VPI Prolog User Manual

By John W. Roach and John Deighan

TR 90-30

Technical Report SRC-90-004

VPI Prolog User Manual

Dr. John W. Roach and John Deighan

Department of Computer Science
at
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

February 1990

*Work supported by the U.S. Navy through the Systems Research Center under Basic Ordering Agreement N60921-83-G-A165 B044.

*Cross referenced as TR-90-??, Department of Computer Science, Virginia Tech.

ABSTRACT

This paper documents how a user may work with the VPI Prolog compiler. The user interface functions called debugging environment and input/output are all described in detail. Anyone using this manual should be able to program effectively using VPI Prolog.

CR Categories and Subject Descriptors: I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving - *Logic programming*

Additional Key Words and Phrases: Prolog, Inferential Database, Manual

Table of Contents

Acknowledgements....................i
Caveats....................ii
Preface....................iii
Conventions used in this manual....................iv
CHAPTER 1 - User's Guide for Getting Started....................1
Calling VPI Prolog and Loading Files....................1
Environments....................2
Command-Line Options....................3
The Startup Sequence....................3
On-line help....................4
The VPI Prolog Debugger....................4
Infinite Looping: A Common PROLOG Bug....................5
CHAPTER 2 - Programming in Prolog....................8
VPI Prolog Syntax and Semantics....................10
Rule Syntax....................10
Term Syntax....................11
Metapredicate....................13
Query Syntax....................14
Data Structures and Data Types....................17
Matching....................22
Control and Backing Up....................25
User Control: The Cut Operator....................28
The Relationship of Input and Output Arguments....................29
Numeric Calculation Routines....................31
CHAPTER 3 - Built-in Functions and Predicates....................33
Control....................33
Adding and deleting facts and rules....................34
Comparison....................35
Math....................36
Strings....................37
Lists....................39
Input/Output Predicates....................39
File Opening, File Closing, and Stream Connecting....................40
Input....................41
Output....................41
File Loading....................42
Example....................43
Miscellaneous....................43

CHAPTER 4 - Database Capabilities....................45
Asserting Database Facts....................46
The loaddb predicate....................46
An Example....................48
CHAPTER 5 - Types and Inheritance....................49
Declaring Types....................50
Example 1....................50
Example 2....................52
Example 3....................53
Appendix 1 - Lexical Conventions....................54
Appendix 2 - Prolog Syntax....................55
Appendix 3 - File Naming Conventions....................57
Appendix 4 - Historical Perspective....................58
Appendix 5 - Error Messages....................62
Appendix 6 - Glossary of Terms....................70
References....................71

// Acknowledgements

The authors wish to acknowledge the helpfulness of Dr. Daniel Chester, University of Delaware. The original Prolog system was built by Glenn Fowler based on a small Pascal implementation given us by Dr. Chester. The current VPI Prolog compiler was heavily influenced by this interpreter.

The design of the VPI Prolog compiler was even more heavily influenced by the abstract machine design by David H. D. Warren, as set forth in the paper An Abstract Prolog Instruction Set (SRI International Technical Note 309, Oct. 1983). We have made many enhancements to the abstract instruction set given therein, but kept the basic core instructions.

In addition, the authors would like to acknowledge the help of a large number of M.S. students of the Department of Computer Science at Virginia Tech who contributed to construction of this compiler: Chandan Chitale (inheritance), Guru Comarapalyama (indexing), and Steve Haugh (paging). Without their help, it would have been impossible to build this rather large system in such a short time.

Caveats

The following features are documented in the User Manual, but are not operational in the most recent compiler delivered to NSWC. Most will be operational in the version to be delivered Jan. 29, and all will be operational in the final version to be delivered in mid-February.

1. '->' is not accepted as a synonym for 'if' (see p. 7).
2. functor terms are not implemented (p. 13).
3. Inheritance and typing are not operational (see Chapter 6).

Preface

This manual documents a PROLOG compiler that has been developed at Virginia Tech. In this document, the word PROLOG (upper case) will be used to refer to the language itself, and our implementation will be referred to as VPI Prolog. The code is written in C and has proven to be highly portable to other systems.

PROLOG is a member of a class of logic programming and rule-based languages that are being explored by artificial intelligence researchers as a tool for working with symbolic data. Experience has shown that there are a number of problems with these languages: program execution is often slow, the user environment is primitive, input/output facilities are poor, and the interface to other languages has not been specified. Most of these problems are due to the relative youth and lack of development of these languages. Time has solved most of these difficulties.

More extensive experience with PROLOG shows that programs become quite large, obviating many of the advantages of the declarative style of programming. In addition, large programs tend to have clauses with large numbers of arguments causing difficulties in programming (mistyped variable names or misplaced arguments).

In recent years, it has become clear that PROLOG has more promise than as merely a language for Artificial Intelligence researchers. The model of computation inherent in PROLOG strongly suggests the possibility of extending the lanuage to handle large scale databases. Construction of database facilities in the language coincides with the relational data model. Addition of object oriented facilities is also being actively investigated.

This report documents the construction of a PROLOG compiler that incorporates a database facility and a type structure appropriate for objects.

Conventions used in this manual

The font called New York is used throughout this manual for explanatory text (such as this). The font called Helvetica is used for text which can be entered at the VPI Prolog prompt (sometimes the prompt will also be shown, but usually only the text to be entered is shown). The Helvetica font can be easily recognized because it has no serifs. An example of the Helvetica font is:

This is the font called Helvetica

In presenting the syntax for a command, the following conventions will be used. A name in angle brackets such as <int> represents a class of possible values. In this case, <int> represents any integer. If a name is enclosed in tall square brackets, it is optional and may be omitted. For example, in the command:

step [<int>]

the command step may be followed by an integer, or may appear by itself. If a name is enclosed in tall curly brackets, it may appear zero or more times. For example, in the description of the print predicate:

((print { <term> }))

the word "print" may be followed by zero or more terms.

If one of several alternatives is allowed, they may be presented separated by a tall vertical bar, like this:

<atom> | <number>

which means that either an atom or a number may appear.

CHAPTER 1 - User's Guide for Getting Started

This chapter describes using VPI Prolog, including how to start VPI Prolog, how to load files, and methods for debugging programs. Practical details and advice are included to facilitate program development. VPI Prolog is an interactive environment, so programming is easier than programming with a compiled language. Normally, a program is more easily debugged in such an interactive environment.

Calling VPI Prolog and Loading Files0

The command line to invoke VPI Prolog is:

```
$ prolog
```

where $ is the operating system prompt (which may differ between computers). VPI Prolog displays several lines of information, then presents the prompt ?-, indicating that it is ready for input. Commands, assertions, and queries may be entered at the prompt. Commands consist of a command word, possibly followed by arguments (for example the name of a file to be operated on), all on a single line. The syntax of assertions and queries is covered in Chapter 2. To quit Prolog, simply enter the command "quit".

As an example of a command, consider the "echo" command. The user can enter the command:

```
?- echo <filename>
```

and any text appearing on the terminal screen thereafter will be placed in a file called "<filename>.ech". Echoing can be disabled later with the command

```
?- echo off
```

VPI Prolog input may be placed in an external file using any text editor, for example, vi on Unix systems. By convention, VPI Prolog input file names end with ".hc" (for Horn Clause, the logic upon which the PROLOG language is based). There are two basic methods for retrieving input from external files:

1. Use the read command to read an external file if the external file contains commands and/or queries in addition to assertions. For example, the command:

   ?- read readme

   will read input from a file named "readme.hc". Note that the ".hc" ending is not required. Input will be read by VPI Prolog as if typed at the terminal, and any input which is valid when typed at the terminal is valid in the file. Read will not echo the input it reads to the terminal (use "readv", which stands for "read verbose", rather than "read" if this is desired).

2. Use the load predicate if the external file contains only assertions (to allow display of progress information, the file may also contain lines of the form: print <string>). For example, the command:

   ?- ((load readme))

   will read input from a file named "loadme.hc". Once again, the ".hc" ending is not required. Any file which can be read using the load predicate can be read using the read command. However, the load predicate can be invoked as a goal, the read command cannot. Load will not echo the input it reads to the terminal (use "loadv", which stands for "load verbose", rather than "load" if this is desired).

Very large lists of facts may be stored in a Prolog database. For efficiency, databases may be stores in a more compact manner than normal Prolog rules, and such files can be loaded using the "loaddb" predicate. For more information, see Chapter 4.

Environments

An environment consists of a set of clauses, including rules and facts stored in a database. Environments may be saved during one session, and then later loaded again during a later session. This is much faster than using read or load since parsing is completely bypassed. An environment is stored in a <name>.env file on your disk (it is not a human-readable file). When Prolog is running, the set of all currently defined rules is called the current environment, and has a name. Initially, the current environment is set to the

empty string. When you enter the "quit" command, if the current environment has a non-empty name, the environment is automatically saved.

The following commands control saving and loading of environments:

saveenv <name>	- save the current environment
loadenv <name>	- load a previously saved environment makes a copy of <name>.env & uses it
setenv <name>	- set current environment name
clearenv	- forget all rules & set current environment to the empty string
savecurenv	- save environment under current name
useenv <name>	- use environment with given name uses <name>.env directly

These are commands, and therefore may not be invoked as a goal. Loadenv and useenv completely replace any environment that you may have been using, i.e. any rules and/or facts which you had defined will be lost.

Command-Line Options

The following command line options (presented here in the style used on Unix systems) are recognized by VPI Prolog:

-r <name>	- read the file <name>.hc
-e <name>	- load a copy of the environment <name>.env
-u <name>	- use the environment <name>.env directly
-n	- don't automatically load the file BasePreds.hc
-h <size>	- allocate a heap array of <size> kilobytes
-s <size>	- allocate a stack array of <size> kilobytes
-c <size>	- allocate a code array of <size> kilobytes
-q <size>	- allocate a query array of <size> kilobytes

The Startup Sequence

When VPI Prolog is first invoked, The following sequence of events occurs:

1. If the -e or -u command-line option was invoked, set up the indicated environment (automatically appending the ".env" ending to the given file name if not present).

2. If neither the -e, -u, or -n command-line option was invoked, and if a file named BasePreds.hc can be found, load that file.

3. If the -r command-line option was invoked, read the given file (automatically appending the ".hc" ending to the given file name if not present).

4. If neither the -e, -u, or -r option was invoked, display some help information.

5. Repeatedly prompt for and process commands, assertions, and queries, until the "quit" command is entered.

6. If the current environment has a non-empty name, save it in a file called <current environment name>.env

On-line help

On-line help is available via the help predicate. The following invocations of help should help you get started with the on-line help system:

```
?- help              ; equivalent to "help help"
?- help topics
?- help general
?- help syntax
?- help builtins
```

The VPI Prolog Debugger

Enter the VPI Prolog debugger via the debug command:

```
?- debug
D-
```

The prompt will change to reflect the fact that you are now in debug mode. In debug mode, you can use the following commands to step through your program:

step [<int>] - match the next rule
solve [<int>] - completely solve the next goal
runto <predicate> - run until <predicate> is the next goal
clear - forget the current query so another one may be entered
rerun - restart the current query (however, asserted clauses are not automatically retracted)

<int> indicates a repeat count, which is optional. Omitting the repeat count is equivalent to entering a 1. Step will execute a single inference step (see Control and Backing Up in Chapter 2). Solve will take the next goal, and execute steps until that goal, any subgoals generated by that goal and their subgoals are removed from the goal list. In other words, if you take the current goal list, solve will either run until the goal list becomes that list with the topmost goal removed, or fail. Runto will execute steps until the given predicate is the name of the next goal to be executed. Rerun will reset the current query and start executing it all over again. In debug mode, a new query may be entered only if no query is currently active. To clear any currently active query, use the clear command. These commands are recognized only in debug mode.

Infinite Looping: A Common PROLOG Bug

Programs with infinite loops can be easily created; a bug known as "left recursion" often occurs in PROLOG programs. To help the programmer, a means of breaking computations has been provided. The C language allows signals to be defined to perform specific actions. In VPI Prolog, the signal called SIGINT interrupts a computation and displays the VPI Prolog prompt (the method for generating the signals SIGINT and SIGQUIT are system-dependent, see your system administrator). The user may then continue the computation using the 'step' command to see which predicates are being executed. Execution can be continued using the 'run' command. The SIGINT break allows the user to take stock of the situation and decide whether an infinite loop has occurred. On UNIX systems, the command (at the UNIX prompt):

```
$ stty intr ^B
```

sets the keyboard character Ctrl-B to generate the SIGINT signal. If SIGINT does not work, the user may resort to the SIGQUIT signal. This will result in the prompt:

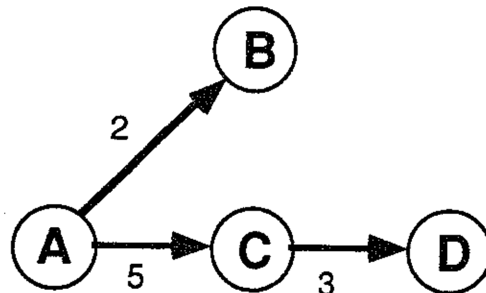
```
Core Dump(y/n)?
```

Whether the user answers y or n, VPI Prolog will terminate, and the system prompt will appear. On UNIX systems, the command (at the UNIX prompt):

```
$ stty intr ^C
```

sets the keyboard character Ctrl-C to generate the SIGQUIT signal. Basically, this feature has been included to keep programmers from sitting in front of terminals wondering whether the system is slow or whether the program is stuck in an infinite loop.

A simple program that goes into an infinite left recursive loop can be demonstrated with the following graph problem:



Finding the distance between points A and D in the graph would seem to be solvable using this program:

```
?-(assert
        ((distance a b 2))
        ((distance a c 5))
        ((distance c d 3))
        ((distance ?point1 ?point2 ?ans) if
             (distance ?point1 ?mid ?x)
             (distance ?mid ?point2 ?y)
             (:= ?ans [+ ?x ?y]))
        )
```

but ?-((distance a d ?ans)) causes an infinite loop. The fourth rule matches on the left, and the system starts to hunt for a midpoint between a and d. The first match that it finds for ?mid is b. We know that b does not lie between a and d, but VPI Prolog does not know that. VPI Prolog next tries to solve (distance b d ?y) but no facts match, only the fourth rule matches; the fourth rule says to find a midpoint between b and d by generating the subgoal

(distance b ?mid ?x). No facts match this subgoal; only the fourth rule matches, generating a new subgoal, namely, (distance b ?mid ?x) among others, the same subgoal that matched the left side of the rule. Thus, the subgoal (distance b ?mid ?x) generates (distance b ?mid ?x) which generates ..., and VPI Prolog is hung in a loop. If this process is unclear, try asserting the rules, enter "debug" mode, enter the query, and "step" through the program.

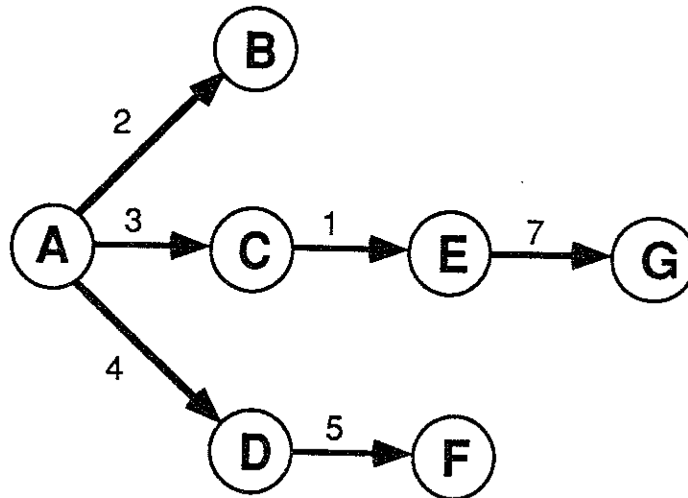
The infinite loop can be prevented in several ways: the first rule describing an end-point can be put third, for example, or the subgoals on the right hand side of the fourth rule can be reordered, the most general solution. Working with this program can help the user understand the problems inherent in left recursive looping.

CHAPTER 2 - Programming in Prolog

In this chapter, elementary aspects of programming in VPI Prolog are discussed and the syntax and semantics of the language are presented. The reader should get some idea of how programs are written as well as the capabilities of the language. This chapter cannot turn a novice into an expert PROLOG programmer; Clocksin and Mellish [3] and Kowalski [8] are recommended for more complete expositions. Learning is best achieved by doing, so the user is encouraged to log on and to try some simple programs as the chapter progresses (the system interface is discussed in the next chapter). The syntax of the version presented here is not the same as Edinburgh versions, but this should not be bothersome due to the simple structure of the rules.

Many programming languages already exist in the world today, so why bother learning a new language? FORTRAN, for example, can compute anything that PROLOG can compute. In answer to this question, PROLOG computes answers to problems more conveniently than conventional algebraic languages for certain application domains. These domains typically require some form of symbolic processing as opposed to the usual numeric calculation routines. PROLOG can process numeric data, fairly naturally in fact, but the main emphasis remains symbolic manipulation. Coelho, and Pereira [4] give a good selection of PROLOG programs including sorting programs, robot problem solvers, theorem provers, natural language processors, and algebraic manipulation routines.

As a simple example of a Prolog program, consider the following directed graph:



where the arcs are labelled with the distance between end points. The program to represent this graph including the rule to determine distances follows (?- is VPI Prolog's prompt):

```
?-(assert
        ((distance a b 2))
        ((distance a c 3))
        ((distance a d 4))
        ((distance c e 1))
        ((distance d f 5))
        ((distance e g 7))
        ((distance ?point1 ?point2 ?ans) if
            (distance ?mid ?point2 ?x)
            (distance ?point1 ?mid ?y)
            (:= ?ans [+ ?x ?y])
        )
)
```

By typing ((distance a g ?x)) at the VPI Prolog prompt ?-, the user can determine the distance between points a and g in the graph; the answer will be returned in the ?x variable:

```
?- ((distance a g ?x))
YES
        ((distance a g 11))
```

VPI Prolog Syntax and Semantics

Rule Syntax

Programs in Prolog are made up of collections of rules, each of which has a particularly simple format. Each rule has the form:

```
( <head> if
        { <subgoal > }
        )
```

There may be zero subgoals. The <head> and each <subgoal> are called relational expressions. The format of a relational expression is (there may be zero arguments) :

```
(<predicate name> { <term> } )
```

Consider this rule, for example,

```
((zebra ?x) if
        (ungulate ?x)
        (blackstriped ?x))
```

which says that ?x can be shown to be a zebra if ?x can be shown to be an ungulate and if ?x can be shown to have black stripes. Notice that the names of relations and the number of arguments are constructed by the programmer to fit the problem domain. The outer parentheses demarcate the boundaries of a rule; inside the outer parentheses are relations, each of which must also be enclosed by parentheses.

The zebra rule can be viewed in several different ways. The left side can be viewed as a goal and the right side as a list of subgoals to be reached or demonstrated. Alternatively, it can be viewed as a rewriting system in which the left hand side can be rewritten as the elements on the right hand side. This concept is supported by a syntax convention: the zebra rule could have been written

```
((zebra ?x) -> (ungulate ?x) (blackstriped ?x))
```

with the "->" rather than with the "if". Note that the arrow here refers to a rewriting operation, not to logical implication. In either case, a pattern on the left must be matched and replaced by new patterns from the right. The rule syntax with relations on both

sides of the "if" is the most general form, but one other form derives from it. With no right hand side a rule looks like

(<goal> if)

and means that the "goal" is defined to be a fact, since when the left hand side is matched no subgoals need be proven to demonstrate the truth of the left hand side. Note that the "if" part of such a rule is optional in this case. Thus, facts may be expressed as:

((blackstriped henry) if)

or as

((blackstriped henry))

If the programmer errs by writing

(blackstriped henry)

instead, Prolog cannot use it; outer parentheses are needed to show that this relation is a rule.

Term Syntax

A term may be an atom, or a structured term. A term represents some entity about which a logical statement may be made. In the above example, henry is a term representing an animal about which we may make the logical statements "henry is blackstriped" and "henry is a zebra", for example. An atom may be a string atom or a numeric atom. A string atom may be enclosed in single or double quotes, and must be enclosed in quotes if it contains any of the following characters: '.', '(', ')', '[', ']', ':', ';',a blank, a single quote, or a double quote. It must also be enclosed in quotes if it starts with a '?' or '*' so it won't be taken as a variable. In the preceding example, henry is a string atom. Other lexical conventions concerning string atoms such as including control characters in a string atom are covered in Appendix 1.

A numeric atom may contain a decimal point, and may include an 'E' or 'e' followed by an exponent. A numeric atom is never equal to a string atom. For example, the atom "123" is taken to be a string atom of length 3 because of the surrounding quotes, and is

not the same as the number 123. Functions are available for converting a numeric atom to a string atom and vice versa.

The special value nil is also considered an atom; however, it is not a string atom, and is not equal to the atom "nil" which, because of the surrounding quotes, is taken to be a string atom. The value nil can also be written as (), i.e. a left parenthesis followed by a right parentheses. This is because the value nil is used to represent an empty list in rules which manipulate lists.

A term may also be a structured term. There are 3 kinds of structured terms, functor terms, function terms, and lists. A functor term consists of a string atom, called the functor, followed by one or more terms, called the arguments, all enclosed in square brackets. For example, the following is a functor term:

```
[student john CS]
```

Any two functor terms with the same functor must always be followed by the same number of arguments (called the arity of the functor). VPI VPI Prolog determines the arity of each functor from the first time is sees that functor used in a functor term. Functor terms can be stored compactly since the size of the term is fixed.

Syntactically, a function term looks just like a functor term. The difference is that a function term is evaluated when it must be unified with another non-variable term. A function term can be recognized by its functor, since the list of functions is fixed. For example, there is a function called "strcat" which returns the concatenation of its arguments. The term

```
[strcat abc def]
```

is recognized as a function since "strcat" is the name of one of the built-in functions, and when it is evaluated, the result is the string atom "abcdef".

A list consists of an arbitrarily long sequence of terms enclosed in parentheses. A list is a general purpose method of grouping data elements together. For example,

```
(eggs butter milk (bluecheese cheddarcheese) cereal)
```

is a grocery shopping list containing a sublist of cheeses to buy. Arbitrary nesting of lists is permitted; thus any level of list

complexity can be achieved. Examples of other lists include vocabulary lists, class rosters, airline flights between cities, library catalogs, genealogy tables, parts inventories, legal chess moves from a given position (itself expressed as a list of piece positions), television shows for a given day, and so on. Lists are an important tool for Prolog programmers.

A variable is a term that looks like a string atom, but starts with a '?' or '*' and cannot be enclosed in quotes .A question mark by itself: ?, is called a void variable, and is not considered the same as any other variable, even another ? in the same clause. An asterisk by itself: * is the multiplication symbol, and not a variable. A variable represents an unknown quantity, whose value may become known during execution of a goal.

A term is called a ground term if it contains no variables. An atom is an example of a ground term. A functor term is a ground term if each of its arguments are ground terms, and a list is a ground term if every term in the list is a ground term.

Metapredicate

There are 2 additional forms which are allowed as a subgoal, i.e. after the "if" or "->". The first is a goal in which the predicate name is replaced by a variable, e.g.

```
?- (assert
        ((P ?x ?y) if
            (?x a b)
            )
        )
```

At run time, ?x must be bound to an atom, otherwise Prolog stops and issues an error message. This implies that when the rule is defined, the variable in the predicate position must occur earlier in the rule, else it could never be bound when used. The second form is one where the entire subgoal is replaced by a variable:

```
?- (assert
        ((P ?x ?y) if
            ?x
            )
        )
```

in this case, the variable must be bound to a list at run time, and the first element of that list must be an atom. Once again, this implies that the variable must occur earlier in the rule.

Query Syntax

The following should be typed:

```
?- ( { <goal> } )
```

when the user wants to invoke VPI Prolog to compute an answer. Essentially, the Prolog system is being asked to use previously defined rules to establish the conjunction of the subgoals. A typical program has the following form (a semicolon (';') begins a comment; Prolog ignores it and any characters remaining on that line) :

```
?-(assert    ; assert some rules
        (<head>)
        (<head> if )
        (<head> if <subgoal1> ... <subgoaln>)
        ...
  )

?- (<goal1> <goal2> ... ) ; goals to solve
```

The construct

```
( assert { <rule> } )
```

defines the rules to be used, and the construct

```
( { <goal> } )
```

is called a query and defines the questions (that is, goals) that Prolog answers using the rules. To simplify entering the query, the following short forms may be used: 1) if the query consists of a single goal, the parentheses demarcating the query may be omitted and 2) if the entire goal is entered on a single line, the parentheses demarcating the goal may also be omitted.

Here is an example of a complete program (adapted and simplified from Winston and Horn [16]):

```
?- (assert

        ((mammal ?x) if
            (hasHair ?x))
        ((mammal ?x) if
            (givesMilk ?x))

        ((bird ?x) if
            hasFeathers ?x))
        ((bird ?x) if
            (flies ?x)
            (oviparous ?x))

        ((carnivore ?x) if
            (eatsMeat ?x))
        ((carnivore ?x) if
            (hasPointedTeeth ?x)
            (hasClaws ?x)
            (hasForwardEyes ?x))

        ((ungulate ?x) if
            (mammal ?x)
            (hasHoofs ?x))

        ((type ?x cheetah) if
            (mammal ?x)
            (carnivore ?x)
            (colorTawny ?x)
            (hasDarkSpots ?x))

        ((type ?x tiger) if
            (mammal ?x)
            (carnivore ?x)
            (colorTawny ?x)
            (hasBlackStripes ?x))

        ((type ?x giraffe) if
            (ungulate ?x)
            (hasLongNeck ?x)
            (hasLongLegs ?x)
            (hasDarkSpots ?x))

        ((type ?x zebra) if
            (ungulate ?x)
            (hasBlackStripes ?x))
```

```
          ((type ?x ostrich) if
               (bird ?x)
               (notFly ?x)
               (hasLongNeck ?x)
               (hasLongLegs ?x)
               (colorBlackWhite ?x))

          ((type ?x penguin) if
               (bird ?x)
               (notFly ?x)
               (swims ?x)
               (colorBlackWhite ?x))

          ((type ?x albatross) if
               (bird ?x)
               (flysWell ?x))

          (hasHair henry))
          ((eatsMeat henry))
          ((colorTawny henry))
          ((hasDarkSpots henry))
)

;
;
;         now find out what henry is
;
?- ((type henry ?x))
```

Prolog responds:

```
YES
          ((type henry cheetah))
```

This small program encodes knowledge about the characteristics of animals and declares certain facts about a particular animal called "henry." When the question, "what kind of animal is henry?" is asked, the reply comes back ((type henry cheetah)). Prolog uses the asserted axioms to find the existence of a symbolic constant or variable that satisfies the question. Since it must show existence of an answer, it stops with the first instance for ?x, in this case cheetah, that answers the question. If the user asks ?-((type henry cheetah)), Prolog will find a sequence of rules to confirm the question. If the user asks ?-((type henry bird)), Prolog will not be able to find any sequence of rules that confirms this question and will return NO as the answer. The NO answer means that the

axioms given so far do not allow Prolog to conclude that the question is true.

Notice that an animal can be shown to be a mammal in two different ways. ?x is a mammal if ?x has hair or ?x is a mammal if ?x gives milk. The has-hair rule is tried first when a query (that is, subgoal) to establish mammal-ness arises. In general, when Prolog is hunting for a rule to solve a subgoal such as (mammal ?x) generated by the cheetah rule, it always starts searching sequentially from the top.

The only variable in this program is ?x which is local to each rule in which it appears; that is, each rule can be thought of as a subprogram with its own local variables. The left side of the rule defines the subprogram entry point, and the right side gives the relations needed to compute an answer. Each relation on the right hand side can be thought of as a call to another procedure. Unlike a normal algebraic language, however, a relation on the right side may fail without causing an error message. Consequently, if a programmer makes a mistake such as not declaring a variable by forgetting to put in the ?, or misspelling a predicate name, a goal will fail without giving an error indication. Global variables accessible by different rules do not exist in this language. Note that the rules are free format, i.e. a rule may span several lines, and several rules may occur on the same line. Most importantly, notice how easily knowledge about animals is expressed in this language. Additional knowledge about animals is easily added by asserting new rules. The new rules are added at the end of the existing rule list.

Data Structures and Data Types

Expressing programs as rules requires an ability to create and manipulate data. Symbolic constants, such as "giraffe" or "bob," numbers (numeric constants), and variables are available in Prolog, as well as means of using them, including user defined and built-in relations. Arithmetic operations on numbers, such as addition, multiplication, trig functions, along with list operations are included.

Symbolic constants, or string atoms, can be thought of as strings of characters. Strings can be indefinitely long. Note that a variable is just a string with a beginning question mark. Unusual strings that have special characters embedded in them (such as a period

or blank) must be delimited by single or double quote marks. Thus, "parse.hc" would be a legal, special string specifying a VAX filename which without the quotation marks would cause trouble. As another example, the goal (print "enter the next rule") would allow the string within double quotes to appear on the user's terminal. Control or other unusual characters can be represented using the \ followed by 2 hex digits: \0D is the ASCII CR, for example. Hex digits include '0'..'9' and 'A'..'F'.

List manipulation predicates are defined using the dot operator. The dot convention comes from the programming language LISP (=LISt Processing language). The dot operator allows the user to add list elements onto the front of a list or to break up a list into a first element (called the head of the list), and the rest of the list (called the tail of the list). The dot operator can be applied repeatedly, ultimately reducing a list to emptiness. The empty list is denoted by nil or (). It should be noted that nil is at the end of every non-empty list. Thus, the list

```
((mammal ?x) if (has ?x hair))
```

is stored internally as

```
((mammal ?x . nil) if (has ?x hair . nil) . nil)
```

Nil will not match (?x . ?y) or any other dotted expression. The dot operator is used so frequently that the Prolog system provides an infix notation for it. For example, here is a program for printing the elements of a simple (one level) list:

```
?- (assert
      ((printeach (?x . ?y)) if (println ?x) (printeach ?y) )
      )
```

where 'println' is a built-in predicate that prints its arguments, including symbolic constants, numbers, variable names, or lists, then appends a newline character. Invoking Prolog with

```
?-((printeach (roses are red and violets are blue)))
```

produces

```
roses
are
red
and
violets
are
blue
NO
```

As the program starts running, ?x matches roses and ?y matches the rest of the list, namely, (are red and violets are blue). The right hand side of the rule instructs Prolog to print what ?x has been matched to and then recursively call printeach with what ?y has been matched to. In effect, the front elements of the list are successively stripped off and printed, and the rule is then "called" with the rest of the list. The program stops printing after "blue" since nil and (?x . ?y) do not match (this fact is explained further in the matching section below). Since no rules match, Prolog prints NO to indicate a failure to compute an answer. Note that variables in this example can match both symbolic constants, such as "roses", and also complete lists, as when ?y is matched to "(are red and violets are blue)". Variables are allowed to match different kinds of objects, such as numbers, symbolic constants, other variables, or lists. Prolog is therefore a "type free" language since variables are not specially reserved for one particular kind of data item.

Operations on lists other than the dot operator are needed to program effectively. Definitions of standard utility functions that append two lists, reverse a list, get the last element of a list, etc. are normally included in programs. The appending of (a b c d) with (e f a) results in (a b c d e f a). The rules needed for this operation are:

```
((append nil ?x ?x))
((append (?x.?y) ?z (?x.?w)) if
        (append ?y ?z ?w))
```

Consider the first two arguments of each 'append' relation to be the input and the third argument to be the output. The first rule says that if the first list is empty then the result of appending it to the second list is the second list itself. The second rule says that to append a non-empty list to a second list, take the tail of the first list, append that to the second list, and then construct the final list

by using the head of the first list as head, and the list just constructed as the tail. Chapter 3 discusses practical details necessary for running this program such as calling Prolog, loading files, and common programming errors.

As an example of list manipulation, the following code defines a recursive program called "quicksort". The input to "quicksort", a list of data to be sorted, comes in through the first argument, and the answer is returned in the second argument. Quicksort works by using the first element of the list to partition the remaining elements into two lists, one of all smaller elements (called 'less' in the program below) and one of all greater than or equal elements (called 'more' below) each of which is then sorted using "quicksort". The first two arguments to 'partition' are input, the partition element and the list to be partitioned; the third and fourth arguments are output, the partitioned results, the less than list and the greater than or equal list. Hand tracing through this program using the example call at the bottom will illustrate many of the data structures principles discussed in this section.

```
?-(assert   ; -------- quicksort program --------
;
; ------------------- predicate quicksort ;
;
((quicksort nil nil))                    ; base case
((quicksort (?x.?y) ?ans) if             ; induction case
        (partition ?x ?y ?less ?more)
        (quicksort ?less ?less1)
        (quicksort ?more ?more1)
        (append ?less1 (?x.?more1) ?ans))
;
; ------------------- predicate partition
;
((partition ?x nil nil nil))          ; base case
((partition ?x (?y.?z) (?y.?less) ?more) if
        (< ?y ?x)
        (partition ?x ?z ?less ?more))
((partition ?x (?y.?z) ?less (?y.?more)) if
        (>= ?y ?x)
        (partition ?x ?z ?less ?more))
;
; ------------------- predicate append
;
((append nil ?x ?x))
((append (?x.?y) ?z (?x.?w)) if
        (append ?y ?z ?w))
)       ; end assert

; example call to QSORT
;
?-((quicksort (jim bill carl allen george) ?x))
YES
        ((quicksort (jim bill carl allen george)
                (allen bill carl george jim)))
```

Lists can also be viewed as sets, as long as duplicate elements in a list are not allowed. Set relations such as element of, equality, union, and intersection can then be defined. Mathematical theorems can be proven using set theory by giving VPI Prolog the axioms or rules of inference needed in the proof, and then asking for a proof of a theorem. Lists, therefore, have many different roles in Prolog programs.

Matching

Pattern matching triggers the rule-based computation or inferencing in Prolog. The implementation of matching depends on a method known as unification, or more specifically, "most general unifier." "Most general unifier" has a precise, mathematical definition, see Robinson [11] for full details. Rather than dwell on the details of the formal definition, the exposition here intends to give a feeling for how unification takes place. Some rules of thumb and some examples will suffice to show what's happening when a subgoal from the right hand side of some rule is unified with the left hand side of a rule.

When matching a goal with the head of a rule (two relational expressions), the predicates must match and the number of arguments must be the same. Two atoms in the same argument position must be equal. A variable will match anything in the corresponding argument position of the other relational expression. For example, given the two relational expressions

```
(nextto robot ?x)
(nextto robot couch)
```

they are unified by matching ?x with the string atom "couch"; given

```
(grandparent ?x ?y)
(grandparent (John Blyth Clay) (Robert Alan Rose))
```

these two relational expressions can be unified, with ?x matching (John Blyth Clay) and ?y matching (Robert Alan Rose).

Dotted expressions are used to break down or build up lists. For example, given

```
(quicksort (?x.?y) ?z)
(quicksort (bill pat john alan cathy) ?v)
```

these two relational expressions are unified when ?x matches bill, ?y matches (pat john alan cathy) and ?z matches ?v. The expression (bill pat john alan cathy) which matches (?x.?y) is the input to the rule containing (qsort (?x.?y) ?ans), so the dot operator is breaking down the original name list, (bill pat john alan cathy).

In the append predicate discussed in the section on data types and data structures, in the rule

```
((append (?x.?y) ?z (?x.?w)) if
        (append ?y ?z ?w))
```

the dot operator usually breaks down input lists that match (?x.?y) and builds up lists from components ?x and ?w with (?x.?w) for outputting an answer. As an example of building lists, suppose that ?x matches alan; ?y matches nil; ?z matches (bill cathy john pat); and ?w matches (bill cathy john pat) after the right hand side of the rule is successfully calculated. The dotted expression (?x.?w) is the output expression for the rule and builds the following answer, (alan bill cathy john pat). Thus, a call to append such as

```
((append (alan) (bill cathy john pat) ?x))
```

would result in ?x in the calling statement being unified with (alan bill cathy john pat). An example where a dotted expression will not match occurs when the corresponding argument in the relation to be matched is not a list and cannot be decomposed into first and rest parts. Thus, (?x.?y) cannot be unified with a constant such as "pat".

A matching convention that can be very helpful involves an extension of the dot operator to matching lists. For example, (verbphrase (hit the ball)) and (verbphrase (?x ?y ?z)) can be unified with ?x matching hit, ?y matching "the", and ?z matching "ball". In the example, the same number of elements must be present in each list. This technique fails when the length of either list is unknown. In such cases, the following can be used: (verbphrase (?x ?y . ?z)) which will unify with (verbphrase (rose with a roar and a mighty burst of flame)) where ?x matches "rose", ?y matches "with", and ?z matches the rest of the list. This form of matching is also frequently used for building output or answer lists. For example, (prepphrase (?x.?y) (prep ?u ?v ?w)) would be used to return an answer involving a prepositional phrase in which values for ?u, ?v, and ?w are computed elsewhere in the rule.

Up to now, the rules governing unification have been simple. The algorithm for unification can in fact be expressed as Prolog rules. But there is a special case not yet discussed that complicates the issue. In some expressions, variables must be substituted for

themselves in order to define a unification; this leads to an infinite regression. Consider, for example, the abstract expressions

```
(relation ?x (P ?x))
(relation (P ?x) ?x)
```

Attempting a unification here results in ?x matching (P ?x), so that ?x is recursively defined. (Note that had the variable in the second relational statement been named ?y instead of ?x, the problem would remain the same.) These two expressions cannot be legally unified; they should not match. Expressions of this kind seldom occur, and the unification algorithm employed here does not check for such situations (This check is called an "occur check" since it checks to see if the variable occurs in the term it is being unified with). The unification algorithm will succeed (illegally) and create a circularly linked structure. The compiler will enter an infinite loop if an attempt is made to print the unification value of a variable bound to the circularly linked structure. The unification algorithm without the occur check runs in time proportional to n, where n is the number of arguments in the relation being unified. Checking for the infinite regression problem, however, causes the algorithm to run in time proportional to n squared, resulting in a considerable performance degradation. Unification is the most often performed operation in this language, so reducing the time spent in unification is quite important. Unification is the heart of resolution, and this language is a resolution based theorem prover.

An additional matching convenience is provided by a wild card match. The symbol ? is used to allow matching on arbitrary objects in the language. The ? wild card will match an arbitrary variable, constant or list. In addition, if ? occurs more than once, each occurrence can match a different expression. Thus, (loves ? ?) unifies with (loves jack jill). The ? can be thought of as an anonymous variable whose matched value is not retained. Using ? together with the . operator allows Prolog to match any number of arguments. For example, the following two expressions match:

```
(cities ?x ?y.?)
(cities boston newark austin seattle denver)
```

with ? matching (newark austin seattle denver).

Control and Backing Up

Control structures such as GOTO, DO, DOWHILE, CASE, and so on are absent from PROLOG. PROLOG provides an automatic control structure that the user must understand to use the language effectively. This control structure functions quite differently from normal programming languages; no statements invoke the control because it is present at all times, just as the default control in normal algebraic languages, statement following statement, is present at all times. The control component of PROLOG automatically provides recursion and backup from failures (as in "non-deterministic" programming, see Floyd [5]). Much of the control that normally must be specified by the user is thus built into the PROLOG logic program processor. This feature of the language has prompted Kowalski [8] to characterize programs written in PROLOG as composed of separate logic and control segments. Execution time of programs can be enhanced either by improving the logic of the program or the control algorithm of the interpreter. Programming this way allows the user to concentrate more on the logic of programs and less on the control. In theory, since many programming errors are control errors, PROLOG programs can be more easily developed; the absence of control statements allows programmers to concentrate on the meaning of the problem to be solved. Such programs also tend to be much more concise.

The ideas behind the control procedures will be explained, illustrated at the end by a simple logic program. For each rule in a program, for example "(<g1> if <sub1> <sub2> <sub3>)", if the left hand side is matched by a goal <g> that is currently being solved, then the subgoals on the right hand side of the rule are solved in the same order as they are written in the rule. Thus, <sub1> would be attempted before <sub2> and <sub2> before <sub3>. Prolog keeps a goal stack and when the front goal on that stack, for example <g>, matches <g1>, the right hand side of the rule, <sub1> <sub2> <sub3> (with the appropriate substitutions made that were necessary to unify <g> and <g1>), replaces <g> on the goal stack. If solving the subgoals <sub1> <sub2> <sub3> does not work, then the goal stack is restored to the state before <g> matched <g1>, that is, with <g> on the top of the stack, and the search for additional rules that match <g> continues. For example, if another rule following the rule above were given by "(<g1> if <sub4> <sub5>)", and the user

typed ((<g1>)), the system would first attempt to solve subgoals <sub1> <sub2> <sub3>. If these three subgoals could not be solved, only then would Prolog continue on to the next rule and attempt to solve <sub4> and <sub5>.

The control structure for one aspect of the PROLOG system has been explained. But another aspect, namely how a conjunction of subgoals such as <sub1> <sub2> <sub3> are solved, remains to be explored. Suppose that sub1 has been solved successfully. Often this requires that certain variables have values assigned through unification that then affect variables in <sub2> and <sub3>. If <sub2> is now attempted and fails, the system backs up to <sub1>, undoes the variable bindings already established, and tries another rule to satisfy <sub1>. If a new solution to <sub1> is found, the system again attempts to solve <sub2> starting at the top of all rules that can match <sub2>. Thus, if <sub2> keeps failing, all possible ways of solving <sub1> will be explored since solutions of <sub2> are usually affected by solutions of <sub1>.

Consider the following simple logic program as an illustration of the control and backup ideas:

```
?-(assert
        ((beautiful aphrodite))
        ((beautiful helen))
        ((beautiful hera))
        ((real helen))
        ((lovable ?x) if
            (beautiful ?x))
        ((loves ?x ?y) if
            (lovable ?y)
            (real ?y))
)

?-((loves paris ?x))
```

The answer returned by this program is

```
YES
        ((loves paris helen)).
```

To understand how the system computed this answer, consider the following table which represents the memory structure of the processor at the critical point in the computation:

	Goals Remaining	Unified Rule	Bindings	Alternates
1.1	((loves paris ?x))	(loves paris ?x) if (lovable ?x) (real ?x))		none
1.2	((lovable ?x) (real ?x))	((lovable ?x) if (beautiful ?x))		none
1.3	((beautiful ?x) (real ?x))	((beautiful aphrodite))	?x/aphrodite	beautiful/2
1.4	((real aphrodite))	FAIL		

Each line represents a step that VPI Prolog takes in attempting to solve the query ((loves paris ?x)). In the "goals remaining" column is the list of remaining goals, initially the query. VPI Prolog always attempts, at each step, to match the first remaining goal. When a goal is first encountered, Prolog attempts to match it with the first rule defined with the same predicate name. The column labeled "unified rule" gives the result of unifying the goal with that rule, unless unification is unsuccessful, in which case the word FAIL appears. If unification succeeds, any bindings created are listed in the column "bindings", and the next alternate rule which could have been tried is listed in the "alternates" column.

When unification fails, Prolog searches for alternates it could have tried, undoes any variable bindings created since then, and tries that alternative. If the number of the line corresponding to the point at which the most recent alternate rule is found is n.m (e.g. 3.4), the new line, following the FAIL line, is numbered n+1.m (e.g. 4.4). This indicates that step n.m, and all steps n.i for i>m, led to failure (these are called failed steps). They could be removed from the table if one is only interested in how a successful answer was derived, and not in any failed computation (in the table they will be marked with an '*'). Any bindings in the "bindings" column of these rows is undone by Prolog. The computation continues as follows (Note how the variable ?x reappears in line 2.3 since ?x was bound to "aphrodite" in a failed step):

Goals Remaining	Unified Rule	Bindings	Alternates

1.1	((loves paris ?x))	(loves paris ?x) if (lovable ?x) (real ?x))		none
1.2	((lovable ?x) (real ?x))	((lovable ?x) if (beautiful ?x))		none
*1.3	((beautiful ?x) (real ?x))	((beautiful aphrodite))	?x/aphrodite	beautiful/2
*1.4	((real aphrodite))	FAIL		
2.3	((beautiful/2 ?x) (real ?x))	((beautiful helen))	?x/helen	beautiful/3
2.4	((real helen))	((real helen))		
2.5	(())	SUCCEED		

After a failure, VPI Prolog will attempt to match the first goal in the "goals remaining" column with the second or later rule for a given predicate. In that case, /<n> is appended to the predicate name, where Prolog is attempting to match the goal with the <n>th rule defined for that predicate (see, for example, step 2.3). If the list of remaining goals is empty, the word SUCCEED appears in the "unified rule" column, and Prolog returns the answer. The answer is constructed by taking the original query, and applying any variable bindings created which are not associated with failed steps. It is possible to ask the system to look for more answers as described in Chapter 3. (In this case, however, there are no more answers). The backtracking described here is completely automatic; essentially, an exhaustive depth first tree search algorithm has been built into PROLOG.

User Control: The Cut Operator

Including automatic backup in a language can cause problems. Large tree searches that have no possibility of finding the correct answer are automatically computed, for example, and waste large amounts of time and space even when the programmer knows better. Allowing the programmer some degree of control is therefore desirable. The cut operator helps the Prolog programmer control the automatic tree search. The cut operator prevents the system from backing up and trying different alternatives; that is, control can pass through the cut, but cannot cross back through it. The cut is like a fish weir: it is easy to pass through the cut, it always succeeds, but it is impossible to go back past it. The cut is used in a group of rules when the programmer is certain that if a particular rule is invoked no alternative matches should even be

tried. That rule should then contain a cut. Consider the following program that computes the union of two sets:

```
(assert
        ((union nil ?x ?x))
        ((union (?x.?r) ?y ?z) if
            (member ?x ?y)
            (cut)
            (union ?r ?y ?z))
        ((union (?x.?r) ?y (?x.?z)) if
            (union ?r ?y ?z))
)
```

The cut operator in the second rule prevents backing up from any failure of the (union ?r ?y ?z) relation of the second rule to try a different solution to the member relation, or to try the third rule of the union program.

The cut operator is not defined in logic; it has a purely procedural definition. The degree to which a programmer uses it is a matter of style. If used indiscriminately, it can cause programs to execute incorrectly, pruning branches of the search tree that contain answers. Used correctly, it can help speed up the execution of a program even to the point of saving an otherwise useless program.

The Relationship of Input and Output Arguments

Most programming languages allow subprogram definitions that include standard input arguments and standard output arguments as specified by the programmer. The subprogram expects certain inputs and algorithmically produces certain outputs. Prolog, however, defines relationships between arguments and thus does not insist on a variable being an input or output argument. Prolog rules are therefore non-directional with regard to inputs and outputs. Inputs can be passed into a set of rules in any pattern and Prolog will compute the relation, if possible, of the inputs to other un-unified variables which become the effective outputs. Since relations cannot always be defined as one to one functions, not all patterns of inputs will produce unique outputs. Consequently, Prolog must "choose" between possible answers to be output through un-unified variables. Since Prolog incorporates automatic backtracking, the choice function can be seen as an implementation of the non-deterministic programming ideas explained in Floyd [5].

For example, consider the append rules defined above in the <u>Data Structures and Data Types</u> section (repeated here for reference),

```
((append nil ?x ?x))
((append (?x.?y) ?z (?x.?w)) if
     (append ?y ?z ?w))
```

If the third argument is given as input with the first two arguments variables, as in

```
?-((append ?x ?y (this is a test)))
```

then the following answers are all possible:

```
((append nil (this is a test) (this is a test)))
((append (this) (is a test)(this is a test)))
((append (this is) (a test) (this is a test)))
((append (this is a) (test) (this is a test)))
((append (this is a test) nil (this is a test)))
```

Any one of these answers could be selected as the answer to the ((append ?x ?y (this is a test))) query. Due to backtracking, the choice of which output to return cycles through all possible outputs if necessary. The following program and query illustrate this idea:

```
?- (assert
       ((test ?list) if
            (append ?x ?y ?list)
            (println ?x ?y)
            (fail))
?- ((test (a b c)))
()(a b c)
(a)(b c)
(a b)(c)
(a b c)()
NO
```

Essentially, rules can be seen as defining constraints between the variables. If the value of one of the variables becomes known, the values of other variables related by a set of rules becomes considerably constrained. In situations where a program must search for an answer among different possibilities, the constraint properties of relations can often be used to help cut down the search space. Considerable research into constraint satisfaction as a problem solving tool has shown the efficacy of this approach in describing knowledge about a domain (Stefik [13], Waltz [14]). The

ability to formulate problem solving strategies using the constraint satisfaction property of PROLOG relations remains to be tested.

Numeric Calculation Routines

PROLOG is intended to be used for symbolic manipulations. Functions for calculating numeric data, however, are available and easily used. This section describes the use of numbers and built-in arithmetic functions.

All numbers in Prolog are converted into floating point format to avoid conversion problems between types such as integer, floating point, double precision, and so on. The programmer may write the number three as 3, 3., 3.0, 3.0E0, or 3E0, but the system will simply type 3 when a print is requested. If the programmer types 3.1415926 as a value of pi, it will be rounded to 3.14 automatically when printed. Due to the input conversion algorithm and the word length of the target machine, the actual accuracy extends to the sixth digit after the decimal point. Note that numbers such as five tenths must not be written as .5 because the period is taken as a dot operator by the interpreter; 0.5 must be typed.

Standard arithmetic operations are available. PROLOG does not have an assignment operator but rather uses the unification process. To force evaluation of an arithmetic function, the := sign should be used. For example,

```
((answer ?x ?y) if
        (:= ?y [+ 3.1416 [tan ?x]]))
```

will compute a value for ?y and replace all occurrences of ?y in that rule with the calculated value given an ?x for which (tan ?x) exists. Note that function applications should be surrounded by '[' and ']' rather than parentheses. If parentheses are used, the correct results will be obtained, but this is included in our implementation to maintain compatibility with an earlier implementation, and carries with it a performance penalty.

Since arithmetic operations are implemented as compiled procedures rather than as rule based computations, it is not possible to run the functions "backwards," unlike other computations defined by rules. For example, (:= 10 [+ ?x ?y]) will

not result in generating two values for ?x and ?y that will then add up to ten. System defined functions are documented in chapter 4.

CHAPTER 3 - Built-in Functions and Predicates

This section lists the system defined functions and predicates available to the user. The programmer may not redefine a system defined function or predicate.

Built-in predicates may be part of the Prolog compiler itself, or may be loaded from a utility file that contains the definitions of many useful predicates. The name of the utility file is "BasePreds.hc", and it is automatically loaded when Prolog is invoked. There is a system-wide BasePreds.hc file; however, if a file named BasePreds.hc is found in the default directory, it will be loaded instead. Predicates contained only in BasePreds.hc will be noted in the documentation below.

All Prolog function calls have the same general form. The function and its arguments are enclosed in brackets, with the function name first:

[<function name> { <term> }]

If the programmer wants to retain an answer that has been returned from a function the := predicate, which forces evaluation as described below, should be used.

Control

Control functions manipulate execution flow.

(and <goal1> <goal2>)

Succeeds if <goal1> is true and <goal2> is true, else fails. Useful inside an 'or' goal. Loaded from BasePreds.hc.

(or <goal1> <goal2>)

Succeeds if <goal1> is true OR if <goal2> is true, else fails. Loaded from BasePreds.hc.

(! <goal1>)

Succeeds if <goal1> fails, else fails. Loaded from BasePreds.hc.

(cut)

Controls the proof backup mechanism.The cut always succeeds when first encountered. However, when proof backup attempts to back up over (cut), the current rule fails, and the remainder of the rule list associated with the current rule is ignored.

(quit)

Terminates Prolog and returns to the local operating system.

Adding and deleting facts and rules

Rules can be added or deleted using these predicates.

(assert { <rule> })

Asserts new rules to the rule database. Each rule is appended at the tail of the rule list associated with the rule name. If the (assert ....) statement occurs on the right-hand-side of a rule, then only one rule may be asserted at a time.

(assert0 <rule>)
(asserta <rule>)

Asserts new rules to the rule database. Each rule is inserted at the head of the rule list associated with the rule name.

(retract <rule>)

Deletes from the rule database the first rule that can be unified with <rule>. Retract will retain bindings from the unification of <rule> and the matching rule.

(retractall <rule>)

Deletes from the rule database all rules that can be unified with <rule>. Retractall will NOT retain bindings from the unification of <rule> and the matching rules.

(retract0 <predicate-name>)
(retracta <predicate-name>)

This function retracts from the rule base the first rule or fact for the given predicate

(retractz <predicate-name>)

This function retracts from the rule base the last rule or fact for the given predicate

Comparison

(== <arg1> <arg2>)
(= <arg1> <arg2>)

Unifies <arg1> and <arg2>. Fails if the unification fails.

(!= <arg1> <arg2>)

Succeeds if <arg1> cannot be unified with <arg2>. Fails if <arg1> can be unified with <arg2>. Note therefore, that != can never create any variable bindings.

(:= <Prolog-variable> <arg>)

Similar to ==, except that if <arg> is a function call, it will be evaluated (== will simply bind the variable to the call without evaluating it). If <arg> is a function call, <arg> has the form [function-name { <term> }]. If an argument is a list, it is evaluated as a function. If some of the arguments needed to evaluate the function have not been assigned values, then the variable is bound to the original list.

(> <arg1> <arg2>)

Succeeds of <arg1> is strictly greater than <arg2>. Note that either both args must be numbers, or both args must be strings; otherwise, an error occurs. Numbers are compared numerically, and strings are compared lexicographically.

(>= <arg1> <arg2>)

Succeeds if <arg1> is greater than or equal to <arg2>. Note that either both args must be numbers, or both args must be strings; otherwise, an error occurs.

(< <arg1> <arg2>)

Succeeds if <arg1> is strictly less than <arg2>. Note that either both args must be numbers, or both args must be strings; otherwise, an error occurs.

(<= <arg1> <arg2>)

Succeeds if <arg1> is less than or equal to <arg2>. Note that either both args must be numbers, or both args must be strings; otherwise, an error occurs.

Math

Prolog provides a variety of math and trig functions. Most other math functions can be built from the basic set. All operations are performed in floating point. Integer operations can be simulated by using the ceil and floor functions. All angles are measured in radians.

[+ <arg1> <arg2>]

Returns the algebraic sum of <arg1> and <arg2>

[- <arg1> <arg2>]

Returns the value of <arg1> minus <arg2>.

[neg <arg>]

Returns the negated value of <arg>.

[* <arg1> <arg2>]

Returns <arg1> times <arg2>.

[/ <arg1> <arg2>]

Returns <arg1> divided by <arg2>.

[% <arg1> <arg2>]
[mod <arg1> <arg2>]

Returns the remainder of dividing <arg1> by <arg2>.

[floor <arg>]

Returns the greatest integer <= <arg>.

[ceil <arg>]

Returns the smallest integer >= <arg>.

[abs <arg>]

Returns the absolute value of <arg>.

[sqrt <arg>]

Returns the square root of <arg>.

[exp <arg>]

Returns 'e' to the power of <arg>.

[log <arg>]

Returns the natural logarithm of <arg>.

[sin <arg>]

Returns the sin of <arg>.

[cos <arg>]

Returns the cosine of <arg>.

[tan <arg>]

Returns the tangent of <arg>.

[pow <arg1> <arg2>]

Returns <arg1> to the power <arg2>.

Strings

There is no difference between atom names and strings in Prolog. Atom names (and strings) are permanent Prolog objects. No mechanism is available for destroying atom names. Control or other unusual characters can be represented using the \ convention: \0D is the ASCII CR, for example. The empty string is specified as "".

[strcase <string> upper]
[strcase <string> lower]

Converts the case of alpha chars according to the option:
    upper: all alpha chars are translated to upper case
    lower: all alpha chars are translated to lower case

[strcat { <string> }]

Concatenates the string arguments and returns the concatenated string as its value.

[stratm <str>]
[strexp <str>]

Expands <str> into its constituent characters and returns a single list containing these characters.

[strlen <str>]

Returns the number of characters in <str>.

[strpos <str> <old> <num>]

Locates the substring <old> in the string <str>. If <str> does not contain <old> then -1 is returned. <num> specifies how many characters to examine.

[numtostr <number> <precision>]

Converts a Prolog number into a string. <precision> controls the number of decimal digits.

[strsub <str> <from> <num>]

Returns the substring of characters in <str> from character position <from> counting <num> number of characters. The first character is at position 0.

[strmid <str> <from> <to>]

Returns the substring of characters in <str> from character position <from> up to, but NOT including character position <to>. The first character is at position 0.

[strdelete <str> <num> <from>]

Returns the string obtained by deleting characters in <str>. The number of characters to delete is <num>, the first character to delete is at position <from>. The first character is at position 0.

[strinsert <str> <insstr> <pos>]

Returns the string obtained by inserting <insstr> into <str>. <insstr> is inserted at position <pos>. The first character is at position 0.

(match <str> <regexpr> <start> <end>)

Perhaps one of the most useful predicates for string searching. <regexpr> is a regular expression (or pattern) like that used by the Unix editors ed and vi. The first 2 arguments must be atoms, but the last 2 arguments may be used in 3 ways: 1) if both are variables, match will fail if no instance of <regexpr> can be found in <str>, else it will return with <start> at the starting position if the first instance of <regexpr> in <str>. If several sequences of characters starting at that position match <regexpr>, the longest possible sequence will be matched. <end>, on return, will actually be bound to one position <u>past</u> the end of the matching character sequence. This facilitates both determining the length of the matching sequence, and passing to another string function or predicate the starting position of the rest of the string. 2) If <start> is a number, match will fail if no sequence of characters in <str> beginning at <start> matches <regexpr>, else <end> will be set to one greater than the end position of the longest possible match. 3) If both <start> and <end> are numbers, match will simply succeed or fail, depending on whether the characters in <str> from position <start> to <end>-1 match <regexpr>.

Lists

(member <element> <list>)

Checks if <element> is an element of <list>. Succeeds if <element> is in <list>, else fails. Loaded from BasePreds.hc

Input/Output Predicates

Prolog input/output is implemented using character streams. Files are therefore treated uniformly as a sequence of bytes even if they have different types. Basic character stream operations are the same across all devices, although some limitations may be imposed by the devices.

Many I/O predicates accept a <stream> argument, where <stream> is a user-supplied stream name. There are 3 predefined streams: stdin is the standard input stream, stdout is the standard output stream, and stderr is the standard error stream. These streams are usually attached to an interactive terminal, but the local system invocation of Prolog may cause redirection to other devices.

To achieve input or output with a file, the file must be associated with a stream name using the open function. A stream name is a character string (an atom) assigned by the programmer. All subsequent input/output operations on the file must refer to the stream name, not the file name. File names follow the local operating system conventions. Here are the available I/O predicates:

File Opening, File Closing, and Stream Connecting

(open <stream> <filename> <mode>)

This command opens the file <filename> and associates it with the <stream> descriptor. <mode> must be read, write, or append

(close <stream>)

This command closes the file associated with the <stream> descriptor. The open command is the converse of this operation.

(eos <stream>)

This predicate succeeds if <stream> is at end-of-stream. Output streams are always at end-of-stream, a terminal is never at end-of-stream, and input file streams are at end-of-stream if the associated file is at end-of-file.

(connect <stream1> <stream2>)

Connects output stream <streaml> to output stream <stream2> such that any output to <streaml> is also sent to <stream2>, but not vice versa. The connection remains until (disconnect <streaml>). Recursion is obviously disallowed.

(disconnect <stream1> <stream2>)

Breaks the connection between <stream> and <stream2>.

(divert <stream1> <stream2>)

Forgets previous connections of <streaml> and connects it to <stream2>. The diversion remains until (revert <stream>), which restores all previous connections.

(revert <stream>)

Reverts the I/O stream <stream> to the state before the most recent divert.

Input

(getchar <stream> ?x)

Returns the next character on the input stream <stream>.

(ungetchar <stream> <char>)

Places a single character <char> back onto the input stream <stream>. The next read on <stream> will return <char> as the first character. Only one character pushback is implemented. If <stream> is omitted then <stdin> is assumed. Note that this only affects the logical operation of Prolog reads and does not modify physical entities associated with <stream>.

(getline <stream> ?x)

Returns the next line on the input stream <stream>.

Output

(print { <term> })

Writes the <term>s to stdout. Quoted strings are printed without enclosing quotes.

(println { <term> })

Writes the <term>s to stdout and then writes a newline. Quoted strings are printed without enclosing quotes.

(printterm { <term> })

Writes the <term>s to stdout. Any strings containing special characters, including blanks, will be quoted.

(printn <stream> { <term> })

Writes the <term>s to stdout <number> times. Quoted strings are printed without enclosing quotes. Useful, for example, for printing a given number of blanks.

(write <stream> { <term> })

Writes the <term>s to output stream <stream>. Quoted strings are written without enclosing quotes.

(writeln <stream> { <term> })

Writes the <term>s to output stream <stream>, and then writes a newline. Quoted strings are written without enclosing quotes.

(writeterm <stream> { <term> })

Writes the <term>s to output stream <stream>. Any strings containing special characters, including blanks, will be quoted.

(writen <stream> <number> { <term> })

Writes the <term>s to output stream <stream> <number> times. Quoted strings are written without enclosing quotes. Useful, for example, for writing a given number of blanks.

File Loading

(load <file>)
(loadv <file>)

Reads the named file <file> as if it were typed at the keyboard. The "loadv" form ('v' for verbose) sends input characters to stdout, thus displaying the input on the screen. If the file being loaded contains load commands, then PROLOG will recursively load the files. This feature permits the programmer who is developing a very large and complex application to create one central file that loads all other files. This command works only for files containing PROLOG assertions, load, loadv, or loaddb commands, and lines of the following forms:

```
print <string>
println <string>
```

(loaddb <relationname> [<filename>])

Allows the programmer to load a database file. See full documentation in Chapter 5.

Example

The definition of a Prolog predicate which will display the contents of a text file on the terminal screen follows:

```
(assert

((cat ?filename) if
        (open catfile ?filename read)
        (showlines fd)
        (close catfile))
((showlines ?fd) if
        (eos ?fd)           ; if at end-of-stream, then done
        (cut))              ; don't try 2nd rule in this case
((showlines ?fd) if
        (getline ?fd ?line)       ; get a line
        (println ?line)     ; display the line
        (showlines ?fd))  ; show rest of the lines
) ; end assert
```

Miscellaneous

The following predicates and functions display user help, test the value of terms, allow the generation of unique atoms, facilitate timing and debugging tests, and allow commands to be passed on the the local operating system.

(help [<argument>])

This is one of the most useful predicates for beginning users. It provides on-line help from within the Prolog system. The (help) command will give the user a menu from which to get specific help using (help <argument>). <argument> can take the values "general" (to get the allowed commands), "syntax" (for Prolog syntax), and "builtins" (for builtin functions and predicates).

(numberp <arg>)

Succeeds if <arg> is a numeric constant.

(atomp <arg>)

Succeeds if <arg> is a string atom or a number.

(boundp <arg>)

Succeeds if <arg> is a bound variable.

(variablep <arg>)

Succeeds if <arg> is a variable.

[gensym]

Returns a unique atom which is guaranteed different from any atom in the current session.

(cpumark)

This marks a time from which to compute elapsed time. This function always succeeds.

[cpu]

Returns the time since the last 'cpumark' in 1/60 sec increments.

(time <month> <day> <year> <hour> <minute> <second>)

Determines the current date. All the arguments must be variables.

(sys <cmd1>)
(system <cmd1>)

Sends the commands <cmd1> to the local system to be executed by the system command interpreter.

CHAPTER 4 - Database Capabilities

PROLOG is a relational programming language. Facts declared in PROLOG look like and function the same way that facts in large, relational databases do. Unfortunately, many of the powerful facilities of a relational database are not normally built into a PROLOG compiler. This chapter documents how the VPI Prolog system has been extended to include the facilities one normally finds in a relational database: most importantly, B+ tree indexing. Relational databases are not normally built within a powerful programming language like PROLOG and must therefore include a separate query processor. Using the query answering capabilities of VPI Prolog, however, we get a much more powerful query processor. The PROLOG computational model, moreover, does not require that facts be placed in third normal form in order to process queries efficiently. The resulting system therefore combines relational database and PROLOG relational language technology. This chapter describes how to use the database facilities built into VPI Prolog.

The specific relational database technology that has been incorporated VPI Prolog includes a buffered paging system and a B+ tree indexing system. The user need not understand the internal details of paging and buffering, the details are transparent to the programmer. In order to use the indexing system effectively, however, the programmer needs to know a few simple details about the internal implementation of the system.

B+ tree indexing works by building an indexing structure in memory in addition to the actual data being indexed. Each argument of a relation may have its own separate indexing structure. The amount of storage being used can become considerable, possibly slowing the user's application due to excessive paging. We have therefore provided a facility for the user to limit indexing to a selected set of the arguments of the relation. Normally, a relation does not need to be indexed on all arguments, so large amounts of storage can be saved in this fashion. The ability to limit indexing structures is documented below in the "loaddb" command.

B+ tree indexing works by indexing one argument of a relation at a time. When a database goal is attempted, the database system selects an instantiated argument to use for indexing. If all arguments are variables, indexing cannot be used and all database facts for the given predicate are searched. There are techniques for indexing more than one argument simultaneously, but this would require a different indexing technique.

Although the processing of large databases is performed by routines separate from the normal PROLOG processor, asserting, retrieving, and retracting database facts, saving the compiled image, printing, etc. are entirely transparent and require no special programmer attention. Due to the large size of databases, however, we have designed a compact database file input format and created a special predicate, loaddb (documented below), to load these files.

Asserting Database Facts

As with normal Prolog rules, database facts can be created by asserting the fact directly. However, some way is needed to distinguish database facts from normal Prolog facts; the ":database" following the <predicate> in the following example does this:

```
(assert ((<predicate>:database { <arg1> } )))
```

In a normal assertion statement, the very first occurrence of a predicate to be considered as a database predicate must be tagged by ":database" followed, as usual, by the predicate's arguments. Later occurrences of the predicate need not be tagged. If by accident or design the programmer does include additional tags, the VPI Prolog system will ignore them. If the programmer fails to tag the first occurrence of a predicate but later does include the same predicate with a ":database" tag, this is a programming error, and an error message will be displayed.

The loaddb predicate

The following predicate will load an entire database from a file formatted as described below:

```
(loaddb <predicate> <filename>)
```

The default file name ending for a database file is ".db", and VPI Prolog will automatically append this ending to the given filename if not present. The following form is also allowed:

```
(loaddb <predicate>)
```

in which case the <filename> to load from is assumed to be the same as the predicate name (with ".db" appended). The database input file must have one of two formats:

```
<arg1> <arg2> ... <argn>
... etc.
```

where each line in the file is of this form. Prolog deduces the number of arguments from the number of terms on the first line. Each line becomes a separate fact in the database just as if the following were entered at the Prolog prompt:

```
?- (assert
        ((<predicate>:database <arg1> <arg2> ... <argn>))
        ((<predicate> ...))
        ... etc.
        )
```

and for each argument, a B+ tree indexing structure is created.

The second input format includes a header line, which is used to declare which arguments should be indexed, and allows some type checking of the input:

```
$database <indexstring> <typestring>
<arg1> <arg2> ... <argn>
... etc.
```

where the first line of the file must start with the string $database and the index and type strings each have one character per argument as follows:

indexing

1 | y | Y	yes, index on arguments in this position.
0 | n | N	no, do not index on arguments in this position.

typing

? | *	perform no checking on the argument in this position.
#	the argument in this position must be a number.
+	the argument in this position must be a list.
@ | $	the argument in this position must be an atom.

Type checking on the arguments is performed only during loading of the database. Prolog deduces the number of arguments from the length of the index string. It is not necessary that all the arguments for each fact be on a single line, except for the first line when using the first format. That is, when preparing an input file for a database relation with many arguments, the arguments may span two or more lines. However, the last argument for each fact must be the last argument on its line.

<u>An Example</u>

Consider a database consisting of three arguments, where the first is a state name, the second is the state's population, and the third is a list of all cities in the state with population greater than 1,000,000. The following file (named state.db) is created:

```
$database      ynn        $#+
"New York"     10000000   ("New York City" Buffalo)
California     50000000   ("Los Angeles" "San Francisco")
Alaska         5000000    ()
```

Notice that an atom that contains blanks (or other special characters) must be enclosed in quotes. This database may be loaded with the query:

```
?- ((loaddb state))
```

CHAPTER 5 - Types and Inheritance

The VPI Prolog compiler implements an extension to Prolog which adds types with inheritance to the PROLOG language. Inheritance has become an issue of some interest as a result of excitement over object oriented programming languages. Object oriented languages have certain capabilities not normally found in standard programming languages: encapsulation of procedures (complete information hiding), message passing between procedures, and inheritance of properties in a hierarchy of types. PROLOG already contains mechanisms akin to message passing and encapsulation. The addition of a type hierarchy adds capabilities that help make VPI Prolog more like an object oriented programming language. The extensions we have made are "natural" modifications of the unification algorithm and so fit in well with standard Prolog programming practice. Normal VPI Prolog programs without inheritance will therefore run on this system without modification.

PROLOG is a general purpose programming language equivalent to a Turing machine. The addition of hierarchies and inheritance, therefore, do not actually add some special capability not otherwise programmable within Prolog. Types and hierarchies can be programmed normally, of course, but the code required to implement them is more extensive and must be specially written into every procedure in which the programmer desires to have inheritance. By providing automatic typing, the system allows the programmer to think more naturally in terms of inheritance and objects. The improvement here, then, is one of conceptual economy: the programmer can think more clearly about the problem to be solved.

Poor performance is a typical problem associated with object oriented languages. Our typing system, however, is designed to impact performance as little as possible. In fact, types can in many cases allow Prolog to execute more quickly because the unification algorithm does not have to check as many cases; there are fewer cases to check at runtime since typing has removed some possibilities.

Declaring Types

The programmer must declare the type hierarchy and the elements of types before any rules in the program are defined .The syntax for declaring types in VPI Prolog is as follows:

```
(declare
        <declaration>
        ...
        )
```

where each <declaration has one of the following forms:

```
(name1 < name2),
```

where name1 and name2 are type names. This declares that name1 is a subtype of name2 .

```
(term << name3),
```

where name3is a type name. This declares that the set of objects that unify with term (if term is an atom, a single object), are elements of type name3.

Atoms, variables, and functor terms are extended by allowing a ":<typename> suffix. For example:

```
?x:atom
tree:oak
[student ?x:name cs]:university
```

where atom, oak, name, and university are type names, represent a variable which represents any atom, a tree of type oak, and a student, whose first argument is of type name, and who is of type university.

Example 1.

The program segment consists of the following declarations and rules.

```
(declare
        (animals < livingthings)              ; declarations
        (plants < livingthings)
        (carnivores < livingthings)
        (domestic < carnivores)
        (wild < carnivores)
        (domestic < animals)
        (wild < animals)
        (venusflytrap << carvivores)
        (venusflytrap << plants)
        (pitcherplant << carvivores)
        (pitcherplant << plants)
        (dog << domestic)
        (cat << domestic)
        )

(assert                ; rules
        ((Eatsmeat ?x: carnivore))            ;(i)
        ((Chasesmailman ?x:domestic) if       ;(ii)
            (Barks?x))
        ((Barks Dog))                         ;(iii)
        )
```

The query:

```
:-((Eatsmeat ?x:plant))
```

returns

```
YES
        ((Eatsmeat ?x:annonymous)),
```

where anonymous is an intersection node that holds the two objects venustrap and pitcherplant

```
:- ((Eatsmeat ?x: animal))
```

returns

```
YES
        ((Eatsmeat ?x: domestic|wild))
```

where domestic|wild is an intersection type that is created.

```
:-((Chasesmailman ?x:domestic))
```

returns

```
YES
        ((Chasesmailman Dog))
```

Example 2.

This program segment consists of the following declarations and rules.

```
(declare
        (birds < has_lungs)
        (birds < lays_eggs)
        (reptiles < has_lungs)
        (reptiles < lays_eggs)
        (owls < birds)
        (parrots < birds)
        (snakes < reptiles)
        (polly << parrots)
        (kaa << snakes)
        )

(assert
        ((animal ?x: lays_eggs) if (warmblooded ?x));  ;(i)
        ((animal ?x: lays_eggs) if (coldblooded ?x))          ;(ii)
        ((warmblooded ?y: birds))                       ;(iii)
        ((coldblooded ?y: reptiles))                    ;(iv)
        ((talks polly))                                       ;(v)
        )
```

Example queries:

```
:-((animal ?x: has_lungs))
YES
        ((animal ?x: birds))
```

The original goal unifies with (i) to give (animal ?x: birds|reptiles), then unifies with (iii) to give (warmblooded ?x:birds).

```
:-((warmblooded ?x:lays_eggs))
YES
        ((warmblooded ?x:birds))

:-((talks ?x: birds))
YES
        ((talks polly))
```

Example 3.

Consider the following program segment.

```
(declare
        (undergraduates < university)
        (graduates < university)
        (csdept < university)
        (mathdept < university)
        (cs << dept)
        (math << dept)
        ([student ?x ?y] << university)
        ([student john cs] << undergraduate)
        ([student mark math] << undergraduate)
        ([student ?x cs] << csdept)
        ([student jane cs] << graduate)
        ([student tom cs] << graduate)
        )
(assert
        ((takes AI ?x: csdept))
        ((enrolled ?x: university)
        ((uses_gym [student ?x ?y: dept]: university))
        )
```

Queries:

```
:-((takes AI [student jane cs]))
YES
        ((takes AI [student jane cs]: csdept))

:-((enrolled [student john cs]))
YES
        ((enrolled [student john cs]: university))

:-((uses_gym [student ?x math]))
YES
        ((uses-gym [student ?x math]: university))
```

Appendix 1 - Lexical Conventions

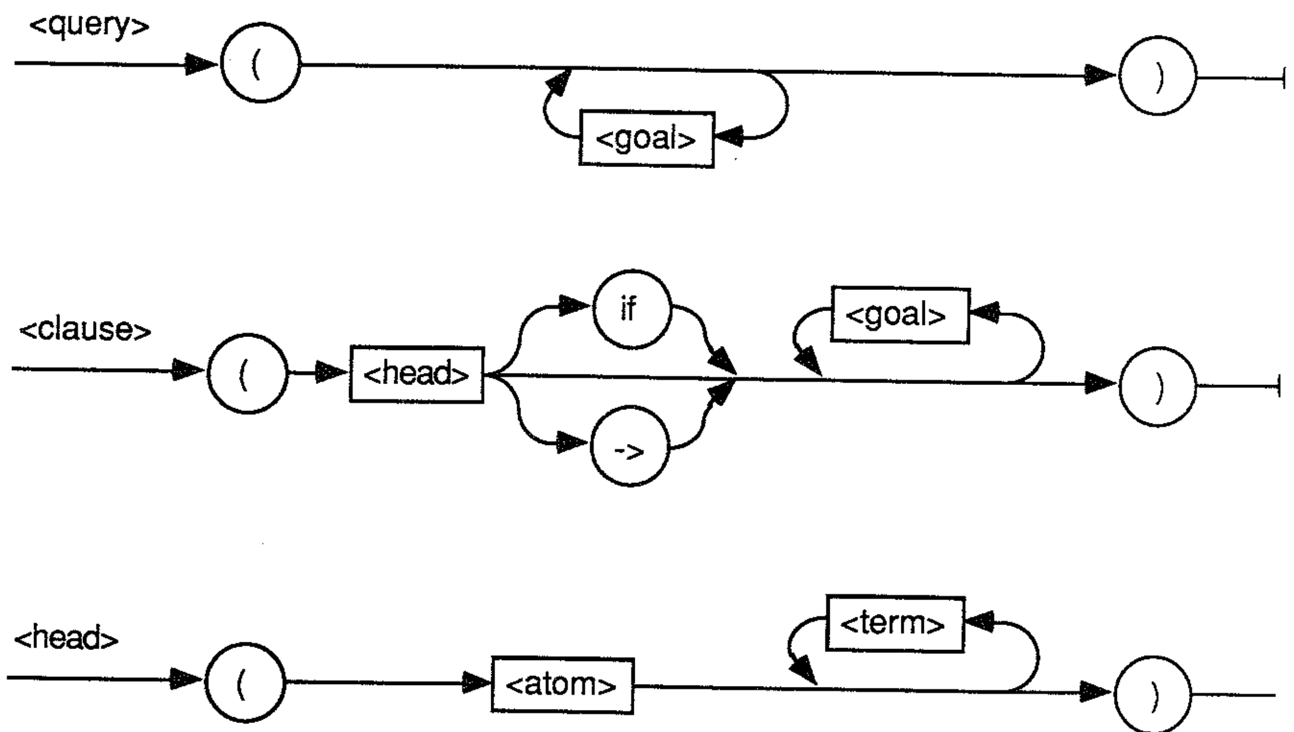
In a string atom (whether or not enclosed in quotes), the following backslash escape sequences represent the characters indicated:

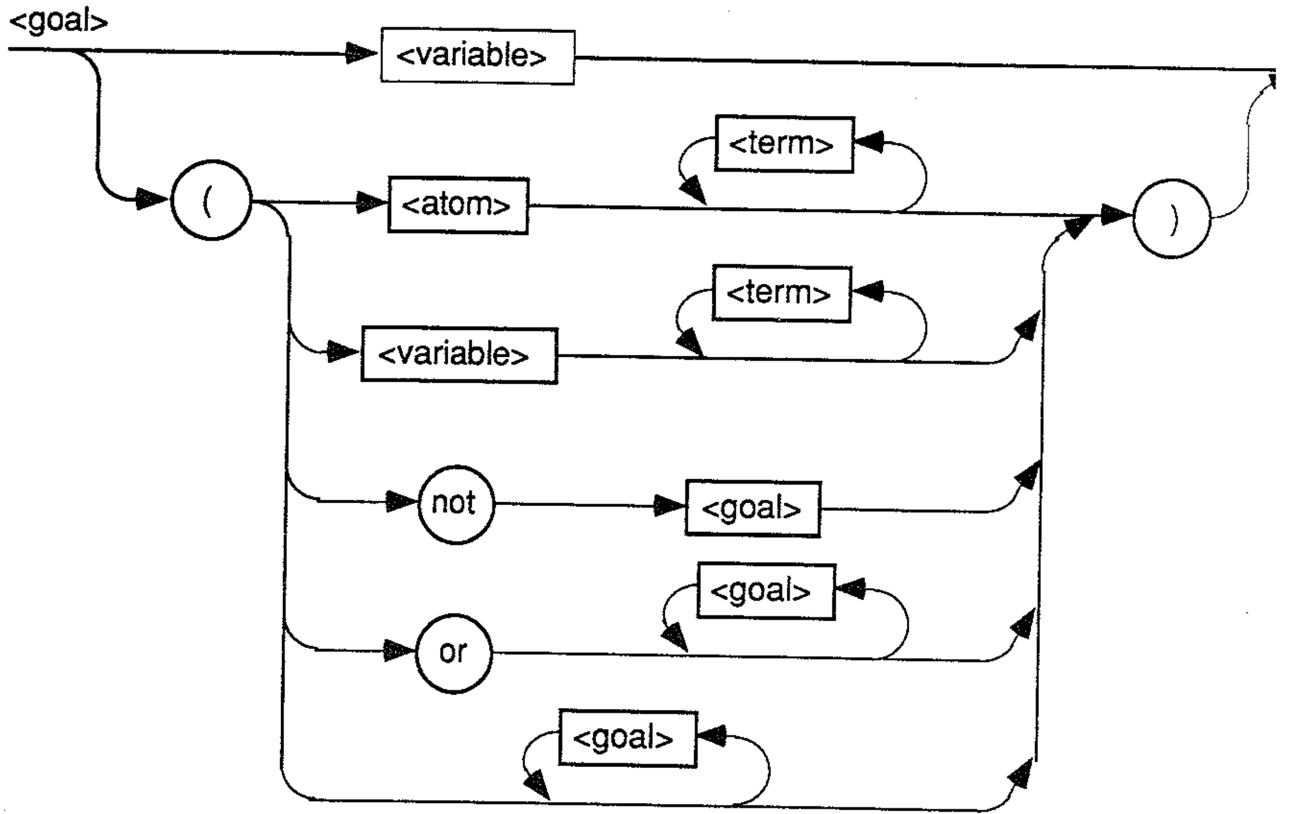
\b	backspace
\e	escape
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\z	bell
\nn	ASCII character nn in hexadecimal

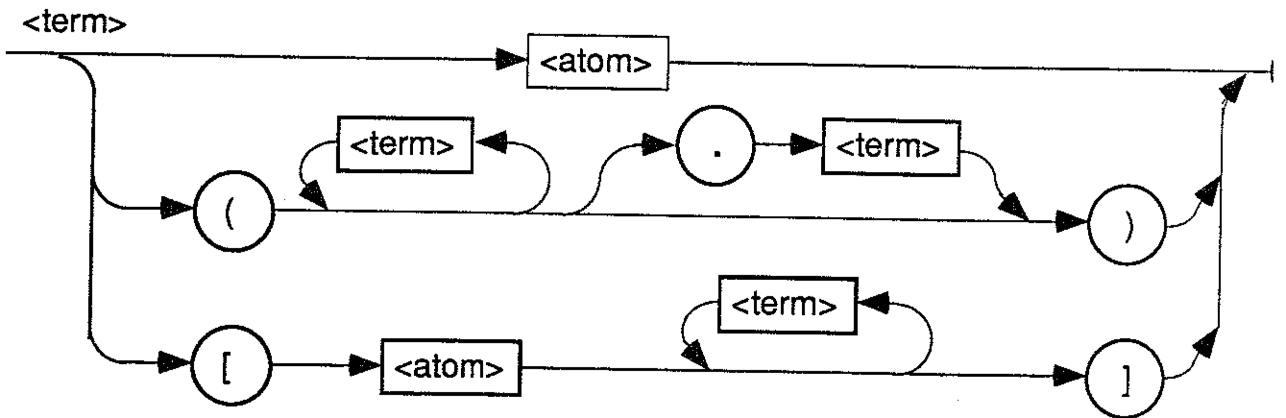
Appendix 2 - Prolog Syntax

The following syntax diagrams define valid constructs in VPI Prolog. Items within a circle or oval are to be taken literally. Items within a square or rectangle denote syntactic constructs defined elsewhere, either in another syntax diagram, or in Appendix 1.

In each syntax diagram, the syntactic object described by the diagram appears on the left, over the incoming arrow. An instance of the object can be constructed by following the arrows and concatenating the objects encountered until the rightmost arrow is reached.




<goal>
<variable>
(
<atom>
<term>
)
<variable>
<term>
not
<goal>
or
<goal>
<goal>


<term>
<atom>
(
<term>
.
<term>
)
[
<atom>
<term>
]

Appendix 3 - File Naming Conventions

The following file name endings are used for files manipulated by VPI Prolog:

".hc"	VPI Prolog input, including commands, assertions, and queries
".db"	Database input files
".env"	VPI Prolog environments
".hlp"	The VPI Prolog help file is named "prolog.hlp"
".ech"	Files produced using the "echo" command. They hold a copy of all output which appeared on the screen during a VPI Prolog session.
".tmp"	The paging file is named "prolog.tmp". Unless execution of VPI Prolog is terminated abnormally, this file is automatically deleted when the session ends.

Appendix 4 - Historical Perspective

The PROLOG implementation described in this report, the VPI Prolog compiler, implements a superset of PROLOG (=PROgramming in LOGic), a language originated in Marseille, France in 1972. PROLOG has a number of unusual features not shared by most programming languages:

1. it can be used as a very high level language in which complex code can be written with few statements.

2. statements in first order logic programs look like "denotational semantics," that is, statements look more like the specification of a problem than steps toward a solution. The language may be viewed as an implementation for a non-vonNeumann style of programming similar to that advocated by Backus [1].

3. it has a clearly defined formal semantics in first order logic.

4. it is the only language that unites the metatheory of computer science, i.e., theories of computation as developed by Godel, Church, Turing, and Post, with a practical syntax to produce a usable programming tool.

PROLOG belongs to a class of languages that have statements written as rules; flow of control does not pass sequentially from statement to statement as in most languages. Instead, rules are "triggered" as appropriate to compute an answer. The programs are thus highly modular and can be extended by adding new rules. Rule based languages are gaining popularity in artificial intelligence applications. They have been used for psychological modelling (Newell and Simon [9]), for modelling human decision making in expert domains such as medicine (Shortliffe [12]), molecular genetics, x-ray crystallography, geological prospecting, signal processing, etc., for natural language parsing and question answering, as a general tool in problem solving (robot problem solving, for example), and for database applications.

Rule based languages can be shown to be fully equivalent to more usual programming languages such as FORTRAN, COBOL, RPG,

and so on. Indeed, since PROLOG has a formal semantics in logic, all the formal language and recursive function theory can be applied to show its general computational power.

Rule based computation can be traced back to Greek mathematical theory, such as Euclid's axioms. Axiom systems were not well understood then, and it was not until the twentieth century that the implications of inference using axiom systems was explored. Godel's seminal paper [7] formalized the inherent difficulties of computation using axiom systems.

The historical precedent for precise studies of rule based computer languages goes back to the symbol manipulation systems of Emil Post [10]. Post proved that all computation can be seen as a set of rules that transform one string of symbols into other strings of symbols. He was able to prove that any such system can be reduced to a particular, simple normal-form. Post's work demonstrated the general power of computation specified using rules, but at the time the work was considered mainly of theoretical interest, since there were no functioning computers at that time and there definitely was no understanding of programming languages. In the early 1950s, A. A. Markov demonstrated the use of rules as a programming language in which algorithms may be expressed ("Markov algorithms").

Several effective rule based languages are currently available--Newell's PSG II, RITA, PROLOG, and the systems used by the Heuristic Programming Project at Stanford. Among these different languages, PROLOG has a very precise semantics defined in first order predicate calculus. Over the years, a number of top artificial intelligence researchers including McCarthy, Hayes, Kowalski, Weyrauch, and now even Newell have championed logic as a general representation method. PROLOG allows knowledge to be represented as inference rules in logic, and thus, logic specifications for representations of knowledge become executable programs. The metaphor for computation in such a system does not follow the standard vonNeumann/algebraic language scheme. Instead, computation can be thought of as controlled deduction or inferencing. The way that programs are executed also differs significantly from algebraic style languages. In particular, the rules in some sense denote the meaning, that is, the logic, of the program to be executed and some separate entity, an interpreter, ensures that the program is executed correctly. An algorithm, then, can be

seen as two separate entities: logic and control. In algebraic style languages following the FORTRAN model, logic and control are normally tied together in the statements of the language, leading to many programming errors.

In rule based languages, the control component causes execution of the rules to follow a non-deterministic, backtracking algorithm. Ideally, rule ordering should not matter for a purely logical specification of an algorithm. When efficiency becomes a primary concern, however, implementation of the control component forces an ordering onto the rule set. PROLOG executes rules in the order in which they were defined. In theory, the correct rule should be executed instead of the first applicable rule, thus maintaining a strict separation of logic and control. The actual compiler forces the programmer to consider some ordering of the rules to avoid infinite, left recursive looping. The method PROLOG uses to search automatically will be explained later in detail.

In PROLOG, the implementation of the control component is achieved using an automatic theorem prover. The logic specification for the program is given to the theorem prover, and the user enters the data that initiates the computation of the desired answer. An answer is derived by proving theorems using the logic specification of the program. A theorem prover, therefore, implements the control component of the system. Theorem provers have in the past run quite slowly, but with appropriate restrictions on the form of the logic specifications (rules), a PROLOG compiler can be shown to run at a speed comparable to compiled LISP and PASCAL. True logical negation and if-and-only-if logical operators cannot be efficiently implemented at this time. Some problems can be expressed more naturally with these logical conventions, so restricting the language to exclude them can occasionally cause programming difficulties. These difficulties can be overcome by programming with meta-level specifications embedded in first order logic , but the resulting programs are beyond the scope of this report. The interested reader is referred to the paper by K. Bowen and R. Kowalski [2]. Weyrauch [15] is a general source for meta-level programming ideas.

The user need not know the deep theory behind logic programming (such as Herbrand universes, satisfiability, models, or resolution) to be an effective programmer in PROLOG, although a thorough understanding of the theory of logic is helpful.

In summary, logic programming has arisen only recently although its theoretical roots go far back. Recent research activity in this area has produced a large number of books and papers of general interest. The reader is referred to Clocksin and Mellish [3], Gallaire and Minker [6], and Kowalski [8]. Logic programming conferences and workshops are held regularly, and a newsletter is available. In Britain, an experiment is under way to educate young children to program using a simplified PROLOG syntax. The interest generated in this language coincides with an increasing level of awareness that programming is enriched by having different metaphors for thinking about computation. The remainder of this report explains the syntax, semantics, and functions of VPI Prolog, including tutorial examples for the beginning logic programmer. Really learning to program in this system, however, requires consultation with an appropriate text on PROLOG and hands-on experience.

Appendix 5 - Error Messages

Error Types

There are nine types of error messages. The first eight types are documented here; the ninth type is for compiler errors. They contain no information useful to Prolog programmers, and their occurance should be reported immediately to those responsible for maintainance of the Prolog compiler. These error messages are documented in the Prolog Implementation Manual. The error types are:

1. warning: doesn't necessarily signal an error, but indicates an unusual condition which has a high probability of being an error, i.e. an attempt to open an already open file.

2. predicate error: a built-in predicate was passed an argument of an incorrect type or value, or an error occurred during its execution.

3. function error: a built-in function was passed an argument of an incorrect type or value, or an error occurred during its execution.

4. command error: a command was passed an argument of an incorrect type or value, or an error occurred during its execution.

5. syntax error: The construct being parsed has incorrect syntax.

6. O.K. limit error: a Prolog compiler limit was reached. The current computation will abort, but Prolog will continue to run.

7. bad limit error: a Prolog compiler limit was reached, and Prolog cannot continue. An error message is displayed, and Prolog is terminated.

8. compiler error: These should never occur, and therefore indicate a bug in the Prolog compiler.

In an error message, any text within curly braces ('{' and '}') will be replaced by appropriate text when the error message is actually issued. For example, if an error message is listed below as:

Error opening file {filename}

and the name of the file which couldn't be opened was "BasePreds.hc," then the actual error message issued will be:

Error opening file BasePreds.hc

Listed below are error messages issued by VPI Prolog, grouped by error message type.

Warnings

Source	Message	Explanation
lex	token too long, truncated	an input token was too long, and was truncated
plm	No clauses defined for {pred}	A goal was found for which no clauses were defined

Predicate Errors

Predicate	Error Message
assert	head must be a list
assert	head must start with an atom
assert	invalid head predicate
assert	keyword 'database' must follow ':'
assert	keyword 'database' must follow ':'
assert	dotted pair not allowed here
assert	dotted pair not allowed here
assert	goal must start with an atom
assert	dotted pair not allowed in goal
assert	argument must be a list
assert	database can only contain facts
assert	dotted pair not allowed as goal

assert	'if' expected
assert	dot not allowed as goal
assert	Goal must be a list
<=	both args must be atoms or numbers
>=	both args must be atoms or numbers
<	both args must be atoms or numbers
>	both args must be atoms or numbers
loaddb	Invalid char in TypeString: '{ch}'
loaddb	Only 1 term allowed after '.'
loaddb	':' not allowed, surround atom with quotes
loaddb	variables not allowed in database
loaddb	unexpected ')'
loaddb	Bad term
loaddb	last argument in fact {} not last on line
loaddb	unexpected end of file
loaddb	Unable to open database file
loaddb	error on 1st line of database file
loaddb	invalid input format
loaddb	wrong # of args
loaddb	invalid predicate
loaddb	type mismatch, fact # {nfacts}
loaddb	usage: loaddb <predatom> [<filename>]
loaddb	usage: loaddb <predatom> <filename>
help	Help file is empty!
help	Unable to find help file
help topics	Unable to open help file
usage	Unable to open help file
retractn	Delete rule #{n}, pred {} has only {n} clauses!
nthrule	1st arg ({term}) should be a predicate
nthrule	2nd arg must be an instantiated number

nthrule	2nd arg must be a number or variable
retract	argument not a list
retract	list argument's 1st elem not a list
retract	start of head not an atom
retract	database retract with RHS
retract	invalid head, end not nil
retractall	argument not a list
retractall	list argument's 1st elem not a list
retract	start of head not an atom
forget	argument must be an atom
forget	not a predicate atom
forget	not a predicate atom
connect	connection would create cycle
match	invalid pattern
match	arg3 must be a number or variable
match	arg4 must be a number or variable
open	Invalid mode, must be read, write, or append
connect	2nd arg must be an open stream
disconnect	unknown stream
divert	2nd arg must be an open stream
read	NOT IMPLEMENTED YET

Function Errors

Function	Error Message	Explanation
/	arg 2 can't be 0	attempted division by 0
eos	Unknown stream	

Command Errors

Command	Error Message
useenv	Unable to open file {name}
useenv	Invalid environment file ({s} section)
loadenv	Unable to open file {name}
loadenv	Invalid environment file ({s} section)
saveenv	Use 'savecurenv' to save the current environ
saveenv	Unable to open file {name}
savecurenv	Unable to open file {name}
read	Too many open input files
read	Unable to open file {name}
echo	Unable to open file '{filename}'
echo	Echo to what file?
read	Usage: read <filename>
debug	Invalid debug arg
step	usage: step <number>
goals	Goals not synchronized
runto	usage: runto <predatom>

Syntax Errors

Error Message
can't redefine builtin predicate {name}
variable invalid as a head predicate
Predicate {pred} expects {n} arguments, got {n}
invalid head predicate
metapredicate could never be instantiated
Predicate {} expects {n} args, got {n}
invalid goal predicate
invalid call kind
lex.EscapeChar: uncompleted hex escape char
invalid hex escape
uncompleted escape

uncompleted escape
unexpected EOF
Exponent expected
quoted string not terminated
Only 1 term allowed after '.'
Bad term
atom (predicate) expected
invalid head predicate
Unknown token following ':'
can't redefine user pred. {pred} as a database
atom (predicate) or variable expected
'(' expected
database can only contain facts
'(' or ')' expected
')', 'if' or ':-' expected
'(' or ')' expected
Wrong # of args for predicate {pred} in metapred.
clause or ')' expected
atom or goal expected
No new queries allowed, exit debug mode with 'debug off'

O.K. Limit Errors

Module	Routine	Error Message
codegen	AllocTempReg	too many registers allocated
codegen	EndClause	Out of code space!
codegen	BeginGoal	at most {n} goals allowed per rule
codegen	BeginMetaGoal	at most {n} goals allowed per rule
database	DbCreate	Too many databases, max = {n}
database	DbCreate	Too many databases, argument overflow
parse	VarTermPtr	clause has too many variables
stacks	SetVariable	Stack Overflow, E={n}, TR={n}
stacks	Trail	Trail Overflow, TR={n}, E={n}

stacks	HeapTerms	Heap overflow, H={n}, StackBottom={n}
stacks	HeapNumber	Heap overflow, H={n}, StackBottom={n}
stacks	AddEnv	too many environments
symtab	FoundVar	symbol table overflow
terms	GetUndefVarName	out of variable array space

Bad Limit Errors

In the following error messages, "out of memory" means that an attempt to allocate memory failed (the message is followed by the data structure which was being allocated). The messages "open error", "read error", "write error", and "seek error" refer to errors involving the paging file. All of these errors are considered very serious and result in termination of VPI Prolog.

Module	Routine	Error Message
atoms	TheAtom	too many atoms, Max={n}
atoms	AllocHashTable	out of memory (hash table)
codeutil	AllocCodeArray	out of memory (code array)
codeutil	AllocCodeArray	out of memory (query array)
lex	GetToken	token longer than {n}
page	PageToReplace	All pages are locked!!
page	WriteZeroPage	write error ({errno})
page	WritePage	open error {errno} on page file
page	WritePage	seek error ({errno})
page	WritePage	write error ({errno})
page	ReadPage	seek error ({errno})
page	ReadPage	read error ({errno})
page	RestorePageFile	open error ({errno})
page	RestorePageFile	read error ({errno})
page	RestorePageFile	write error ({errno})
page	SaveDirtyPages	open error {n} on page file
page	SaveDirtyPages	write error {n} on page file
specnum	init_specnum	out of memory (SpecAtomSpace)
stacks	AllocHeapStack	out of memory (heap & stack)
stacks	AllocHeapStack	out of memory (heap type array)

utils	init_utils	out of memory (String)

Appendix 6 - Glossary of Terms

clause - Consists of a goal (the 'head'), optionally followed by 'if' and a series of subgoals. At run time, a rule which is invoked either succeeds (possibly binding some variables in its argument list), or fails, forcing backtracking.

command - described throughout this manual. Commands may only be entered at the Prolog prompt, or be contained in files loaded using the read command. They cannot appear as goals, and may not be enclosed in parentheses.

fact- a rule with no subgoals

function - function calls are bounded by square brackets ('[' and ']') and may used as arguments to a goal, or arguments to other function calls. Functions are evaluated, producing a value (i.e. a number, atom, or list), which logically replaces the function call.

rule - Consists of a goal (the 'head'), optionally followed by 'if' and a series of subgoals. At run time, a rule which is invoked either succeeds (possibly binding some variables in its argument list), or fails, forcing backtracking.

infix - an infix operator is one which appears between the two operands it operates on, for example, in the expression 1+2, + is an infix operator

predicate - strictly, the word which follows the '(' beginning a goal. However, the term predicate is often used to refer to the set of rules which have a particular predicate name starting the head goal.

relation - A set of rules which have the same predicate name.

References

1. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Communications of the Association for Computing Machinery, September, 1977.

2. K. Bowen and R. Kowalski, "Amalgamating language and metalanguage," in K. L. Clark and S.-A. Tarnlund, Logic Programming, London: Academic Press, 1972.

3. W. Clocksin and C. Mellish, Programming in Prolog, Berlin: Springer, 1981.

4. H. Coelho and L. Pereira, Prolog By Example, Berlin: Springer, 1989.

5. R. Floyd, "Nondeterministic algorithms," Journal of the ACM, Oct. 1967, 636-644.

6. H. Gallaire and J. Minker, editors, Logic and Databases, New York: Plenum Press, 1978.

7. K. Godel, "On formally undecidable propositions of Principia Mathematica and related systems, I," see J. van Heijenoort, From Frege to Godel, A Sourcebook in Mathematical Logic, 1879-1931, Cambridge, MA: Harvard University Press.

8. R. Kowalski, Logic for Problem Solving, NY: North-Holland, 1979.

9. A. Newell and H. Simon, Human Problem Solving, Englewood Cliffs, NJ: Prentice-Hall, 1972.

10. E. Post, "Formal reductions of the genral combinatorial decision problem," American Journal of Mathematics, 1943, 197-268.

11. J. A. Robinson, "A machine-oriented logic based on the resolution principle," Journal of the ACM, January, 1965, 23-41.

12. E. Shortliffe, Computer-Based Medical Consultations: MYCIN, NY: Elsevier, 1976.

13. M. Stefik, "Planning with constraints," Artificial Intelligence Journal, 1981, 111-139.

14. D. Waltz, "Understanding line drawings of scenes with shadows," in P. Winston, The Psychology of Computer Vision, NY: McGraw-Hill, 1975.

15. R. Weyrauch, "Prolegomena to a theory of mechanized formal reasoning," Artificial Intelligence Journal, 1980, 133-170.

16. P. Winston and B. K. P. Horn, Lisp, Addison Wesley

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Field	Entry
1a. REPORT SECURITY CLASSIFICATION	Unclassified
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
3. DISTRIBUTION / AVAILABILITY OF REPORT	Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	Systems Research Center SRC-90-004
5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION	Systems Research Center
6b. OFFICE SYMBOL *(If applicable)*	
7a. NAME OF MONITORING ORGANIZATION	Naval Surface Warfare Center
6c. ADDRESS *(City, State, and ZIP Code)*	320 Femoyer Hall Virginia Tech Blacksburg, Virginia 24061
7b. ADDRESS *(City, State, and ZIP Code)*	Dahlgren, Virginia 22448
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	Naval Surface Warfare Center
8b. OFFICE SYMBOL *(If applicable)*	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(City, State, and ZIP Code)*	Dahlgren, Virginia 22448

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

Field	Entry
11. TITLE *(Include Security Classification)*	VPI Prolog User Manual
12. PERSONAL AUTHOR(S)	Dr. John W. Roach and John Deighan
13a. TYPE OF REPORT	Final
13b. TIME COVERED	FROM 2/10/89 TO 2/9/90
14. DATE OF REPORT *(Year, Month, Day)*	February 1990
15. PAGE COUNT	72
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This paper documents how a user may work with the VPI Prolog compiler. The user interface functions called debugging environment and input/output are all described in detail. Anyone using this manual should be able to program effectively using VPI Prolog.

Field	Entry
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	☑ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS
21. ABSTRACT SECURITY CLASSIFICATION	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	
22b. TELEPHONE *(Include Area Code)*	
22c. OFFICE SYMBOL	

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE

Unclassified