ASSOCIATIVE RETRIEVAL

Many of the new languages recognize that provision of the capability to store and access data associatively (i.e., on the basis of some property it possesses with no regard to name or location) a basic necessity for current and future research. CLISP, SAIL, LEAP, POPLER, and TELOS all provide for an associative data type and therefore, associative data base organization. This powerful mechanism is critical to the construction of goal-oriented systems, where procedures to be tried for reaching a goal are identified by description rather than by name. Furthermore, it enables the appending of procedural or declarative chunks to a knowledge-based data store. Subsequent use of these chunks is determined by some control abstraction (e.g., a demon, on the lookout for an instance of a certain situation) whose specification most likely preceded the entry of the chunks into the system [59].

In general, an association data structure is a three-tuple of data elements of the form item1 * item2 = item3 [18]. For specified sets of items an association is executable. For example, the following three-tuple:

MAKE * AUTOMOBILE = FIAT

could be analogous to the executable statement:

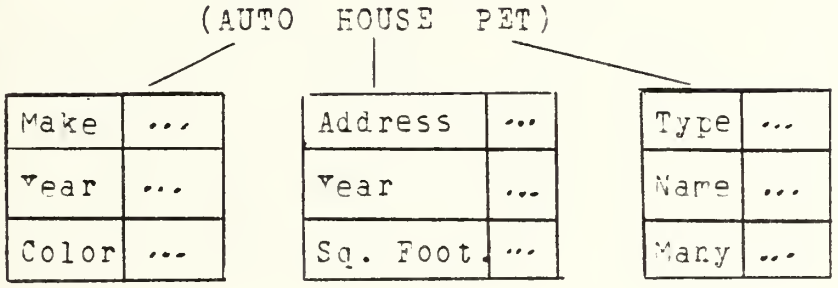
foreach x such that x in AUTOMOBILE do MAKE.x = FIAT

which instructs that all elements(x) in the set "AUTOMOBILE" will have their "MAKE" property set to "FIAT". Thus, the

primary interest is in attribute values of one or more set elements. In the example, AUTOMOBILE ('item2) is the set, MAKE (item1) is the attribute, and FIAT (item3) is the value of the attribute [4].

One obvious application of association occurs with the implementation of a property list as a list of pairs (property, value) linked to an object. However, from a user point-of-view this scheme is handicapped, as it is one-way. For instance, there could exist a situation where a list contains sublists of three atoms each, representing three things that a family might own such as an automobile, a house, and a pet. Each has a property list as shown in figure 3a. The problem arises when one has a value with no idea of what attribute it belongs to. The value could be a number like 1972. This could stand for the year of the auto, the year of the house, or the square footage of the house. Associative retrieval based on wanting to know the value for a specified attribute is acceptable since the order and determination of attributes is known for each p-list. There is no way to find an attribute based on an arbitrary value save to brute-force a pattern match through every cell of every p-list for each element of the set. Such a search would require an enormous (if not unreasonable) expense in bookkeeping and time. SAIL and LEAP, however, provide a storage scheme that allows such a "backward" associative retrieval operation on p-lists. It is done by designating a

continuous block of storage for each p-list in which each attribute is assigned a relative address. A hashing scheme on values is then used to store and retrieve the attribute data [18]. This setup allows the flexible access of items in a p-list either by attribute or value. However, it also implies a requirement of twice as much storage as usual.



Property Lists.

Figure 3g.

H. CONTEXTS AND FRAMES

There is not universal agreement on the meaning of the words context and frame. Some use them synonymously, yet others draw a distinction, albeit a subtle one. For instance, context represents an implemented programming mechanism, whereas a frame refers to a data structuring concept only. This policy is sufficient for the purposes of this discussion. In a rough sense a context may be thought of as a "viewpoint on a data base [47]." More specifically, it is a data structure representing previously stored chunks of stereotyped knowledge (e.g., driving to work, making a

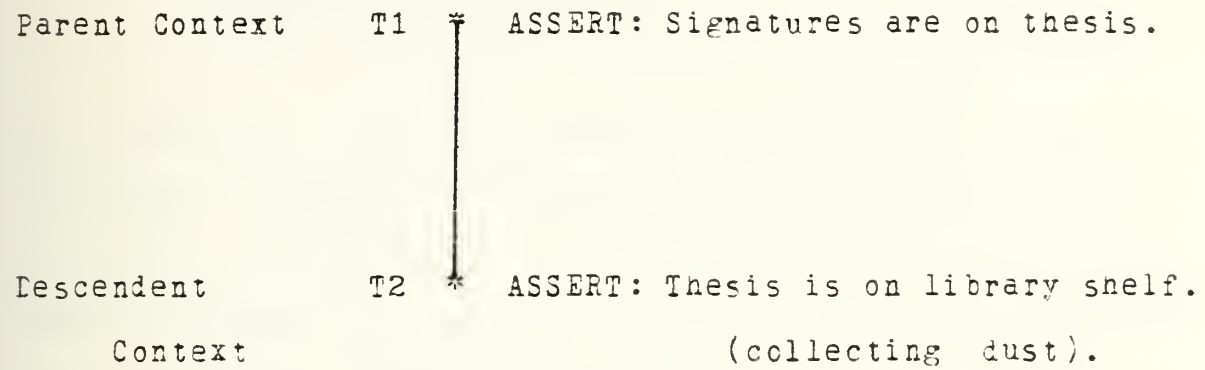
phone call, typing a paper, etc.) [9]. It is provided as a limited attempt to implement Minsky's complex frame data structure idea in QLISP, CONNIVER, and POPLER. In these languages, the mechanism works in the following manner: all expressions in the data base are factored into segments, each of which may be referred to as a "context." Assertions may be entered and subsequently retrieved from the data base with respect to a context. (This is analogous to a "block" in a block-structured language.) The context may be manipulated explicitly or by default (i.e., based on the structure of the flow of program control) [4]. For example, consider the CONNIVER program module:

```
1.        (PROG(Y)
2.        (SETQ Y 100)
3.        (SETQ VAR2 (TAG LABL))
4.        (SETQ VAR1 (FRAME))
5.        (PRINT 'HELLO)
6. LABL   (PRINT 'GOODBYE)
            .
            .
            .
7.        (SETQ Y 50)
8.        (PRINT (EVAL 'Y VAR1)
            .
            .
           )
```

The routine above demonstrates the use of two special commands offered in CONNIVER for manipulation of contexts. The commands are called FRAME which creates a pointer to the current frame so it may be modified or executed again later, and TAG which is similar but it also specifies a starting location within the frame [4]. With respect to the example, one could issue a CONTINUE VAR1 instruction causing processing to commence at statement 4. A CONTINUE VAR2 instruction would send control to statement 6. The CEVAL command in statement 8 would cause the value Y (=100) within the context of VAR1 to be printed. The mechanism in each of the three languages works in very much the same way, and provides the same principal value, that of allowing the programmer to consider alternative situations, without changing a global data base, by specifying the scopes of variable bindings, i.e., switching informational "contexts" or "frames" of reference. The frame structure has been referred to by many other names including "script" (Schank & Abelson), "units" (Bobrow & Winograd), "depictions" (Hayes), and "common sense algorithms" (Rieger) [9].

A new "context" is created, in a program, whenever a user-defined block is entered (or a CLAMBDA function is encountered in CLISP). The current "context" is set to be a descendent of the previous "context." This is known as "pushing" a context [61]. This means that variable bindings and assignment of properties to expressions that are local

to a context are accessible only from that context or a descendent context.



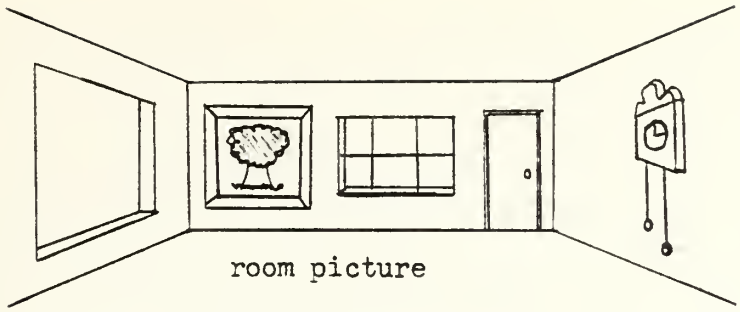
With respect to the drawing above, scope rules dictate that any assertion made with respect to T1 will be available to queries made with respect to T2. Thus, if asked whether signatures are on thesis, with respect to T2, the answer will be "yes". But, assertions made with respect to that descendent are invisible to queries made with respect to its parent, or any other context aside from its own descendents, For instance, if thesis is asserted to be on the library shelf, with respect to T2, that information will not be available to queries made with respect to T1 [61].

Marvin Minsky was the first to seriously put forth the concept of a frame data structure. He suggests a frame be thought of as a network of nodes and relations. He further explains his idea with the following elaboration:

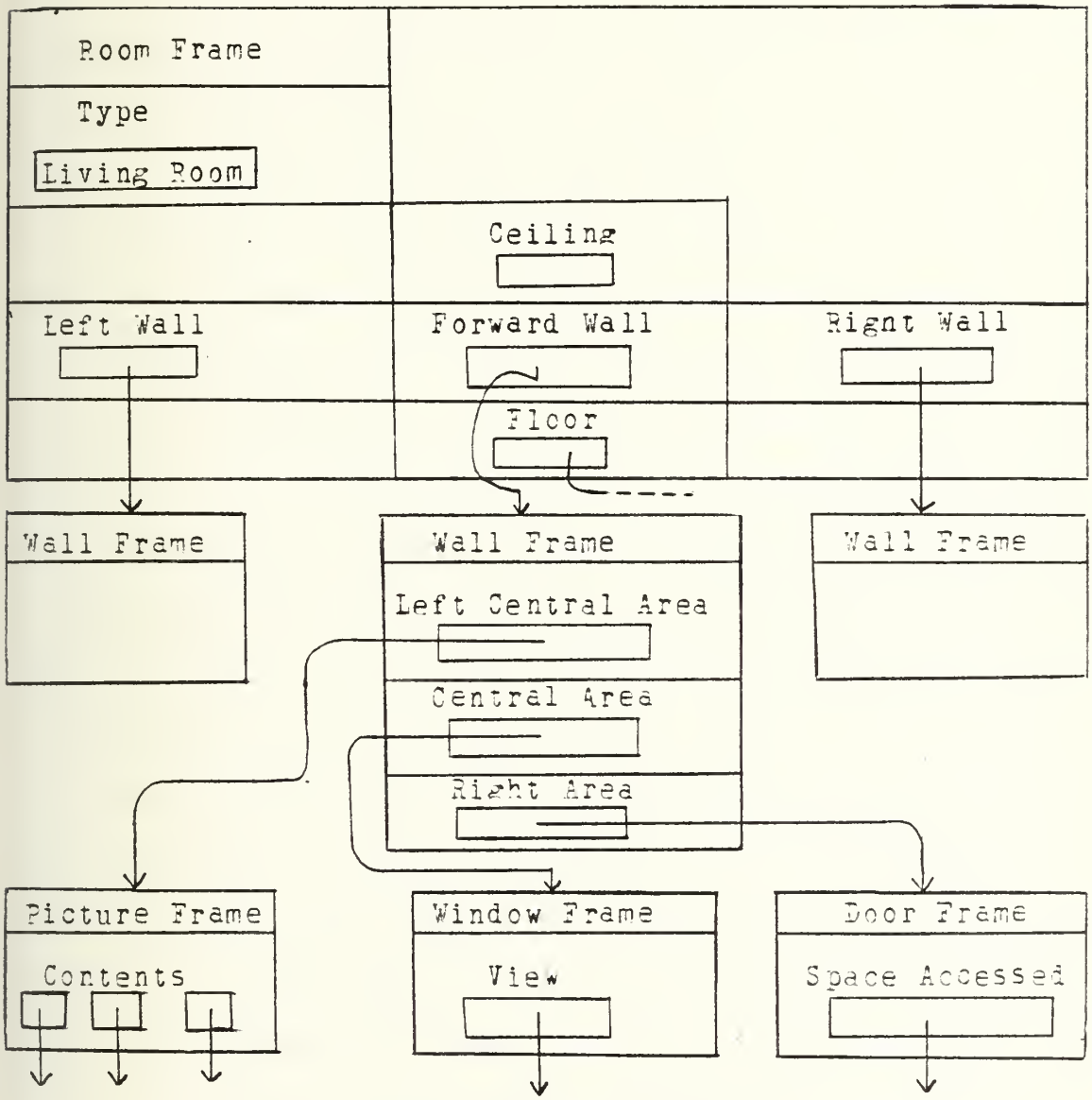
> The "top levels" of a frame are fixed, and represent things that are always true about the supposed situation. The "lower levels" have many "terminals"-"slots" that must be filled by specific instances or data. Each terminal can specify conditions its assignment must meet [32].

The "conditions" can be either simple (i.e., requiring assignment be made to a person, object or pointer to another assignment) or complex (i.e., a relation existing among some things assigned to several terminals) [64]. All terminal nodes are loosely assigned default values thereby conferring on a frame many details whose supposition is not specifically warranted by the situation. These assignments may be easily displaced by new items that are discovered to better describe the current situation.

An example of a typical use of the frame concept is borrowed from Winston to how a room could be described with a frame in figure 3h [64]. A frame may contain subframes. That is, a frame "slot" may simply contain a pointer to another frame. In the figure each frame slot for a wall contains a pointer to a wall frame, which may again contain subframes.



room picture



Room Frame - Winston.

Figure 3h

The frame concept has stimulated an area of current investigation namely, the "frame problem." "This is the problem of creating and maintaining an appropriate informational context, or frame of reference, at each stage in certain problem-solving contexts [42]." The essence of the "problem" is bookkeeping. There are an enormous number of details to keep track of in the hypothetical worlds resulting from alternative actions which could be taken in order to carry out a given task. Much of the problem is due to complexity. For example, a typical situation might require a frame description of 1,000 elementary facts. It is not unreasonable to think that an average of 6 different actions is plausible in any situation and that 4 successive actions are required to achieve a goal state. This implies that there are $6^4 = 1296$ possible intermediate and terminal situations to be considered. Storing 1,000 facts to describe each of more than 1,000 situations means storing over one million facts, which is not yet a feasible thing to do if all facts must be available in main memory [44]. To relieve this problem current research is concerned with investigating the possibilities of using state variables (each node would carry only change information). This allows the incorporation of the "STRIPS assumption [61]" into context mechanisms. The STRIPS assumption is that an action leaves all the relations in the model unchanged, unless otherwise specified. Generally, the specification of change

is made in "add lists" and "delete lists" which accompany the application of an operator to a situation. For example, consider a situation where a car with a driver and a sleeping passenger is going down the road. The state may be represented by the assertion of all the following predicates:

MOVE (car forward)
HAS (car fourdoors)
IS (car red)
AT (driver steeringwheel)
SIEEPS (passenger backseat)
MILES (car 60K)
IN (sparetire trunk)

Now apply the Flat Tire operator. It has add and delete lists as follows:

ADD LIST	DELETE LIST
AT (driver tire)	AT (driver steering wheel)
IS (driver mad)	MOVE (car forward)
MOVE (car nowhere)	IN (sparetire trunk)
On (sparetire axle)	
In (flattire trunk)	

The application of the flat tire operator creates a new state which is different from the first as prescribed by the add and delete lists for that action. The entire model need

not be regenerated since it is known to be the same as before with the exception of a few modifications. In the example above for instance, without having to be explicitly stated it is known that the passenger is still asleep in the backseat. The resulting state is analogous to a descendent context.

Similar to a frame is Schank's notion of a script. Whereas a frame usually depicts a static scene, a script is used to represent an event, something that includes an implicit time ordering. Figure 3i shows a script generated by Charniak about grocery shopping.

The advantage of a frame or script-type representation is that it will contain information expected to construct a specific ordinary event whether static or dynamic. A program operating on a frame-oriented data base need be told explicitly only that information which is not accurate in the frame. For example, a shopper may not use a basket at a grocery store. The slot for that aspect is loosely bound to the value that is initially installed there. Both frames and scripts may be stored, retrieved, and modified. Retrieval may be done via a pattern match on a goal (frame heading) or via a link from a frame slot to another frame or frame slot.

A great deal of work remains to be done in order to fully realize the details of Minsky's frame abstraction. But, any progress made along that continuum will contribute to the simplification of inherently complex programming

tasks as one would find in a question-answering syster or a problem-solving system.

```
GOAL: SHOPPER owns PURCHASE-ITEMS
SHOPPER decide if to use basket,
   if so set up cart-carry Frame Image
SHOPPER obtain BASKET  *cart-carry
SHOPPER obtain PURCHASE-ITEMS
   method-suggested
   DO for all ITEM  PURCHASE-ITEMS
    SHOPPER choose ITEM  PURCHASE-ITEMS-DONE
    SHOPPER at ITEM
      side-condition DONE at item also
         method-suggested
         cart-carry(SHOPPER,BASKET,DONE,ITEM)
    SHOPPER hold ITEM
    ITEM in Basket *cart-carry
    DONE<-DONE+ITEM
   END

SHOPPER at CHECK-OUT-COUNTER
      side-condition PURCHASE-ITEMS at CHECK-OUT COUNTER also
         method-suggested
         cart-carry(SHOPPER,BASKET,PURCHASE-ITEMS,
                    CHECK-OUT-COUNTER)
SHOPPER pay for PURCHASE-ITEMS
SHOPPER leave SUPERMARKET

* means get the following frame if appropriate.
```

Supermarket Script - Charniak

Figure 3i

I. DEFINITIONS FOR DATA STRUCTURES

1. Tree

A "tree" is similar to a natural tree except that it is drawn upside down. Technically, it is a directed graph in which each node has at most one predecessor. The root node (at the top) has no predecessor ever. A node with no successor is called a terminal node. The lines (all having a downward direction) connecting nodes are called arcs.

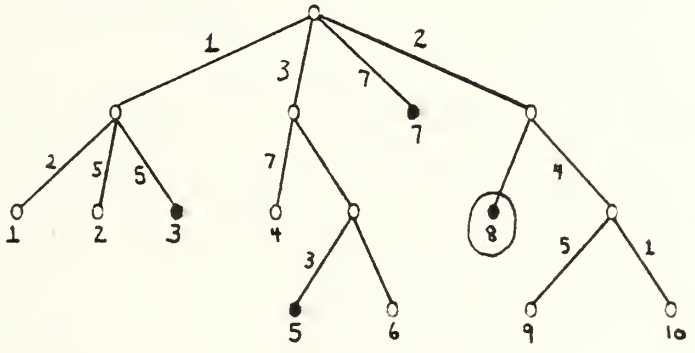
Each node is associated with a level of the tree. The level of a node is one more than its distance from the root node. All the nodes on each distinct horizontal plane represent another level. The depth of the tree is equal to the number of levels it has.

Trees may be used to conceptually represent the organization of a data base, or the search space for a problem.

Tree nodes are examined for solution in a certain order. The most common orderings are breadth-first, depth-first and best-first. Breadth-first examines the nodes on each level, starting at the root, in a left to right direction usually. Depth-first search, on the other hand, travels all the way down the left most branch of a tree to the end, or as far as a pre-specified bound. The search successively makes its way rightward always going down the left most branch till a solution is found. When there is a

cost associated with each arc and the problem is to find the least cost goal node, best-first search is the most efficient method, although it is expensive in terms of storage. This method will choose to explore past a node if the cost on any arc leading out of that node (added to the total cost so far on the path) is less than or possibly equal to the cost of a path leading into another node at the same level. current least cost path. The first goal node found will be the one at the end of the total least cost path.

Figure 3j shows a tree with cost values on the arcs. The goal nodes are blackened. Results from each of the three search search methods is shown also.



Order of Discovery of Goal Nodes:

Depth-first	Breadth-first	Best-first
3	7	8
5	3	-
7	8	-
8	5	-

Evaluation of a Tree

Figure 3j

2. Atom

Atom is a term derived from LISP. It is a symbol used to name an object, such as a variable, function, or some object in the world in which the program is operating. In a robot world an atom could be a table, or box. An atom is the smallest meaningful unit of information. The following examples could be atoms in a program: A, X9, SQRT, WINDOW, FROG, etc.

3. S-expression

This is a LISP term that is used when referring to lists or atoms in general, and neither in particular.

4. Garbage Collection

This is a facility some languages offer for examining memory at periodic intervals to determine which locations are not currently in use and returning these to the free list. This is done by following the chains of pointers from active list variables. Any cells which cannot be reached are considered no longer needed [66]. When this mechanism is supplied by the language the programmer is freed from having to keep track of used and unused memory which in a list processing language without garbage collection requires very careful and attentive programming. See also [54].

IV. CONTROL STRUCTURES

Control structures are an integral part of the programming environment. They provide the framework or operations for specifying rules directing the flow of processing and interpretation for programs or parts of programs [21]. For example, a control structure might include a general rule that dictates that statements be executed one at a time in order as they appear in the program description. A discussion of control then must also include consideration of program sequencing rules (or the lack of such) or even the indeterminancy of the desired sequencing.

In early programming languages, flow of control was strictly hierarchical, meaning every module (procedure, function or any primitive program unit) was expected to complete its task before returning control to the module that activated it [4]. Processing occurred sequentially except when interrupted by iteration or conditionals or branches like go to's [21]. Such limitations on flexibility were too constraining when it came to programming complex AI systems that accessed and manipulated very large stores of knowledge and more often than not, required subsequent program control to be dependent on the result of an access or manipulation

It is little surprise then to discover that with the advent of the "first" AI language , IPL, a new control structured was introduced.

IPL was originally designed to implement the LOGIC THEORIST (LT) theorem-proving system of Simon and Newell. LT's task was to prove theorems in propositional calculus. In so doing it would iteratively apply a variety of logical rules to the proposition. A mechanism was needed to halt the application and pursue proof of a subproblem and then possibly continue where processing had earlier left off. The "generator" was that mechanism and it enabled repetitive, non-hierarchical processing. It would produce a sequence of outputs to which it iteratively applied a specified process. Any iteration producing a false (boolean) value could terminate the generator, but not before ensuring a possible later reentry at that point[21].

Of course IPL was also a list processing language and as such, one of its basic control structures was recursion, which is a naturally hierarchical structure. Each recursive call creates a separate activation at a new level.

COMIT, a string processing language that appeared about the same time as IPL, was characterized by a primarily sequential program flow. A program was simply a prioritized list of rules for replacing characters in string data. However, only the rules that contained a specified pattern in a substring of the data were executed.

Much of the power and sophistication of the control structures of AI languages, over and above what is available in the conventional programming languages, is due to the inclusion of a variety of deductive mechanisms. These are mechanisms that provide an automatic or at least semi-automatic data base search capability. The more automatic they are the more nonprocedural the language turns out to be. The most nonprocedural language is PLANNER and it indeed offers the most automatic deductive mechanisms. PLANNER's primary tool for deduction is the consequent theorem. It has two parts, a pattern statement (P) and a procedure body (Q) which are employed in a Q=>P manner. Thus if the program Q were successfully executed then the assertion matched by the pattern P would be proven [4]. Frequently the program Q requests that an assertion be proven, i.e., that a subgoal be achieved. The consequent theorem may thus set up a backward chaining mechanism for searching the data base. For example, consider the situation wherein a goal is to assert that there are giraffes in San Diego:

GOAL(IN GIRAFFES SANDIEGO)

In the data base might be found the following assertions:

(IN ZOO SANDIEGO)

(IN GIRAFFES ZOO)

The assertion desired is not explicitly available so the next step is to find a consequent theorem that would enable

its derivation. The theorem below would do just that.

```
[CONSEQUENT
 (IN X Z)
 (GOAL IN X Y)
 (GOAL IN Y Z)]
```

The first step is to match the target pattern (IN X Z) against the desired assertion (IN GIRAFFES SANDIEGO). The body of the consequent theorem instructs that two subgoals be achieved. The data base is searched for an assertion that matches (IN GIRAFFES ?Y) where ? means that any value found in this position will be acceptable. In the data base is found (IN GIRAFFES ZOO) which matches. One subgoal has been achieved. The second causes a data base search to find a match for (IN ZOO ?Z). It finds the fact that (IN ZOO SANDIEGO). Thus the consequent theorem has been successfully executed and the assertion has been proven to be true. Pattern matching capability is fundamental to the success of this type of deduction.

Other languages that offer mechanisms as described above include CONNIVER, QLISP, and POPLER. In all four languages the data base search is initiated by an explicit goal statement. The differences between them is the amount of programmer interference allowed in constraining the search. In PLANNER there is little or no opportunity for the programmer to explicitly direct the search process whereas

in CONNIVER and QLISP there is a great deal. These two languages provide a frame construct which specifies different data and control contexts. A new frame is created each time a non-atomic expression is encountered. The frame contains all the information associated with every activation of the access module in which that expression resides. Furthermore, the programmer has access to all components of a frame including links to module continuation points, bound variables and free variable environments.

The CONNIVER analog to the PLANNER consequent theorem is the IF-NEEDED demon, for QLISP it is the combination of the 'is' and 'cases' commands, where 'is' directs the data base search and 'cases' performs pattern directed execution. Likewise POPLER offer similar features in the form of the 'achieve' and 'infer' commands which are each a form of consequent theorems. 'Achieve' will attempt to assert a specified goal whereas 'infer' will merely check the status of a specified goal. Basically, all these mechanisms provide a capability to "deduce desired logical expressions from previously specified expressions in a manner similar to the deducton of theorems from axioms in theorem-proving programs [4].

In keeping with the philosophy that "use of control structures better suited to a programming task can simplify that task and expose the significant problems in that problem," a variety of control structures for AI programming

languages suited to a variety of AI tasks have been developed and refined continuously since the contributions of IPL and COMIT were made [21].

The remainder of the chapter describes many of the new structures employed in AI programming, and discusses their implementations in the AI languages. The following structures and concepts are included: Data-driven programming, demons, coroutines, procedural networks, chaining, backtracking, deduction systems, pattern-directed invocation, and discrimination networks. Some miscellaneous terms and concepts are defined at the end of the chapter.

A. DATA-DRIVEN PROGRAMMING (DISPATCHING)

> Data-driven programming is a style of programming in which the boundary between programs and data becomes even more blurred than usual [64].

Data-driven programming is a method of attaching programs to data, thereby relieving the programmer of the detail of predetermining much of the program control. The incoming data actually takes control by causing invocation of pre-specified, data-type dependent routines for application and execution [9].

This may be done by supplying a calling procedure with the user or argument input data that has been associated with procedures in the data base. This causes those procedures to be invoked and executed [51]. In particular,

in LISP this is done by storing type specific functions in the form of LAMBDA expressions on the property lists of type-describing atoms. These functions are then automatically fetched later when an argument of a given type appears.

A simple example may be contrived to demonstrate the idea of attaching programs to data. Consider the situation in which one would like to assert the relationship COMPONENTS(finger, hand, 5) or COUSINS(mary,bob). In order to do so, a general purpose procedure 'store', with its arguments, must be called. This procedure will allow the programmer to assert different relations, each type of which may very well require a different method of storage into the data base. For example the 'components' relation may use a hashing scheme, whereas the 'cousins' relation may be pushed onto a stack. In any case, there might be several different types of relations that are asserted at various times. It is highly desirable then to have a general function, in this case 'store', that would take as arguments the relation and its arguments. This function would then proceed to look on the property list of that relation to find and execute the associated storing function. In a situation such as this the general function (e.g., store) is said to "dispatch" to (i.e., send off or away with promptness) procedures associated with relation names.

A benefit of data-driven programming is that program

surgery is avoided whenever additional information is made available to the system [64]. If a new relation were created or an old one altered in the example it would be taken care of via property lists and the program code need never be changed. Furthermore, it is a necessity for the implementation of pattern directed invocation and demons that are invoked on the basis of recognition of a pre-specified data type [51].

B. DEMONS

The idea of a demon is well-known in many other programming areas besides AI, e.g., operating systems development. A demon is a procedure that is automatically invoked when a pres-pecified event takes place, such as a condition becoming true, a certain value being changed, or some relation being altered in a certain way [22]. Most commonly they are invoked by data base additions or removals. Activation occurs with a successful match between a data item and a pattern associated with a demon [64]. Frames are in essence a demon-based scheme. A frame may provide the capability of a demon by containing a pattern that invokes a certain procedure. Programming languages that offer demon-like facilities include PLANNER, CONNIVER, QLISP, and TELOS.

In QLISP, the programmer is allowed sufficient flexibility to design an efficient system in which a demon

is activated only at appropriate times, or a more carefree system wherein the demon is invoked at every opportunity [4]. Groups of functions may be defined to act as "teams" of demons that may be associated with an arbitrary database storage or retrieval command.

In PLANNER, there are the antecedent and consequent theorems. The former are, in effect, demons that act independently to add and delete facts to the data base as other facts arrive. An antecedent theorem or demon represents a fact in the form of: pattern (p) and body (b). Whenever anything is asserted (entered into the data base) all antecedent theorems are checked against the data base. If the recent assertion matches the p part then the b part is immediately executed. Consequent theorems, on the other hand, are not really demons but rather "fact-finders." They are deductive mechanisms used to establish facts that are comparatively unimportant and that by not having been asserted do not clutter the data base. Their information is easily derivable. Similarly, CONNIVER has IF-NEEDED and IF-ADDED demons that are analogous to PLANNER's consequent and antecedent theorems.

In TELOS, the use of demons occurs "in a clearly demarcated textual scope." A demon-like event mechanism is used to allow the programmer to detect and trap error conditions and, if desired, to limit the effects of their scope during execution [59].

Demons are used, among other places, in story understanding programs that attempt to make inferences while reading. For example, a demon with the pattern "person P is outside" could contain in its body the fact:

If it is (or will be) raining and P is outside

then P will get wet.

While reading, the program may come across the sentences:

It was raining. Jack was outside.

The first sentence would likely invoke a routine (called a base routine or high level demon) about rain that alerts all demons related to that topic. The next sentence would then invoke the specific demon described above.

An example of a simple demon in CONNIVER that is invoked whenever person P loses his or her job may be defined to alter the content of the data base in the following manner:

```
DEFINE DEMON
IF-ADDED
(P LOSES JOB)
(Remove 'P is-a Happy Person)
(Add 'P is-a Sad Person)
(Add 'P has-no Steady Income)
(Add 'P pays-no Income Taxes)
END DEFINITION
```

Statements may be added at the end to have the demon deactivate itself when it is through or no longer needed.

Demons are very useful as they "add knowledge to a system without specification of where it will be used [64]." They are easily attached and encapsulate the bookkeeping operations that otherwise litter programs. Thus, programs are more readable.

C. COROUTINES

Coroutines are "specialized control mechanisms for situations in which the natural division of a process into subtasks is not hierarchical [21]." Three program structures that are analogous to coroutines, and serve to demonstrate their appropriateness are: 1)Mutual subroutines, a simplistic point of view wherein each of several procedures are to be written so that each procedure may call the others as if the others were subroutines, 2)The most characteristic perspective of coroutines is that of procedures with 'own' storage such that a procedure's variables retain their values between calls. The 'own' variables retain not only local procedure data, but also the state of processing within the procedure so that processing will continue from that point at the next invocation, and 3) Symmetrical control achieved by separating programs into logically disjoint parts and describing each part separately. Their various stages of execution are then interleaved [21]. This

last view may be considered more as a side-effect of the implementation of a coroutine regime than analogy.

Any implementation of coroutines implies that at each point of invocation the calling routine, which was executing, is suspended, not to be resumed except by explicit call, and the routine which is called is resumed at the point from which it last left off, with all of its internal variables unchanged, that is, the previous state is restored [52].

The advantage of using coroutines is that each of several processes can be described as a main routine (vs. subroutine) with minimal concern for the interface with other processes [21]. A coroutine may be given more than one continuation point and use only one of them depending on the result of some test [9].

A simple example would be producer-consumer situation involving two routines A and B. Routine A's job is to search a data base for a certain type of expression or datum. When it has found one not previously found it fetches it, leaves it in a certain location, suspends itself and resumes the processing of routine B. Routine B's job is to fetch from a certain location an expression on which it is to perform some computation. Once this is done it suspends itself and resumes processing of routine A. Routine A continues where it left off in its data base search and fetch. This back and forth processing continues until A can no longer find

expressions or until B has performed a pre-specified amount of computation. Both routines in this situation operate as though each were a main routine.

In TELOS, there is a mechanism (function) called NewProcess which provides a coroutine capability. NewProcess creates a coroutine as a process with specified actual parameters. The value returned by NewProcess is a reference to the newly created process. The state in which the new process begins execution will be that which was operative when NewProcess was executed [59].

In CONNIVER, there are "generators" which are a kind of coroutine process. A generator will generate requested data items, one or a few at a time, to which some specified process will be applied. The environments of the process and the generator are independent and not hierarchically defined. If more data items are later requested, the generator resumes in the data context of the previous request. An analogy would be a FOREACH statement wherein, the head would represent the generator and the body of the statement would represent the specified process. But, imagine that that particular body may be interchanged with another body perhaps of another FOREACH statement in another data context. Then execution would continue in different data and control contexts for the two parts.

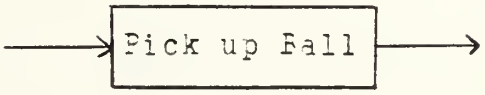
SAIL also provides a limited coroutine capability. With the use of a command called SPROUT a new process may be

created to commence execution at the point it is created and run parallel to the process that executed the command. True parallelism is not possible on a single processor computer so the SAIL run time 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. The RESUME command will suspend the current process and reactivate a named suspended process. But, no capability exists for one process to pass any information to another except via side effects.

D. PROCEDURAL NETWORKS

Procedural nets are usually used for representing plans created by problem-solving systems. A problem-solver such as STRIPS (see Appendix B) will generate hierarchical plans beginning at the highest level of abstraction of the goal state to be achieved until a sufficiently detailed level is reached to allow execution.

The simplest possible procedural net consists of a single node with two directed arcs, one leading in and one leading out; this would specify the single step in a one-step plan.



A refined procedural net describing a problem involving reconfiguring arrangement 1 to get arrangement 2 is shown below. In the problem, only 1 letter may be moved a distance of one space in any direction that an empty space exists.

arrangement 1:

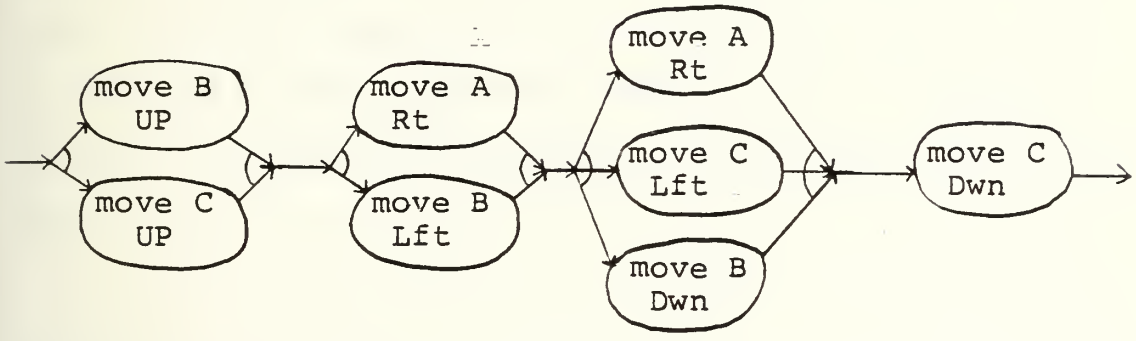
INPUT ORDER(A B C)

A	B	C

arrangement 2:

GOAL ORDER(B C A)

B	C	A



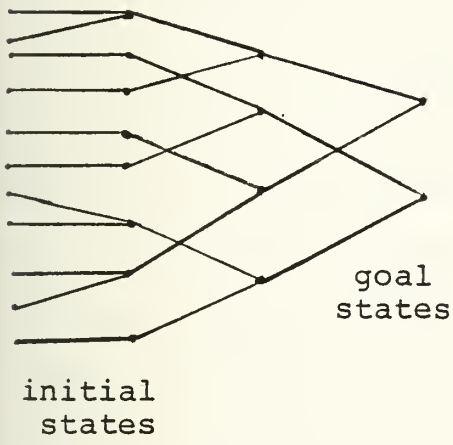
(note: steps stacked vertically may be done in any order.) On more complex nets the nodes are the steps or actions to be taken, whereas the edges (directed arcs) imply the sequencing of the steps.

An important feature of procedural nets is that when a plan is generated it is often unclear what will be the best order to carry out some of the steps. The order is allowed to be left unspecified until some later time when the correct ordering does become clear [23]. This may not take place till the plan is actually executed. Languages which allow this type of feature include PLANNER, CONNIVER, CLISP, and PROLOG.

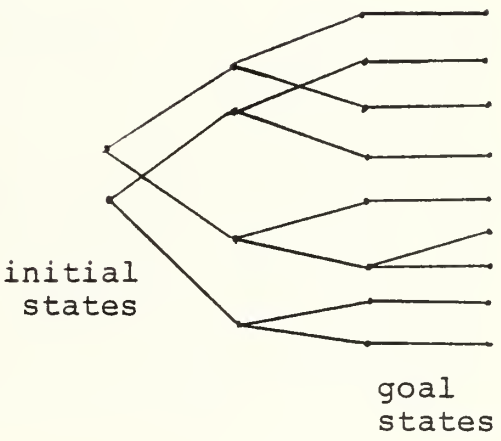
E. CHAINING

There are two types of chaining: forward and backward. If the goal of a search is to discover all that can be deduced from a given set of facts then forward chaining is appropriate. Progress is made as the system works from and initial state toward a goal state. The shape of the state space is fan-in. Figure 4a. is an example of such a space.

A forward chaining technique (means-end analysis) is employed in GPS (see Appendix B). At the start there are a number of differences existing between the initial state and goal state. At each point GPS selects what it figures to be the most prominant difference and attempts to eliminate it [64].



Forward Chaining
(Fan-In)
Figure 4a



Backward Chaining
(Fan-Out)
Figure 4b

Backward chaining works in just the opposite manner. First of all, the shape of the state space is fan-out, as in figure 4b. The goal is to verify or deny a particular conclusion. Thus, progression is from the goal state toward the initial state. Often backward chaining will lead to dead ends, i.e.,subgoals which cannot be deduced from information available in the database. In that case, a return to the state that induced the subgoal takes place and a new subgoal is selected, if one exists[64].

Backward chaining techniques are employed in rule-based systems [27] such as MYCIN and DECAIDS, in PLANNER-like language systems, and in theorem-proving systems, such as BUILD and HACKER (see Appendix B)

F. NONDETERMINISM and BACKTRACKING

These two programming concepts go hand-in-hand. Backtracking is usually applied to nondeterministic programs. A nondeterministic program is one that may have its solution space represented as a search tree, where each terminal node is a potential solution. The task is to traverse the tree starting at the root node, and find a solution node [21].

Backtracking is basically an exhaustive depth-first search procedure. The algorithm to traverse the tree is executed, making choices at each node regarding what path to take next according to some choice function. When a solution

node turns out to be a failure, the algorithm "backs up", restoring all variable values to the last choice point at which one or more feasible choices remain, indicating alternative paths [21].

Backtracking has been considered as a special coroutine regime (in the sense that a coroutine is characterized by the ability to suspend a processing state of a routine that may then be resumed at a later time where it left off). Instances of the program environment (called modules) are saved at the choice points, and restored in a LIFO sequence when subsequent modules "fail" [4].

In PLANNER, a nondeterministic language, backtracking is provided completely automatically. The advantages of this are that the programmer does not have to worry about writing such a complex search strategy, and the program text becomes relatively easy to read. However, this latter advantage can turn into a disadvantage when the programmer is unable to infer from the text precisely what is happening. For example, a typical task for the SHRDLU problem-solver (see Appendix B) written in Micro-PLANNER, a subset of PLANNER, could be to "locate the big, blue block, or the small pyramid shaped one." SHRDLU would have to set up the following three goals:

1. Look for a block.
2. Check to see if it is blue or pyramid shaped.
3. Check the size.

If the block is blue the program will proceed to determine if it big. If not, it will back up to the previous decision point to see perhaps if it pyramidal. If it is, it will check the size again since the result of the previous check is unavailable now. If the test fails again the program will backtrack again to get another block this time. Backtracking must be used here as there is no way to determine in advance which block, B1, B2, B3, etc. is the one which will have the three properties. This tends to encourage the construction of programs that rely too heavily on blind search. An additional disadvantage of automatic backtracking in PLANNER is that there is no way to determine why a particular failure occurred, as the environmental context at the point of failure is deleted[5].

The development of CONNIVER was an attempt to retain all of PLANNER's good features and replace the bad ones, namely that of automatic backtracking. CONNIVER provides a capability for a failed routine not only to tell a higher level module why it failed but to even pass on information about the successive world changes it encountered along the way [5]. The power of this facility allows systems to be implemented in CONNIVER that attempt to learn from their mistakes. This is not possible with PLANNER. The problem-solving systems HACKER (written in CONNIVER) and SHRDLU (written in PLANNER) are examples of this difference (see Appendix B).

PROLOG provides for a backtracking capability. Since PROLOG is a predicate logic-based system, it searches or tries to deduce a clause to match a goal clause. When it fails and rejects a clause the original goal clause is reconsidered and an attempt is made to find a subsequent clause which also matches the goal [59].

G. PRODUCTION RULES and DEDUCTION SYSTEMS

Production rules, as they occur in deduction systems, are general statements in the form of implications, which the system uses to deduce new or implicit knowledge. Conceptually a rule would have the format

antecedent => consequent.

The antecedent part of a rule (lhs: left-hand-side) represents a set of assumptions or conditions, and the consequent part (rhs: right-hand-side), a set of goals or actions [27]. Thus, the control structure amounts to a simple "recognize-act" paradigm.

> Each rule is designed to be ideally, an independent chunk of knowledge with its own statements of relevance (either the conditions of the lhs, as in a data- driven system, or the action of an rhs, as in a goal-directed system [10].

The significance of the production rules is revealed when they are considered together with a data base of facts and an interpreter for the rules. With this combination, a

deduction system may be constructed. For example, the deduction system below called DS.CV has two productions, DS1 and DS2, and one fact, F1, also found in the data base.

DS.CV: (DS1, DS2)

DS1: ((car is FIAT) and (type is Sportscar) =>
(car is Spyder))

DS2: ((make is Foreign) and (size is Small) =>
(car is Sportscar))

F1: FIAT is foreign-made.

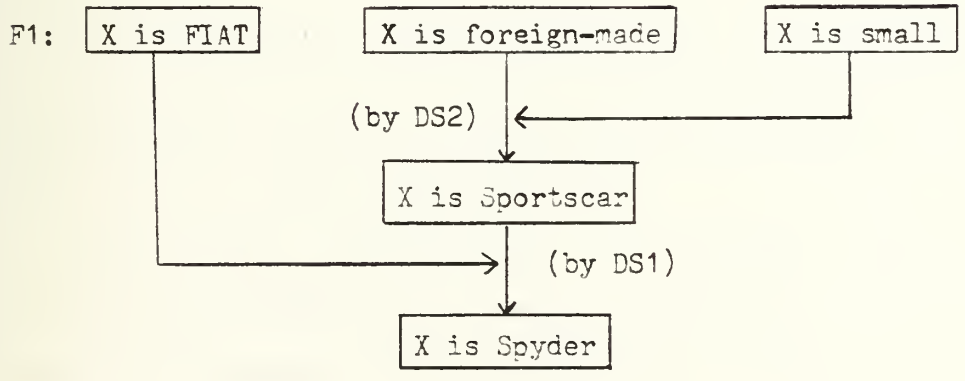
Given that we have a small FIAT, we would like to determine if it is a Spyder. This can be done by applying the rules and the facts in the following manner:

1) Apply F1 => foreign-made

2) Apply DS2 => car is a Sportscar

3) Apply DS1 => Spyder (which was what we wanted).

This graphs into:



Any conclusion is usually derived from many productions. The A conclusion in an individual production follows from a

conjunction of the facts,all asserted to be true, in the antecedent part. A conclusion reached by more than one production is done so through the disjunction of those productions. This translates into an And/Or tree representation of the productions and conclusions. Figure 3c is an example of such a representation.

Working from known facts to new, or deduced facts (as in the above example) utilizes forward-chaining. However, a deduction system can also employ backward-chaining by hypothesizing a conclusion and using the productions to work backward toward an assertion of all the facts necessary to support the hypothesis [64].

The advantages of deduction systems include:

1)They seem to provide a decent model of human problem-solving behavior. Humans informally deduce much from their own observations, experiences and education.

2)The knowledge base provided by all the productions may grow incrementally with the simple addition of new productions.

3)Data-driven programming is induced. Thus, a piece of knowledge (that is, procedure) may be applied whenever it is appropriate without having to have been planned for ahead of time.

The second advantage can become more a problem if the

growth of the knowledge base is allowed to get unreasonably out of bounds. A way to prevent this is to partition the facts and productions into subsystems (based on some relationship) so that at any given time only a manageable number are available.

PLANNER, CONNIVER, and POPLER are nicely suited for implementation of a deduction system. PLANNER provides consequent theorems, CONNIVER offers IF-NEEDED demons, and POPLER has a condition-consequent conditional. All are analogous to the "situation=>action"-type production needed to represent knowledge in the data base. Also, PLANNER's antecedent theorem and CONNIVER's IF-ADDED demon will add and delete facts from the data base as new facts are introduced.

Deduction systems are being well-exploited for the development of knowledge-based expert systems (e.g., MYCIN, DENDRAL, DECAIDS, CRYSALIS, and PROSPECTOR, see Appendix B), where formal and informal knowledge of the domain, and even the way the expert thinks in that domain (e.g., MYCIN), is crystallized into a set of production rules [10].

H. PATTTERN-DIRECTED INVOCATION and RETRIEVAL

Pattern-directed invocation of functions and pattern-directed data retrieval are deductive techniques that have recently become quite popular and for some tasks absolutely necessary. The phrase "pattern-directed invocation is somewhat self-explanatory. It refers to the method of invoking procedures indirectly, i.e., based on a pattern match, rather than an explicit call to a location. The match takes place between the desired output and and a procedure header. Pattern-directed data retrieval is based on a similar notion. One might retrieve an assertion (fact) from the data base by matching a pattern containing the known components and their order against all the assertions in the data base. In this manner large stores of symbolic data may be manipulated efficiently.

The essence of these techniques is a pattern-matching algorithm which will perform symbolic expression comparisons,atom-by-atom. It will allow an expression to be specified as a template against data items, in the case of data retrieval. An example of this idea may be taken from CONNIVER, whose pattern matcher may be used on arbitrary LISP data. It is desired to fetch items from the data base that match the pattern (template):

((EUREKA ?X) ?REST)

The above template is found to match both

((EUREKA HUMBOLDT) COUNTY) and

((EUREKA I'VE) FOUND IT).

generating the association lists

((X HUMBOLDT) (REST (COUNTY))) and

((X I'VE) (REST (FOUND IT)))

where X and REST are specified as open pattern variables. They are so indicated by the prefix "?'. In general pattern variable must have type declarations which are done via special prefix symbols [4].

> Templates provide a very flexible and easily understood way to specify the form of data items required, without the user having to be aware of how the items are stored [4].

In the case of subroutine selection, a template is specified as part of the definition of each subroutine. The subroutine will be executed if its template matches the actual arguments of the expression that invoked it. An example from [4] demonstrates this idea. Consider the following two functions, PLUSSINGLE and PLUSZERO, which might be part of an algebraic simplifier.

```
PLUSSINGLE : (LAMBDA (PLUS <-X) Xs)

PLUSZERO : (LAMBDA (PLUS 0 <-<-X)
                   ('(PLUS $$X))).
```

The single arrow prefix (<-) indicates that a single argument is to be matched. A double arrow prefix (<-<-) on a pattern variable indicates that any number of arguments will be accepted but considered as a single entity. PLUSSINGLE will, given any form of PLUS followed by any single element, return that single element. PLUSZERO presented with any form of PLUS followed by any number of elements, one of which is 0,will return the form PLUS followed by all the other elements of the argument [47].

Functions invoked by patterns are typically defined for an application toward a specified purpose. Some functions are used for consequent reasoning, and some for antecedent reasoning. In the first case, a function Consequent reasoning is required when trying to determine the truth of as assertion not explicitly represented in the data base. Some consequent procedure would be invoked by a successful match of the assertion in questionto a pattern that indicates the type of assertion proven true by a successful execution of the body of the procedure. On the other hand, antecedent reasoning is used when a program attempts to cause effects on the data base [47]. Thus, consequent functions are tried when a goal is desired and antecedent functions when the data base is to be updated. This allows a top-level program to invoke a subroutine to produce a certain data structure, without having to know which subroutine will respond to the request [4].

Pattern variables are special variables used in a pattern specification or template to indicate what the context or binding of the variable must be in order to make it eligible for a match. In general, a pattern variable is typed as being "open" or "closed". An open variable will match anything whereas a closed variable will match only an item equal to the previously assigned type-value of the variable. Although PLANNER and CONNIVER have no closed pattern variable, they,like QLISP and INTERLISP (which do have a closed pattern variable type),provide a semi-open type. It will match any item, if it has not been previously matched,otherwise it will match the previously assigned one. Thus, it acts as an open variable the first time and a closed variable thereafter. Instead of a semi-open pattern variable, CONNIVER has a type "macro". The macro has two values, one which instructs substitution of a CONNIVER value and another to substitute the LISP value in the template [4]. POPLER offers the greatest variety; four types, two modes, and restrictions for any of the types. The restrictions are in terms of data type or some user-specified tests. The variable types include one closed, one semi-open and two types of open. One that permanently assigns values and another that temporarily assigns a value, setting up a failure-action to restore the old value.

In addition to CONNIVER, PLANNER, QLISP, and INTERLISP, the languages SAIL, TELOS, and PROLOG also possess pattern

matching facility. In SAIL, the mechanism is limited since a template is simply created by specifying one or more items of an association. A search on the data base then occurs in order to find all the associations whose items match the specified items in the template. For example, the association

MONTEREY * _____ = _____.

could be provided as a template. All the triples in the data base with item1=MONTEREY would be returned as matches, and each triple would be executed in turn.

In TELOS, as well as SAIL, the primary use of patterns is in associative referencing. For instance, the instruction

GET(person[? age :greater (20) ?])

can serve as a template for retrieving a data base item, where the "greater"is a pattern function [59].

In PROLOG, there is a facility called "unification" which is explained as "pattern matching + logical variable[62]". 1977]. A logical variable is distinguished by the fact that it is unprefixed. That is, there is no distinction made by the programmer as to whether it is open, close, semi-open, etc. The programmer need never be concerned with whether it is assigned or bound or not. The system manages that aspect. To execute a goal, the system searches for the first clause (recall that PROLOG is a predicate logic based language)

whose head (i.e., procedure entry point) matches or "unifies" with the goal. The process of unification then finds the most general common instance of the two terms. If a match is found the found-clause is activated by executing in turn each of the goals within its body. Absence of a match causes backtracking to occur to the point that the original goal may be reconsidered and an attempt made to find a subsequent clause that also matches the goal [62].

QLISP also offers a "unification" mechanism for pattern matching. It does not have the same effect as that of PROLOG, however. In QLISP, this mechanism will let each of two expressions act as templates for one another. The facility is invoked by the keyword MATCHQQ. An example of its usage may be shown as follows:

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

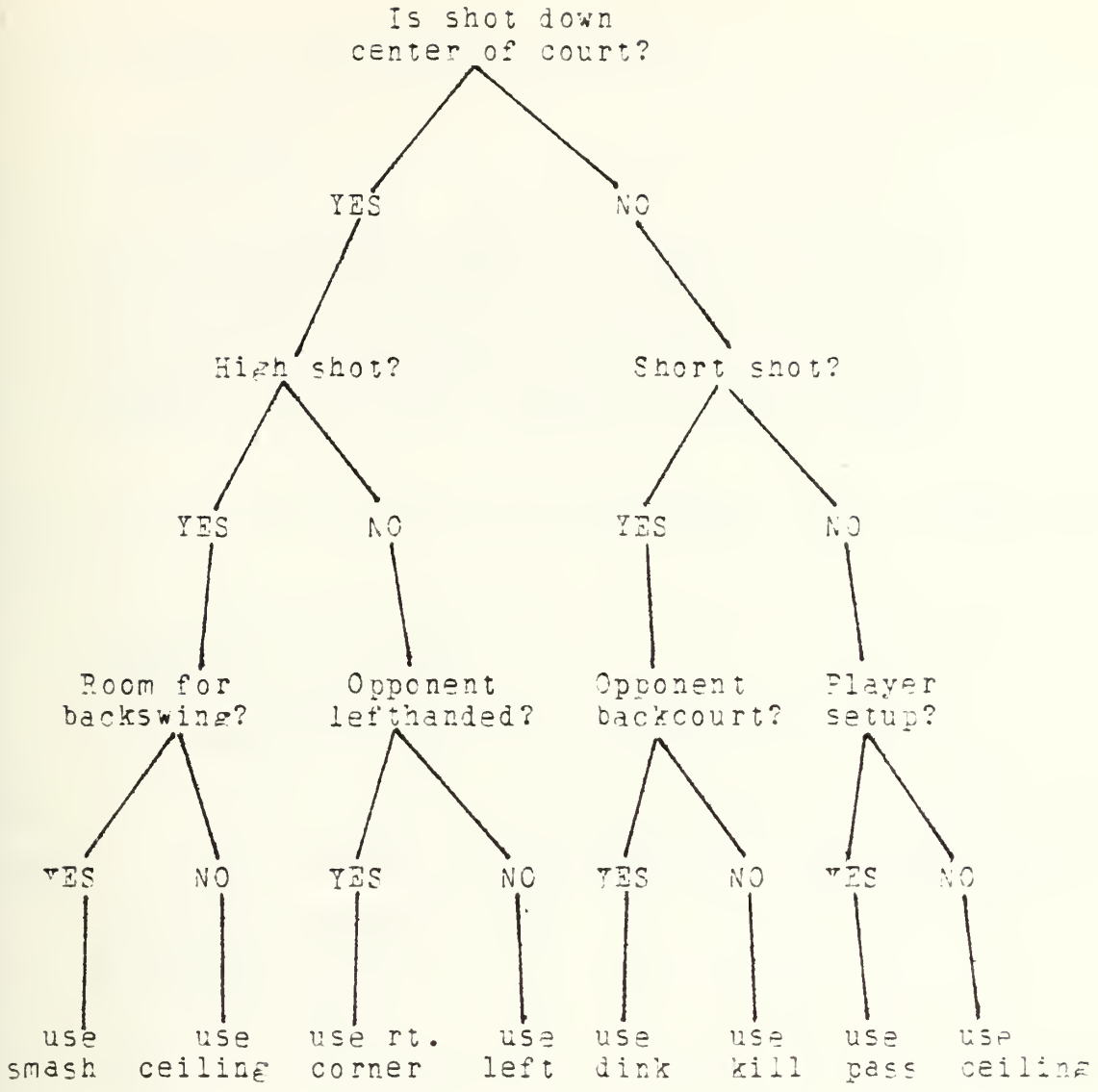
Here X, Y, and Z are open variables. The effect produced is that X is matched to A, Y to (A (B C)) and Z to (B A).

I. DISCRIMINATION NETWORK

A discrimination net classifies information on the basis of some of its properties. It is a tree-like structure in which the nodes represent tests to apply to an expression and the arcs represent values returned by the tests. These tests can then be set up to find out if some specified fact (in the form of a list of atoms or an arbitrary expression) is stored in the data base [9].

In QLISP, a discrimination net is used to represent the form of a data base. All data is stored in a common net so that equivalent expressions may be represented uniquely. Only one instance of an expression may occur, so before an expression is entered into the data base it is transformed into a canonical form [47]. This enables an expression in the net to possess a permanent property list just like that of a LISP atom [4]. For example, the discrimination net in figure 4c contains the expression USE (dinkshot) which would possess a property list that includes the following information:

Location	Frontside
Shot Length	Short
Type of Return	Defensive



Discrimination Network.

Figure 4c

I. DEFINITIONS FOR CONTROL STRUCTURES

1. Recursion

> A recursive technique is one with an essentially hierarchical character, that can be naturally described in several levels of detail. And a recursive procedure is one that can refer to and operate on itself, so that it can be "nested" within itself to an indefinite number of hierarchical levels [5].

The factorial routine below is an example of a recursive subroutine.

```
     SUBROUTINE FACTORIAL (N)
     FACTORIAL 1 = 1
     IF N > 1, FACTORIAL N = N * FACTORIAL (N-1)
     END SUBROUTINE
```

Using iteration instead of recursion this routine would appear somewhat longer and more awkward to read and write:

```
     SUBROUTINE FACTORIAL (N)
     FACTORIAL = 1
     IF N <= 1 THEN X:
     FOR I = 2 TO N
     FACTORIAL = FACTORIAL * I
     NEXT I
 X:  RETURN
```

2. Unification and Instantiation

Unification attempts to find substitutions of terms for varables to make expressions (well-formed formulas) identical.

> Very briefly and according to [39], a well-formed formula (wff) is any legitimate expression of the predicate calculus. For example,
>
> ON(BOAT, LAKE)
> ON(x, y).
>
> The elementary components of predicate calculus are: 1)predicate symbols-represents a relation in a domain of discourse, e.g., ON.
>
> 2)variable symbols-permits one to be indefinite about which entity is being referred to, e.g., x and y.
>
> 3)function symbols-denote functions in the domain of discourse. For example in the statement "The fast car beat the slow car" which may be represented by the expression BEAT(fast(CAR), slow(CAR)), fast and slow are the function symbols.
>
> 4)constant symbols-simple term used to represent a physical object or entity in the domain of discourse, e.g., BOAT, LAKE.
>
> The wffs above are atomic. More complex wffs may be formed with connectives such as (and), V (or), => (implies). Example: ON(BOAT, LAKE) IN(FISHERMAN, WATER).
>
> Any wff, atomic or complex, will evaluate to a true or false value.

With regard to unification, wff W2 may be obtained from the wffs of the form W1 and W1=>W2 via Modus Ponens inference rule. Furthermore, the universal specialization rule of inference allows the wff W(A) to be derived from the wff

(Vx) W(x), where A is any constant symbol. Using the two rules together produces the wff W2(A) from the wffs (Vx)[W1(x)=>W2(x)] and W1(A). Thus unification is required to find the substitution "A for x" that makes the well-formed formulas W1(A) and W1(x) identical [39]. There are two reasons for performing unification:

1. Resolution. If the two atoms that were unified (made identical) occur in different clauses and with opposite "signs" (one positive, one negative), then the clauses in which the two occur can be resolved.

2. Factoring. If the two atoms that were unified occur in the same clause with the same sign, then the clause contains two identical literals. The duplicate literal can be eliminated. This is called "factoring", and the clause with the substitutions made and the duplicate literal eliminated is a "factor" of the original clause [23].

Instantiation occurs when the name of a particular individual or object is substituted for a variable. The individual is then an "instance" of the variable.

3. Predicate Logic

Predicate logic breaks up propositional clauses (statements) in order to consider individually the items about which something is being asserted. For example, the proposition,

"the frog is in the water"

would be dealt with as one statement or entity in propositional logic, about which one would assert its truth or falsity. This statement however contains two items from the real world, namely "the frog" and "the water." The relationship between the two is expressed by the words "is in." The phrase "is in" may be represented by IN and is considered the predicate. The individuals, "the frog" (FROG) and "the water" (WATER) are the arguments for the predicate. Thus, the original proposition may be restated in the following way:

IN (FROG, WATER)

Any of the operations of propositional logic may still be applied to the statement.

The power that predicate logic adds is in the fact that the arguments of the predicate need not be named explicitly (instantiated). They may be variables, i.e., IN(x, y), a much more general statement yielding a more general and flexible reasoning capability [23].

4. Resolution Logic

The basic idea of the resolution principle is to prove the the converse of a theorem is false. Thus, resolution is a method for proving that a false statement os indeed false, but not necessarily that a true statement is true. Nevertheless, the advantage of this method is that if a theorem is true, a proof will be produced after a finite number of steps [26]. The fundamental technique employed is the conversion of an implication to an expression that contains no implication. That is, if 'p implies q' then it is eqivalent to state 'not p or q.'

Given a set of premises, the procedure for deriving a conclusion is as follows:

1. Form a new set of premises from the given premises and a negation of the conclusion.

2. Derive a contradiction from this new set.

3. Assumption of the original premises to be true and the derived conclusion to be false leads to a contradiction, therefore the desired conclusion must be true whenever the premises ar true. Thus the desired conclusion follows from the premises [23].

In general, resolution is more easily programmed than the other computational logics. It can handle complex premises and conclusions. However, because of combinatorial explosion it can't be used to prove deep mathematical theorems verifying complex, computer programs or to aid a robot cope with real world complexities. This is a result of

the need to derive many clauses (through unification, resolution and factoring) that are relevant to deriving a conclusion. A great deal of time is wasted though following lines of reasoning that come to dead ends [23].

Resolution can be used as the logical mechanism for theorem-provers. It has also been used in natural language understanding systems, formula manipulation and symbolic integration systems, and STRIPS-style problem-solvers.

One language especially well-suited for resolution programming is PROLOG [60].

V. SUMMARY

Summary Chart of AI Languages and Their Data and Control Structures

LANGUAGE:	DATA STRUCTURE:	CONTROL STRUCTURE:
IPL	List	Sequential
COMIT	String	Sequential
LISP	List property lists	Hierarchical recursion
INTERLISP	same as LISP plus strings, arrays, and user defined data types	as in LISP plus pattern matching and frame-oriented control
QLISP	same as above plus tuples, bags, classes, and vectors, discrimination net data base	same as above plus demons
PLANNER	List, data base made of assertions and antecedent and consequent theorems	global backtracking pattern matching, demons recursion

CONNIVER	List, If-added, If-needed theorems in a context oriented data base	pattern matching, demons, coroutine capability recursion
LEAP	Sets, associative data base, property lists	pattern matching
SAIL	same as LEAP plus non-recursive list structure	pattern matching, coroutine capability recursion
POPLER	List, context-oriented data base with PLANNER-like theorems	pattern matching recursion
PROLOG	List, record, frame, assertional data base	pattern matching, backtracking recursion
ABSET	Sets	
TELOS	as in PASCAL plus encapsulated data structuring	pattern matching, coroutine capability, demons, recursion

VI. CONCLUSIONS

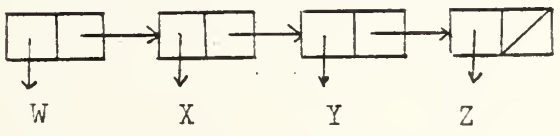
> Artificial Intelligence is the use of programs as tools in the study of intelligent processes, tools that help in the discovery of the thinking-procedures and epistemological structures employed by intelligent creatures [5].

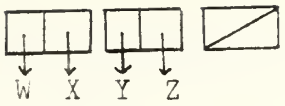
This statement is generally representative of the feelings of many AI researchers. The role of programming languages in AI is an undeniably major one. Moreover, the requirements of AI are complex, ambitious, and demanding to the point that general purpose conventional languages are inadequate. In fact, it is likely that data representation techniques or control structures invented for today's AI needs will inevitably be incorporated in conventional languages for more general purpose computing [22].

There is some mild difference of opinion as to the importance of data structures versus control structures. The argument is that the data structures needed in Artificial Intelligence do not really differ from those needed in general. Thus the key to efficient processing really lies in the available control mechanisms of a programming language [22]. However, this statement becomes less accurate as the distinction between data structure and control structure becomes increasingly fuzzy. For instance, frames and contexts are considered as data structures in Chapter III.

The reasoning was that they can be considered data entities with which are associated certain operations for their manipulation, e.g., create, delete, and modify. It may also be argued however that these are control structures. A frame slot for instance may cause the invocation of some procedure or the retrieval of another frame. Therefore in order to be thorough in the coverage of the special features required for AI programming, data structures as well as control structures must be considered equally important, to the possible extent at times that their distinction may seem entirely arbitrary.

Without exception, the languages employed in AI are symbol manipulators. More specifically, as mentioned earlier, they are either list processors or string processors (although a string may be manipulated as a linear list). The idea of a string processing language is becoming an obsolete one, as strict string manipulation is too restrictive for most applications. The current trend is to incorporate string processing capabilities in list processing languages. The notion of 'CDR-encoding' was intended to achieve just that for LISP. It implies that segments within a list will be stored in sequential locations in memory. Instead of the normal representation for a list, e.g., LIST(W X Y Z):



it might look something like: 

This would be nicely suited to records or packed data structures that are generally manipulated as whole entities. The advantage is a substantial savings of storage as locations containing pointers to the next list item are no longer needed. This is the very advantage that string processing has always had over list processing [8].

By now, there can be little disagreement as to the primacy of the list structure in AI programming. Data that is represented internally in a list structure conveniently allows for dynamic storage allocation, a basic necessity for AI computing. Those languages which don't provide it as a built-in feature provide good facilities for allowing the user to define such a data type as well as operations for its creation, deletion, and manipulation. Although list processing is a necessary capability for a good AI language to possess, there are other capabilities that should be considered highly desirable as well. The qualities listed below are seven of the most important.

1. Languages should be based on a single compiler or interpreter so that the basic control structures needed are easily accessed and AI systems may be developed by writing packs of procedures [22].

QLISP, CONNIVER, and PLANNER must be translated into LISP as a halfway stage. Most systems written in these languages have some modules or procedures written in LISP or INTERLISP. In some cases it is even necessary to write a special purpose language in order to do processing on a certain task efficiently (e.g., STRIPS, SHRDLU, see appendix B). The result is that most systems are implemented in INTERLISP.

2. Languages should offer simple control structures as opposed to complex, elaborate ones [22].

Elaborate control structure can hinder as often as help the user not intimately familiar with its manner of operation. The well-used example of global backtracking in PLANNER demonstrates this point. Its casual usage by a naive programmer will lead to considerable inefficiency.

3. The ways in which complex control structures are used should be studied to determine if more efficient implementations could be designed into a language [22].

For example, recursion is generally space consumptive and slow. It was discovered that many uses involve recursing an expression to get a value that is then merely passed back up through the chain in which the expressions themselves do not change. This is not an efficient use of recursion. Tail-recursion is a compromise between recursion and iteration, was invented to make recursion more efficient

under such circumstances. In a tail-recursion situation, recursion is used to compute a value, which is then passed back up the chain at which point the procedure can be converted to a simple iterative procedure. The value of this technique is in the substantial space (and time) savings, since the intermediate activation records for each recursive step need not be saved.

4. Garbage collection remains as one of the most effective and simple methods for reclaiming abandoned storage cells.

5. The use and manipulation of patterns is becoming an increasingly important capability for variety of operations including data retrieval, procedure invocation, and unification.

Currently the general pattern matching operation returns too little information. Either a failure is reported or a match is with no indication as to how close the match might have been in the case of failure. It would be useful to have a measure of 'fuzzy' matching for dealing with expressions on a semantic level [47].

6. Data storage and retrieval mechanisms should be incorporated that will not just distribute data randomly throughout the store [47].

It would be useful to many applications to have data stored on a semantic basis rather than a syntactic one as is most often the case. That is, some method of storing that considers some relation among some data items such that they are linked or stored in close proximity to one another.

7. Facilities for user defined data types should be considered more desirable than a multitude of built-in types that may turn out to be insufficient for unanticipated needs.

LISP is one of the oldest high level programming languages, the second oldest in fact according to its inventor, John McCarthy [34] (FORTRAN being considered the oldest). It is also the most popular for AI programming. For no one particular reason, but rather a combination of several (including the rapid growth of AI research), LISP has become the AI programming standard. There have been and still are many contenders for that position but so far none offer sufficient improvement over LISP's capabilities to warrant majority acceptance. The fact that LISP gives good access to the features of the host machine and its operating system, the availability of its interpreter as a command language for driving other programs, along with its internal list structure that makes it a good target for translating from yet higher level languages, make it a convenient vehicle for higher level systems for AI [34].

LISP possesses nearly all the qualities of a good AI language that were listed above. It has the distinction of having the first compiler written in the language to be compiled [34]. The LISP interpreter is also written in LISP and is about one page in length.

LISP has a small set of selector and constructor operations that are expressed as functions. Simplicity is derived from the small number of these functions. The fact that these operations were made into functions together with the ability to have branching within function definitions enabled LISP to become an entirely applicative language. That is, everything in a LISP program is presented as an expression. Separate procedures for function definition are not needed. There is no need for statements (such as goto's and assignments) of the type found in most languages. LISP is based on the ideas of the Lambda calculus. Conditional expressions may be recursively employed for building computable functions. Programs may be represented as data, and data as programs for that matter. These aspects contribute to the partial, but significant, achievement of one of McCarthy's original design goals, that is, to provide the capability for programs to be proved correct.

The availability of a property list structure, automatic garbage collection combined with features already mentioned (such as recursion and data-program indistinguishability) have made LISP extremely well-suited to past AI programming

needs. This has led McCarthy to make the claim and justly so that "LISP will become obsolete when someone makes a more comprehensive language that dominates LISP practically and also gives a clear mathematical semantics to a more comprehensive set of features [34]." Although LISP is adequately suited to most current AI programming needs, capabilities such as pattern matching, meaningfully organized data storing, and user defined data structures, all of which are becoming increasingly important to the implementation of applications systems, are not offered. However, LISP does provide the framework for building higher level languages that do offer these features. QLISP, CONNIVER, and PLANNER are examples of such languages. In seeking a higher level of program abstraction certain flexibilities are inevitably sacrificed. This is most often the case when the new extensions are convenient for certain types of applications. If an application is an unusual one and efficiency is required the designer is usually forced back to a familiar basic language-such as LISP-to either utilize directly or build a new special purpose language upon (e.g., the PROGRAMMAR language in Winograd's SHRDLU system, see Appendix B). PLANNER fell victim to such a circumstance. PLANNER's emphasis is on its ability to procedurally embed knowledge. Global backtracking was included to relieve the programmer of having to specify how it should be done, thus making the language as nonprocedural

as possible. What was not anticipated was the number of applications that preferred a PLANNER-like language for implementation but required programmer control over backtracking, as in the case of learning systems. PLANNER has nevertheless made a significant contribution to language design. It has pointed out a path toward the ultimate desirable characteristic of any AI language, that of nonprocedurality (see appendix A).

CONNIVER is an improvement over PLANNER (e.g., global backtracking removed) and it is an implemented language whereas full PLANNER never has been. CONNIVER is of course a PLANNER-like language. Therefore one may program instructions to the system such as "Imagine you were to do that, and tell me what would happen if you were to find that such and such were true," or "What did you learn about this while you were trying out that hunch that eventually you abandoned? [5]" Here, as in most CONNIVER programs, the programmer is relieved of the considerable detail of specification of the manner of achieving various goals. "Such intellectual subtleties are not straightforwardly expressible [5]."

Another PLANNER-like language is PROLOG. Unlike PLANNER it is not based on LISP and it does have working compilers. Pattern matching is extensive and an assertional data base is provided [36]. The efficient usage of this language is highly dependent on the expertise of the programmer.

Although current users are nearly fanatical in their devotion, it is not an easy language to learn and will likely not acquire massive popularity in the near future [36].

The POPLER language is again PLANNER-like. It is written in a lower level language called POP-2 which is similar to LISP in many respects. POPLER combines features of PLANNER and CONNIVER. It has borrowed the concepts of assertions and theorems from the former and context mechanisms from the latter. Also, as in PLANNER, a failure mechanism and backtracking are an important part of the control facility. However in POPLER the user is allowed to specify failure actions via an "action list." POPLER is a well-documented and user-friendly system but has not yet been used for any major projects and currently resides only in Edinburgh [4].

There are a few other languages that have no relation to LISP, and are not PLANNER-like, but are AI languages nonetheless. They include TEIOS, ABSET, and SAIL. TEIOS, founded on the PASCAL language, provides pattern-matching, an elaborate user-defineable data structure mechanism, coroutine capability and even demons. It should be able to compete successfully with most of the popular AI languages. No major systems to demonstrate its potential have as yet been implemented in TEIOS.

The ABSET language is founded on ABSYS, a declarative language with heavy emphasis on the use of propositional logic. ABSET is an extensive improvement over its predecessor but maintains the same goals, that is, to distinguish between the ordering of decisions and the ordering of evaluation, and the manipulation of partially evaluated programs [16]. These are valuable ideas for AI applications systems development. Like TELOS, no major system has been written in ABSET making it difficult to remark on its practical merit.

SAIL is an established , relatively well-known language. It is an extension of LEAP which is based on ALGOL 60 constructs. It has its own compiler, a feature that makes it unique among most AI languages. It has a few limitations however, including the fact that erasure of abandoned storage is the programmer's responsibility, as is the specification of which variables to save or restore upon backtracking. Lists are a recently added data type but will not efficiently represent multi-level linked structures such as trees and graphs in a uniform way. However, SAIL does have elaborate process control (see Coroutines, Chapter IV) and some new communication features. It will likely maintain its current, general popularity as it evolves in a stable, reliable manner to meet the needs of its users [4].

Finally, QLISP, another PLANNER-like language, offers a rich variety of data and control structures over which the user may opt for control. The most hopeful prospect for a general, efficient, and flexible programming system for AI lies in a QLISP/INTERLISP marriage [4]. INTERLISP is an extension of LISP 1.5 and currently the preferred language for applications systems implementation. Its popularity is growing as it offers the user all the desirable basic features of LISP plus aids that make it more user-friendly. The two-language programming system is presently under development. It is likely that AI programming will proceed in the direction of QLISP/INTERLISP until a major conceptual breakthrough is made in the way in which programming is approached.

Although LISP has been the standard AI language for over two decades it has only been in the last few years that it has acquired most of the popular attention that it now receives. Either more users and system implementers discovered a need for its capabilities or were simply finally ready to accept its unfamiliar appearance. Likewise, the next major advance in programming languages will probably be based upon an old idea or a current one perhaps. For instance, it may include Backus's notions of functional programming, that would liberate programming from its von Neumann, word-at-a-time, style. No doubt, this would ultimately be beneficial to AI programming needs.

APPENDIX A

PROCEDURALNESS

In the roughest sense, proceduralness refers to the degree to which a programming language will allow the programmer to specify how operations on data are to be carried out. Thus, a lack of proceduralness constrains a programmer to merely specifying what is to be done to the data, while the system is left to worry about how to do it. Procedural vs. nonprocedural may also be viewed as the difference between explicit and implicit specification for processing of information [25]. At one end of the spectrum exist these nonprocedural, or declarative, languages (i.e., the "what" languages), while at the other end can be found the procedural languages (i.e., the "how" languages"). Among the latter, as far toward the end as one may go, are machine languages (in which a programmer must give the most detailed instructions to the computer). At the other extreme, one would find all the problem describing languages, which are declarative by nature [48]. Included in this group are the report generation languages, in which a sample command would be "calculate the payroll for the ABC Company."

According to Sammet[48], a language demonstrates its nonprocedural character in one of two manners. Either the user is required to submit an ordered sequence of steps, each of which is "somewhat nonprocedural", or a set of

executable operations whose sequence is not user-specified. Report generators (e.g., RPG) are constrained in application and the user need only specify the input and desired output without any specific indication as to the procedures needed. Thus they are exemplary of the most nonprocedural languages.

It has been contended that progress along the how-to-what spectrum may be considered as a measure of progress in AI research, provided that a proper goal therein is to create sophisticated declarative higher level languages. Nilsson[38] feels that AI languages are already moderately far along toward the "what" end of the spectrum. PLANNER is probably the most declarative of the current AI languages. It is a goal-directed language in which the user may "specify high level goals in general terms without individuating all the particular objects and operations involved in their achievement [5]." It has established the model for nonprocedural, or PLANNER-like, languages.

Here again, the user specifies "what" rather than 'how'. In PLANNER, one states a goal that the system then matches to the index of general patterns and for which it attends to certain bookkeeping matters. However, PLANNER has been found to be "inefficient and hard to control [38]." Only a portion of it, called Micro-PLANNER has ever been fully implemented. In fact, it seems that PLANNER's major drawback is its inflexibility. Most users want more control than PLANNER allows, to tailor some procedures such as depth-first search

and backtracking (automatic features of the language), to more efficiently fit the particular problem at hand. Out of this type of frustration, Sussman and McDermott [55] created the CONNIVER language. CONNIVER is more procedural, and was designed to implement Sussman's problem-solving system, HACKER (see Appendix B). During the implementation he discovered potential inefficiencies and dead ends of which he would otherwise have remained unaware had he continued to program the system in PLANNER. CONNIVER allows greater programmer control over backtracking for more efficient tree-searching.

The basic argument in favor of working toward the development of increased nonproceduralness is that declarative languages are relatively easier for people to understand and communicate. The programmer is also freed to think about a problem at a more abstract level. But, in order to accomplish the goal of developing a sophisticated nonprocedural programming language for AI the computer must have stored within it a large amount of background and knowledge of the subject matter of a request. Thus the more nonprocedural a language is, the more knowledge it must have procedurally embedded within itself. PLANNER, without a doubt, is furthest along in this design. Hewitt has developed and incorporated into PLANNER a "Thesis of procedural embedding" which states that "intellectual structures should be analyzed through their procedural

analogues." For example, the analogue of a data type is the set of procedures which create, destroy, recognize, and transform data, or, the analogue of a drawing is a procedure for making the drawing [27].

A language that is even more nonprocedural than PLANNER but, still far less nonprocedural than a report generator, is an older language, COMIT, which was created to facilitate string data handling. Everything to be done in this language is embodied within a so-called "rule". Any action to be taken depends upon notation and position within the rule [48].

The format of a COMIT rule is:

statement label target pattern = action // go-to

In each rule, the programmer specifies what the pattern of interest is, what is to be done if the target pattern is found, and what location to branch to in order to find the next rule to evaluate. An entire COMIT program is nothing more than a series of such rules.

Nonprocedural is really a relative term that in fact changes with the state of the art. As compilers are developed to cope with increasingly complex sentences, the nature of the term changes. Thus, what is considered nonprocedural today may well be procedural tomorrow [45].

The debate over the amount of proceduralness an AI language should possess will likely continue for some time. The principle is that of the proceduralist versus the

declarativist philosopy of knowledge representation. The claim of the proceduralist is that knowledge should be represented procedurally, rather than factually. For example, the information that

"all weekends are relaxing"

would be represented in a declarativist's data base in exactly that form, i.e., as a single specific fact. A proceduralist, on the other hand, would represent it in a such a way as to allow it to be used more flexibly. The fact above could represented with four statements, each with a slightly different viewpoint:

1. Something is relaxing if ever it is found to be a weekend.
2. (Alternatively) if one wants to show something is relaxing, then trying to show that it is a weekend may be a good way.

(These are the antecedent and consequent choices that cannot be shown with a declarative representation.) Then there are the parallel possibilities taken from the negations:

3. If something is shown to be not relaxing, then one may deduce that it is not a weekend.

4. (Alternatively) if it is desired to show that something is not a weekend, then showing it is not relaxing may be reasonable [64].

A proceduralist does not like to defer the decision about how a fact may be used. The declarativist however, would prefer to do so while claiming intent to avoid predjudicing the future use of the fact.

There are some sound arguments favoring both sides, such as, more nonproceduralness in a language implies a lower amount of programming effort to produce a working program. However, it also implies a loss of programmer control over I/O functions and inability to minimize memory usage or execution time through more direct control of hardware operations. In figures A1 and A2, Fletcher has clearly identified the advantages and disadvantages of procedural and nonprocedural languages, in general.

The dichotomy among proficient programmers for preference of explicit control over various mechanisms has prompted some language experts to suggest the establishment of a "proceduralness factor" [50]. If machine code represents the most-procedural end and a true problem-describing language the most-nonprocedural, then all other languages could be each assigned some value

indicating how close it is to one end or the other. This would be an interesting exercise, however the final result would be little more than informational (see fig. A3). A programmer looking for a language in which to express a problem would not have an appreciation for the difference in utility of two closely valued languages (like CONNIVER and PLANNER for instance) unless he or she has prior knowledge of the basic differences between the two languages. This is particularly true when considering languages that may actually be a mixture of procedural and nonprocedural parts.

It would be more useful, but of course more complicated as well, to devise some type of "measuring scale by which the amount of proceduralness, to be specified by the user, can be defined for any given language [50]." A feature such as this in PLANNER would have obviated the necessity of creating CONNIVER.

Nonprocedural advantages:	Procedural disadvantages:
-less attention to utilization of H/W resources	-use of H/W resources involved in planning program
-little need to concentrate or mechanical relationships of I/O processing	-must carefully think through the program sequence in the order the computer will perform it
-focuses programmer's attention on the problem itself, and not on the requirements of the computer	-programmer's time divided between understanding the problem and understanding the computer
-mostly self-documenting	-requires additional effort to properly document
-relatively simple to learn	-moderately difficult to learn
-easier to produce working programs due to more extensive diagnostics and automatic inclusion of internal structure of object program	-requires more time to debug programs

D.A. Fletcher
(Reprinted from The Encyclopedia of Computer Science, 1976)

Figure A1

Nonprocedural disadvantages:	Procedural advantages:
-doesn't provide complete control over I/O functions	-relatively complete control over I/O functions
-requires considerable knowledge of the implentation of control mechanisms in order to use the language most effectively	-little knowledge of implementation of control mechanisms needed as these are programmer responsibilities
-less efficient use of memory	-programs may be written to minimize memory usage
-programs require somewhat more time to be executed	-programs can be written to minimize execution time by taking into account specific H/W characteristics

D.A. Fletcher
(Reprinted from The Encyclopedia of Computer Science, 1976)

Figure A2

NONPROCEDURAL: Problem-describing languages

COMIT

PLANNER

PROLOG

CONNIVER

POPLER

QLISP
ABSET
INTERLISP

TELOS

SAIL
LEAP
LISP

IPL

PROCEDURAL: Machine Code

Relative Proceduralness
of AI Languages.

Figure A3

APPENDIX B

APPLICATIONS SYSTEMS

Practical applications of Artificial Intelligence research can be found in medicine, chemistry, geology, general science, psychology, and commonsense problem-solving. Systems currently operating in some of these fields are being utilized as knowledge experts, or consultants. Some prominent and successful examples of such systems are included in this appendix.

The following information is presented for each system:

1. Principal designer(s)
2. Location of system development
3. Date of introduction
4. Implementation language
5. Functional description
6. General design characteristics
7. Where possible, accomplishments and/or limitations of the system are presented.

Some common terms and concepts referred to in the descriptions of several of the systems discussed are explained below.

i. Theorem prover:

A program whose function is to deduce theorems from an axiomatic base of knowledge by using strictly logical methods of reasoning. Some programs employ very general methods, which are universally applicable in principle although not necessarily always sufficient in practice to solve the problem at hand. Resolution is an example of such a general method, whereby an assertion is shown to be a theorem by proving that its negation is impossible [5].

ii. Question-answerer:

A Program which allows the user to interrogate a data base (generally, domain-specific) via a natural language (e.g., English). Ideally, the system (program) should be able to "store a large number of facts and respond to reasonable questions whose answers could be deduced from these facts [38]."

One of the first question-answering (Q-A) systems to have been developed was one called BASEBALL [36]. This system, written in IPL-V, could answer questions asked in ordinary English about the month, day, place, teams, and scores for each baseball game played in the American League in one year. The primary goal of the development of BASEBALL was to provide some insight to possible basic mechanisms for language comprehension.

A Q-A system such as BASEBALL is made up of two basic parts, a linguistic part and a processor. The linguistic part syntactically parses (semantically as well where possible) the question and determines what information is being given about the data being requested. The processor searches through the data base for the appropriate information, processes the results and reports the answer, usually in an abbreviated form [35].

iii. Generation-and test:

A technique first introduced in GPS, in which plausible guesses are made of solutions (regarding differences to reduce between an initial state of a problem and a goal state) and then tests are made to see how well the guess fits the circumstances [35].

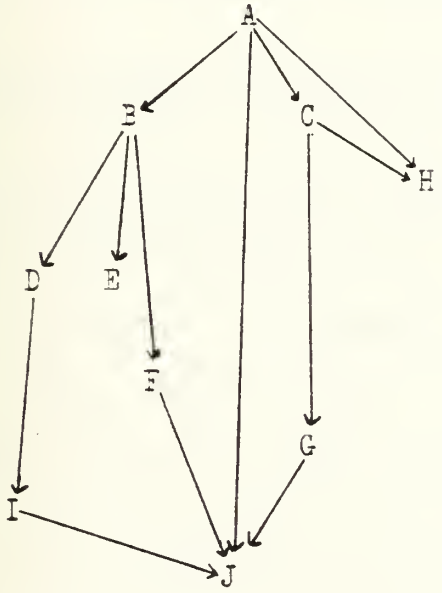
Different systems have different generation processes for proposing solution hypotheses. The particular process is usually dependent on the nature of the task. For example, DENDRAL employs a combinatorial algorithm that can produce all the topologically legal candidate structures for an unknown organic molecule. Whereas MYCIN uses a logical rule of inference (backward chaining of production rules) [17].

iv. Hierarchical vs. Heterarchical control:

In hierarchical control (the most usual type) one program has overall control. All others are subordinate to it as subroutines are to a master routine. Subordinate programs

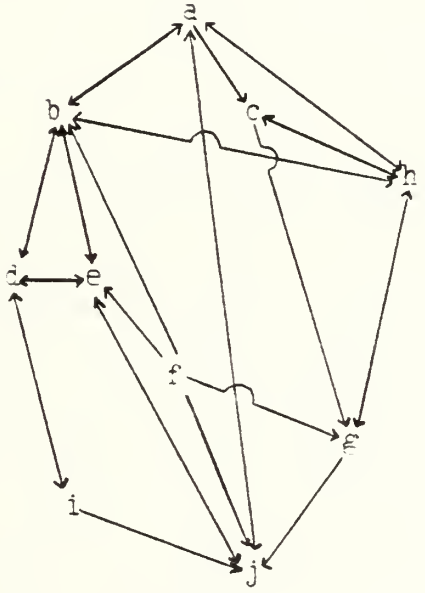
need not be directly accountable to the highest one, since there may be several levels of 'nierarchy.' However the flow of control passes in one direction only, downward. Moreover, no 'sideways' communication is allowed [5]. Figure B1 from [5] is an example of hierarchical control between programs.

In heterarchical control, the responsibility for control is more equally distributed usually throughout the system with a greater degree of internal communication. Programs can address or call one another either 'up,' 'down,' or 'sideways.' Furthermore, they may do so at different points in their processing [5]. Figure B2 demonstrates a heterarchical control scheme [5].



Hierarchy

Figure B1



Heterarchy

Figure B2

1. AM

Douglas B. Lenat
Carnegie-Mellon University
1976
INTERLISP

Functional description:

AM's task is to formulate scientific theory. In particular to make new definitions, explore new concepts and judge the "interestingness" of its discoveries.

AM is a theorem proposer, as opposed to a theorem prover, in a primarily mathematical knowledge domain.

General design characteristics:

AM is a heuristic rule-guided system, consisting of a set of a few hundred rules.

It works on packages called "concepts" which are made up of many "facets". The facets are attributes of some concept that AM would be exploring at some point. Examples of facets are concept name, associated definition, examples, analogies, etc. AM makes repeated attempts to fill in values for (make discoveries about) these concepts. Each concept is represented as an active, structured knowledge module. One hundred very incomplete modules are initially supplied, each

one corresponding to and elementary set-theoretic concept (e.g., Union). This provides a definite but immense space which AM begins to explore.

An agenda of "jobs" is maintained. A job could be finding a value for a facet in a concept. When a job is chosen for processing all potentially relevant heuristic rules are gathered and executed. A typical agenda entry would be:

```
ACTIVITY:  Fill in some entries
   FACET:  for the GENERALIZATIONS facet
 CONCEPT:  of the PRIMES concept.

 Reasons:  because
     (1)   There is only 1 known generalizaton of
           PRIMES so far.
     (2)   The worth rating of PRIMES is now very high.
     (3)   Focus of attention: AM just worked on PRIMES.
     (4)   Very few #s are PRIMES; a slightly more
           plentiful concept may be more interesting.
Priority:        350 (on a scale of 0-1000) [29].
```

Accomplishments/Limitations:

In one hour of CPU time discovered the obvious finite set theoretic concepts and relationships such as DeMorgan's laws and singletons. AM also discovered Squaring, Squareroot, Natural #s, fundamental theorem of arithmetic (unique factorization into primes), Goldbach's conjecture (every # > 2 is the sum of 2 primes), and literally hundreds of other common concepts [29].

AM quickly finds its limitations as it has no ability to discover or create new heuristic rules. Another system under development by Lenat, called EURISKO, attempts to improve on AM in just that regard [27].

2. BUILD

S.E. Fahlman

?

1974

CONNIVER

Functional description:

BUILD is tasked with solving logic or common sense problems occurring in a "blocks world".

General design characteristics:

BUILD's knowledge base includes facts about the physics of weight and levers, stability and friction.

BUILD is heterarchically organized around seven high level procedures. These modules each have memory and reasoning power that allow them to call one another when in trouble.

A typical task would be to rearrange an odd assortment of blocks into a specific order. BUILD starts by drawing up

a very simple plan, which is gradually elaborated as necessary or at the request of other experts. The first plan follows BUILDS comparison of goal state with initial state and is simply a list of the blocks that are not yet in their correct position. The list is ordered, its first members being those blocks whose supports (table or other blocks) are already in position. Starting with these, BUILD plans to call the PLACE routine to deal with them one by one. If PLACE succeeds immediately in each case, the problem is solved [5].

Accomplishments/Limitations:

The modular programming approach yields easier understanding of the functions of the various parts. Also BUILD need not compute thoroughly beforehand all the facts that may be relevant. The system is capable of backtracking and utilizing all pre-faiure information before choosing an alternative path. When it fails BUILD does not back up to the previous choice-point and pick an alternative method at random; instead it uses its understanding of its failure to select the method most likely to succeed. If the failed method is due not to unfortunate choice of method but rather to a small local difficulty in applying it, BUILD can adjust this as required and restart the failed method [5].

3. CRYSALIS

E. Feigenbaum, H.P. Nii, R. Engelmore

Stanford University

1977

INTERLISP

Functional description:

The CRYSALIS system hypothsizes the structure of a protein from a map of electron density that is derived from x-ray crystallographic data.

General design characteristics:

CRYSALIS is a knowledge-based, rule-guided inference system where the rules are of the form:

situation => action

The 'situation' can be thought of as the current hypothesis. The 'action' is a process that modifies the current hypothesis. Internally, the situation-hypothesis is represented as a node-link graph with distinct levels, each representing a degree of abstraction. A node represents an hypothesis and a link represents support for a hypothesis. The situation-hypothesis is formed incrementally via a sequence of local generate-and-test activities.

A simplifying strategy is employed that maintains only one best situation-hypothesis at a time and modifies it incrementally as necessary as the data changes [17].

4. DECAIDS

T. Buscemi and J. Masica

Naval Postgraduate, Monterey

1979

INTERLISP

Functional description:

DECAIDS was designed to assist in the organization of "decision aiding characteristics relating to computer resource allocation alternatives [7]." Information concerning the organization's task, technology, environment, and structure characteristics is sought from the user during an interactive consultation session. Recommendations are then provided by the system.

General design characteristics:

DECAIDS is a knowledge-based production rule system that employs a backward chaining search strategy. The knowledge base consists of 41 rules and 23 parameters represented in an implied AND/OR tree.

The framework of Stanford University's EMYCIN inference engine (a product of the MYCIN project) was adapted for DECAIDS[7].

5. DENDRAL/MetaDENDRAL

E. Feigenbaum and J. Lederberg

Stanford University

1965

INTERLISP

Functional description:

DENDRAL's task is to assist the analytic chemist in enumerating plausible structures (atom-bond graphs) for organic molecules. Given an observed fragmentation pattern, DENDRAL hypothesizes the best structural explanation of the data. Its primary source of empirical data is a mass spectrometer-an instrument that fragments molecules of a chemical sample (using an electron beam) and records the results. A mass spectrum, the output of the mass spectrometer, is a 2 dimensional record of the abundance of various fragments plotted as a function of their molecular weights. A secondary source of data is a nuclear magnetic resonance spectrometer(NMR).

DENDRAL's output is a graph, i.e., a topological model of the molecular structure of the unknown compound, or the output is a list of plausible molecular graphs, rank ordered with their relative plausibility scores.

The total size of the problem space is the number of topologically possible molecular structures generated with valence considerations alone.

MetaDENDRAL works with DENDRAL to automatically form, test, and modify inference rules. It infers rules of mass spectrometry by induction from empirical data for possible later use by DENDRAL.

General design characteristics:

DENDRAL is a heuristic rule-guided system. Rules are of the form:

situation => action

each with an associated probability of occurrence.
Goal-seeking and achievement is done via the three stage process, plan-generate-test [17].

Accomplishments/Limitations:

Heuristic DENDRAL has solved hundreds of structural inference problems, most recently of structures in the family of organic amines, for which the analysis is reasonably complex. The improvement in running speed of solving these problems using the specialized heuristic methods found in DENDRAL over solving them by more general methods is estimated to be as great as a factor of 30,000 [6].

6. ELIZA

J. Weizenbaum

MIT & Stanford University

1966

COMIT

Functional description:

ELIZA is a question-answering system that is intended to simulate the conversations between a psychoanalyst (the machine) and a patient. The point of interest is the two-person conversation [5].

General design characteristics:

Noncommital natural English statements are generated by ELIZA in response to questions tendered in English as well. The system does not 'understand' questions but rather attempts to make sense (i.e., find an appropriate response) of them through extensive pattern matching and classification [35]. It looks for key words, e.g., I, you, alike, father, etc. If found the sentence is transformed according to a rule associated with the word. If no key word is found ELIZA responds with a content-free formula such as "Why do you think that?" or with a reference to an earlier remark.

The knowledge base is relatively small, a consequence of having chosen the psychiatric mode of conversation. Recent versions employ frames to make ELIZA more knowledgeable. A frame is an organized body of data, in this case comprising a specific set of key words and associated transformation rules [5].

Accomplishments/Limitations:

ELIZA has no capability for understanding the semantics of the conversation in which it participates. However, it successfully presents the illusion of such. It has lured many knowledgeable individuals, even sophisticated computer scientists, into engaging in personal conservations [35].

7. EPAM

E. Feigenbaum and H. Simon

UC, Berkeley and RAND Corp.

1961

IPL-V

Functional description:

EPAM (Elementary Perceiver and Memorizer) was designed to model the human cognitive process of the rote learning of nonsense syllables. It is an information processing psychology model of a classical phenomenon (rote learning) well-known in the literature. It serves also as a simulation of verbal learning behavior [53].

General design characteristics:

EPAM utilizes a discrimination network as the basis for an associative memory scheme. The net is an adaptive structure that can grow over time to incorporate new stimulus objects that need to be recognized in an efficient manner [5].

Accomplishments/Limitations:

The discrimination net concept of EPAM has been adapted for other applications, including chess playing programs, where it used to represent the patterns of chess boards during play [35].

EPAM successfully provided a clear and simple information processing structure; the patterns were easy to understand and gave rise to complex behavior and interesting explanations of phenomena that were well understood experimentally [53].

B. GENERAL PROBLEM SOLVER--GPS

A. Newell and H. Simon

Carnegie Tech., and RAND Corp.

1973

IPL-V

Functional description:

GPS grew out of the LOGIC THEORIST machine [37]. It was aimed at trying to solve a variety of unrelated logic problems. Emphasis was placed on attaining generality in problem solving. The quality of the problem solving process was a secondary consideration [37].

General design characteristics:

GPS employs a heuristic search paradigm. This method consists of two entities, objects and operators. An operator when applied to an object produces a new object or indicates inapplicability. A heuristic search problem is:

Given: An initial situation represented as an object.
A desired situation representation as an object.
A set of operators.

To Find: A sequence of operators that will transform the initial situation into the desired situation.

The operators are rules for generating objects and thus

define a tree of objects. Each node represents an object and each branch from a node represents the object produced by the application of the operator.

A method for solving a heuristic search problem is to search the tree defined by the initial situation and the operators for a path from the initial situation to the desired situation [37].

The effectiveness of this method is determined by its rules for selecting operators to be tried (rules for guiding the search) There are two basic criteria for selecting operators:

Desirability--the operator should produce an object that is similar to the desired situation.
Feasibility--the operator should be applicable to its input object.

GPS uses the heuristic search paradigm directly; a problem is given to GPS in terms of objects and operators. It employs a general technique called means-ends analysis to guide its tree search. Means-ends analysis is accomplished by taking differences between what is given and what is wanted, e.g., between two objects, or between an object and the class of objects to which an operator can be applied. A difference designates some feature of an object that is incorrect. GPS uses the difference to select a desirable operator--one that is relevant to reducing the difference.

If an operator is not applicable to an object, an attempt to apply it will result in a difference--the reason it is not applicable. If the difference is not too difficult, GPS will tackle it as it would any difference between two objects. If it is successful, a new object will be produced that hopefully may have the operator applied to it.

GPS has four types of goals that application of the operators attempt to achieve:

1. Transform object A into object B.
2. Reduce difference D on object A.
3. Apply operator O to object A.
4. Select the elements of set S which best fulfill criterion C [37].

Accomplishments/Limitations:

The introduction of means-ends analysis provided a new and widely applicable method of goal achievement [5].

GPS successfully solved a variety of problems including: the Missionaries and the Cannibals problem, Integration of problems such as (sin (ct)cos t(ct) + t)dt, the Towers of Hanoi problem, the Bridges of Konigsberg problem, Letter series completion (e.g., complete the series: BCBDBEBF...) problems, and many others [37].

9. HACKER

G. Sussman

MIT

1975

CONNIVER

Functional description:

HACKER operates in a blocks world where its task is to develop procedures for getting a block or group of blocks from an initial configuration to a specified goal configuration.

General design characteristics:

HACKER uses means-ends analysis (see GPS) to accomplish its task. The data base maintains information on operators (actions), as well as preconditions and effects of the operators.

Initial plans are rough so that when mistakes are made HACKER must analyze what went wrong and where. Special debugging programs are supplied for this. Mistakes are then classified, generalized and placed on a list of traps to be avoided later.

Accomplishments/Limitations:

HACKER may be considered a system for automatic programming, in that it is a problem solving program that itself writes and improves programs and learns to do so better with practice [5].

10. INTERNIST

H. Pople
University of Pittsburgh
1975
INTERLISP

Functional description:

INTERNIST is a system developed to provide support in the formation and solution of difficult clinical problems in internal medicine.

General design characteristics:

INTERNIST is a knowledge-based system wherein the knowledge is represented in two different element types: disease entities and manifestations. There are about 400 disease entities and over 2000 manifestations (including symptoms, lab data, etc). Each disease entity has an associated list of manifestations. A value between one and five is assigned to each manifestation estimating its frequency of occurrence. By the same token, each manifestation has an associated list of disease entities, each of which has a weighting factor between zero and five.

A partitioned semantic network is used to represent a hierarchy of disease categories, organized primarily around

the concept of organ systems, e.g., liver, lung, kidney diseases, etc. [40].

The number of distinct disease entities known to a practicing physician is about 40. However, a patient may demonstrate symptoms of the co-occurrence of several hitherto unrelated diseases. If an upper bound of ten is imposed on the number of concurrent disease processes possible, then the number of diagnostic categories required to classify arbitrary patients is of the order of ten to the 40th.

Accomplishments/Limitations:

INTERNIST II is a recent enhancement of the original INTERNIST system. It embodies strategies of concurrent problem formation that yield more rapid convergence to a correct diagnosis in many cases, and at least some cases provide more accurate results [40]. One complex case on which INTERNIST II consulted produced a diagnosis consisting of: primary cardiomyopathy, congestive heart failure with pleural effusion, transudative ascites, cardiac cirrhosis resulting from chronic hepatic congestion, and acute tubular necrosis of kidneys caused by cardiogenic shock. Evidence of systemic embolism was also reported [40]. The system required 90 seconds of execution time to generate that particular diagnosis of what was considered a relatively complex case.

11. MYCIN

E. Shortliffe, R. Davis, B. Buchanan

Stanford University

1976

INTERLISP

Functional description:

MYCIN was designed to assist a physician by providing consultative advice on diagnosis of and therapy for infectious diseases--in particular bacterial infections in the blood. The system is capable of handling interactive dialogue regarding the diseases and capable of supplying coherent explanations of its results [11].

General design characteristics:

MYCIN is a rule-guided inference system that employs a backward chaining search strategy. A certainty factor between 0 and 1 (i.e., probability) is associated with each conclusion.

Each rule embodies a single, modular chunk of knowledge and states explicitly in the premise all necessary context. Specifically, a rule is a simple conditional statement (IF/THEN or premise/action format). The premise is a Boolean expression. The action part contains one or more

conclusions. Each is completely modular and independent of the others. They are invoked in a backward unwinding scheme that produces a depth-first search of an AND/OR goal tree (similar in some respects to PLANNER's consequent theorems) of more than 200 rules. The maximum number of rules for a single subgoal is 30. Meta-rules are available as strategy rules for suggesting the best approach to a given subgoal. They have the same format as the clinical rules but can indicate that certain clinical rules should be tried first, last, before others, or not at all. These then provide a capability for pruning the search tree, thereby reducing the feasible search space.

Accomplishments/Limitations:

In mid 1974 a semi-formal study was conducted in which five infectious disease experts not associated with the project were asked to evaluate the system's performance on 15 cases of bacteremia selected from current patients. The experts approved of MYCIN's therapy recommendations in 72% of the evaluations. This is a significant appraisal of MYCIN's success especially considering that the experts were not unanimous in their own recommendations [11].

12. PROSPECTOR

R. Duda, P. Hart, P. Barret, J. Gaschnig,
K. Konolige, R. Reboh, J. Slocum

AI Center, SRI International

1976

INTERLISP

Functional description:

PROSPECTOR was designed to assist exploration geologists in interpreting and evaluating data on specific mineralized sites or prospects [15].

General design characteristics:

PROSPECTOR is a knowledge-based, rule-guided inference system organized internally as a partitioned semantic network. The partitioning is hierarchical, in order to represent a taxonomy of minerals. Top-level nodes of the net correspond to top-level hypotheses about the presence of various types of ore deposits. Lower-level nodes may correspond to directly observable geologic data, or to intermediate concepts that can be inferred from observables. A principal task is to infer probabilities for the top-level hypotheses on the basis of available observations [14].

Rules are of the form, A => B. A probability factor is associated with the conclusion B. Two additional numbers measure the degrees to which A is necessary and sufficient for B.

Accomplishments/Limitations:

This system has been employed successfully as a teaching tool for learning about types of ore deposits.

PROSPECTOR has a slight handicap in that a body of observations, often uncertain in nature, must be interpreted with the aid of a knowledge base that supports plausible reasoning but not strict logical inference [14].

13. SHRDLU

T. Winograd

MIT

1971

Micro-PLANNER, LISP, PROGRAMMAR

Functional description:

The SHRDLU program represents a robot that can respond verbally as well as actively. Emphasis is placed on language interpretation rather than generation. SHRDLU's area of concern and universe of discourse is problem solving in a blocks world.

General design characteristics:

The SHRDLU program is heterarchically organized. Its knowledge is comprised of those types required for abstract general problem solving, linguistic processing, and reasoning in the specific domain of discourse. The combined store of knowledge is represented procedurally, due in part to the use of micro-PLANNER as the primary language of implementation.

The heterarchy of the system is expressed with three major programs each written in a different language. The deduction program, written in micro-PLANNER, has its own

body of knowledge regarding the specific universe of discourse chosen. It solves problems of various kinds in the course of exploring the consequences of facts presented, planning actions, and answering questions. The semantic program, written in LISP, is actually a set of programs dealing with meaning (whether of words, word groups, or whole sentences). The third program does grammatical parsing. Embodied within it is a particular theory of English grammar which is used to recognize the structure of sentences [5]. For this, Winograd wrote a special purpose language called PROGRAMMAR.

Accomplishments/Limitations:

Although SHRDLU can 'converse' sensibly only about pyramids and other inhabitants of the blocks world, it can parse sentences containing non-Blocks words, like 'eggs, cake, mother, and recipe,' provided the minimal relevant semantic information (such as that mother is an animate noun) is included with the definitions of the words in question. For instance, the program can parse (though not reply to) the sentence, "How many eggs would you have been going to use in the cake if you hadn't learned your mother's recipe was wrong? [5]".

14. STRIPS/ABSTRIPS

R. Fikes, N. Nilsson, B. Raphael,
T. Garvey, R. Waldinger, J. Munson/
E. Sacerdoti

Stanford Research Institute

1971/1974

CLISP, INTERLISP

Functional description:

STRIPS is used to formulate plans (i.e., a series of actions that together establish preconditions necessary for the final action) for the robot SHAKEY, which spends most of its time in a world consisting of 7 rooms variously connected by 8 doors and containing several large boxes to be pushed from one place to another [5].

ABSTRIPS (Abstracton-based STRIPS) is a modification or improvement on STRIPS. Preconditions necessary for the attainment of goals or subgoals are ordered by criticality. This insures that the most critical preconditions are satisfied first. Planning is done therefore in a hierarchy of abstraction spaces [46].

General design characteristics:

STRIPS searches a space of 'world models' to find one in which a given goal is achieved. Each world model includes a

large number of facts and relations dealing with the position of the robot and the attributes of various objects, open spaces, and boundaries. It is represented by a set of well-formed formulas (wffs) of the first order predicate calculus [20]. The problem space is composed of three entities:

1. Initial world model--a set of wffs describing the present state of the world.
2. A set of operators--including a description of their effects and precondition wff schemata.
3. A goal condition stated as a wff--the problem is solved when STRIPS produces a world model that satisfies the goal wff.

For searching the space of world models, STRIPS uses a GPS-like means-ends analysis strategy. The combination of means-ends analysis and formal theorem proving methods allows objects (world models) to be much more complex and general than any of those used in GPS. It also provides more powerful search heuristics than any of those found in theorem proving programs [20].

Accomplishments/Limitations:

STRIPS is able to solve a variety of problems of the following nature:

1. Turn on a lightswitch (must climb on a box first--must find a box before that).
2. Push three boxes togetner.
3. Go to a location in anotner room [20].

15. TALE-SPIN

J. Meehan
UC, Irvine
1977
INTERLISP

Functional description

TALE-SPIN is a program that generates stories by using diverse sources of knowledge including English, physical space, problem-solving, story structure, social relationships, and bodily needs. The purpose of the work is to test the combination and interaction of many sources of knowledge [31].

General design characteristics:

TALE-SPIN is a, top-down, goal-oriented problem-solver. Its output may be regarded as a trace through problem solving procedures [31]. It may be viewed as a program that simulates rational behavior by characters in the world. It is composed of three parts: a problem solver, an assertion mechanism (adds events to memory), and an inference mechanism (produces consequences of an event).

TALE-SPIN draws on frame-oriented knowledge to direct story development. The user may initially specify the characters, environment, and a problem for each character to solve, or may specify a moral and the system generates the problems. Each character in a story has a goal stack associated with it. An example of a goal is 'hunger' which possesses a set of rules such as, 'If you are hungry and see some food, you'll want to eat it,' and 'If you are trying to get some food and you fail, you will get sick.' Achievement of the goals developes the story.

Problems are associated with an area of knowledge, i.e., the problem domain. This is defined by set of representational primitives, a set of goal states or problems expressed in terms of those primitives, and procedures for solving those problems [31].

Accomplishments/Limitations:

A sample story generated by TALE-SPIN:

Once upon a time George Ant lived near a patch of ground. There was a nest in an ash tree. Wilma Bird lived in the nest. There was some water in a river. Wilma knew that the water was in the river. George knew that the water was in the river. One day Wilma was very thirsty. Wilma wanted to get near some water. Wilma flew from her nest across a meadow through a valley to the river. Wilma drank the water. Wilma wasn't thirsty any more.

George was very thirsty. George wanted to get near some water. George walked from his patch of ground across the meadow through a valley to a river bank. George fell into the water. George wanted to get near the valley. George couldn't get near the valley. George wanted to get near the meadow. George couldn't get near the meadow. Wilma wanted George to get near the meadow. Wilma wanted to get near George. Wilma grabbed George with her claw. Wilma took George from the river through the valley to the meadow. George was devoted to Wilma. George owed everything to Wilma. Wilma let go of George. George fell to the meadow. The end [reprinted from 31].

LIST OF REFERENCES

1. Allen, J., Anatomy of LISP, McGraw-Hill Book Co., San Francisco, 1978.

2. Backus, J., "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs," 1977 ACM Turing Award Lecture, Communications of the ACM, Vol. 21, No. 8, Aug 1978.

3. Bobrow,D., Raphael,B., "A Comparison of List-Processing Languages," Communications of the ACM, Vol. 7, No. 4, April 1964.

4. Bobrow,D., Raphael,B., New Programming Languages for AI Research, XEROX, Palo Alto Research Center, Palo Alto, CA, July 1976.

5. Boden,M., Artificial Intelligence and Natural Man. Basic Books Inc., New York, 1977.

6. Buchanan,B., Feigenbaum,E.,"DENDRAL and Meta-DENDRAL: Their Apllications Dimension," Artificial Intelligence, Vol. 8, No. 24, Nov 1978.

7. Buscemi,T., Masica,J., DECAIDS: A Prototype Production Rule for Decision Support Systems, MS Thesis, Naval Postgraduate School, June 1978.

8. Campbell,J., "Symbolic Computing With and Without LISP," Record of the 1980 LISP Conference, Aug 1980.

9. Charniak,E., Riesbeck,C., McDermott,D., Artificial Intelligence Programming, Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

10. Davis,R., King,J., "An Overview of Production Systems," Machine Intelligence 8, Elcock and Michie, eds., Halsted Press, NY, 1977.

11. Davis,R., Buchanan,B., Shortliffe,E., "Production Rules as a Representation for Knowledge-Based Consultation Programs," Artificial Intelligence, Vol. 8, No. 1, Feb 1977.

12. di Forno,A., "Programming Languages," Advances in Information Systems Science, Vol. I, J. Tou, editor, Plenum Press NY, 1969.

13. Doran,J., "Knowledge Representation for Archaeological Inference," Machine Intelligence 8, Halsted Press, NY, 1977.

14. Duda,R., Hart,P., Nilsson,N., Sutherland,G., "Semantic Network Representations in Rule-Based Inference Systems," Pattern-Directed Inference Systems, Waterman and Hayes-Roth, editors, Academic Press, NY, 1978.

15. Duda,R., Hart,P., Barret,P., Gaschnig,J., Konolige,K., Reboh,R., Slocum,J., Development of the PROSPECTOR Consultation System for Mineral Exploration, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1978.

16. Elcock,E., Foster,J., Gray,J., McGregor,J., Murray,A., "ABSET, A Programming Language Based on Sets: Motivation and Examples," Machine Intelligence 6, Meltzer and Michie eds., American Elsevier Publishing Co., Inc., New York, 1971.

17. Feigenbaum,E., "The Art of Artificial Intelligence --Themes and Case Studies of Knowledge Engineering," Proceedings of the National Computer Conference, 1978.

18. Feldman,J., Rovner,P., "An ALGOL-Based Associative language," Communications of the ACM, Vol. 12, No. 8, Aug 1969.

19. Feldman,J., Low,J., Swinehart,D., Taylor,R., "Recent Developments in SAIL--An ALGOL-Based Language for Artificial Intelligence," Proceedings of the AFIPS Conference, 1972.

20. Fikes,R., Nilsson,N., "STRIPS: A New Approach to the Application of Theorem Proving," Artificial Intelligence, Vol. 2, No. 3/4, 1971.

21. Fisher,D., Control Structures for Programming Languages, PhD Dissertation, Carnegie-Mellon University, May 1976.

22. Foster,J.,"Programming Language Design for the Representation of Knowledge," Machine Intelligence 8, Elcock and Michie, eds., Halsted Press, NY, 1977.

23. Graham,N., Artificial Intelligence, TAB Books Inc., Blue Ridge Summit, PA, 1979.

24. Guttag,J., Horowitz,E., Musser,D.,"The Design of Data Type Specifications," Current Trends in Programming Methodology, Vol. I", Yeh,R., editor, Prentice-Hall, Inc., NJ, 1978.

25. Hewitt,C.,"Procedural Embedding of Knowledge in PLANNER," Proceedings of the 2nd IJCAI, Sept 1971.

26. Hunt,E., Artificial Intelligence, Academic Press, NY, 1975.

27. Klahr,P.,"Planning Techniques for Rule Selection in Deductive Question-Answering," Pattern-Directed Inference Systems, Waterman and Hayes-Roth, editors, Academic Press, NY, 1978.

28. Lenat,D.,"The Ubiquity of Discovery," Proceedings of the 5th IJCAI, 1977.

29. Lenat,D.,"Designing a Rule System That Searches for Scientific Discoveries," Pattern-Directed Inferences Systems, Waterman and Hayes-Roth, eds.,Academic Press, NY,1978.

30. Lenat,D.,"On Automated Scientific Theory Formation: A Case Study Using the AM Program," Machine Intelligence 9, Hayes, Michie and Mikulich, eds., John Wiley and Sons, NY, 1979.

31. Meehan,J., TALE-SPIN,"An Interactive Program That Writes Stories," Proceedings of the 5th IJCAI, 1977.

32. Minsky,M., "A Framework for Representing Knowledge," The Psychology of Computer Vision, Winston,P., ed., McGraw-Hill Book Co., New York, 1975.

33. McCarthy,J., "LISP-Notes on Its Past and Future," Record of the 1980 LISP Conference, Stanford University, 1980.

34. McCarthy,J., "History of LISP," ACM-SIGPLAN, History of Programming Languages Conference, June, 1978.

35. McCorduck,P., Machines Who Think, W.H. Freeman and Company, San Francisco, CA, 1979.

36. McDermott,D., "The PROLOG Phenomenon," SIGART Newsletter No. 72, July 1980.

37. Newell,A., Ernst,G., GPS: A Case Study in Generality and Problem Solving, Academic Press, NY, 1969.

38. Nilsson,N., "Artificial Intelligence," Proceedings of the IFIP Conference, 1974.

39. Nilsson,N., The Principles of Artificial Intelligence, Tioga Publishing Co., Palo Alto, CA, 1980.

40. Pople,H., "The Formation of Composite Hypotheses in Diagnostic Problem Solving," Proceedings of the 5th IJCAI, 1977.

41. Quillian,R., "Semantic Memory," Semantic Information Processing, Minsky,M., ed., MIT Press, MA, 1968.

42. Raphael,B., "The Frame Problem in Problem-Solving Systems," Artificial Intelligence and Heuristic Programming, Findler and Meltzer, eds., American Elsevier Publishing Co., New York, 1971.

43. Raphael,B., Bobrow,D., Fein,L., Young,J., "A Brief Survey of Computer Languages for Symbolic and Algebraic Manipulation," Proceedings of the IFIP Working Conference on Symbol Manipulation Languages, Bobrow,D., ed.,1971.

44. Raphael,B., The Thinking Computer, Mind Inside Matter, W.H. Freeman and Company, San Francisco, 1976.

45. Reboh,R., Sacerdoti,E., A Preliminary QLISP Manual, Artificial Intelligence Center. SRI International, Menlo Park, CA, 1973.

46. Sacerdoti,E., Planning in a Hierarchy of Spaces, Artificial Intelligence Center, SRI, Menlo Park, CA, 1974.

47. Sacerdoti,E., Fikes,R., Reboh,R., Sagalowicz,D., Waldinger,R., Wilber,B., "QLISP-A Language for the Interactive Development of Complex Systems," AFIPS Conference Proceedings, 1976.

48. Sammet,J., Programming Languages: History and Fundamentals, Prentice-Hall, NJ, 1969.

49. Sammet,J., "Roster for Programming Languages," Communications of the ACM, Vol. 19, No. 12, Dec 1976.

50. Sammet,J., "Programming Languages: History and Future," Selected Computer Articles. DCPCI. Washington, D.C., July 1976.

51. Sandewall,E., "Some Observations on Conceptual Programming," Machine Intelligence 8, Elcock and Michie eds., Halsted Press, NY, 1977.

52. Shaw,A., The Logical Design of Operating Systems. Prentice-Hall, NJ, 1974.

53. Simon,H., "The Simulation of Verbal Learning Behavior," Western Joint Computer Conference Proceedings, Vol. 19, May 1961.

54. Siklossy,L., Let's Talk LISP, Prentice-Hall, Englewood Cliffs, NJ, 1976.

55. Sussman,G., McDermott,D., "From PLANNER to CONNIVER, a Genetic Approach-Why Conniving is Better than Planning," Proceedings AFIPS WJCC, Vol. 41, Dec 1972.

56. Swinehart,D., Sproull,B., SAIL, Stanford AI project Op Note 57.2, Stanford Univ., 1971.

57. Teitelman,W., INTERLISP Reference Manual, XEROX, Palo Alto Research Center, 1978.

58. Tonge,F., "Summary of a Heuristic Line Balancing Procedure," Computers and Thought, Feigenbaum and Feldman, eds., McGraw-Hill Book Co., Inc., NY, 1963.

59. Travis,L., Honda,M., LeBlanc,R., Zeigler,S., "Design Rationale for TEIOS, A Pascal-Based AI Language," Proceedings of the Symposium on AI and Programming Languages, ACM, 1977.

60. van Emden,M., "Programming With Resolution Logic," Machine Intelligence 8, Elcock and Michie eds., Halsted Press, NY, 1977.

61. Waldinger,R., "Achieving Several Goals Simultaneously," Machine Intelligence 8, Elcock and Michie, eds., Halsted Press, 1977.

62. Warren,D., Peireira,L., Peireira,F., "PROLOG-The Language and its Implementation Compared with LISP," Proceedings of the Symposium on AI and Programming Languages, ACM, 1977.

63. Weissman,C., LISP 1.5 Primer, Dickenson Publishing Co., Inc., Belmont CA, 1967.

64. Winston,P., Artificial Intelligence, Addison-Wesley Publishing Co., Menlo Park,CA,1979.

65. Wirth,N., Systematic Programming: An Introduction, Prentice-Hall, NJ, 1973.

66. Woodward,P., "List-Programming," Advances in Programming and Non-numerical Computation, Fox,L., ed., Pergammon Press, NY, 1966.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Cheron Vail 8616 Lepus Rd San Diego, CA 92126	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
5.	Dr. Lance R. Wilson Forecasting Division Land O'Lakes, Inc. 614 McKinley Place Minneapolis, MN 55413	1
6.	Dr. Richard Sorenson Code 03 NAVPERSRANDCEN San Diego, CA 92152	1
7.	CDR Ronald A. Vail 1301 Bobcat Ln Alpine, CA 92001	1
8.	LCDR Stephen L. Reitz NSTR 1902 W. Minnehaha Ave. St. Paul, MN 55104	1
9.	Dr. James R. Callan Code 302 (n) NAVPERSRANDCEN San Diego, CA 92152	1

Thesis
V124 Vail
c.1 Data and control
structures required for
artificial intelligence
programming languages.
193722