An experimental expert system development environment for the VAX computer operating under the VMS operating system
/ by T. Gomi, N. Nakamura

Government of Canada | Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP-85-046

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: AN EXPERIMENTAL EXPERT SYSTEM DEVELOPMENT ENVIRONEMENT FOR THE VAX COMPUTER OPERATING UNDER THE VMS OPERATING SYSTEM .

AUTHOR(S): T. Gomi
N. Nakamura

ISSUED BY CONTRACTOR AS REPORT NO: AAIS 84-005

PREPARED BY: Applied AI Systems, Inc.
P.O. Box 13550
Kanata, Ontario
K2K 1X6

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 06ST.36001-3-4454

DOC SCIENTIFIC AUTHORITY: R.A. Millar

CLASSIFICATION: UNCLASSIFIED

DATE: 6/5/85

rec'd May 6/85

(2)

An Experimental Expert System Development Environment for the VAX Computer operating under the VMS Operating System

Technical Report No. AAIS-84-005

By
(1)
T. Gomi
N. Nakamura

Applied AI Systems, Inc.
P.O. Box 13550
Kanata, Ontario
K2K 1X6

Under DSS/DOC contract 06ST.36001-3-4454

Version: 20FEB85

C O N T E N T S

G L O S S A R Y

AASC Advanced Autonomous Spacecraft Computer, a spacecraft computer system concept developed at CRC (CRC/AASC)

AI Artificial Intelligence, a subdiscipline of Computer Science (Computer Science/AI)

ASPTP Almost as Simple as Possible Theorem Prover, an AI tutorial theorem prover developed for Smart Systems Technology by Drew McDermott of Yale University. (AI/Languages/ASPTP)

ATN Augmented Transition Network, a language parsing methodology proposed by William Woods, then of BBN. Here, ATN is a tutorial software system developed for Smart Systems Technology by Drew McDermott of Yale University designed to teach and explore the basic concepts of NL parsing.(AI/NL/ATN/'ATN')

CLisp A dialect of LISP, and its language system including an interpreter developed at the University of Massachusetts at Amherst (AI/Languages/LISP/CLisp)

CMU Carnegie-Mellon University

CRC Communications Research Centre, Department of Communications (DOC/CRC),

DOC Department of Communications, Government of Canada

DUCK A deductive retrieval system developed by Drew McDermott of Yale University. It is an AI system language with the ability to develop non-monotonic logic systems. (AI/Languages/DUCK)

KBS Knowledge-Based System (AI/KBS). Synonym for Expert System, except in the KBS the knowledge source is not necessarily attributed to an expert.

MIT Massachussets Institute of Technology

MPROLOG A prolog language system developed and marketed by Logicware of Toronto. A Prolog dialect (AI/Languages/Prolog/MPROLOG).

NL Natural Language

OPS5 A production system development language developed by Carnegie-Mellon University (AI/Languages/OPS5).

PDSS Program Development SubSystem, a software development environment for MPROLOG (AI/Environment/PDSS).

POC Proof of Concept.

PSN Procedural Semantic Network. A semantic network description language developed at the University of Toronto (AI/Languages/PSN)

SAMS Spacecraft Autonomy Management System, a substructure of the hierarchical design of the AASC (AASC/SAMS)

SANS Simplified Associative Network System, a simplified semantic network language developed by Kenneth Hayes of Smart Systems Technology. (AI/Languages/SANS)

WM OPS5 Working Memory (AI/Languages/OPS5/WM)

Acknowledgements

The Spacecraft Autonomy Management System (SAMS) was developed by the authors for the Communications Research Centre (CRC) of the Federal Department of Communications (DOC) under contract to the Department of Supply and Services (Contract Number 06ST.36001-3-4454). Authors are thankful for the support given by Dr. S.P. Altman and Mr. R.A. Millar of the Communications Research Centre.

Summary

The SAMS is conceived as the top layer of the Advanced Autonomous Space Computer (AASC) hierarchy developed at the CRC during the past three years. The SAMS layers are characterized by their use of Artificial Intelligence (AI) techniques. A set of expert systems were developed in 1984 as a Proof of Concept (POC) experimental system, and a series of experiments were conducted using them.

This document describes the software development environment used for developing the expert system and other AI systems. The environment was established on the VAX-11/780 computer (running VMS) at the Simulations & Analysis Laboratory of the CRC at Shirleys Bay, Ottawa, Canada. This work was accomplished during the course of the POC experimental system development. The environment exists as a collection of AI languages and tools. Example programs are given for each of the software packages that constitute the environment.

1. Introduction

Early AI development environments were constructed on main frame computers such as Digital Equipment Corporation's PDP-10. The DEC-20 series of computers such as DEC 2060 was another AI standard in earlier days.

In 1980, MIT completed the first Lisp machine prototype. This was quickly taken up by two commercial interests, Lisp Machine Inc, now of Los Angeles California, and Symbolics Inc. of Cambridge, Massachusetts. Both companies are today successfully marketing these machines after several revisions of hardware and software. The most important difference between this class of machine and a conventional computer is that in these machines Lisp functions are directly executed by the microcoded or hard-wired control units of the hardware, rather than by software emulation. One of the major drawbacks of this class of machine, however, is an extremely poor cost performance. Imported price of an avarage AI work station is about $200,000. Yet such a workstation only supports one user per installation. Multi-user versions of these machines started to appear in the market, but per user cost is still much higher than in conventional workstations for non-AI computing.

In spite of impressive throughputs and amenities offered by AI workstations, the Lisp machine has had a limited penetration into the AI communities of North America and of the world. The rest of the communities have gradually shifted from mainframe machines to super minicomputers, most notably, Digital Equipment Corporation's VAX-11 family of machines. This shift became dominant in the late 70s and early 80s. That happened to be the period when UNIX was gaining popularity, first in universities, then in industry and government agencies, as students trained in UNIX became a main force in non-academic computing. Development by the University of California at Berkley of Franz Lisp, and its inclusion into the Berkley UNIX created a standard AI development environment which was accepted by many university departments, corporate laboratories, and government agencies.

However, UNIX never gained popularity as an operating system for real-time applications. As the popularity of VAX computers itself increased, and their use in on-line, real-time applications grew, the relative importance of VMS as a real-world operating system increased. Still very limited in number and variety, there are now several software packages that can be used under a VMS operating system as AI development tools.

An AI development environment built on a VAX machine is far less costly than the AI workstations mentioned above. It is most suitable for the earlier phases of building up an in-house AI capability. Building fair size prototype AI systems or small target AI systems can be done without draining system resources. However, once a major target system begins its production run, the squeeze is often felt by itself and other programs running on the same machine. Hence a large VAX machine, either running UNIX or VMS, can be made into a multi-user AI development system for several users safely, but never as a satisfactory AI target system.

An important development in AI software is the emergence of industrial grade AI system shells (expert system shells and natural language shells). These software packages can be fitted with a knowledge base specifically developed for an application domain, and made into a more or less customized AI system. This approach not only cuts short the development time for AI systems drastically, but also in most cases increases the reliability of the developed AI system.

Software development for AI systems, such as expert systems, requires a set of tools somewhat different from conventional programming tools. Most of the differences are found in the nature of AI processes which are drastically different from conventional numerical computation. The language for developing expert systems and other AI systems used to be almost exclusively Lisp. The situation has changed after the growth in popularity of Prolog and other logic programming languages. Another trend is the use of conventional computer languages such as FORTRAN, PASCAL, C, BLISS, etc., in the implementation of AI systems. These are often a second implementation of a system first built in Lisp. Hence the emphasis in these cases is on performance and increased portability.

2. The over-all structure of the expert system development system

2.1 Objectives of the environment

The expert system development environment described in this document has been developed to fulfill the following objectives:

(1) To allow construction of expert systems rich enough to be considered a proof of concept system.

Simple AI programs can be easily written using basic AI languages like Lisp or Prolog. These languages may be used to construct more serious applications, but that often takes more experience in knowledge representation and reasoning techniques. High level AI languages or expert system shells may be useful to guide initial attempts at programming.

(2) To provide a reasonably easy entry point for those who intend to enter the field of AI programming.

Programming AI systems such as expert systems is non-trivial to programmers experienced only in conventional programming methods. Software packages that can demonstrate the significance of the difference readily will be highly educational.

(3) To cover all major approaches of expert system building (software packages selected for the aproach are shown in bracket):

- rule-based knowledge representation and reasoning (CLISP, OPS5, DUCK, MPROLOG),
- logic programming (ASPTP, DUCK, MPROLOG),
- problem solver (ASPTP, DUCK),
- inference network (DUCK, SANS, MPROLOG),
- frame-based semantic network (SANS, PSN),
- introductory parsing techniques (ATN).

Earlier expert systems almost exclusively used rules or productions as the basic knowledge representation mechanism. Most commercially available expert systems in

the market today adopt this knowledge representation. Semantic network representation, which was popular but experienced a failure in the late-60s, is making a come back with the improved nodal expression and enhanced taxonomy, and is often used in conjunction with a production system.

It is now well understood that, in order to create 'deep' or detailed causal models of an expertise, production systems are not adequate. Frame-based semantic networks are often viewed as vehicles to support such serious, sophisticated, and richer knowledge representation concepts.

The present state of the art in semantic networks is still far from practical. This is in spite of intensive R&D activities in the subfield of knowledge representation. In order to make the new representation techniques feasible, one may need the fifth generation computer hardwares with massively parallel computational elements. However, this is not a valid excuse for not pursuing this technique using whatever is available today. It is the theoretical understanding and acquisition and fluent usage of design know-how that will take the longest time. These studies can easily take longer than the development of the first highly parallel fifth generation hardware.

Logic programming has been 'discovered' by several large scale AI projects, such as national Fifth Generation Computer Systems (FGCS) projects of several countries. These projects' basic premise is that logic is probably the most important single area of study in AI system development. With the emergence of more implicit logic and re-investment in the study of common sense, their claim seems to have a foundation.

With the increased understanding of the capabilities and the limitations of existing expert system technology, and with existing pressure to bring expert systems into practical application, many will begin to realise that interfacing expert systems to a real-world application is often more important than issues internal to expert systems themselves. Language parsing technique is a basis of Natural Language (NL) systems. NL systems may be connected to expert systems to create an intelligent interface, and hence their technical bases must be understood.

(4) To construct an AI programming environment that is highly cost effective.

An AI workstation costs between $35,000 and $200,000. They are designed primarily for single users, those which are not are very expensive. Today, not many can justify the expense when nobody has successfully demonstrated the universal usefulness of the expert system technology. Depending solely on basic AI languages, such as Lisp and Prolog, on the other hand, costs the developer a long build up time. By carefully selecting a set of software tools, a less expensive but moderately powerful development environment may be created using a popular multi-user computer.

2.2 Structure of the environment

The development environment consists of a number of software packages. They are additional layers to the VMS operating system running on a VAX-11/780 super minicomputer, as shown in Figure 2.1. All of them may be used to develop an expert system. The height in the diagram indicates the level of abstraction the software package represents. For example, using PSN, one can represent events and objects more abstract than those representable by SANS, and OPS5 than CLISP.

PDSS is not a programming language, but is a development support system for MPROLOG. However, since they together create an appearance of a more abstract language interface to the user, it is represented in the diagram as it is.

The version of Franz Lisp, a famous Lisp dialect, used here as a basis for a number of AI languages is a private copy of the language which was developed at Carnegie-Mellon University (CMU) in the late 70s. While it can be accessed from languages that lay on top of it, it is an older version (Opus 34) and some aspects of the language are already different. Similarly, NISP is an understructure of DUCK (a macro library) and again accessible from DUCK. Because of the nature of Lisp, a macro, or a compound command automatically becomes a command that operates at a higher level of abstraction. These two software packages are not treated as independent software modules.

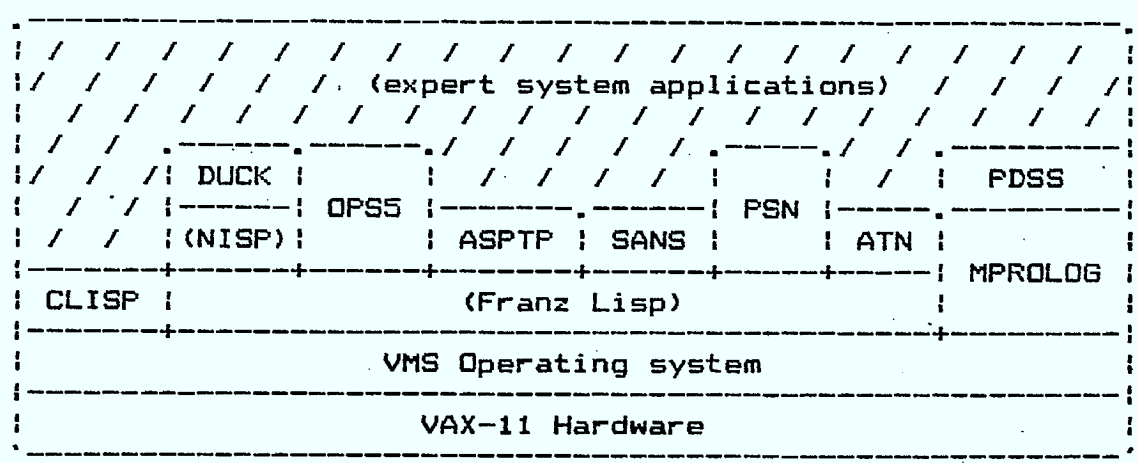


Figure 2.1 The expert system development environment

CLISP is a Lisp dialect developed at the Computer Science department of the University of Massachussets at Amherst for the past several years. It has relatively limited facilities but runs efficiently in a VAX/VMS environment.

DUCK is an AI language system developed by Prof. Drew McDermott of the Massachusetts Institute of Technology (MIT) and Yale University. Its truth maintenance features are unique among expert system shells. It has rather limited input/output capabilities and command syntax is terse. DUCK has a natural way of merging different programming styles such as logic programming, functional programming, and rule-based system descriptions.

OPS5 was originally developed by John McDermott and Charles Forgy of CMU as a series of production system languages (OPSn). Its theoretical background goes back to Simon/Hubert's study of human reasoning models studied in Cognitive Science. The language has gained popularity recently, particularly after the success of XCON (R1) and XSEL expert systems written in OPS5. Its weakness is the lack of truth maintenance features and ability to handle uncertainties. The former is being tackled by a group in IBM Yorktown. There are commercialized versions of this software available in the market which offer improved performance and technical support. Unless immediate major expert system development is planned using the language, it is the opinion of the authors that the current version described herein is sufficient for research purposes.

ASPTP is a tutorial problem solver developed for Smart Systems Technology (SST) and is based on a logic programming paradigm. Its formalism is much like that of DUCK, simply because it was developed by the same author as a simplified version of DUCK. It allows both forward and backward chaining and serves as a good introductory logic programming language/ problem solver.

SANS was also developed for SST by Dr. Ken Hayes. It is a simple frame-based semantic network language for constructing small to medium size semantic networks. Though relatively simple, it possesses all the basic features a semantic network language should have.

PSN, on the other hand, is a highly elaborate and sophisticated version of a frame-based semantic network system. It has been developed at the University of Toronto over the past several years. Portions of PSN are still under development. It is suitable for studying highly complex semantic network systems. A large and rich system may be developed using PSN, but its performance on VAX computers is limited because of its complexity.

ATN parser is not directly connected to expert system building. It is a tutorial parser written by Drew McDermott for SST for educational purposes. It has basic mechanisms to practice both syntactic and semantic parsing.

MPROLOG is a dialect of Prolog developed in Europe during the last few years and imported to North America by Logicware of Toronto. It is a well-debugged, well-packaged production quality AI language. Compared to some other versions of Prolog, such as Sigma Prolog of the U.K., it may be judged less elegant. However, its strengths are a facility to allow modular construction of Prolog programs (original Prolog is not modular), and a very rich set of well-appointed built-in predicates. PDSS is built around MPROLOG and serves as an environmental support to the user of the language.

3. The logic programming paradigm

Logic programming is an approach to AI, originating in Europe. It is currently used in various AI projects in Britain, other European countries and Japan, including the Fifth Generation Computer System (FGCS) projects of these countries. Its application in North America has been limited because of the dominance of Lisp as the standard AI language. However, there is a move there to reevaluate its potential. Universities (Stanford, Syracuse, among others) and private companies (IBM, SRI International, Honeywell, and several others) have been showing an active interest in the logic programming approach to AI.

The concept of logic programming can also serve as a unifier of recent innovations in the field of software engineering, database technology, computer architecture, and AI. Logic programming is also the missing link between knowledge engineering, a powerful but an expensive process, and parallel computing, a known solution to some computing power problems. It also bridges the gap between the new software technology based on reasoning and the new computer architecture epitomized by the recent emergence of non-Von Neuman machines. A paper by Kowalski [Kowalski 83] (attached as Appendix *5) of Imperial College, London, gives a concise summary of the programming method.

Prolog is a language system that implements a notion of logic programming called Horn clause logic programming. It is important that the two are not confused.

3.1 MPROLOG

3.3.1 Features of MPROLOG

MPROLOG is a dialect of Prolog which originated at the University of Edinburgh in the mid-1970s. A version of Edinburgh Prolog was transported to Hungary by visiting researchers, and developed there from 1979-83. Currently the language system is being developed for enhancements and North American adaptation by Logicware Inc. of Toronto.

A Prolog program consists of a collection of predicates formed into Horn clauses. A predicate can either be a rule (often called an implication) or an assertion. Rules are stored in rule bases, while assertions are stored in a scratch pad memory, a temporary storage, or a database.

MPROLOG is easily transportabile and is currently implemented on several machines including the following:

- IBM VM/CMS,
- DEC VAX-11/750, 780, 782, running under VMS or UNIX,
- Motorola 68000 based machines such as SORD and the SUN Micro work station,
- Tektronix 4404 AI Workstation,
- IBM PC and XT.

Except for the IBM PC version, MPROLOG software is accompanied by a comprehensive software development environment called the Program Development Sub-System (PDSS). It contains the MPROLOG interpreter, a pretranslator, a consolidator (linker), an editor, a tracer, a librarian, a help facility, a run control mechanism, and a module management facility. A compiler is to be added to some versions in the future (VM/CMS version of PDSS has a compiler now).

The PDSS features the following:

- Interactive program editor,
- On-line help facility,
- Program trace,
- User-defined exception handling,

- Windowing,

- Automated garbage collection,

- About 240 built-in predicates (except for the IBM PC version)

MPROLOG programs may be developed in modules using the module management facilities of the PDSS and MPROLOG. Modules are connected non-hierarchically and argument values exchanged via inter-module channels created by pdss' import/export, global/local, visible/hidden, and other interface commands.

Interlanguage communication supports in MPROLOG are very limited. An MPROLOG-FORTRAN linkage is about to be completed on the VM/CMS version, followed by other versions (except for IBM PC version). For the VAX/VMS version, attempts to link modules written in different languages using the mailbox facility has been successful. This approach will allow, for example, a module written in PASCAL to exchange parameters with an MPROLOG program. Since logic programming, particularly its Horn clause subset represented by Prolog has its limitations, it is desirable to establish generous inter-language links. More effort will be necessary to improve this capability of the language.

Another limitation is the language's ability to handle numbers. There is presently no provision for handling floating point numbers in MPROLOG. Hence, no built-in functions such as trigonometric functions exist. All representations and calculations of numbers must be done using integers, the maximum absolute value of which must be less than 1000000 (i the VMS implementation). Again, work is underway at Logicware to support floating point numbers and operations using them.

The original Edinburgh's DEC-10 Prolog syntax may be made acceptable to MPROLOG by using a switch in the PDSS. The switch has an additional position at which rules can be expressed in pseudo-English style of "If...Then..." format. However, the switch controls the over-all PDSS environment. No mixture of formats is allowed among modules or within a modules.

The planned future enhancements of the MPROLOG language system includes the following:

- semi-intelligent tracing

- compiler

- screen-oriented editor

- window management

- optimizer

3.1.2 Using MPROLOG on VAX/VMS

The MPROLOG commands are documented in "MPROLOG Language reference" and the PDSS commands in "MPROLOG Development System reference" manuals [Logicware 84]. Assuming that PDSS and MPROLOG are installed and made available to the user, the following steps exemplify a typical PDSS/MPROLOG session:

```
$ pdss                          Invoking PDSS

MPROLOG (Vx.y) Program Development SubSystem x.y  Rev.
(c) 1984 LOGICWARE Inc., Toronto Canada

                                Herald message, x.y = version number.
                                PDSS prompts the user with a ':'.

:consult  <file specification>

                                Bringing in file(s) containing user's
                                developed MPROLOG codes, rules, and
                                assertions.  This command will be
                                omitted when building an entirely new
                                program.

<a list of predicates being loaded>
        .
        .
        .
                                Predicates are displayed as they  are
                                read in in the form

                                    <predicate-name/N>

                                where,

                                    N = number of arguments.

<file specification> CONSULTED.
```

End of a consultation sequence. Any number of **consult** commands may be issued.

:<pdss commands>

.
.
.

PDSS commmands are used to create, modify, and delete MPROLOG predicates.

:?<predicate>

Request to the PDSS for the execution of a predicate. In place of '?', the following prompts may be used:

?-, :, :-

Also, any of the PDSS commands may be issued here to further edit the predicates created or **consulted**.

:bye

Terminates a PDSS session.

Normal exit from MPROLOG PDSS

Termination message by system

$

End of a PDSS session. Back to VMS environment.

During the PDSS session described above, a user may enter the VMS DCL (Digital Command Language) environment by pressing the ENTER key to the ':' prompt. Any of the VMS commands may be issued then, including another PDSS. A **LOGOFF** command brings the user back to the last PDSS environment.

A PDSS session may be interrupted by an exception. For example, upon reaching the alloted call count limit (the number of times a predicate is invoked), the PDSS interrupts the session by informing,

```
call limit reached
 In call of <predicate which was interrupted>
        Limit = 10000
Function (h for help)?
```

By entering **h**, one gets the following menu of commands which can be used to manage the interruption:

```
 p   - enter new PDSS level
 b   - backtrace
 a   - abandon execution
 c   - continue
 f   - fail
 s   - contents of the stack
 r   - redo the broken call
 i   - user handled interrupt
 h   - help
Function (h for help)?
:
```

In addition to this set of exception management commands, invoking a second copy of PDSS (use **p** command), the PDSS commands may be issued to modify or inquire about the PDSS run time environment. Issuing a PDSS command without creating a second copy of PDSS will result in an automatic termination of the current run followed by the execution of the entered PDSS command.

```
p
Entering level 2 of PDSS
:
```

From within the new PDSS level, the PDSS **set** command may be used to change the parameters of the old PDSS run time environment. For example, the call limit (number of predicate invocations) may be increased from the default (10000 calls) by entering,

```
set/ call_limit = 100000
call_limit = 100000
```

A **bye** and a **c** command (continue) must be entered to resume the interuupted PDSS session.

```
bye
```

```
Exit from level 2 of PDSS
Function (h for help)?
c
```

Alternatively, the run may be terminated by entering an **a** command upon interruption. The rationale for the call limit provision is to provide a way of regaining control from an infinite loop, which may be caused by an error in a predicate definition. Another commonly used way of causing an exception in PDSS execution is to arbitrarily interrupt a run using ctrl-c:

```
^C
. . .
external interrupt
   In call of <interrupted predicate>
Function (h for help)?
```

The procedure explained above for examining or altering the PDSS run time environment is applicable here.

3.1.3 Examples of MPROLOG program

3.1.3.1 Predicate to compute length of a list

A predicate which computes the length of a list is presented as a simple example of MPROLOG program. The predicate, list_length, takes two arguments: the length of the list, and the list itself. The example demonstrates Prolog's power in defining 'what should be calculated' as opposed to 'how should be calculated'. It simply states that,

- the length of a list is zero, if it is empty

- otherwise, if the length of the tail of the list is L, then the length of the list itself including the head will be L+1.

The predicate uses a recursion. However, unlike more common tail-recursion, it recurses on a clause which is not the last element of the predicate. The program listing is followed by an example run, which was traced using the PDSS trace facility to show the step-by-step execution of the recursion.

```
list_length(0,[]) .
list_length(Z,[X|Y]) :-
     list_length(L,Y), plus1(L,Z) .

plus1(L,Z) :-
     plus(L,1,Z) .
```

```
?list_length(L,[a,b,c,d,e,f,g,h]).
OK
    L = 8
Continue (y/n) ?
```

```
trace list_length
list_length/2 TRACED
trace plus1
plus1/2 TRACED
?list_length(L,[a,b,c,d,e,f,g,h]).
> list_length(_311,[a,b,c,d,e,f,g,...])
 > list_length(_346,[b,c,d,e,f,g,h])
  > list_length(_380,[c,d,e,f,g,h])
   > list_length(_414,[d,e,f,g,h])
    > list_length(_448,[e,f,g,h])
     > list_length(_482,[f,g,h])
      > list_length(_516,[g,h])
       > list_length(_550,[h])
        > list_length(_584,[])
        + list_length(0,[])
        > plus1(0,_550)
        + plus1(0,1)
       + list_length(1,[h])
       > plus1(1,_516)
       + plus1(1,2)
      + list_length(2,[g,h])
      > plus1(2,_482)
      + plus1(2,3)
     + list_length(3,[f,g,h])
     > plus1(3,_448)
     + plus1(3,4)
    + list_length(4,[e,f,g,h])
    > plus1(4,_414)
    + plus1(4,5)
   + list_length(5,[d,e,f,g,h])
   > plus1(5,_380)
   + plus1(5,6)
  + list_length(6,[c,d,e,f,g,h])
  > plus1(6,_346)
  + plus1(6,7)
 + list_length(7,[b,c,d,e,f,g,h])
 > plus1(7,_311)
 + plus1(7,8)
+ list_length(8,[a,b,c,d,e,f,g,...])
    L = 8
Continue (y/n) ?
n
OK
```

3.1.3.2 Family tree problem

A relatively large family tree or a lineage is created in terms of assertions, such as **father** (Bob, John)., and relations or rules, such as:

grandfather (A, B) :- father (A, C), father (C, B).

The relations are common to all family trees that may be built and examined using this program, while the assertions are particular to an individual family tree. As described earlier, the former will be stored in a knowledge base, while the latter will be placed in a database in the toy expert system. The example chosen here is a family tree of some of the Greek gods. The structure of the tree is shown in Figure 3.2.

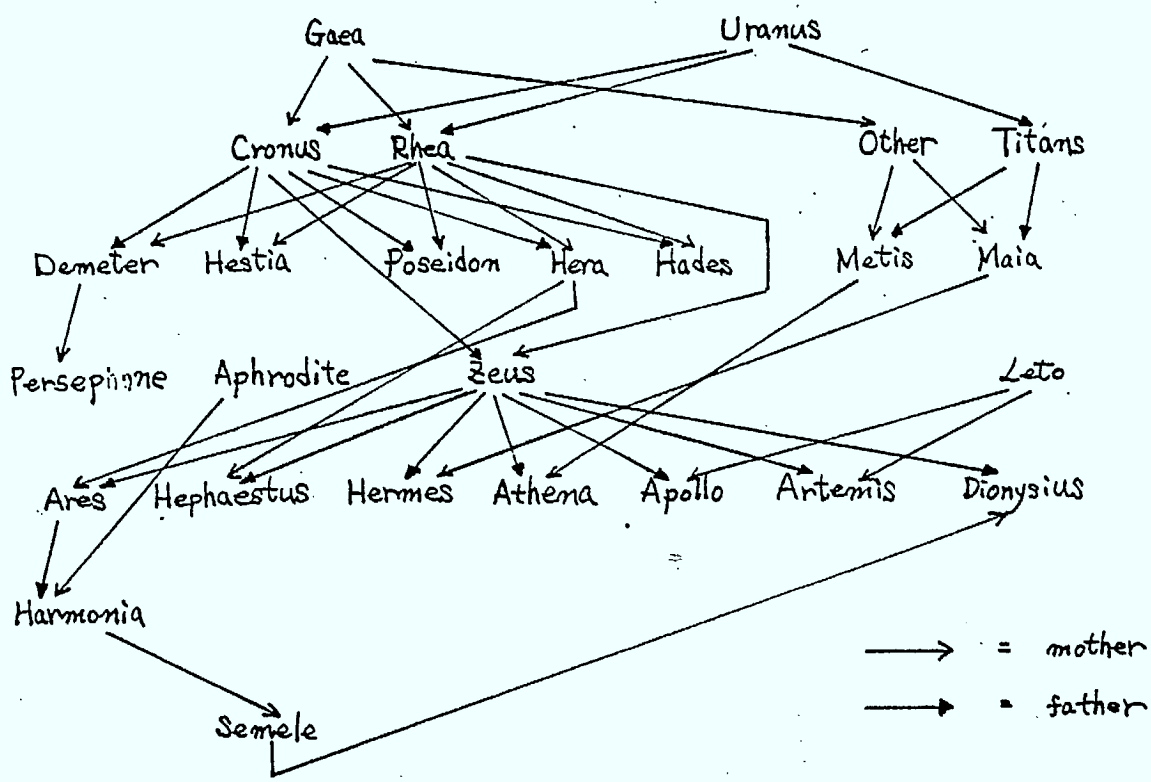


Figure 3.2 Family tree of Greek gods

```
$-ty-greek.log
/*****************************************************/
/*                                                   */
/*     MPROLOG example of family relationships       */
/*                                                   */
/*****************************************************/

/*                                                   */
/*     Rules                                         */
/*                                                   */

     father(F,C) :-
        parent(F,C),male(F).

     mother(M,C) :-
        parent(M,C),female(M).

     grandfather(GF,GC):-
        parent(GF,P),
        parent(P,GC),
        male(GF).

     grandmother(GM,GC):-
        parent(GM,P),
        parent(P,GC),
        female(GM).

     siblings(SX,SY):-
        mother(M,SX),mother(M,SY).
     siblings(SX,SY):-
        father(F,SX),father(F,SY).

     immediateSiblings(I_SX,I_SY):-
        mother(M,I_SX),
        mother(M,I_SY),
        father(F,I_SX),
        father(F,I_SY).

     ancestor(A,D):-
        parent(A,D).
     ancestor(A,D):-
        parent(Z,D),ancestor(A,Z).


     child(C,P):-
        parent(P,C).

     descendant(D,A):-
        child(D,A).
     descendant(D,A):-
        child(Z,A),descendant(D,Z).
```

```
/*                                                          */
/*        listing_of_conclution                             */
/*                                                          */

     is_parent_of(Child,Parent_list):-
        setof(X,parent(X,Child),Parent_list).
     is_parent_of(Child_list,Parent):-
        setof(X,parent(Parent,X),Child_list).

     is_child_of(Parent,Child_list):-
        setof(X,child(X,Parent),Child_list).
     is_child_of(Parent_list,Child):-
        setof(X,child(Child,X),Parent_list).

     is_grandfather_of(Grandchild,GrandFather_list):-
        setof(X,grandfather(X,Grandchild),
        GrandFather_list).
     is_grandfather_of(Grandchild_list,Grandfather):-
        setof(X,grandfather(Grandfather,X),
        Grandchild_list).


     is_grandmother_of(Grandchild,Grandmother_list):-
        setof(X,grandmother(X,Grandchild),
        Grandmother_list).
     is_grandmother_of(Grandchild_list,Grandmother):-
        setof(X,grandmother(Grandmother,X),
        Grandchild_list).

     is_siblings_of(Sibling,Sibling_list):-
        setof(X,siblings(X,Sibling),Sibling_list).

     is_immediateSiblings_of(I_sibling,I_sibling_list):-
        setof(X,immediatesiblings(X,I_sibling),
        I_sibling_list).

     is_ancestor_of(Descendant,Ancestor_list):-
        setof(X,ancestor(X,Descendant),
        Ancestor_list).
     is_ancestor_of(Descendant_list,Ancestor):-
        setof(X,ancestor(Ancestor,X),
        Descendant_list).


     is_descendant_of(Ancestor,Descendant_list):-
        setof(X,descendant(X,Ancestor),
        Descendant_list).
     is_descendant_of(Ancestor_list,Descendant):-
        setof(X,descendant(Descendant,X),
        Ancestor_list).
```

```
/*                                                          */
/*        Database of the Greek Gods                        */
/*                                                          */

     parent(gaea,cronus).
     parent(gaea,rhea).
     parent(gaea,other).
     parent(uranus,cronus).
     parent(uranus,rhea).
     parent(uranus,titans).
     parent(cronus,demeter).
     parent(cronus,hestia).
     parent(cronus,zeus).
     parent(cronus,poseidon).
     parent(cronus,hera).
     parent(cronus,hades).
     parent(rhea,demeter).
     parent(rhea,hestia).
     parent(rhea,poseidon).
     parent(rhea,zeus).
     parent(rhea,hera).
     parent(rhea,hades).
     parent(other,metis).
     parent(other,maia).
     parent(titans,metis).
     parent(titans,maia).
     parent(demeter,persephone).
     parent(hera,ares).
     parent(hera,hephaestus).
     parent(metis,athena).
     parent(maia,hermes).
     parent(aphrodite,harmonia).
     parent(zeus,ares).
     parent(zeus,hephaestus).
     parent(zeus,hermes).
     parent(zeus,athena).
     parent(zeus,apollo).
     parent(zeus,artemis).
     parent(zeus,dionysius).
     parent(leto,apollo).
     parent(leto,artemis).
     parent(ares,harmonia).
     parent(harmonia,semele).
     parent(semele,dionysius).
```

```
female(gaea).
female(rhea).
female(other).
female(demeter).
female(hera).
female(metis).
female(maia).
female(aphrodite).
female(leto).
female(harmonia).
female(semele).
female(hestia).
female(persephone).
female(athena).
female(artemis).

male(uranus).
male(cronus).
male(titans).
male(zeus).
```

```
?is_parent_of(hestia,X).
    X = [cronus,rhea]
Continue (y/n) ?
?is_parent_of(X,cronus).
OK
    X = [demeter,hades,hera,hestia,poseidon,zeus]
Continue (y/n) ?
?is_child_of(rhea,X).
OK
    X = [demeter,hades,hera,hestia,poseidon,zeus]
Continue (y/n) ?
?is_child_of(X,zeus).
OK
    X = [cronus,rhea]
Continue (y/n) ?
?is_grandfather_of(hera,X).
OK
    X = [uranus]
Continue (y/n) ?
?is_grandfather_of(X,uranus).
OK
    X = [demeter,hades,hera,hestia,maia,metis,poseidon,zeus]
Continue (y/n) ?
?is_grandmother_of(hera,X).
OK
    X = [gaea]
Continue (y/n) ?
?is_grandmother_of(X,gaea).
OK
    X = [demeter,hades,hera,hestia,maia,metis,poseidon,zeus]
Continue (y/n) ?
?is_siblings_of(athena,X).
OK
    X = [apollo,ares,artemis,athena,dionysius,hephaestus,hermes]
Continue (y/n) ?
?is_immediatesiblings_of(ares,X).
OK
    X = [ares,hephaestus]
Continue (y/n) ?
?is_ancestor_of(harmonia,X).
OK
    X = [aphrodite,ares,cronus,gaea,hera,rhea,uranus,zeus]
Continue (y/n) ?
?is_ancestor_of(X,rhea).
OK
    X = [apollo,ares,artemis,athena,demeter,dionysius,hades,harmonia,
        hephaestus,hera,hermes,hestia,persephone,poseidon,semele,zeus]
Continue (y/n) ?
?is_descendant_of(X,zeus).
OK
    X = [cronus,gaea,rhea,uranus]
Continue (y/n) ?
```

```
?is_descendant_of(uranus,X).
OK
    X = [apollo,ares,artemis,athena,cronus,demeter,dionysius,
hephaestus,hera,hermes,hestia,maia,metis,persephone,poseidon,
rhea,semele,titans,hades,harmonia, ...]
Continue (y/n) ?
?is_descendant_of(X,zeus).
OK
    X = [cronus,gaea,rhea,uranus]
Continue (y/n) ?
```

3.2 ASPTP

3.2.1 Features of ASPTP

ASPTP is a problem solver included in the SST tutorial software package. It is written entirely in Lisp and allows the user to conduct simple problem solving sessions. A session is conducted interactively, and consists of entering facts and rules in the form of assertions, and then posing questions in the form of a hypothesis (or a theorem) to be proven. The sessions are effctive as a tutorial of the theorem proving paradigm, and as an introduction to the more sophisticated problem solver, DUCK.

There are only two commands (predicates) in the ASPTP: **assert** and **bc**. Facts are entered as a simple assertion using the **assert** command, while a rule is entered as a Horn clause, again using the same command. A hypothesis to be proven is presented as a goal of a goal-driven inference, using the **bc** (backward chainig) command. A rule may be presented either as a backward or a forward chaining rule. When a forward chaining rule is added to the database, new assertions may be made using that and other rules which may become relevant because of the new rule. There will be no automatic assertions when a backward chaining rule is added. It will be invoked only in the process of proving a theorem.

3.2.2 Using ASPTP

The SST tutorial software (ASPTP, SANS, OPS5, ATN) is stored under the directory

SYS$SYSDISK:[PACKAGE.SST.TUTORIAL.SSTC.AIC1]

The following sequence of commands, which is common to all of the tutorial software, must be issued to access them:

```
$ set default sys$sysdisk:[package.sst.tutorial.sstc.aic1]
```

User's default directory is set to that of the tutorial software

```
$ @sstcourse
```

Assigns a version of Lisp that is appropriate to the tutorial package. System responds with the following:

```
Previous logical name assignment replaced
Previous logical name assignment replaced
Previous logical name assignment replaced
Previous logical name assignment replaced


$ lisp
```

Enter the lisp environment.

```
 54c00 bytes read into 2c00 to 577ff
Franz Lisp, Opus 34

         SMART SYSTEMS TECHNOLOGY
      Artificial Intelligence Course

A.I. Course Software Selections
Type (asptp)  for ASPTP deductive retriever
     (ops5)   for OPS5 productive system
     (sans)   for SANS associative network system
     (parser) for an ATN natural language parser
     (loadloop) for examples of control structures
```

Note: above steps must be followed by all SST Tutorial software.

```
-> (asptp)
```

Select ASPTP. Note all inputs are in lower case letters. ASPTP responds with its herald messages:

```
[fasl sst$lib:asptp.0]

Leaving Workspace: background
In Workspace: asptp

Leaving Workspace: asptp
In Workspace: background
Type (navig) to load in the NAVIG database
     (arith) to load in arithmetic plus and times
     (family) to load in the Family Tree database
nil
->
```

Assert facts and rules using the **assert** predicate of ASPTP to the prompt "->", to construct a problem, and then to activate the theorem prover by entering one or more of the **bc**

predicates. The assert predicate has the following format:

-> (**assert** '<assertion>)

where, <assertion> is either a fact or an implication (rule). Examples of a fact would be:

(brother ted chris) Ted is a brother of Chris, or Chris is a brother of Ted.

(is-east-of victoria vancouver)

Victoria is located east of Vancouver.

(meaner_than lucy marcie)

Lucy is meaner than Marcie.

Rules are, in general, defined using a variable. A variable is any lowercase alphanumeric string preceded by a "?". Examples of an implication would be:

(<- (wife ?x ?y) (and (spouse ?x ?y) (female ?x)))

If x is a spouse of y and x is a female, then x is y's wife.

(-> (is female ?p) (is woman ?p))

If p is a female then p is a woman.

In these examples the relational operator "<-" implies a backward chaining or a goal driven inference, while "->" means a forward chaining, or data driven inference. In ASPTP, there is no distinction between a knowledge base and a database. Both rules and facts are written into a database.

In forward chaining, when a rule is asserted, any facts that may be justified by the new rule will be asserted in the database automatically. In the above forward chaining example, if the database already had an assertion

(female alice)

then an assertion

(woman alice)

will be made immediately after the rule is asserted.

Once facts and rules are entered in the database, one can request the ASPTP's resolution mechanism to prove various hypotheses. The format of the request is:

-> (bc '<hypothesis>)

where, **bc** stands for backward chain, signifying the fact that the ASPTP tries to resolve a hypothesis using a goal-driven inference.

A hypothesis has the identical format as an assertion discussed above. In fact, there is no actual distinction between the two. In problem solving, one tries to prove that there are supporting evidences (assertions) that can be used to prove a hypothesis (assertion). The process may chain to whatever depth necessary using available rules.

As one might suspect from the syntax of the ASPTP, the problem solver is implemented entirely in Lisp, as a set of Lisp functions. As such, other Lisp functions may be used in conjunction with the ASPTP codes.

Also available to the ASPTP and other SST tutorial software (OPS5, SANS and ATN), and the DUCK is the Lisp Workspace Manager. A workspace is a set of related functions and data that are in the main memory. A workspace can be entered by executing (**workspace** '<workspace name>). Anything defined after this will be done so in the workspace. In order to preserve a workspace, execute (**wsave** '<file specification>). The current workspace will be saved in file <file name>. A sister function (**load** '<file specification>) will load the file and restore the workspace. When switching a workspace, issue (**workspace** '<workspace name>). Issue as many (**wsave** '<file specification>) as necessary to save functions defined thereafter. The rule applies to assertions, productions, and grammars defined in ASPTP and DUCK, OPS5, and ATN parser, respectively. In order to detatch a session from all workspaces, execute (**workspace** nil).

The function (**wsym** '<symbol>) associates <symbol> with the current workspace. All its properties will be saved with the workspace. The default saving monitor normally saves any

symbols that are likely to become necessary in future sessions. (**unwsym** '<symbol>) flushes <symbol> from the current workspace. (**cursyms**) returns the symbols in the current workspace. Functions **editp** and **editf** permit in-memory editing of predicates (assertions) and Lisp functions, respectively. (**workspace-push** '<workspace name>) pushes the current workspace on a stack and goes to a new workspace, <workspace name>. (**workspace-pop**) restores the last workspace. (wsmerge '<workspace name>) merges the current workspace with workspace <workspace name> and makes it current.

3.2.3 ASPTP program example

The sequence below demonstrates a simple ASPTP session:

```
-> (assert '(is_a fred male))

                                "Fred is a male" is entered as
                                 a fact.
Asserting (is_a fred male))
asserted
                                 System confirms what is
                                 asserted by reciting.

-> (assert '(-> (is_a ?x male) (is_a ?x human)))

                                 "If x is a male, x is a human."
                                 is entered as a rule.

Asserting (-> (is_a (|?| x) male) (is_a (|?| x) human))

                                 System confirms.

Asserting (is_a fred human)
asserted
                                 Because the rule is a forward
                                 chaining one, "then Fred is a
                                 human" is implied and asserted
                                 automatically by the system.

-> (assert '(<- (is_a ?x human) (is_a ?x female)))

                                 A rule which says, "If x is a
                                 female then x is a human" is
                                 entered as a backward chaining
                                 rule.
```

```
Asserting (<- (is_a (!?! x) human) (is_a !x! x) female))
asserted
```

System recites, but since the rule is a backward chaining one, no implication occurs.

```
-> (assert '(is_a lucy female))
```

"Lucy is a female" entered.

```
Asserting (is_a lucy female)
asserted

-> (bc '(is_a fred human))
```

An enquiry, "Is Fred a human?".

```
Goal: (is_a fred human)   Queue length: 0
(nil)
```

System takes up the hypothesis as a goal to be solved. However, this goal has been asserted as a result of the forward chaining rule.

Note: (nil) in ASPTP means a "Yes". Some intermediate results are not shown here.

```
-> (bc '(is_a lucy human))
```

"Is Lucy a human?" is asked.

```
Goal: (is_a lucy human)   Queue length: 0
Implication: (<- (is_a (!?! v1) human) (is_a (!?! v1) female))
```

A rule that supports the goal is found. It is a backward chaining rule,

```
Subgoal: ((is_a lucy female))
```

of which condition is "If x is a female." This now becomes a goal to be proven. Since this is one of the assertions entered,

```
Discharged: (is_a lucy human)
```

the original goal is now proven.

```
Goal: (is_a lucy female)   Queue length: 0
Assertion (is_a lucy female)
```

System also tries to prove the enquiry directly, by looking for the goal itself in the database. Since this assertion does not exist there,

```
RESULT: nil
```

"no" is returned from this search. **Note:** (nil) = "Yes", nil = "No".

```
Discharged: (is_a lucy female)
```

And the attempt is given up.

```
1 Chainings
(nil)
```

Number of inferences reported. The over-all answer to the question is a "Yes".

```
-> (bc '(is_a bill human))
```

"Is Bill a humman?"

```
nil
```

System does not know anything about "bill". So it answers no.

```
-> (bc '(is_a ?x human))
```

A question "Who are human?" is asked.

```
Goal: (is_a (!?! x) human)   Queue length: 0
```

System tries to find a straight assertion of the form "x is a human."

```
Assertion: (is_a fred human)
```

And finds the one asserted as a result of the firing of the forward chaining rule.

```
RESULT: ((x fred))
```

x = Fred is given as an answer.

```
Implication: (<- (is_a (!?! v1) human) (is_a (!?! v1) female))
```

The backward chaining rule picked up.

```
Subgoal: ((is_a (!?! x) female))
```

The rule has "If x is a female" as a condition.

```
Goal: (is_a (!?! x) female)
```

Queue length: 0

Which is posted as a goal to be proven.

```
Assertion: (is_a lucy female)
```

An assertion which matches the goal is found in the database.

```
RESULT: ((x lucy))
```

x = Lucy is found.

```
Discharged: (is_a (!?! x) female)
```

The goal discharged.

```
1 Chainings.
(((x lucy)) ((x fred)))
```

Answers listed.

```
-> (bc '(is_a ?x person))
```

"Who is a person?"

```
nil
```

ASPTP does not know any "person" as that concept does not exist in database.

3.3 DUCK

3.3.1 Features of DUCK

DUCK is a hybrid AI language for developing predicate-calculus rules that may consist of one or more of the following programming styles.

- Rule-based knowledge representation,

- Logic programming,

- Functional programming.

DUCK is best suited for constructing non-monotonic reasoning systems and intelligent databases in which deductive retrieval of information is conducted using built-in inference rules. The consistency of the database is maintained using a truth maintenance system. DUCK is currently the only commercially available system which can handle non-monotonic logic. Other AI programming applications in which Lisp or Prolog is normally used can also be written in DUCK. Its drawback is in its slow execution.

DUCK was developed during the past decade both at the MIT and Yale University by Professor Drew McDermott. The software is now being marketed by Smart Systems Technology of McLean Virginia, and runs on VAX/VMS, VAX/UNIX, and Symbolics 3600 Series computers.

DUCK combines four programming paradigms successfully used in AI applications:

(1) Logic Programming

First order predicate calculus is supported. Both conjunctive (AND) and disjunctive (OR) operators are used to form relations to be stored in the knowledge base, and to issue queries. Unification and backtracking are used as in Prolog as the basic execution control mechanisms. In fact, these are about the only execution control mechanisms in DUCK. Semantic information is separated from the algorithm in knowledge and data bases, unlike conventional procedural languages. This is a strength DUCK and other logic programming languages share, and it makes program update easier.

One of the applications of DUCK based on this characteristic is rapid prototyping. Rules and assertions are defined in knowledge and data bases, after their extraction from an expert. Subsequent testing is easy using DUCK's control mechanisms.

(2) Rule-based systems

Unlike conventional databases or systems built around a database, DUCK builds a rule-based system. A rule here can be thought of as a deduction: a conclusion of true beliefs from true premises. Rules offer a significant increase in computational power over conventional databases. DUCK allows data to be deduced, rather than explicitly stored or computed by procedures. Such a data form based on rules is sometimes called virtual data. Structural changes to virtual data need not be explicitly made. Since rules are, as described in (1) above, independent from program control structure, they can be added or altered more easily than in conventional procedural programming.

(3) Non-monotonic reasoning

Handling of reasoning with assumptions or inconsistent information is achieved through a technique called dependency directed backtracking. DUCK maintains a history of data dependencies during its reasoning, so that changes to an earlier assumption can be reflected throughout the database. Considering that human reasoning includes many adjustments in its process due to newly discovered assertions or data, this feature is very important in creating highly flexible intelligent systems. In fact, non-monotonicity of a reasoning mechanism will likely become a basic requirement in future AI system design. DUCK is most advanced in this respect among similar AI tools.

Using the truth maintenance system, DUCK can at first assume default values for variables whose values are unknown. These values are traced throughout subsequent deductive processes. If at a later time an assumption is found to be wrong, the correct value is assigned and corresponding updates are made to other assertions in the database.

DUCK also maintains 'data pools'. This mechanism allows hypothetical 'what if' situations to be specified in the database. In effect, a data pool creates a copy of the data base by saving the differences between the original. This mechanism provides an opportunity to explore several hypothetical situations with minimum memory overhead.

(4) Deductive search

DUCK uses both forward and backward search techniques. The chaining process begins when rules invoke other rules in the knowledge base. In forward chaining, the implications of a given predicate are added to the database. Backward

chaining begins with a goal and searches for assertions which will support that goal. The processing time required for many applications using searches increases exponentially as the size of the search tree grows. By mixing forward and backward chaining strategies, DUCK reduces significantly the amount of search. This contrasts with Prolog's backward chaining only control strategy (though Prolog may be used to program a strategy to do otherwise) and OPS5's forward chaining only control strategy.

In addition, heuristic search may be performed under the user's control by using facilities provided in DUCK. This may (depending on the heuristics introduced) result in a further reduction in search time. Also, DUCK has a mechanism to allow partial searches. This feature not only reduces the search time but also aids the debugging.

Further details on the language system are described in [Mcdermott 83] (Appendix *3). Unfortunately, this rather unreadable manual/functional description is the only document available for the system.

3.3.2 Using DUCK on VAX/VMS

Assuming that DUCK is installed in a system directory, the following sequence initiates a DUCK session:

```
$ DUCK (or duck)

154c00 bytes read into 2c00 to 1577ff
duck version !DUf8-4.131!
```

DUCK herald message. User may require an additional memory allotment from the system manager. DUCK uses an arrow (->) as prompt.

```
-> (load '<file-specification>)
```

The contents of a file (DUCK predicates) are loaded into memory. This may not be the case when starting an entirely new DUCK program.

```
->
```

enter, modify, or delete

```
                                        predicates in knowledge
                                        base using commands available
                                        under DUCK.

-> (exit)
                                        Exiting DUCK.

$
                                        Back to the VMS.
```

3.3.3 Examples of DUCK program

Two programs are shown below as examples of a DUCK program.

3.3.3.1 A classification expert

This program has a small amount of knowledge in its knowledge base to conduct simple discrimination tasks among animals.

At the beginning of a session the user imagines an animal in his/her mind. The program asks the user a number of questions concerning the features of the animal that the user chose. The sequence with which these questions are asked is governed by a set of rules so that unintelligent questions such as "Does it nurse its young with milk?" be asked following a "no" answer to "Is it a warm blooded animal?". The user must answer these questions consistently in regard to the animal.

Answers to these questions are stored in templates in the database. The template has a structure:

```
(answer <question-id> <reply>)
```

where,

question-id is an identifier of the question asked,
reply is a **yes/no** answer to the question.

The filled template then becomes a clause in the antecedent of rules which are used to identify animals. Matching is sought in backward chaining to identify the animal. Obviously, only a rule with all its AND conditions asserted fires.

Shown below is a program listing, followed by a log of trial sessions:

```
;       A classification expert
;
;-----------------------------------------------------------------------
(workspace-push 'duck-dumd-animal)
;
;       Define the various animals
;
(defsymtype ANANIMAL SYMBOL)
(duclare ANANIMAL chicken crocodile dog dolphin frog mosquito
 robin snake tiger tuna whale worm)
;
;       Define the various features of animals
;
(defsymtype FEATURE SYMBOL)
(duclare FEATURE backbone warm-blooded nurse
 water huge domesticated gills
 gills-then-lungs legs begin fly)
;
;       Define the questions
;
(defpred (question FEATURE ?f STRING ?q)
        (question backbone "Does it have a backbone?")
        (question warm-blooded "Is it a warm-blooded animal?")
        (question nurse "Does it nurse its young with milk?")
        (question water "Does it live in the water?")
        (question huge "Is it huge?")
        (question domesticated "Is it a commonly domesticated
        animal?")
        (question fly "Can it fly?")
        (question gills
        "Does it have gills and live all its life in the water?")
        (question gills-then-lungs
        "Does it start life with gills and then become an air breather?")
        (question legs "Does it have legs?")
)
;
;       Define what is allowed for answers
;
(defsymtype YNANSWER SYMBOL)
(duclare YNANSWER yes no)
;
;       Define question order
;
(defpred (next-question FEATURE ?old YNANSWER ?tnil FEATURE ?new)
        (next-question begin yes backbone)
        (next-question backbone no fly)
        (next-question backbone yes warm-blooded)
        (next-question warm-blooded yes nurse)
        (next-question warm-blooded no gills)
        (next-question nurse yes water)
        (next-question nurse no fly)
        (next-question water yes huge)
        (next-question water no domesticated)
        (next-question gills no gills-then-lungs)
        (next-question gills-then-lungs no legs)
        )
```

```
;
;       User responses are kept as ANSWERs
;               We use begin as a way to start questions
;
(defpred (answer FEATURE ?f YNANSWER ?yesno)
        (answer begin yes))
;
;       Rule sequencing:A question about ?feature can be
;                       asked if it has not already been
;                       asked and its precusor has been
;                       answered correctly.
;
(defpred (askable FEATURE ?f)
        ((- (askable ?feature)
                (and (next-question ?prev ?ans ?feature)
                        (answer ?prev ?ans)
                        (thnot (answer ?feature ?yn))))
        )
;
;       Now we define the correspondence between animals
;       and features.
;
(duclare animal (fun PROP (ANANIMAL) ()))
(::: animal template :::
 ((animal ?a) (?a " is the animal")))

 (rule its-a-worm
        ((- (animal worm)
        (and (answer backbone no)
                (answer fly no)))
 )

(rule its-a-mosquito
        ((- (animal mosquito)
        (and (answer backbone no)
                (answer fly yes)))
)

(rule its-a-snake
        ((-     (animal snake)
                (and    (answer backbone yes)
                        (answer warm-blooded no)
                        (answer gills no)
                        (answer gills-then-lungs no)
                        (answer legs no)))
)

(rule its-a-crocodile
        ((-     (animal crocodile)
                (and    (answer backbone yes)
                        (answer warm-blooded no)
                        (answer gills no)
                        (answer gills-then-lungs no)
                        (answer legs yes)))
)
```

```
(rule its-a-frog
        ((-     (animal frog)
                (and    (answer backbone yes)
                        (answer warm-blooded no)
                        (answer gills no)
                        (answer gills-then-lungs yes)))
)

(rule its-a-tuna
        ((-     (animal tuna)
                (and    (answer backbone yes)
                        (answer warm-blooded no)
                        (answer gills yes)))
)

(rule its-a-chicken
        ((-     (animal chicken)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse no)
                        (answer fly no)))
)

(rule its-a-robin
        ((-     (animal robin)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse no)
                        (answer fly yes)))
)

(rule its-a-tiger
        ((-     (animal tiger)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse yes)
                        (answer water no)
                        (answer domesticated no)))
)

(rule its-a-dog
        ((-     (animal dog)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse yes)
                        (answer water no)
                        (answer domesticated yes)))
)
```

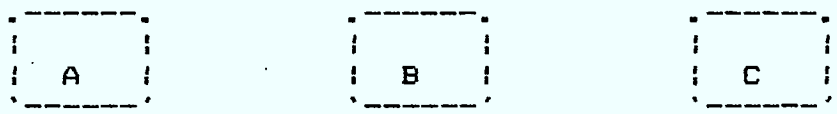
```
(rule its-a-dolphin
        ((-     (animal dolphin)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse yes)
                        (answer water yes)
                        (answer huge no)))
)

(rule its-a-whale
        ((-     (animal whale)
                (and    (answer backbone yes)
                        (answer warm-blooded yes)
                        (answer nurse yes)
                        (answer water yes)
                        (answer huge yes)))
)

(:= first-dp* dp*)
-----------------------------------------------------------------------
(de whatanimal ()
        (do [(a-ans (fetch '(animal ?x)) (fetch '(animal ?x)))
                (dp* (dp-push first-dp*))]
                [a-ans (progn (dp-kill dp*)
                        (cadr (assoc 'x (caar a-ans)))]
                (for-first-ans (fetch '(askable ?feature))
                (for-first-ans (fetch '(question ?feature ?english))
                        (princ ?english)
                (cond [(is-yes (read)) (premiss '(answer ?feature yes))]
                        [t (premiss '(answer ?feature no))])
                        )
                )
        )
)
(de animals ()
        (do [(la nil)]
                [nil]
                (princ "Do you want to play animal?")
                (cond [(is-yes (read))
                        (:= la (whatanimal))
                        (terpr)
                        (terpr)
                        (princ "The Animal is a ")
                        (princ la)
                        (princ " !")
                        (terpr)
                        (terpr)]
                [t (return)])
        )
)
(princ "ANIMAL.DUC Loaded")
(terpr)
(princ "TYPE (animals)")
(terpr)
```

```
=> (animals)
Do you want to play animal?yes
Does it have a backbone?yes
'Is it a warm-blooded animal?yes
'Does it nurse its young with milk?yes
'''''Does it live in the water?no
'Is it a commonly domesticated
        animal?no
''''''

The Animal is a tiger !

Do you want to play animal?yes
Does it have a backbone?yes
'Is it a warm-blooded animal?yes
'Does it nurse its young with milk?no
'Can it fly?no
'''''

The Animal is a chicken !

Do you want to play animal?yes
Does it have a backbone?no
'Can it fly?no
'''

The Animal is a worm !

Do you want to play animal?yes
Does it have a backbone?yes
'Is it a warm-blooded animal?no
'Does it have gills and live all its life in the water?no
'Does it start life with gills and then become an air breather?no
'Does it have legs?yes
''''''

The Animal is a crocodile !

Do you want to play animal?no
nil
-> (exit)
```

3.3.3.2 Data pool control

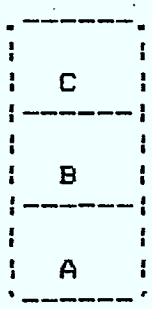
A test program was written and run to test the datapool feature of DUCK. In DUCK, datapool is used jointly with the truth maintenance system to maintain dependency directed backtracking. With this mechanism, changes to data can be reflected throughout the data base.

In the following example, situations created in a "block's world" that has three boxes A, B, and C is saved in different data pools. For instance, data pool 1 records a situation in which the three boxes are placed side by side on the table, as ahown in Figure 3.3a, while data pool 3 has



a. Data pool 1

b. Data pool 2



c. Data pool 3

Figure 3.3 A block's world example

them stacked up in the order of A. B, and C, from the surface of the table up. These situations may depict intermediate states created by a robot executing a task. The task could be "Begin with situation in datapool 1 and end with a situation in which box C sits on top of A and box A sits on top of box B.

The DUCK's truth maintenance system would allow the robot to 'go back' to an earlier situation and try a different sequence of actions from there. Such switching of context without tracing back actions taken in the past in reverse chronological order, can be done if a history of data dependencies is maintained. The reasoning mechanism of DUCK may switch data pools and conduct inference on assertions particular to the context represented by the new data pool.

The program shown below demonstrates creation of three data pools. Note in a logic programming system without this facility, contradiction among assertions, such as "B is on A" as in data pool 2 and 3 and "B is on table" as in data pool 1, cannot be tolerated, drastically limiting the real-world applicability of the approach.

```
;
;     Data pools
;
(workspace-push 'datapool)

(deftype object SYMBOL)
(duclare object table A B C)

(defpred (ON object ?x object ?y))

(:= first-dp dp*)

(premiss '(ON A table))
(premiss '(ON B table))
(premiss '(ON C table))

(terpr)
(princ "***** Contents of data pool 1 *****")(terpr)
(terpr)
(for-each-ans (fetch '(ON ?x ?y))
   (princ "        ")
   (princ ?x)
   (princ " is on ")
   (princ ?y)
   (princ ".")
   (terpr)(terpr)
)
```

```
(:= second-dp (dp-push dp*))
(let ((dp* second-dp))
   (premiss '(ON B A))
   (erase '(ON B table))
```

```
   (terpr)
   (princ "***** Contents of data pool 2 *****")(terpr)
   (terpr)
   (for-each-ans (fetch '(ON ?x ?y))
      (princ "       ")
      (princ ?x)
      (princ " is on ")
      (princ ?y)
      (princ ".")
      (terpr)(terpr)
   )
   (:= third-dp (dp-push dp*))
   (let ((dp* third-dp))
      (premiss '(ON C B))
      (erase '(ON C table))

      (terpr)
      (princ "***** Contents of data pool 3 *****")
      (terpr)(terpr)
      (for-each-ans (fetch '(ON ?x ?y))
         (princ "       ")
         (princ ?x)
         (princ " is on ")
         (princ ?y)
         (princ ".")
         (terpr)(terpr)
      )
    )
)

(terpr)
(princ "***** Contents of data pool 1 *****")
(terpr)(terpr)
(for-each-ans (fetch '(ON ?x ?y))
   (princ "       ")
   (princ ?x)
   (princ " is on ")
   (princ ?y)
   (princ ".")
   (terpr)(terpr)
)
```

```
=> (load 'datapoolg.duc)
IN WORKSPACE datapool
Autosave mode: save
''''
***** Contents of data pool 1 *****

        C is on table.

        B is on table.

        A is on table.

'''''
***** Contents of data pool 2 *****

        B is on A.

        A is on table.

        C is on table.

'
***** Contents of data pool 3 *****

        C is on B.

        B is on A.

        A is on table.

***** Contents of data pool 1 *****

        C is on table.

        B is on table.

        A is on table.

t
```

4. The Semantic Network paradigm

4.1 SANS

4.1.1 Semantic network and SANS

Semantic network is a knowledge representation technique initially proposed in the early 60s. The idea originated from the Cognitive Science camp of a then loosely formulated school of AI. The formalism has fundamental psychological and physiological overtones. The approach of capturing and accessing human thought processes based on a method similar to that found in these sciences caught popularity but died away by the early 70s. Failure was due to weakness in formalism and too much flexibility in the interpretation of the meaning provided by semantic network.

A new breed of semantic networks began to reappear in the latter half of the 70s following Minski's historical 'frames' declaration of 1975 [Minsky 75]. Semantic network approach was then re-instated with 'frames' as its central concept. One of the weaknesses of the earlier semantic network was the loose and freer definition of nodes and arcs in the network. If one replaces arbitrarily defined nodes with frames, and arcs with taxonomical and similarity/dissimilarity links among frames, an entirely new type of semantic network formalism is created. This is indeed what was done.

KL-ONE, or Knowledge Language One [Brachman 78] is the first well known semantic network system of this generation. Others include Carnegie-Mellon University's Schema Representation Language, or SRL [Fox 78], Stanford University's UNITS system [Stefik 80], and Schubert's efforts [Schubert 76]. Current research centers around the methods of procedural attachment to slots of a frame, of including stronger deduction mechanisms to the network and of interconnecting frames using production rules. Toronto University's PSN [Mylopoulos et al 83], Krypton being developed at Schlumberger Palo Alto Research (SPAR) [Brachman et al 83], and KEE 20 system developed by IntelliCorp [Kunz et al 84], respectively, are examples of current development projects.

SANS is a frame based semantic network (called associative network in the SANS for historical reasons) system developed for tutorial purposes. It is mostly aimed at deepening the understanding of the semantic network concept, while allowing development of simple applications using

semantic network representation. Basic concepts of semantic network organization, generic vs. instantiated nodes, valued slots of a node, property inheritance, and demons as a form of procedural attachment are all included in the systems.

SANS uses nodes, slots, values, and demons to construct a semantic network. A node is also called an object, much in the same sense the term is used in object-oriented programming. Objects, or nodes are described in SANS in terms of their properties and the relationships among them. Further details of the SANS' features and its access commands are found in the manual [Hayes 83] (Appendix *2) and a tutorial note [Berg 84] (Appendix *1).

Not all reasoning problems are suitable for semantic network representation. In fact, the present application of semantic network is still very limited because of its limited ability to represent. Only classification problems and certain types of diagnosis problems are effectively solved using semantic network approach. SANS has the limitation too. It is best suited in problems where there is a strong taxonomy in the application. Basic understanding of the semantic network paradigm can be obtained by reading text books [Winston 84a] (Chapter 8), [Winston 84b] (Chapter 22), Nilsson 80] (Chapter 9), [Cohen and Figenbaum 82] (All three volumes, use index to look for 'semantc network').

Assuming that an appropriate application domain is defined, in order to develop a SANS-based system, one proceeds as follows:

(1) Describe the application in the form of a taxonomy. This may involve clarification and definition of basic concepts (eg., managers, workers, superiors, subordinates, departments, merchandise, customers, equipment, etc.) and their relationships to the other concepts,

(2) Develop a template node structure using commands in SANS for that purpose. The template node defines a generic concept in the system in terms of attribute slots and their default values. Template nodes for all basic concepts evident in the application must be developed. Then they must be connected according to the taxonomy developed in step (1).

(3) Define and implement, again using SANS commands, procedures to be attached in the form of demons to some of the slots in the template nodes,

(4) Using system commands provided in SANS for that purpose, develop an instantiated node structure that corresponds to actual instances of the template node (generic concept) structure. For example, the concept of 'APEX Corporation' may be developed as an instantiated case for the generic concept 'company', and 'Shipping dept.' for 'department',

(5) Using commands to activate demons, execute attached procedures and compute values or cause actions desired. A possible action may be to fill a slot of another node.

4.1.2 Using SANS on VAX/VMS

The SST tutorial software (ASPTP, SANS, OPS5, ATN) is stored under directory

SYS$SYSDISK:[PACKAGE.SST.TUTORIAL.SSTC.AIC1].

Follow the steps shown in Section 3.2.3 above until the tutorial software menu is displayed.

```
-> (sans)
```

Select SANS. All inputs in lower case.

```
[fasl sst$lib:rsans.O]

Leaving Workspace: background
In Workspace: sans

Leaving Workspace: sans
In Workspace: background
Type (rstest) to load in example associative network
nil

-> (workspace 'mysans)
```

Define your own workspace.

```
-> (load '<file specification>)
```

Load predefined SANS program. This command may not succeed if the user does not have sufficient priviledge. Use SANS interactively, if not.

.
.
.

SANS commands follow.

4.1.3 SANS program examples

The following is an example SANS program which deals with basic statistics of Canadian provinces. Two sets of nodal (or frame) structures are constructed: template and instance. For each structure, nodes (or frames) are created by defining their slots and the value of the slots. Lisp functions are written to go around the defined frames and collect statistics by tabulating values from a specified slot.

Both template and instance frames are displayed below, followed by the results of the run.

```
;
;     SANS example
;
;     An associative network for geographical
;     information about provinces in Canada
;-----------------------------------------------------------------
;
     (workspace  'mysans)
;
;
;     Define Template-nodes and slots
;
     (make-template  'country 'nil)
     (add-slot  'country 'capital 0)

     (make-template  'province 'country)
     (add-slot  'province 'provincial-capital 0)
     (add-slot  'province 'area 0)
     (add-slot  'province 'population 0)
     (add-slot  'province 'floral-emblem 0)
     (add-slot  'province 'date-become-province 0)
;
;
;     Define Instance-nodes
;
     (make-instance  'nil 'country 'Canada)
     (put-value  'capital 'Ottawa 'Canada)
```

```
(make-instance  'Canada 'province 'B.C.)
(make-instance  'Canada 'province 'ALTA.)
(make-instance  'Canada 'province 'Sask.)
(make-instance  'Canada 'province 'Man.)
(make-instance  'Canada 'province 'Ont.)
(make-instance  'Canada 'province 'P.Q.)
(make-instance  'Canada 'province 'Nfld.)
(make-instance  'Canada 'province 'N.B.)
(make-instance  'Canada 'province 'N.S.)
(make-instance  'Canada 'province 'P.E.I.)

(put-value  'provincial-capital 'Victoria 'B.C.)
(put-value  'provincial-capital 'Edomonton 'Alta.)
(put-value  'provincial-capital 'Regina 'Sask.)
(put-value  'provincial-capital 'Winnipeg 'Man.)
(put-value  'provincial-capital 'Toronto 'Ont.)
(put-value  'provincial-capital 'Quebec 'P.Q.)
(put-value  'provincial-capital 'ST.Johns 'Nfld.)
(put-value  'provincial-capital 'Fredericton 'N.B.)
(put-value  'provincial-capital 'Halifax 'N.S.)
(put-value  'provincial-capital 'Charottetown 'P.E.I.)

(put-value  'area '948596 'B.C.)
(put-value  'area '661185 'Alta.)
(put-value  'area '651900 'Sask.)
(put-value  'area '650087 'Man.)
(put-value  'area '1068582 'Ont.)
(put-value  'area '1540680 'P.Q.)
(put-value  'area '404517 'Nfld.)
(put-value  'area '73437 'N.B.)
(put-value  'area '55490 'N.S.)
(put-value  'area '5657 'P.E.I.)

(put-value  'population '2184621 'B.C.)
(put-value  'population '1627874 'Alta.)
(put-value  'population '926242 'Sask.)
(put-value  'population '988247 'Man.)
(put-value  'population '7703106 'Ont.)
(put-value  'population '6027764 'P.Q.)
(put-value  'population '522104 'Nfld.)
(put-value  'population '634557 'N.B.)
(put-value  'population '788960 'N.S.)
(put-value  'population '111641 'P.E.I.)

(put-value  'Floral-Emblem 'Flowering-Dogwood 'B.C.)
(put-value  'Floral-Emblem 'Wild-Rose 'Alta.)
(put-value  'Floral-Emblem 'Prairie-Lily 'Sask.)
(put-value  'Floral-Emblem 'Pasqueflower 'Man.)
(put-value  'Floral-Emblem 'White-Trillium 'Ont.)
(put-value  'Floral-Emblem 'White-Garden-Lily 'P.Q.)
(put-value  'Floral-Emblem 'Pitcher-Plant 'Nfld.)
(put-value  'Floral-Emblem 'Violet 'N.B.)
(put-value  'Floral-Emblem 'Trailing-Arbutus 'N.S.)
(put-value  'Floral-Emblem 'Ladys-Sipper 'P.E.I.)
```

```
(put-value  'Date-Become-Province '1871 'B.C.)
(put-value  'Date-Become-Province '1905 'Alta.)
(put-value  'Date-Become-Province '1905 'Sask.)
(put-value  'Date-Become-Province '1870 'Man.)
(put-value  'Date-Become-Province '1867 'Ont.)
(put-value  'Date-Become-Province '1867 'P.Q.)
(put-value  'Date-Become-Province '1947 'Nfld.)
(put-value  'Date-Become-Province '1867 'N.B.)
(put-value  'Date-Become-Province '1867 'N.S.)
(put-value  'Date-Become-Province '1873 'P.E.I.)
;
;
;
;      LISP functions
;
(de  provincial-capital (x)
   (get-value  'provincial-capital x ) )

(de  area (x)
   (get-value  'area x ) )

(de  population (x)
   (get-value  'population x ) )

(de  Floral-Emblem (x)
   (get-value  'Floral-Emblem x ) )

(de  Date-Become-Province (x)
   (get-value  'Date-Become-Privince x ) )
```

-> (provincial-capital B.C.)
victoria
-> (provincial-capital Alta.)
edmonton
-> (provincial-capital Man.)
winnipeg
-> (provincial-capital N.S.)
halifax
-> (provincial-capital Ont.)
toronto
-> (area B.C.)
948596
-> (area Alta.)
661185
-> (area Man.)
650087
-> (area N.S.)
55490
-> (area Ont.)
1068582
-> (population B.C.)
2184621
-> (population Alta.)
1627874
-> (population Man.)
988247
-> (population N.S.)
788960
-> (population Ont.)
7703106
-> (Floral-Emblem B.C.)
flowering-dogwood
-> (Floral-Emblem Alta.)
wild-rose
-> (Floral-Emblem Man.)
pasqueflower
-> (Floral-Emblem N.S.)
trailing-arbutus
-> (Floral-Emblem Ont.)
white-trillium
-> (Date-Become-province B.C.)
1871
-> (Date-Become-province Alta.)
1905
-> (Date-Become-Province Man.)
1870
-> (Date-Become-Province N.S.)
1867
-> (Date-Become-Province Ont.)
1867

4.2 PSN

4.2.1 Features of PSN

Based on Hecter Levesque's 1977 proposal, Procedural Semantic Network has been developed at the Computer Science Department of the University of Toronto under Professors John Mylopoulos and John Tsotsos during the past seven years, involving many research staff at the department. The system is one of the most sophisticated and advanced Knowledge Representation (KR) systems in the world today. While present implementation of the language is not efficient enough to be used in a great number of applications, it has already been proven useful in large scale prototypes of advanced expert systems [Tsotsos 81] [Shibahara et al 83].

The most salinus aspect of PSN is its rigid definition of the structural aspects of knowledge. Classes and relations are defined as entities representing generic concepts - like person, house, flower - while relations represent generic relationships such as parent_of, above, and citizen_of. Tokens and links are instantiated entities corresponding to classes and relations. Procedural elements are introduced into the language in terms of four access primitives attached to a class: TO-GET, TO-REM, TO-TST, and TO-PUT, for creating, deleting, testing and collecting objects.

There are three fundamental relationships defined in the PSN: IS-A, INSTANCE-OF, and PART-OF. Of these, IS-A relation is similar to that in many other semantic network systems and implies a generalization/specialization taxonomy. PART-OF relation is for aggregation/division, and INSTANCE-OF for catagorization. Most other semantic network languages, including the popular KL-ONE, do not distinguish the taxonomical differences as in PSN, which are very subtle and hard to handle properly. An application system with very elaborate descriptions of its components and relationships may be constructed using PSN. However, the performance of such a system will be poor and impractical for running on a VAX-11/780.

PSN has a hierarchical structure. Each layer of the hierarchical language offers a set of representational features that includes features of an inner layer. PSN/0 is the most fundamental layer supporting only the INSTANCE-OF relation. PSN/1 adds IS-A and a simple form of PART-OF to depict organizational knowledge in a system. PSN/2 introduces the more sophisticated PART-OF, along with similarity links and exceptions. Similarity links connect classes of similar attributes, and suggest other classes to be tried when a match fails between a given class and input data. When a

match failure occurs, an exception is raised. It determines which similarity link should be used to suggest other classes to be tried for matching. Although the development group has plans for further expansion (ie., PSN/3 on), it is unlikely that such development will happen.

Appendix *4 is a copy of PSN User's Manual.

4.2.2 Using PSN on VAX/VMS

There are two versions of PSN interpreter, PSN1 and PSN2, installed on A&SL VAX-11/780. Use PSN2 as follows:

```
$psn2
 77800 bytes read into 2c00 to 7a3ff
```

Note Franz Lisp is also loaded.

```
(include <user psn source file>)
```

Use include command to load PSN definitions.

```
[*list:436{68%}; fixnum:2{0%}; ]
[*list:446{67%}; fixnum:2{0%}; ]
[*list:456{66%}; fixnum:2{0%}; ]
              .
              .
              .
t
```

User PSN file loaded.

```
-> (Flora-Emblem Alta.)

Floral emblem of Alta. is Wild-Rose
```

PSN is ready for access using user defined knowledge base.

4.2.3 PSN program example

An example very similar to the one made for the SANS (Section 4.1) is written for PSN. The knowledge base stores in a structure, facts about Canada: population, land area, floral emblem, capital. Same sets of information are also stored for the provinces. A set of Lisp functions are provided to access the classes (frames) in which this

knowledge is stored. Some of them simply retrieve the knowledge, while others compute a value (eg., population density). In the last set of examples, PSN's own fetch function (:$) is used to retrieve information from frames. Shown below is the knowledge base developed, and the results of runs performed using the developed knowledge.

```
;
;
;     PSN example
;
;     A procedural semantic network for geographical
;     information about provinces in Canada
;
;-----------------------------------------------------------
;
;
;     Class definition for various geographical elements
;
;
     (:+ class (ident Name) nil nil nil)

     (:+ class (ident Geographical-unit)
        '((to-put stdputms))
        '(class)
        '((Head-slot slot)))

     (ident Head-slot)

     (:+ class (ident Geographical-entity) nil nil nil)

     (:+ Geographical-unit (ident Province) nil
        '(Geographical-entity)
        '((Provincial-capital Name)
          (Area number)
          (Population number)
          (Floral-Emblem Name)
          (Date-Become-Province number)))

     (ident Provincial-capital Area Population
            Floral-Emblem Date-Become-Province)

     (:+ Head-slot Province Provincial-capital)

     (:+ Geographical-unit (ident Country) nil
        '(Geographical-entity)
        '((Capital Name)
          (Area number)
          (Population number)
          (National-Emblem Name)
          ))
```

```
(ident Capital National-Emblem)

(:+ Head-slot Country Capital)

(:+ relation (ident Contains)
   '((domain Geographical-entity)
     (range Geographical-entity)) nil)

;
;
;-----------------------------------------------------------
;
;
;
;
;
;     Knowledge base definition
;
;
(ident Victoria Edomonton Regina Winnipeg Toronto
       Quebec ST.Johns Fredericton Halifax
       Charottetown)

(ident Flowering-Dogwood Wild-Rose Prairie-Lily
       Pasqueflower White-Trillium White-Garden-Lily
       Pitcher-Plant Violet Trailing-Arbutus
       Ladys-Sipper)

(ident Canada)

(ident B.C. Alta. Sask. Man. Ont. P.Q. Nfld. N.B.
       N.S. P.E.I.)
;
(mapcar (f:l (name) (:+ Name name nil))
   '(Victoria Edomonton Regina Winnipeg Toronto
     Quebec ST.Johns Fredericton Halifax
     Charottetown
     Flowering-Dogwood Wild-Rose Prairie-Lily
     Pasqueflower White-Trillium White-Garden-Lily
     Pitcher-Plant Violet Trailing-Arbutus
     Ladys-Sipper Ottawa Maple))
;
;
;     Class "Canada"
;
(:+ Country Canada '((Capital Ottawa)
                     (Area 9970000)
                     (Population 21830000)
                     (National-Emblem Maple)))
```

```
;
;     Provinces defined as a class
;

(:+ Province B.C. '((Provincial-capital Victoria)
                    (Area 948596)
                    (Population 2184621)
                    (Floral-Emblem
                     Flowering-Dogwood)
                    (Date-Become-Province
                     1871)))

(:+ Province Alta. '((Provincial-capital Edomonton)
                     (Area 661185)
                     (Population 1627874)
                     (Floral-Emblem
                      Wild-Rose)
                     (Date-Become-Province
                      1905)))

(:+ Province Sask. '((Provincial-capital Regina)
                     (Area 651900)
                     (Population 926242)
                     (Floral-Emblem
                      Prairie-Lily)
                     (Date-Become-Province
                      1905)))

(:+ Province Man. '((Provincial-capital Winnipeg)
                    (Area 650087)
                    (Population 988247)
                    (Floral-Emblem Pasqueflower)
                    (Date-Become-Province 1870)))

(:+ Province Ont. '((Provincial-capital Toronto)
                    (Area 1068582)
                    (Population 7703106)
                    (Floral-Emblem White-Trillium)
                    (Date-Become-Province 1867)))

(:+ Province P.Q. '((Provincial-capital Quebec)
                    (Area 1540680)
                    (Population 6027764)
                    (Floral-Emblem
                     White-Garden-Lily)
                    (Date-Become-Province 1867)))

(:+ Province Nfld. '((Provincial-capital ST.Johns)
                     (Area 404517)
                     (Population 522104)
                     (Floral-Emblem
                      Pitcher-Plant)
                     (Date-Become-Province 1947)))
```

```
(:+ Province N.B. '((Provincial-capital Fredericton)
                    (Area 73437)
                    (Population 634557)
                    (Floral-Emblem
                     Violet)
                    (Date-Become-Province 1867)))

(:+ Province N.S. '((Provincial-capital Halifax)
                    (Area 55490)
                    (Population 788960)
                    (Floral-Emblem
                     Trailing-Arbutus)
                    (Date-Become-Province 1867)))

(:+ Province P.E.I. '((Provincial-capital
                       Charottetown)
                      (Area 5657)
                      (Population 111641)
                      (Floral-Emblem
                       Ladys-Sipper)
                      (Date-Become-Province 1873)))

;
;
;     Canada - Provinces taxonomy
;

(:+ Contains Canada B.C.)
(:+ Contains Canada Alta.)
(:+ Contains Canada Sask.)
(:+ Contains Canada Man.)
(:+ Contains Canada Ont.)
(:+ Contains Canada P.Q.)
(:+ Contains Canada Nfld.)
(:+ Contains Canada N.B.)
(:+ Contains Canada N.S.)
(:+ Contains Canada P.E.I.)

;
;-----------------------------------------------------------
;
;
;     LISP functions to fetch the contents of PSN.
;-----------------------------------------------------------

;
;     To calculate the population density.
;
(defun Population-Density (P)
   (Population-Density-of P)(terpri)(terpri))
```

```
(defun Population-Density-of (P)
   (terpri)(terpri)
   (princ "Population density of ")
   (princ P)
   (princ " is ")
   (princ ( quotient (:$ Population P nil)(:$ Area P nil)))
   (princ " persons per square kilometer.")(terpri)
)

;
;     To calculate the  population density of all
;     provinces of country.
;
(defun Nationwide-Population-Density-for (C)
   (foreach province (:$ Contains C nil)
      (Population-Density-of province))(terpri)(terpri))

;
;     To get the National emblem
;
(defun National-Emblem (C)
   (terpri)(terpri)
   (princ "National emblem of ")
   (princ C)
   (princ " is ")
   (princ (:$ National-Emblem C nil))
   (terpri)(terpri)(terpri))

;
;     To get the Floral emblem of province
;
(defun Floral-Emblem (P)
   (terpri)(terpri)
   (princ "Floral emblem of ")
   (princ P)
   (princ " is ")
   (princ (:$ Floral-Emblem P nil))
   (terpri)(terpri)(terpri))
;
```

```
-> (National-Emblem Canada)

National emblem of Canada is Maple


nil
=>-(Floral-Emblem Nfld.)

Floral emblem of Nfld. is Pitcher-Plant


nil
-> (Population-Density Ont.)

Population density of Ont. is 7 persons per square kilometer.


nil
-> (Population-Density P.E.I.)

Population density of P.E.I. is 19 persons per square kilometer.


nil
-> (Nationwide-Population-Density-for Canada)

Population density of P.E.I. is 19 persons per square kilometer.

Population density of N.S. is 14 persons per square kilometer.

Population density of N.B. is 8 persons per square kilometer.

Population density of Nfld. is 1 persons per square kilometer.

Population density of P.Q. is 3 persons per square kilometer.

Population density of Ont. is 7 persons per square kilometer.

Population density of Man. is 1 persons per square kilometer.

Population density of Sask. is 1 persons per square kilometer.

Population density of Alta. is 2 persons per square kilometer.

Population density of B.C. is 2 persons per square kilometer.
```

```
-> (:$ Capital Canada nil)
Ottawa
-> (:$ Area Canada nil)
9970000
-> (:$ Population Canada nil)
21830000
-> (:$ National-Emblem Canada nil)
Maple
-> (:$ Provincial-capital N.S. nil)
Halifax
-> (:$ Area N.S. nil)
55490
-> (:$ Population N.S. nil)
788960
-> (:$ Floral-Emblem N.S. nil)
Trailing-Arbutus
-> (:$ Date-Become-Province N.S. nil)
1867
-> (exit)
```

5. The production system paradigm

5.1 CLisp

5.1.1 Feature of CLisp

CLisp was developed between 1979 and 1983 by a group at Computer and Information Science Department of the University of Massachussets at Amherst. It is aimed to be run on VAX-11 family of computers running under VMS operating system. This is the first serious non-UNIX Lisp for this computer before Common Lisp. It has been the base language for a numnber of AI projects at that department, including the well-known HEARSAY-II speech understanding system project conducted there by Prof. Victor Lesser and Dr. Dan Corkill.

CLisp has an extensive on-line help facility which explains virtually all built-in functions. Entering

(help)

user gets a list of functions explained by the facility. According to (**help** HELP), one of the explanations under this facility, help for a particular CLisp function can be obtained by typing:

(**help** <category> <function>)

if the user is not using CLisp editor, or

(**clisp-help** <category> <function>)

from within CLisp, anywhere. A <category> <function> may be of the following fromat:

- an alphanumeric string,
- a match-al (wildcard) symbol, "*".
- any of the above folowed by "..."

Examples are :

(**help** misc func)	Prints out the description of the function 'func' from category miscellaneous.
(**help** constructors *)	will print out the descriptions of all constructor functions - functions that returns lists,

S-expressions, attribute-lists, or from their arguments.

(help input_output ...)	Prints out the names of all input output functions.
(help * fc-average)	Prints out the description of function fc-avarage, looking under all categories.
(help ???)	Prints out the names of all categories without help information.
(help *...)	Prints out the entire help document.

In addition, the CLisp help facility follows the identical control/display format as the VMS Help facility. User can access hierarchically structured help information selecting them from the list of topics on which additional information is available. Prompts such as 'Topics?' and 'Subtopc?' guide the process, as in the VMS Help facility.

5.1.2 Using CLisp on VAX/VMS

CLisp interpreter may be accessed by entering the following sequence to a VMS prompt:

$ clisp

Clisp: Enter Lisp functions ... Typically, one or more of the following functions are entered at the beginning of a Clisp session.

CLisp: **(load-file** '<filename>)

Reads in file <filename>. File type must be .LSP.

CLisp: **(create-file** '<filename>)

Defines a new file to be created in the session.

CLisp: **(defun** <fuction name> (<arguments>)
(<function definition> . . .

Defines a new function. More

```
                                  definitions or executions of a
                                  function follow

CLisp: (help <topic>)
                                  Prints out information on the
                                  use of CLisp in general and on
                                  all of its functions.
        .
        .
        .

CLisp: (exit)                     Terminates a session.

End CLisp Run
 dd-mmm-yyyy hh:mm:ss.xx          Date and time of termination
CPU Time (seconds)   = 19.32      and system statistics.
Pagefaults           = 931
Garbage Collections = 6
End CLisp run

$                                 Back to VMS.
```

5.1.3 CLisp program example

The well-known monkey and banana problem is chosen to demonstrate CLisp in problem solving.

At the beginning, a monkey, a table, and a banana all are located separately in any of the three rooms, room 1, 2, or 3. The banana is hung from the ceiling and monky may move the table from ay other room to reach at it. The problem is to write a program that predicts the monkey's movement.

The following rules apply:

- If the monkey, the banana, and the table are all in the same room, the monkey will reach out and eat the banana

- If the monkey and the banana is in the same room, but the table is in another room, the monkey goes to that room to get the table. Then the above rule applies.

- If the monkey, the banana, and the table are all in different room (the initial condition), the monkey first goes to the room where the table is. Then the above rule applies.

Using these rules, a program shown below is written in CLisp:

```
CLisp: (print-file 'monkey)
==================================

USER$DISK1:[AISYS2.SSTJMONKEY.LSP;6      modified:  7-JAN-1985 08:55:34.44

==================================


contents:

        monkey-and-banana
        new-world
        place
        assoc
        get-banana
-------------------------------------------------------------------------------
monkey-and-banana                        modified:   6-JAN-1985 16:22:24.49

        (lambda (L)
           (cond
              ((equal (place L 'banana) (place L 'monkey))
               (cond
                  ((equal (place L 'table) (place L 'monkey))
                   (cons (list 'monkey 'eats 'banana) nil))
                  (t (cons (list 'monkey
                                 'moves
                                 (place L 'monkey)
                                 'to
                                 (place L 'table)
                                 'and
                                 'brings
                                 'table
                                 'from
                                 (place L 'table)
                                 'to
                                 (place L 'monkey))
                           (monkey-and-banana (new-world L
                                                          'table
                                                          (place L 'monkey)))
                           ))))
              (t (cons (list 'monkey
                             'moves
                             'from
                             (place L 'monkey)
                             'to
                             (place L 'banana))
                       (monkey-and-banana (new-world L
                                                      'monkey
                                                      (place L 'banana)))))))
-------------------------------------------------------------------------------
```

```
new-world                                   modified:   6-JAN-1985 16:22:24.60

        (lambda (L X Y)
           (cond ((null L) nil)
                 ((equal (caar L) X) (cons (list (caar L) Y) (cdr L)))
                 (t (cons (car L) (new-world (cdr L) X Y)))))
```

```
place                                       modified:   6-JAN-1985 16:22:24.62

        (lambda (L X)
           (car (assoc X L)))
```

```
assoc                                       modified:   6-JAN-1985 16:22:24.64

        (lambda (X L)
           (cond ((equal X (caar L)) (cdar L)) (t (assoc X (cdr L)))))
```

```
get-banana                                  modified:   7-JAN-1985 08:55:34.87

        (lambda (L)
           (mapc (monkey-and-banana L)
                 '(lambda (Z)
                      (terpri)
                      (terpri)
                      (print Z)))
           (terpri)
           (terpri)
           (terpri))
```

("USER$DISK1:[AISYS2.SST]MONKEY.LSP;6")

Three sets of initial conditions shown in Figure 5.1 are chosen to test the program.

```
.-----------------+-----------------.
| room 1          | room 2          |
|                 |                 |
|     (monkey)    |     (banana)    |
|                 |                 |
+------      ----+-----      ------+
|                                   |
|              (table)              |
|                                   |
| room 3                            |
`-----------------------------------'
```

Case 1

```
.-----------------+-----------------.
| room 1          | room 2          |
|                 |                 |
|     (monkey)    |     (table)     |
|                 |                 |
+------      ----+-----      ------+
|                                   |
|              (banana)             |
|                                   |
| room 3                            |
`-----------------------------------'
```

Case 2

```
.-----------------+-----------------.
| room 1          | room 2          |
|                 |                 |
|     (table)     |     (banana)    |
|                 |                 |
+------      ----+-----      ------+
|                                   |
|              (monkey)             |
|                                   |
| room 3                            |
`-----------------------------------'
```

Case 3

Figure 5.1 Monkey and banana problem initial conditions

The results of the three runs are shown below:

```
CLisp: (get-banana '((monkey room-1)(banana room-2)(table room-3)))

(monkey moves from room-1 to room-2)

(monkey moves room-2 to room-3 and brings table from room-3 to room-2)

(monkey eats banana)


nil
CLisp: (get-banana '((monkey room-1)(table room-2)(banana room-3)))

(monkey moves from room-1 to room-3)

(monkey moves room-3 to room-2 and brings table from room-2 to room-3)

(monkey eats banana)


nil
CLisp: (get-banana '((table room-1)(banana room-2)(monkey room-3)))

(monkey moves from room-3 to room-2)

(monkey moves room-2 to room-1 and brings table from room-1 to room-2)

(monkey eats banana)


nil
CLisp: (exit)
```

5.2 OPS5

5.2.1 Features of OPS5

OPS5 is a Production Language developed at Carnegie-Mellon University (CMU) by John McDermott and Charles Forgy. The term production is used in Cognitive Science, and is synonym of 'rule' or 'rule-based'. It is specifically designed for building expert systems based on the theoretical study by Allan Newell and Hubert Simon of CMU.

OPS5 may be considered a Fortran of AI languages in its practicality and ease of use. One can write expert systems in Fortran, albeit with a great difficulty. The major difference between writing expert system in Fortran (Pascal, Bliss, or C, for that matter) and in OPS5 is that in Fortran, the programmer, acting as an expert or expert's interpreter must code the intelligence in the program as a series of instructions fixed and executed the way it was written. In OPS5, the intelligence lies mostly in the knowledge that is captured and stored separately, and completely detached from the control structure. This form of processing intelligence is much closer to (what we know of) the model of human intelligence.

OPS5 has a working memory (WM) filled with working memory elements. The WM is often related to the human short term memory. It has productions which are if-then rules of the form IF C1...Cn THEN A1...Am, where C1...Cn is a list of conditional elements and A1...Am a list of actions. C1...Cn is called the Left Hand Side (LHS) of the production and A1...Am the Right Hand Side (RHS).

The conditional elements of the production (synonym of rule) are compared against the WM elements. If they simultaneously match some constellation, then the actions can be performed. The set of productions whose conditional elements are satisfied is called the conflict set. It is called so because only one of them can be chosen for execution at a time. A conflict resolution strategy, which is modifiable, is used to select the production to run next from the conflict set. Running the production means performing actions specified by its RHS. Productions can be removed from and added to the conflict set due to actions modifying the WM. The entire set of coded productions is stored in a knowledge base, which is analogous to the long term memory in the model of human thought process.

The default conflict resolution strategy of OPS5 is

described below:

1. Avoid simple infinite loops by never running a production on the identical constellation of the WM elements.

2. Give preference for productions that match more recently defined WM elements.

3. Give preference for productions with longer LHSs that match WM elements of the same age.

4. Randomly pick a production from the set that survive condition 4.

An action on the RHS of a production can be to call a LISP function to:

- create new WM elements

- interrogate knowledge bases

- perform specialized input or output functions for user inferface

WM elements can have status "unasked", "T user", "nil user", or "T PN" or "nil PN" where "user" means the user gave this information and PN means the element was modified directly by a production and PN is its name.

Another function allows list representations of slightly modified Fortran format statements to be printed out at the terminal.

Special WM elements may have to be created to enforce sequentiality if a number of if-then statements must be executed in a particular order. The extra elements added to the appropriate RHS and LHS are called control elements because they encode the state of the production system rather than actually representing the description of the problem or solution.

Looping is also implemented using control elements. 3 productions can be used for this:

-one to initialize the loop by creating a control element

-another to perform the looping function

-a final one to terminate the loop and remove the control element

A double loop is implemented with a "same-name" loop to avoid having to introduce new names when updating fields. Both the "loop trick" and the "conditional sequencing trick" are used to do this.

A menu loop is implemented where the user is shown a menu and can choose which item he would like to see next. The user can continue to get information until he types the item that stops the loop.

The syntax of a production (rule) is of the form:

```
(p rule-name
   (function 1)

   -->

   (function 2)
   (function 3)
   )
```

This would read:

Rule-name is the name of the production. If function 1 is true then do function 2 and function 3.

Any explanation, user dependent or not, must be specially coded into an OPS5 based expert system. The standard OPS5 debugging aid is to run the system and watch which productions run and what changes in the WM. No good explanation facility exists for OPS5.

There is an escape hatch in OPS5 which allows a production to call LISP functions. This allows databases of knowledge to be addressed without filling up WM.

5.2.2 Using OPS5 on VAX/VMS

Follow th steps described in Section 3.2.2 and obtain the SST Tutorial software menu. Select OPS5 by entering

```
-> (OPS5)
[fasl sst$lib:vps2.o]
[fasl sst$lib:ops5e.o]
```

```
Leaving Workspace: background
In Workspace: ops5

Leaving Workspace: ops5
In Workspace: background
TYPE (ops-hanoi) to load Tower of Hanoi in OPS5
     (ops-robot1) to load Simple Robot in OPS5
     (ops-robot2) to load Robot Problem #2 in OPS5
nil
-> (load '<OPS5 application file specification>)
```

User will require suffucuent priviledge to maintain a file on the system disk where OPS5 resides. Interactive sessions do not require the **load**.

```
Leaving Workspace: background
In Workspace: <user defined workspace>

*rule-1 defined.
*rule-2 defined.
       .
       .
       .
*rule-xx defined.
<workspace name> Loaded
```

User defined productions are read in.

```
Type (run) to run <problem name>
```

A file-based OPS5 session begins.

```
       .
       .
       .
-> (exit)
```

The session ends and **exit** brings the user back to the VMS environment.

```
$
```

5.2.3 Example OPS5 program

The following is an example OPS program which works as a descrimination expert. Similar to the descrimination expert

discussed in Section 3.3.3, it asks a number of questions and determines what the user has in mind. The mini-expert system identifies provinces of Canada. The results of a couple of runs are shown following the program listings.

```
; ------------------------------------------------------------------------
; province.ops OPS5 implementation of PROVINCE
; Given hints the program will guess the chosen province.
; ------------------------------------------------------------------------
(workspace-push 'ops-province)
; ----------------------------------------------------------------------
; Create the question working memory and initiate processing.
; This program will be done first after start is added to the working
; memory and the comments on how to run PROVINCE is output. Start is
; deleted from working memory here as it will no longer be needed in
; this run. The questions are all added as elements to the working
; memory with status unasked.
; ----------------------------------------------------------------------
(definepr setup-p
  (start)
  -->
  (remove 1)
  (make oceans unasked "Is it next to the Atlantic or Pacific ocean" "?")
  (make loyalist unasked "Was it settled by loyalists" "?")
  (make great_lakes unasked "Does it contain any great lakes" "?")
  (make island unasked "Is it an island" "?")
  (make potatoes unasked "Are potatoes a major crop" "?")
  (make maritimes unasked "Is it part of the Maritimes" "?")
  (make atlantic unasked "Is it part of the Atlantic provinces" "?")
  (make bilingual unasked "Is it bilingual" "?")
  (make french unasked "Is its official language French" "?")
  (make english unasked "Is its official language English" "?")
  (make prairies unasked "Is it part of the Prairies" "?")
  (make oil unasked "Does it contain the tar sands" "?")
  (make rockies unasked "Does it contain part of the Rocky Mountains" "?")
  (make north unasked "Is it north of the tree line" "?")
  (make trees unasked "Are trees plentiful" "?")
  (make east unasked "Is it easterly" "?")
  ; ----------------------------------------------------------------------
  ;              Sequencing and Choosing of Appropriate Questions
  ;
  ; If the answer to the very first question which is oceans is true then
                                                  erwise prairies is asked.
  ; The next-questions are all added as elements to the working memory.
  ;
  ; (make next-question oldquestion oldanswer newquestion)
  ;
  ; If next-questions are not used then the questions are asked in the
  ; order of last input (most current). That would be east first.
```

```
  ; ------------------------------------------------------------------
  (make next-question oceans yes atlantic)
  (make next-question oceans no prairies)
  (make next-question prairies yes rockies)
  (make next-question prairies no great_lakes)
  (make next-question great_lakes yes english)
  (make next-question great_lakes no french)
  (make next-question atlantic yes island)
  (make next-question atlantic no rockies)
  (make next-question rockies yes oil)
  (make next-question rockies no trees)
  (make next-question trees yes french)
  (make next-question trees no north)
  (make next-question north yes east)
  (make next-question island yes maritimes)
  (make next-question island no maritimes)
  (make next-question maritimes yes bilingual)
  (make next-question maritimes no english)
  (make next-question bilingual yes loyalist)
  (make next-question bilingual no english)
  (make next-question english yes potatoes)
  (make next-question french no trees)
  (write (crlf) "Answer questions with yes,no, or stop" (crlf))
  (make goal restart)
  )
; ------------------------------------------------------------------
; In the production question-asker the goal is to find out which province
; is chosen. The variable prop signifies the question to be asked.
; Thus the question has the form:
;
; (question_name status sentence punctuation)
; ------------------------------------------------------------------
(definepr question-asker
   (goal province)
   (question (prop))
   ((prop) unasked (question) (mark))
   -
   (province)
   -->
   (write (question) (mark))
   (remove 2)
   (modify 3 ^2 (accept))
   )
; ------------------------------------------------------------------
; If the user inputs stop then the program terminates early.
; ------------------------------------------------------------------
(definepr bail-out
   (goal province)
   ((prop) stop)
   -->
   (write (crlf) (crlf) "Bye " (crlf))
   (modify 1 ^2 restart)
   (halt)
   )
```

```
; ---------------------------------------------------------------------------
; Here some error checking is done on user input.
; The braces {} are used to indicate that a value in a working memory
; element must match several things simultaneously. The predicate <> means
; that it will match anything that is not equal to the current binding of
; what is after it. All the known values are checked for. The bad answer
; is thrown away. The status of the question just asked is set back to
; being unasked. The question is added to working memory again.
; ---------------------------------------------------------------------------
(definepr bad-answer
   (goal province)
   ({<> goal <> province <> question <> next-question <prop>}
    {<> unasked <> yes <> no <> stop})
    -->
    (write (crlf) "Sorry, but the only legal answers are:"
           (crlf) "yes,no, or stop.")
    (modify 2 ^2 unasked)
    (make question <prop>)
    )
; ---------------------------------------------------------------------------
; This sequence rule implements random question ordering in the system.
; Without the next-question mechanism set up we would check if the
; question-name was unasked and put the question-name in the variable prop.
; It would be on the LHS and have the form:
; ({<> goal <> province <> question <prop>} unasked)
; Then on the RHS the question would be added to working memory:
; (make question <prop>)
; This production sequence-rule gets the value for the next-question which
; has the form:
; (next-question question answer nextquestion)
; It asks the question and if it succeeds with the correct answer then it
; adds the nextquestion to working memory as the current question.
; ---------------------------------------------------------------------------
(definepr sequence-rule
   (next-question <prop> <answer> <next-prop>)
   (<prop> <answer>)
   -->
   (make question <next-prop>)
   )
; ---------------------------------------------------------------------------
; These productions specify the 10 provinces and the 2 territories.
; When the if-part of the rule succeeds then the then-part will
; add the element of province to the the working memory.
; ---------------------------------------------------------------------------
(definepr its_britishcolumbia
   (goal province)
   (oceans yes)
   (atlantic no)
   (rockies yes)
   -->
   (make province britishcolumbia)
   )
```

```
;
(definepr its_alberta
   (goal province)
   (oceans no)
   (prairies yes)
   (oil yes)
   -->
   (make province alberta)
   )
;
(definepr its_saskatchewan
   (goal province)
   (oceans no)
   (prairies yes)
   (rockies no)
   (trees no)
   -->
   (make province saskatchewan)
   )
;
(definepr its_manitoba
   (goal province)
   (oceans no)
   (prairies yes)
   (rockies no)
   (trees yes)
   -->
   (make province manitoba)
   )
;
(definepr its_ontario
   (goal province)
   (great_lakes yes)
   (english yes)
   -->
   (make province ontario)
   )
;
(definepr its_quebec
   (goal province)
   (french yes)
   (oceans yes)
   (atlantic no)
   -->
   (make province quebec)
   )
;
(definepr its_newbrunswick
   (goal province)
   (bilingual yes)
   (loyalist yes)
   (maritimes yes)
   -->
   (make province newbrunswick)
   )
```

```
;
(definepr its_novascotia
   (goal province)
   (maritimes yes)
   (bilingual no)
   (island no)
   -->
   (make province novascotia)
   )
;
(definepr its_princeedwardisland
   (goal province)
   (maritimes yes)
   (island yes)
   (potatoes yes)
   -->
   (make province princeedwardisland)
   )
;
(definepr its_newfoundland
   (goal province)
   (maritimes no)
   (atlantic yes)
   (island yes)
   (potatoes no)
   (oceans yes)
   -->
   (make province newfoundland)
   )
;
(definepr its_yukon
   (goal province)
   (north yes)
   (east yes)
   (trees no)
   -->
   (make province yukon)
   )
;
(definepr its_northwestterritories
   (goal province)
   (north yes)
   (east no)
   (trees no)
   -->
   (make province northwestterritories)
   )
; ------------------------------------------------------------------------
; This prints out the answer.
; (remove 2) deletes the value for province from working memory in an
; attempt to start cleaning it up for a new run. Its value stays in (x)
; for the write statement regardless of the working memory.
; (halt) is needed to avoid a loop on the final decision.
; ------------------------------------------------------------------------
```

```
(definepr all-over
   (goal province)
   (province <x>)
   -->
   (modify 1 ^2 restart)
   (remove 2)
   (write (crlf) "----------------------" (crlf)
          (crlf)  The province is <x> "!" (crlf))
   (halt)
   )
; ---------------------------------------------------------------------
; The RHS of this production will be done when the goal is restart and
; the user's input matches an acceptable value. It changes the status of
; the current question to being unasked.
; ---------------------------------------------------------------------
(definepr clean-up-rule
   (goal restart)
   (<prop> (<yes no stop>))
   -->
   (modify 2 ^2 unasked)
   )
; ---------------------------------------------------------------------
; (make question oceans) adds the element that the question is oceans to
; working memory thereby causing oceans to become the very first question
; to be asked. All subsequent questions are chosen on the basis of the
; answer to oceans.
; ---------------------------------------------------------------------
(definepr restart
   (goal restart)
   -
   (<prop> (<yes no stop>))
   -->
   (modify 1 ^2 province)
   (make question oceans)
   )
; ---------------------------------------------------------------------
; start is added as an element to the working memory. It is done first
; because it is unconditional.
; ---------------------------------------------------------------------
(make start)
; ---------------------------------------------------------------------
; These also are unconditional outputs so they are done before the other
; programs start. If there was no halt then (run 34) would run the New
; Brunswick case in that many steps.
; ---------------------------------------------------------------------
(princ "OPS-PROVINCE Loaded")
(terpr)
(princ "Choose a province of Canada and I will try to guess which one")
(terpr)
(princ "Type (run) to run PROVINCE")
(terpr)
```

```
OPS-PROVINCE Loaded
Choose a province of Canada and I will try to guess which one
Type (run) to run PROVINCE
t
-> (run)
1. setup-p 1
Answer questions with yes,no, or stop

2. restart 40
3. question-asker 42 43 3 Is it next to the Atlantic or Pacific ocean ?yes
4. sequence-rule 19 46
5. question-asker 42 47 9 Is it part of the Atlantic provinces ?yes
6. sequence-rule 25 50
7. question-asker 42 51 6 Is it an island ?yes
8. sequence-rule 32 54
9. question-asker 42 55 8 Is it part of the Maritimes ?yes
10. sequence-rule 34 58
11. question-asker 42 59 10 Is it bilingual ?no
12. sequence-rule 37 62
13. question-asker 42 63 12 Is its official language English ?yes
14. sequence-rule 38 66
                Are potatoes a major    crop ?yes
16. its_princeedwardisland 42 58 54 70
17. all-over 42 71
---------------------

the province is princeedwardisland !

end -- explicit halt
 20 productions (179 // 297 nodes)
 17 firings (74 rhs actions)
 36 mean working memory size (39 maximum)
 1 mean conflict set size (1 maximum)
 76 mean token memory size (84 maximum)
nil
-> (exit)
Before EXITing: Please tell me where to save the
                following workspaces.  If you type
                NIL, I will throw it away.
Where should I save workspace: ops-province ?nil
```

```
$_lisp
 54c00 bytes read into 2c00 to 577ff
Franz Lisp, Opus 34

          SMART SYSTEMS TECHNOLOGY
      Artificial Intelligence Course

A.I. Course Software Selections
Type (asptp)  for ASPTP deductive retriever
     (ops5)   for OPS5 production system
     (sans)   for SANS associative network system
     (parser) for an ATN natural language parser
     (loadloop) for examples of control structures
-> (ops5)
[fasl sst$lib:vps2.o]
[fasl sst$lib:ops5e.o]

Leaving Workspace: background
In Workspace: ops5

Leaving Workspace: ops5
In Workspace: background
TYPE (ops-hanoi) to load Tower of Hanoi in OPS5
     (ops-robot1) to load Simple Robot in OPS5
     (ops-robot2) to load Robot Problem #2 in OPS5
nil
-> (load 'province.ops)
Leaving Workspace: background
In Workspace: ops-province
*setup-p defined.
*question-asker defined.
*bail-out defined.
*bad-answer defined.
*sequence-rule defined.
*its_britishcolumbia defined.
*its_alberta defined.
*its_saskatchewan defined.
*its_manitoba defined.
*its_ontario defined.
*its_quebec defined.
*its_newbrunswick defined.
*its_novascotia defined.
*its_princeedwardisland defined.
*its_newfoundland defined.
*its_yukon defined.
*its_northwestterritories defined.
*all-over defined.
*clean-up-rule defined.
*restart defined.
OPS-PROVINCE Loaded
Choose a province of Canada and I will try to guess which one
Type (run) to run PROVINCE
t
```

```
-> (run)
1. setup-p 1
Answer questions with yes,no, or stop

2. restart 40
3. question-asker 42 43 3 Is it next to the Atlantic or Pacific ocean ?no
4. sequence-rule 20 46
5. question-asker 42 47 13 Is it part of the Prairies ?no
6. sequence-rule 22 50
7. question-asker 42 51 5 Does it contain any great lakes ?no
8. sequence-rule 24 54
9. question-asker 42 55 11 Is its official language French ?no
10. sequence-rule 39 58
11. question-asker 42 59 17 Are trees plentiful ?no
12. sequence-rule 30 62
13. question-asker 42 63 16 Is it north of the tree line ?yes
14. sequence-rule 31 66
15. question-asker 42 67 18 Is it easterly ?yes
16. its_yukon 42 66 70 62
17. all-over 42 71
---------------------

the province is yukon !

end -- explicit halt
 20 productions (179 // 297 nodes)
 17 firings (74 rhs actions)
 36 mean working memory size (39 maximum)
 1 mean conflict set size (1 maximum)
 73 mean token memory size (81 maximum)
nil
-> (exit)
Before EXITing: Please tell me where to save the
                following workspaces.  If you type
                NIL, I will throw it away.
Where should I save workspace: ops-province ?nil
```

6. Natural Language Processing

6.1 The SST ATN tutor

6.1.1 Features of the SST ATN tutor

The SST ATN tutor is a software package that demonstrates the principle of the Augumented Transition Network grammar, first proposed by William Woods [Woods 70] then of Bolt Beranek and Newman Inc. This method of parsing natural language inputs is still a mainstay of the parsing methods used widely in today's Natural Language (NL) systems. In order to benefit from the tutorial software, the user must have a basic understanding of the ATN grammar and how it is used in a typical NL processing system. Chapter 9 of [Winston 84a] and also Chapter 9 of [Rich 83] constitute an adequate introduction to the theory of parsing, in particular, that of ATN. To those with the basic grasp, the package will act as an effective tool to enhance the understanding of the NL processing methodology.

The parser basically takes in input sentences and parses them. The user will learn if the input was successfully parsed or not. In case of a failure, the user is notified of how the process failed. A successful completion creates a parse tree in memory, which is displayed at the end of the parsing. Unlike actual NL systems which, for example, front-end a database, this tutorial system does not have a code generator for a specific application. This means that, while the parser parses input strings, it does not convert the semantics of the input sentence into an actual command sequence. This is because there is no specific application for which the parser was designed. Instead, it returns a parsed tree in a predicate form.

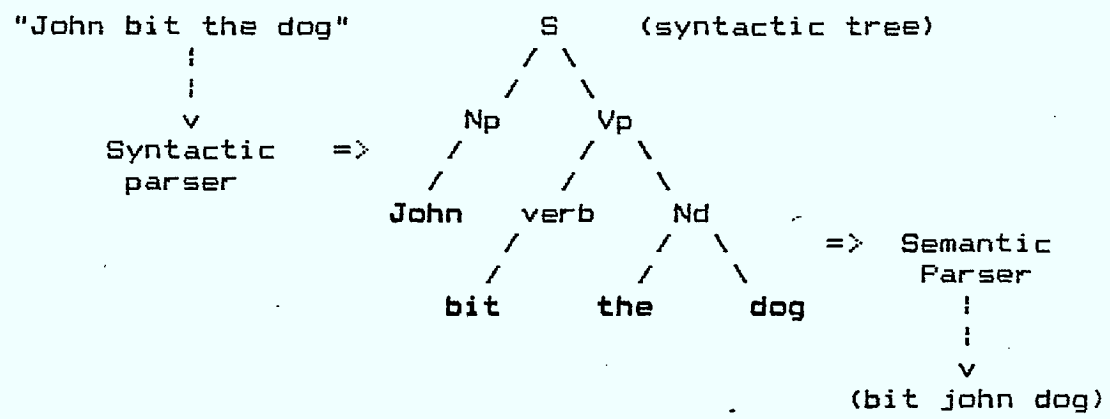


Figure 6.1 Two phases of ATN parsing

The parsing is performed in two steps: the syntactic and the semantic parsing. The first parser parses an input sentence and developes a syntactic tree. The second phase takes in the syntactic tree and creates a predicate calculus representation of the input sentence. For example, an input sentence "John bit the dog" will turn into (bit John dog) after the two phases of parsing. This process is shown in Figure 6.1.

In addition to parsing input sentences, and observing how parsing is done, the user can also modify the structure of the network (ie., ATN), the contents of the dictionary, the description of the grammar, and the definition of semantics, all part of the parser. Any such alteration affects the way the parsing is conducted. There are a small number of commands for carying out such manipulations. See SST ATN Manual (Appendix *6) for further details of how to conduct these operations. Some of these operations are quite involved.

6.1.2 Using the ATN tutor on VAX/VMS

In order to create an environment for the ATN tutor, execute the steps shown in Section 3.2.2, up to the point where the tutorial software menu is displayed. Select ATN by entering

```
-> (parser)
```

A succession of workspace management commands are executed to load necessary modules.

```
Leaving Workspace: background
In Workspace: parser

Leaving Workspace: parser
In Workspace: background
```

The syntactic parser is loaded.

```
Leaving Workspace: background
In Workspace: sematn

Leaving Workspace: sematn
In Workspace: background
```

The semantic parser follows.

```
Leaving Workspace: background
In Workspace: grammar

Leaving Workspace: grammar
In Workspace: background
```

The grammar and dictionary are loaded.

```
Type (lsemantics) to load semantics
nil
```

Semantic parser is not loaded unless specified. See Section 6.1.3.3 for running ATN with the semantic parser.

The ATN parser is now ready to process requests.

```
-> (atn '(the boy saw me))
```

Entering a request for parsing a simple sentence. See Section 6.1.3.1 for the result of this request.

```
-> (words '<class>)
```

List all words in the dictionary that belong to syntactic class <class>.

```
-> (all-words)
```

This command displays all the words the parser knows. Outputs from this and other display commands are shown in Section 6.1.3.2. Also shown there is how to modify the dictionary. The size of the present dictionary is very limited.

There is a set of commands in the SST ATN for defining the ATN itself, its dictionary, and semantic meanings to be attached to the nodes of the ATN. These commands are described below. The details of the commands are not covered in this document but described in the SST ATN Manual.

```
-> (defnet '<network node specification>)
```

Defines a node of ATN. An ATN

may be constructed by a set of **defnet** commands.

-> (**defword** '<word specification>)

Defines a new word in the dictionary.

-> (**defsem** '<description of semantics>)

Defines semantics to be attached to a sentence or a noun phrase.

6.1.3 Example sessions using the ATN tutor

The parser may be used for parsing a sentence, or for directly modifying its dictionary, grammer, or semantics and examining the effects of modifications in subsequent parsings. Both methods of using the parser are described below.

6.1.3.1 Parsing a simple sentence

-> (**atn** '(The boy saw me))

Trying to parse a simple sentece:

"The boy saw me".

All the words in this sentence are known to be in the dictionary.

Parsing (the boy saw me) as np

ATN tries to parse the sentence as a noun phrase, without success.

Parsing (the boy saw me) as vg

Then as a verb group, in vain.

Parsing (the boy saw me) as np-head

As a noun phase at the beginning of a sentence (np-head).

>> the (det in np-head)	Assuming that 'the' is a determinant of the np-head.
>> boy (noun in np-head)	Assuming 'boy' as a noun in the np-head.
Parsing (boy saw me) as ap*	Trying to see if (boy saw me) as an adjective phrase, in vain
Found np-head (the boy)	Now ATN is sure 'the boy' is a np-head.
Left to parse: (saw me)	This is what's left to be accounted for.
Parsing (saw me) as pp*	Parsing it as a prepositional phrase, in vain.
Found np (the boy)	Now it thinks 'the boy' is a noun phrase.
Left to parse: (saw me) Oops (s np) (saw me)	
Parsing (saw me) as vg	Checking if 'saw me' can be a verb group.
>> saw (verb in vg)	Then, saw must be a verb - yes, the dictionary says so.
Found vg (saw)	Only 'saw' accepted as belonging to a verb group.
Left to parse: (me)	One more word to go.
Disagreement compl nil t	
Disagreement trans t nil Oops (s) (me)	
Parsing (me) as np	'me' is a valid noun phrase.
Parsing (me) as np-head	Can 'me' be another np-head?
>> me (pronoun in np-head)	'me' surely can be a pronoun in an np-head.
Found np-head (me)	Lets assume 'me' is an np-head.
Left to parse: nil	Then there is none left to support the np-head assumption.
Found np (me)	'me' must be just a noun phrase.

Left to parse: nil Oops (s np) nil	
	Nothing more to parse.
Parsing nil as pp*	ATN is checking if nil can be interpreted as a prepositional phrase. No.
Found **s** (the boy saw me)	Now the entire sentence (**s**) is parsed.

At this point the parser displays the syntactic parse tree. A terse description of the format of the parse tree is given in the ATN Manual. Readers may require a good understanding of ATN parsers to fully understand the tree. The parser also outputs the summary of the parse following the display of the tree.

```
Result:
(s nil
   ((tns past) (stype decl) (numbers (1 3)))
   (subj nil ((numpers (1 3)))(np-head nil ((numpers (1 3)))the boy))
   (vg nil
       ((compl nil)
        (tns past)
        (vnumpers (or (1 1) (2 1) (1 2) (2 2) (1 3) (2 3)))
        (trans t))
       saw)
   (obj nil ((numpers (1 1))) (np-head nil ((numpers (1 1))) me)))
```

6.1.3.2 Listing the dictionary

Words in the dictionary are defined with its class (syntactic role of the word) and other attributes using the **defword** command (See the SST ATN Manual for the detailed description of the command). As mentioned in Section 6.1.2 above, there are commands to display and manipulate the contents of the dictionary. They are:

(all-words)	Lists all words in the dictionary
(words '<class>)	Lists words that belong to <class>. Classes are: noun, pronoun, verb, adj, det, prep, relpro(relative pronoun).

These display commands are tested below. Note the results of executing the commands do not show the contents of

the dictionary themselves but only its entries. The dictionary contents itself are more elaborate, as shown in the manual.

```
-> (words 'noun)
(block-n blocks-n boy boys fritter-n girl park sheep stand-n
stands-n telescope unknown-noun)
```

<noun>-n or <noun>-noun is a notation to mark the word to be a noun, while it can belong to different classes.

```
-> (words 'pronoun)
(he her him i me she them they us we you)
```

```
-> (words 'verb)
(be block-v blocked blocks-v fritter-v saw see sees sleep
sleeps slept stand-v stands-v stood unknown-verb)

nil
```

<verb>-v or <verb>-verb distinguishes them as verb, while the same word may belong to other classes. The nil in the output has no significance.

```
-> (words 'adj)
(angry big colorless green happy heavy red unknown-adj)

nil
-> (words 'det)
(a every some the)

nil
-> (words 'prep)
(by for in of on over to under with)

nil
-> (words 'relpro)
(that)
```

Finally, all the entries of the dictionary is listed by (all-words) command:

```
-> (all-words)
(unknown-noun unknown-verb a i block angry unknown-word heavy
happy fritten-n fritten-v green colorless every fritter be he
by sheep me in of on blocks we to us sleep slept under stand
```

stood blocks-n blocks-v girl boys john park sees that them block-n they block-v over some with big her boy him blocked for red see she saw the stand-n stand-v sleeps stand-n you stands-v telescope stands unknown-adj)

The defword command may be used to add to the vocabulary, as shown in the command sequence below:

```
-> (words 'pronoun)
                                List all pronouns that are in
                                the dictionary.

(he her him i me she them they us we you)

nil                             Notice a rather limited
                                vocabulary of pronouns.

-> (defword '(my (class pronoun)))

                                Defining a new pronoun.

my                              Done.

-> (words 'pronoun)             Confirming the addition to the
                                dictionary.

(he her him i me my she them they us we you)

                                Entered, alphabetically sorted.
```

Another way of defining a word in the dictionary is by running the parser with a sentence which includes a undefined word. See the following exaqmple:

```
(atn '(The boy saw a kangaroo))
Parsing ...
          .
          .
          .
I DON'T KNOW WORD: kangaroo
 Retype it
or type T and I will define it as noun
or type NIL and I will punt.  >T

                                A new noun is defined during a
                                parse. The word will remain in
                                the dictionary beyond this
                                parse.
```

.
.
.

Found s (the boy saw a kangaroo)

And the parsing completed OK.

6.1.3.3 Parsing with the semantic parser

The ATN parser may be run as a combined syntactic and semantic parser. This is accomplished by executing the (lsemantics) command and then issuing the (atn '<sentence>) command. It loads three files (sst$lib:sem.l, sst$lib:besem.l, and sst$lib:gosem.l) which define semantics for the ATN as defined in the present tutor and by the attached dictionary. The following is the summary of a run with both the syntactic and the semantic parser:

(lsemantics)

Loading semantic parser and definitions.

Leaving Workspace: Background
In Workspace: sem

Entering workspace for semantics.

Moving: unknown-nown from Workspace: grammar to Workspace: sem

Moving: unknown-verb from Workspace: grammar to Workspace: sem

Moving: the from Workspace: grammar to Workspace: sem

Moving: a from Workspace: grammar to Workspace: sem

.
.
.

Moving: telescope from Workspace: grammar to Workspace: sems

Moving: park from Workspace: grammar to Workspace: sems
t

Words in the dictionary are redefined with semantics. Note there are two workspaces that deals with semantics.

```
(atn '(The boy saw me))
                                        Parsing the same sentense.

Parsing (the boy saw me) as np
          .
          .
          .
Found np (the boy)
GOT: (person age young sex male ref def) Score = 0

                                        As the semantics for 'boy' was
                                        defined in the network, the
                                        semantic parser cuts into the
                                        parsing sequence and provides a
                                        semantic interpretation of
                                        'boy'. The semantics normally
                                        affects the further parsing.

Left to parse: (saw me) Oops (s np) (saw me)
          .
          .
          .                             The syntactic parsing continues

Found s (the boy saw me)
GOT: (do action mtrans instr (do action attend organ eye) to
(head (*same actor)) time past) Score = 1000

                                        A semantic interpretation of
                                        'saw' is given.
```

The parse tree created during the parse is again displayed at the conclusion of the process. This time it will have a distinct difference in appearance, representing the effect of semantic parsing. Compare the tree below with the one shown in Section 6.1.3.1. Lines that differ from the syntax only parsing are marked by an asterisk(*).

```
 Result:
  (s (do action
*        mtrans
*        instr
*        (do action attend organ eye)
*        to
*        (head (*same actor))
*        time
*        past)
    ( (tns past) (stype decl) (numpers (1 3)))
*     (subj (person age young sex male ref def)
*           (numpers (1 3)))
```

```
        (np-head nil ((numpers (1 3))) the boy))
 (vg nil
     ((compl nil)
      (tns past)
      (vnumpers (or (1 1) (2 1) (1 2) (2 2) (1 3) (2 3)))
      (trans t))
      saw)
(obj nil ((numpers (1 1)))(np-head nil ((numpers (1 1)))me)))
```

REFERENCES

[Berg 83]
Berg, Bradley, "SANS Associative Network Tutorial", Wang Institute of Graduate Studies, August 83, Revision 1.0.

[Cohen and Feigenbaum 82]
Cohen, Paul R., and Feigenbaum, Edward A., "The Handbook of Artificial Intelligence, Volumes 1, 2, & 3". William Kaufmann, Inc., Los Altos, California.

[Fox 82]
Fox, M.S., "SRL: Schema Representation Language". Technical Report, Robotics Institute, Carnegie-Mellon University, Pittsburg, PA.

[Brachman et al 83]
Brachman, Ronald J., Fikes, Richard E., and Levesque, Hector J., "Krypton: A Functional Approach to Knowledge Representation". In Computer, Vol. 16, No. 10, October, 1983, IEEE, pp. 67-73.

[Hayes 83]
Hayes, Kenneth C., "SANS: Simplified Associative Network System". Smart Systems Technology, February 1983.

[Kunz et al 84]
Kunz, John C., Kehler, Thomas P., Williams, Michael D., "Application Development Using a Hybrid AI Development System". In The AI Magazine, Fall 1984 Issue, American Association for Artificial Intelligence, pp. 41-54.

[Logicware 84]
Logicware Inc., "MPROLOG Manuals", Version 1.4, August, 1984.

[Nilsson 80]
Nilsson, Nils J., "Principles of Artificial Intelligence". Tioga Publishing Co., Palo Alto

[Mylopoulos et al 83]
Mylopoulos, John, Shibahara, Tetsutaro, and Tsotsos, John, "Building Knowledge-Based Systems: The PSN Experience". In Computer, Vol. 16, No. 10, October, 1983, IEEE, pp. 83-89.

[Rich 83]
Rich, Elaine, "Artificial Intelligence," McGraw-Hill Series in Artificial Intelligence, 1983.

[Schubert 76]
Schubert, L.K., "Extending the expressive power of semantic networks." Artificial Intelligence, 7, pp. 163-98.

[Shibahara et al 83]
Shibahara, Tetsutaro, et al., "CAA: A Knowledge-Based System with Causal Knowledge to Diagnose Rhythm Disorders in the Heart," Proc. Int'l Joint Conference on Artificial Intelligence, 1983 (IJCAI-83)

[Stefik 80]
Stefik, M., "Planning with Constraints," Report No. 784, Computer Science Department, Stanford University.

[Tsotsos 81]
Tsotsos, John, "Temporal Event Recognition: An Application to Left Ventricular Performance Evaluation," In Proc. Int'l Joint Conference on Artificial Intelligence, 1981 (IJCAI-81)

[Winston 84a]
Winston, Patrick Henry, "Artificial Intelligence, Second Edition," Addison-Wesley Series in Computer Science, Michael A. Harrison Consulting Editor. Addison-Wesley Publishing Company, Reading MASS, Palo Alto, London, Amsterdam, Don Mills, Sydney, 1984

[Winston 84b]
Winston, Patrick Henry, "LISP, Second Edition," Addison-Wesley Publishing Company, Reading MASS, Palo Alto, London, Amsterdam, Don Mills, Sydney, 1984

[Woods 70]
Woods, William, "Transition Network Grammars for Natural Language Analysis." CACM, Vol 13, No. 10, October 1970, pp. 591-606.

GOMI, T.
--An experimental expert system
development environment...

DATE DUE
DATE DE RETOUR
