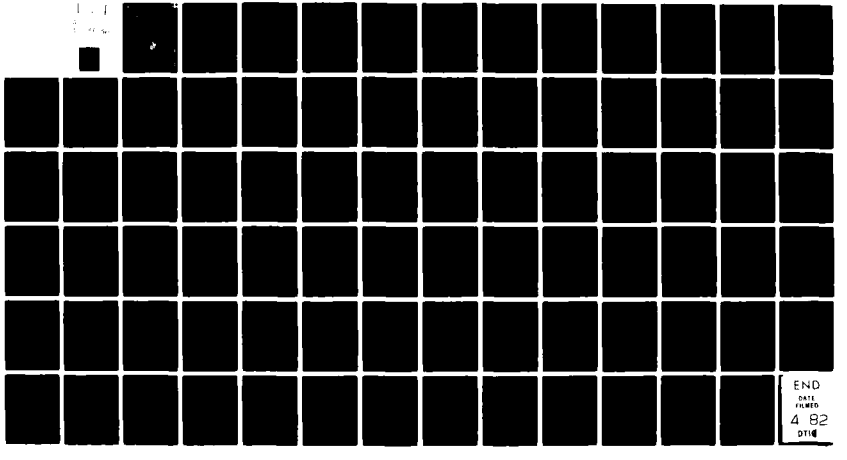
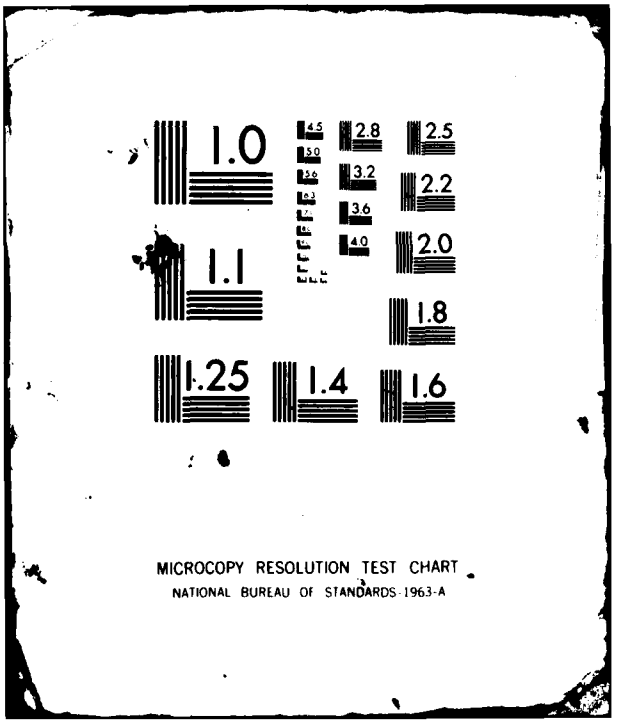
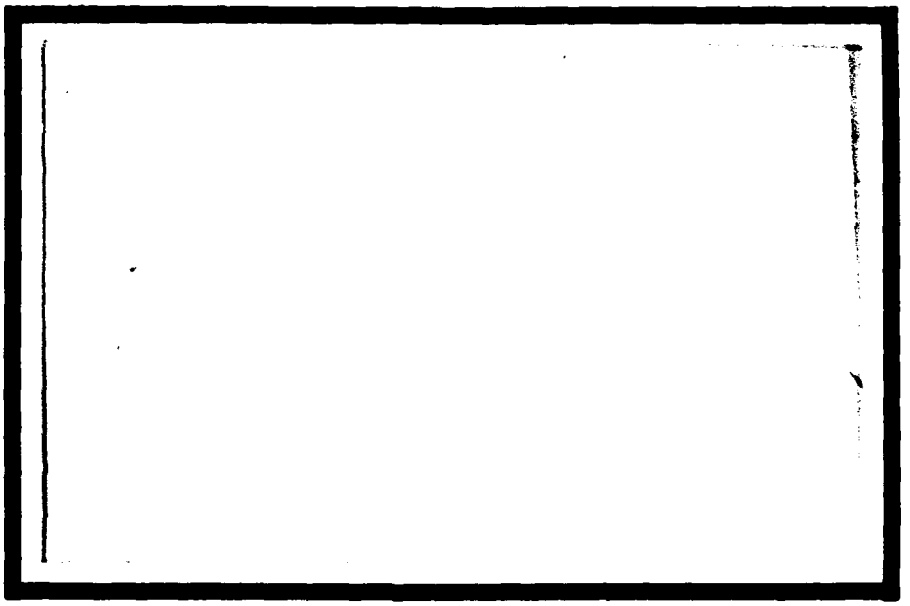
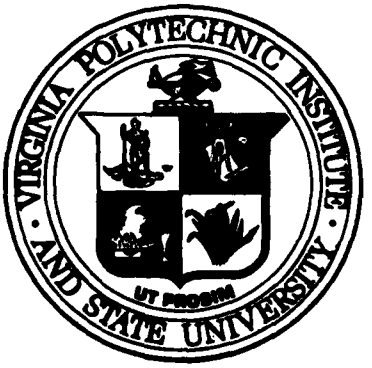
AD-A113 036 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG --ETC F/G 9/2
RULE-BASED PROGRAMMING FOR HUMAN-COMPUTER INTERFACE SPECIFICATI--ETC(U)
JAN 82 J W ROACH, G S FOWLER N00014-81-K-0143
UNCLASSIFIED CSIE-82-5 NL



END
DATE FILMED
4 82
DTIC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A





Virginia Polytechnic Institute and State University

Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

CSIE-82-5

January 1982

(7)

RULE-BASED PROGRAMMING FOR HUMAN-COMPUTER INTERFACE SPECIFICATION

John W. Roach

Glenn S. Fowler

TECHNICAL REPORT

Prepared for
Engineering Psychology Programs, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CSIE-82-5	2. GOVT ACCESSION NO.: AD-A113036
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): RULE-BASED PROGRAMMING FOR HUMAN-COMPUTER INTERFACE SPECIFICATION	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): John W. Roach Glenn S. Fowler	8. CONTRACT OR GRANT NUMBER(s): N00014-81-K-0143
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Virginia Polytechnic Institute & State University Blacksburg, VA 24061	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR SRO-101
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Code 442 800 North Quincy Street Arlington, VA 22217	12. REPORT DATE: January 1982 13. NUMBER OF PAGES: 76
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

dialogue, language, communication, programs, Markov algorithms, procedural languages, production systems, rule-based [illegible]es.

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

The specification of a human-computer interface requires a language in which that interface is expressed. Such a language should have a number of properties:

1) It should not be so syntactically complex that programming nonspecialists who must author dialogues have difficulty learning and using it.
2) It must be expressive and concise so that complicated interfaces can have a simple definition.
3) It ought to model human reasoning processes so that unnecessary formalisms

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

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT.

and constructs are not required of the dialogue author.
A number of types of languages are available for specifying dialogues, including procedural languages, graphical languages, and rule-based languages. Procedural languages have been thoroughly discussed, and it is well known that they are oriented toward explicit handling of control flow. Graphical languages, in which temporal sequences of tactile selections specify control flow are easier to use in specifying sequential events, but they are awkward for expressing more complicated control structures such as iterations. The third alternative; a rule-based language, is most useful in expressing pieces of knowledge and in expressing flow of information through a specific dialogue instance. This report describes an implementation of a rule-based language related to PROLOG for the specification of human-computer interfaces. It is based not upon von Neumann computer architectures but rather upon Post production systems or Markov algorithms, which are the foundations of computer science.

A

ACKNOWLEDGEMENTS

This research was supported in part by the Office of Naval Research under ONR Conract Number N00014-81-K-0143, and Work Unit NR SRO-101. The effort was supported by the Engineering Psychology Group, Office of Naval Research, under the technical direction of Dr. John J. O'Hare.

The authors wish to acknowledge the helpfulness of Dr. Daniel Chester, University of Delaware. The system discussed here is a modification of code that he originally started. The modifications have been considerable, and we must accept responsibilities for the mistakes embodied in this Horn Clause (HC) interpreter. Nonetheless, his advice was quite helpful. We would also like to thank the ONR group for patiently awaiting this report and for its spirited support.

TABLE OF CONTENTS

INTRODUCTION

The so-called procedural languages are by far the most common types of programming languages despite the fact that rule-based languages have fundamental importance in the theoretical foundation of the computer science field. This is probably attributable to the classes of problems that computers have traditionally been called upon to solve. Newell and Simon, on the other hand, believe that production system organization resembles the organization of human programs [10]. Consequently, many believe that production systems are an appropriate framework for modeling human cognitive systems. Since human-computer dialogue is a special case of human communication, it is also appropriate to investigate the usefulness of production systems for modeling and specifying human-computer dialogue. In this report a rule-based language similar to PROLOG is described that is implemented on the VAX 11/780. This implementation is intended, in part, to serve as a means for experimenting with the specification of human-computer dialogues using production systems.

Let us first make a cursory comparison of the two types of languages with two problems.

Problem: Given a list of items, count the number of items in the list.

Program 1.1

```
COUNT = 0
DO WHILE LIST = NIL
    LIST = STRIP (LIST)
    COUNT = COUNT + 1
END DO
```

Program 1.2

```
(COUNT LIST X) if (COUNT* LIST 0 X)
(COUNT* (H.T) X Y) if (= Z (+ X 1))(COUNT* T Z Y)
(COUNT* NIL X X)
```

Program 1.1 is a conventional program with a clear control structure that emphasizes such procedural steps as (1) setting a counter to zero, (2) iterating until the termination condition is reached, and (3) incrementing the counter in each iteration. Program 1.2 starts with a query (COUNT (ABC)X), and starting with the first rule, an attempt is made to match the current predicate with the left hand side of some rule. In this case, the first match occurs in the first rule, and the list ABC matches the variable LIST and the symbol X matches the variable X. Substitutions are made in the right side of the first rule to produce a new predicate, (COUNT* (ABC) 0 X), which now matches rule 2, and so forth. In the end, the predicate will be (COUNT* NIL 3 3) which matches rule 3 and returns the count, 3.

The parallel between Programs 1.1 and 1.2 is that in Program 1.2, rule 1 does the initializing, rule 2 does the counting, and rule 3 handles termination. The fact that these rules can be written in any order shows how control is de-emphasized in production systems. However, it may well be argued that counting, in the sense of the Peano postulates, is really a control issue and that Program 1.1 has greater clarity. However, consider a second example dealing with an issue in a hypothetical anytime reservation system.

Problem: For a given airport, determine the minimum layover time required in order to complete a flight connection.

Program 2

```
(LAYOVER FLT1 FLT2 TIME) if (GAPTIME X Y TIME)
(GAPTIME PI124 BA257 30MIN)
(GAPTIME PI953 EA420 40MIN)
      .
      .
      .
```

Program 2 starts with a query like (LAYOVER PI953 EA420 TIME) which matches rule 1. Substituting in the right hand side of rule one produces the predicate (GAPTIME PI953 EA420 TIME) which matches the third rule, returning a result of 40 minutes. What is dramatic about this example is that it is a program for doing a database search without ever having to confront the issue

of actually implementing it. As more information becomes available, predicates can be added as needed, and in any order. Thus, those aspects of human-computer dialogue that require deduction may well be specified much more effectively in a rule-based language.

CHAPTER 1

HISTORICAL PERSPECTIVE

The language described in this report, the Horn Clause (HC) interpreter, implements a substantial subset of PROLOG (PROgramming in LOGic), a language originated in Marseilles, 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. It has a clearly defined formal semantics in first order logic.

3. 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-von Neumann style of programming similar to that advocated by Backus [1].

4. It is the only language that unites the metatheory of computer science, i. e. theories of computation as developed by Gödel, 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 rule to rule 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 enjoying quite a vogue in artificial intelligence where they have been used for: psychological modelling, (Newell and Simon [10]); for modelling human decision

making in expert domains such as medicine (Shortliffe [13]), molecular genetics, x-ray crystallography, geological prospecting, and signal processing; for natural language parsing and question answering; and as a general tool in problem solving (robots, 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.

The history of rule-based computation goes back to Greece, and Euclid's axioms in particular. Axiom systems were not, however, well understood, and it was not until the twentieth century that the implications of inference using axiom systems was explored. Gödel's seminal paper [8] formalized the inherent difficulties of computation using axiom systems be they for logic or integers (or geometry).

The historical precedent for precise studies of rule-based computer languages goes back to the symbol manipulation systems of Emil Post [11]. 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 to become executable programs. The metaphor for computation in such a system does not follow the standard von Neumann/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. The interpreter scans the rule set from top to bottom and executes the next applicable rule in sequence. In theory, the interpreter should execute the correct rule instead of the first applicable rule, thus maintaining a strict separation of logic and control. The actual interpreter forces the programmer to consider some ordering of the rules to avoid infinite, left recursive looping. The backup structure of PROLOG 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 interpreter can be shown to run at a speed comparable to compiled LISP and PASCAL. 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 [16] is a general source for meta-level programming ideas. The user need not know about Herbrand universes, satisfiability, models, or resolution to be an effective programmer in PROLOG, although a thorough understanding of logic theory is helpful.

Logic programming has arisen only recently although its theoretical roots go far back historically. 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 [4], Gallaire and Minker [7], and Kowalski [9]. Conferences and workshops are held regularly, and a newsletter is available. In Britain, an experiment is under way to educate young children in programming 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 a logic-based language, including tutorial examples for the beginning logic programmer.

CHAPTER 2

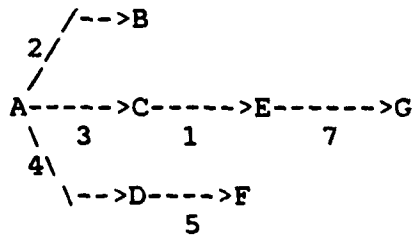
Programming in PROLOG

In this chapter, elementary aspects of programming in HC, the name of the VAX 11/780 implementation of PROLOG available at Virginia Tech, are discussed, the syntax and semantics of the language are presented, and a list of built-in functions and features are given. 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 [4] and Kowalski [9] 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 other versions (there is no standard syntax), but this should not be bothersome due to the simple structure of the rules. Built-in functions, although not standardized, are few and easy to learn.

Many programming languages already exist in the world today, so why bother learning a new language? FORTRAN, for example, can compute anything. In answer to this query, PROLOG computes answers to problems in certain application domains more conveniently than conventional algebraic languages. 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, Cotta, and Pereira [5] 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 an HC 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:

```
(assert
(   (distance a b 2)   )
(   (distance a c 3)   )
(   (distance a d 4)   )
(   (distance c e 1)   )
(   (distance e g 7)   )
(   (distance point1 point2 ans) if (distance mid point2 x)
                                    (distance point1 mid y)
                                    (= ans (+ x y))   )
)
(variable ans mid point1 point2 x y)
```

By typing (if (distance a g x)), the user can determine the distance between points a and e in the graph; the answer will be returned in the x variable. The system responds to the (if

(distance a g x)) statement with the reply, (distance a g 11). The details of what makes this program work are explained below.

Rule Syntax and Semantics in HC

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

(goal if subgoal1 subgoal2 ... subgoaln)

where goal and each subgoal can be expanded as a relation on some arguments. 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, as originally stated, the left side as a goal and the right side as a list of subgoals to be reached or demonstrated, or 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 two other forms derive from it. First, 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 the rule is optional in this case. Thus, facts may be expressed as:

```
( ( blackstriped henry ) )
```

If the programmer writes

```
  ( blackstriped henry )
```

instead, HC cannot use it; outer parentheses are needed to show that this relation is a rule. The second reduction of the general rule form has no left-hand side:

```
(   if   subgoals )
```

and is used to invoke the theorem prover, usually from command level. Essentially, the HC system is being asked to use previously-defined rules to establish the conjunction of the subgoals. A typical program has the following form:

```
(variable x y ... )
(assert
( goal  if  subgoal1 ... subgoaln )
( goal  if )
( goal )
)

(if subgoal1 subgoal2 ... )
```

where the VARIABLE function declares the variables to be used, ASSERT defines the rules to be used, and the final line asks the questions (that is, subgoals) that HC answers using the rules. Here is an example of a complete program:

```
(variable x)
(assert
( (mammal x) if (has x hair) )
( (mammal x) if (gives x milk) )
( (bird x) if (has x feathers) )
( (bird x) if (flies x)(oviparous x) )
( (carnivore x) if (eats x meat) )
( (carnivore x) if  (has x pointedteeth)
                    (has x claws)
                    (has x forwardeyes) )
( (ungulate x) if (mammal x) (has x hoofs) )
( (type x cheetah) if (mammal x)
                      (carnivore x)
                      (color x tawny)
                      (has x darkspots) )
( (type x tiger) if (mammal x)
                    (carnivore x)
                    (color x tawny)
                    (has x blackstripes) )
( (type x giraffe) if (ungulate x)
                      (has x longneck)
                      (has x longlegs)
                      (has x darkspots) )
( (type x zebra) if (ungulate x)
                    (has x blackstripes) )
( (type x ostrich) if (bird x)
                      (notfly x)
```

```
                    (has x longneck)
                    (has x longlegs)
                    (color x blackwhite) )
( (type x penguin) if (bird x)
                    (notfly x)
                    (swims x)
                    (color x blackwhite) )
( (type x albatross) if (bird x)
                      (flyswell x) )
( (has henry hair) if )
( (eats henry meat) if )
( (color henry tawny) if )
( (has henry darkspots) if )
)
;
;   now find out what henry is
;
(if (type henry x) )
```

In response to this query, HC responds:

(type henry cheetah)

(Adapted and simplified from Winston and Horn [17]). 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). Had the system been unable to compute an answer, nil would have been returned indicating a failure. Notice that an animal can be shown to be a mammal in two different ways. X is a mammal if x has hair &O&R 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 the HC interpreter 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. Unlike a normal algebraic language, however, the right side may fail without causing an error message. Consequently, if a programmer makes a mistake such as not declaring a variable, a rule will fail without giving an error indication; it will simply return nil. Global variables accessible by different rules do not exist in this language. Note that the rules are free format and that comments start with a semicolon. Most importantly, notice how easily knowledge about animals is expressed in this language. Additional knowledge about animals is easily added by calling the *assert function* with new rules. The new rules are added to the end of the old 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), variables, and strings are available in HC, as well as means of using them, including user-defined and built-in data relations. Arithmetic operations on numbers, such as addition, multiplication, trig functions, along with list operations are included. Arithmetic operations are discussed in detail below.

Strings are typically used in print and load statements. For example, (print "enter next rule") would allow the string within double quotes to appear on the user's terminal. The correct syntax for using a string requires the double quote on each end. Strings may have up to sixteen characters; strings and symbols exceeding this length produce an error condition. The maximum string length can be changed by modifying the source code of the interpreter. See the section (pp. 48-51) that describes user modifications to the interpreter. String operators such as concatenation and substring matching are not available in the current version.

Lists constitute a general-purpose method of grouping data together. For example,

(eggs butter milk (bluecheese cheddarcheese) cereal)

is a grocery shopping list containing a sublist of cheeses to buy. Note that the parentheses must be included. 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, genealogical 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 HC programmers.

The primitive list-manipulation functions are defined using the so-called dot operator. Actually, 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 and the rest of the list. The dot operator can be applied repeatedly, ultimately reducing a list to emptiness. The empty list is denoted by NIL which is both a symbolic constant and a list (empty). NIL is automatically supplied whenever the user reduces a list to the empty state; if one can manage to reduce the empty list any further, the resulting operation produces emptiness. The dot operator is used so frequently that the HC system provides an infix notation for it. For example, here is a one-line program for printing the elements of a simple (one level) list:

```
( (printeach (x . y)) if (print x) (printeach y) )
```

where x and y are variables and the PRINT relation is a built-in function that prints its arguments--including symbolic constants, numbers, variable names, or lists. Invoking the theorem prover with

```
(if (printeach (roses are red and violets are blue) ) )
```

produces

```
roses
are
red
and
violets
```

```
are
blue
nil
```

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 the prover to print what x has been matched to and then, recursively, to 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 is explained further in the matching section (pp. 22-26). Since no rules match, the system returns nil to indicate a failure to compute an answer; the returned answer, nil, is then printed by the system. 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, or lists. HC 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 concatenate two lists, reverse a list, get the last element of a list, etc., are normally included in programs. The concatenation 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:

```
( (concatenate nil x x) )
( (concatenate (x . y) z (x . w)) if (concatenate y z w) )
```

where w, x, y, and z are all variables. Consider the first two arguments of each CONCATENATE 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 whatever matches x in the second position is the answer of the whole CONCATENATE procedure. The second rule removes the front element x from the first input list, determines the resulting concatenation w of the remainder of the first list y with the second list z, and finally puts x onto the front of the resulting answer w with the (x . w) operation. This may seem mysterious at first, so try it! Use the trace feature described in Chapter 3. Chapter 3 also discusses other practical details necessary for running this program such as calling HC, loading files, and common programming errors.

The following code defines a recursive quicksort program. Input to QSORT, 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 quicksorted. 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-structure principles discussed in this section.

```
;
; quicksort program
;
(variable ans less1 more1 less more x y z)
;
(assert

;
; base case
;
((qsort nil nil))

;
; induction case
;
((qsort (x . y) ans) if (partition x y less more)
                        (qsort less less1)
                        (qsort more more1)
                        (concatenate less1 (x . more1) ans))

;
; base case
;
((partition x nil nil nil))

;
; partition into less than list
;
((partition x (y . z) (y . less) more) if
                        (< y x)
                        (partition x z less more))

;
; partition into greater than or equal list
;
((partition x (y . z) less (y . more)) if
                        (partition x z less more))
;
((concatenate nil x x))
((concatenate (x . y) z (x . u)) if (concatenate y z u))
)
;
; example call to QSORT
;
(if (qsort (jim bill carl allen george) x) )
```

The execution efficiency of this code can be improved by eliminating the CONCATENATE function but at the cost of making the program less understandable.

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 HC 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 HC programs.

Matching

Pattern matching triggers the rule-based computation or inferencing in HC. 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 [12] for full details. Rather than dwell on the minute, difficult details of the formal definition, the exposition here intends to give a feeling for how matching 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 matches the left-hand side of a rule.

When matching two relational expressions, the relations must match and the number of arguments of each relation must be the same. Two constants 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 (nextto robot x) and (nextto robot couch), where x is a variable, the two relational expressions can be unified by matching x with couch; given (grandparent x y) and (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) and (quicksort (bill pat john alan cathy) v), these two relational expressions can be unified if x matches bill, y matches (pat john alan cathy) and z matches v. The expression matching (x . y) is used as input to the rule containing (quicksort (x . y) z), so the dot operator is breaking down the original name list, (bill pat john alan cathy). In the concatenation program discussed in the section on data types and data structures,

((concatenate (x . y) z (x . w)) if (concatenate y z w))

the dot operator both breaks down lists that match (x . y) and builds up lists from components x and w with (x . w). 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). A case 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 "pat", for example.

A matching convention that can be very helpful involves an extension of the dot operator to list matching. 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 HC 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 (function x)) and (relation (function x) x). Attempting a unification here results in x matching (function x), so that x is recursively defined. The upshot is that these two expressions cannot be legally unified; they should not and will not match. Expressions of this kind seldom occur, and the unification algorithm employed here does not check for such situations. The unification algorithm without the 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, causing a considerable performance degradation. Unification is the most often performed operation in this language, so reducing the time spent in this operation 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 wild-card matches. Two symbols, * and ?, are used to allow matching on arbitrary objects in the language. The * wild-card will match an arbitrary list structure, relation with arguments, or a constant; 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. The ? wild-card restricts the matching process more than *; it cannot match arbitrary lists and relations, rather it is restricted to

matching single constants or variables. Thus, (loves ? ?) matches (loves jack jill) but will not match (loves (jack jones) (jill smith)). The * and ? wild-cards extend the power of the unification process considerably, allowing many diverse patterns to be matched.

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 [6]). Much of the control that normally must be specified by the user is thus built into the PROLOG interpreter. This feature of the language has prompted Kowalski [9] 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 errors are introduced by control mistakes,

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.

Now the ideas behind the control procedures will be explained, illustrated at the end by a simple robot-program. For each rule in a program, "g1 if sub1 sub2 sub3", say, 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. The HC implementation of PROLOG keeps a goal stack and when the front goal on that stack, g say, 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 demonstrating, that is 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 front 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 (if g1), the system would first attempt to solve subgoals sub1 sub2 sub3. If these three subgoals could not be solved, only then would the HC interpreter 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 then 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 robot-program which illustrates the control and backup ideas:

```
;
;       1 robot
;       3 boxes
;
(var u v w x y z)
(assert
;
; solution control
;
        ((prob x y) if (solves z y (state0) nil)
                       (reverse z nil x) )

        ((reverse nil x x))
        ((reverse (x . y) z u) if (reverse y (x . z) u))

        ((solves x nil x nil))
        ((solves x nil x (y . z)) if
                (holds y x) (solves x nil x z))
        ((solves x (y . z) u w) if
                (makestrue v y u) (solves x z v (y . w)))

        ((makestrue x y x) if (holds y x))
```

```
;
; invariant truths
;
        ((holds (loose box1) w))
        ((holds (loose box2) w))
        ((holds (loose box3) w))

        ((holds (type box1 box) w))
        ((holds (type box2 box) w))
        ((holds (type box3 box) w))
;
; go next to operator
;
;    add rule
       ((holds (next robot x) ((gonext x) . w)))
;
;    delete rule
       ((holds w ((gonext x) . y)) if
              (!= w (next robot v)) (holds w y) )
;
;    preconditions
       ((makestrue ((gonext x) . w) (next robot x) w)  )

;
; push operator
;
;    add rules
       ((holds (next x y) ((push x y) . w)))
       ((holds (next x y) ((push y x) . w)))
;
;    delete rule
       ((holds w ((push x y) . z)) if
              (!= w (next v x)) (!= w (next x v))
              (holds w z) )
;
;    preconditions
       ((makestrue ((push x y) . u) (next x y) w) if
              (!= x robot) (!= y robot)
              (solves u ((loose x) (next robot x) )
                     w nil) )
       ((makestrue ((push y x) . u) (next x y) w) if
              (!= x robot) (!= y robot)
              (solves u ((loose y) (next robot y) )
                     w nil) )
)
;
; hard problem
;
(if (prob x ((next box1 box2) (next box2 box3)) ) )
```

This program may look formidable, but by tracing through the rules, the backup can be understood. Basically, the first seven rules, the "solution control," define the program and the remaining rules comprise data that encode facts about a simple robot-world. Robot problems are difficult to solve because the solution to one problem may undo the solution to a problem solved earlier. Thus, pushing box1 next to box2 can be undone in the next part of the problem by pushing box2 next to box3. The correct solution returned by the prover is:

```
(prob  (state0 (gonext box1)
               (push box1 box2)
               (gonext box3)
               (push box3 box2)  )
       ( (next box1 box2)(next box2 box3)  )   )
```

The backup necessary to achieve this answer will be explored next. To understand the explanation, the reader will need to refer regularly to the "solution control" section of the rules of the robot program.

The first "solution control" rule says that a problem list y can be solved by z which must be reversed to make it look nice. The rules that define the "SOLVES" relation reduce the problem list one goal at a time (third SOLVES rule) by making true the front goal and then recurring on the rest of the goals list. When no more goals remain to be solved (the goal list is reduced to NIL), the first two SOLVES rules check to make sure that all goals still HOLD, that is, that no later operation undoes an earlier solution to a goal (this is checked by the first two SOLVES rules).

For example, in the robot problem above, after solving the first goal, (next box1 box2), the system tries to solve (next box2 box3) using the following rule

```
( (solves z (next box2 box3)
          ( (push box1 box2)(gonext box1)state0 )
          ( (next box2 box3)(next box1 box2) )   if
    (makestrue v (next box2 box3)
                ( (push box1 box2)(gonext box1)state0 )  )
    (solves z nil v ((next box2 box3)(next box1 box2)) ) )
```

The first makestrue rule that matches, the last rule in the solution control-section, tries to establish that (next box2 box3) already holds in the world description generated from the solution to the first goal, namely ((push box1 box2)(gonext box1)state0). That attempt fails since (next box2 box3) is not the case. The next makestrue rule that matches attempts to make (next box2 box3) true by applying the operator (push box2 box3). The solution thus generated looks like this,

```
( (push box2 box3)
  (gonext box2)
  (push box1 box2)
  (gonext box1)
  state0 )
```

(remember that solutions are generated backwards until they are REVERSEd). The second SOLVES rule determines that (next box2 box3) HOLDs but then determines that (next box1 box2) does not HOLD, so the system begins backing up. Since in the second SOLVES rule, (holds (next box2 box3)) was the last solved subgoal, the system tries to find an alternate proof to (holds (next box2 box3)) on the possibility that (holds (next box1

box2)) can then be demonstrated. No alternate proof to (holds (next box2 box3)) can be found, however, so the second SOLVES rule fails. No other rule matches the goal that originally triggered the second SOLVES rule, namely,

```
(solves z
        nil
        ( (push box2 box3)(gonext box2) ... )
        ( (next box2 box3)(next box1 box2) )   )
```

so this goal fails completely. This goal was generated in the third SOLVES rule after (MAKESTRUE v (NEXT BOX2 BOX3) ((PUSH BOX1 BOX2)...)). The system backs up to the MAKESTRUE and tries again to make (next box2 box3) true. The next rule that can satisfy the (MAKESTRUE v (NEXT BOX2 BOX3) ...) goal says

```
( (makestrue ((push box3 box2) . u)
             (next box2 box3)
             ( (push box1 box2) ... )    if   etc.  )
```

This goal generates ((gonext box3)(push box3 box2)), which when added on to the previously-computed actions gives the correct answer. The backtracking described here is completely automatic; essentially, an exhaustive depth-first tree-search algorithm has been built into PROLOG.

User Control: The Slash 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 and waste large amounts of time and space, for example, even when the programmer knows better. Allowing the programmer some degree of control is

therefore desirable. A slash operator (the / operator) has been implemented to help the user control the automatic tree-search. The / operator prevents the system from backing up and trying different alternatives; that is, control can back up to the / but cannot cross back through it. The slash is like a fish weir: it is easy to pass through the slash, it always succeeds, but it is impossible to go back past it. The slash is used in a group of rules when the programmer is certain that if a particular rule is invoked nothing else should even be tried. That rule should then contain a slash. Consider the following union of two sets program:

```
( (union nil x x) )
( (union (x . r) y z) if (member x y) / (union r y z) )
( (union (x . r) y (x . z)) if (union r y z) )
```

where r, x, y, and z are variables. The slash 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 slash (sometimes called "cut") operator is not an operator 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 by 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,

although this occurs infrequently. Code with many slash operators tends to be more difficult to read and understand.

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. HC, however, defines relationships between arguments and thus does not insist on a variable being an input or output argument. Rules in HC are therefore non-directional with regard to inputs and outputs. Inputs can be passed into a set of rules in any pattern and HC 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, HC must "choose" between possible answers to be output through un-unified variables. Since the HC system incorporates automatic backtracking, the choice function can be seen as an implementation of the non-deterministic programming ideas explained in Floyd [6]. For example, consider the CONCATENATE rules defined (pp. 16-22) in the Data Structures and Data Types section of this report (repeated here for reference),

```
( (concatenate nil x x) )
( (concatenate (x.y) z (x.w)) if (concatenate y z w) )
```

If the third argument is given as input with the first two arguments variables, as in (if (concatenate variable1 variable2 (this is a test))), then the following answers are all possible:

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

Any one of these answers could be selected as the answer to the (concatenate variable1 variable2 (this is a test)) query. Since backtracking is algorithmic, the choice of which output to return cycles through all possible outputs if necessary. Embedding the CONCATENATE relation in the following rule,

```
( (test x) if (concatenate variable1 variable2 x)
               (fail) )
```

and entering the query (if (test (this is a test))), will cause HC to backtrack to the CONCATENATE relation until all possible answers are generated. After the last possible answer is generated, NIL will be returned by the CONCATENATE relation.

Essentially, rules can be seen as 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 [14], Waltz [15]). The ability to formulate problem-solving strategies using the constraint-satisfaction property of HC relations remains to be tested.

System functions cannot be treated as being relationally defined. This means that input and output arguments are fixed since the routines implementing the functions are coded in a conventional programming-language. Thus, the + function cannot take variables as input and calculate values for them: (= 10 (+ x 4)), although logical, is illegal. Similarly, the IF relation cannot allow a variable on the left-hand side of a rule (its first argument) with inputs on the right-hand side.

Numeric Calculation Routines

HC 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 HC are converted into floating-point format to avoid conversion problems between types such as integer, floating point, double precision, bignums, 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 3.1416 automatically when printed. Due to the input-

conversion algorithm and the word length of the VAX, the actual accuracy extends to the sixth digit after the decimal point. The user can control the number of digits that the system displays by typing (= precision 8) at the command level, at a BREAK, or at a continue? 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, trigonometric, logarithmic, and square-root operations are available. By typing (stat f) inside the interpreter, the programmer can get a list of all functions, including arithmetic, that are implemented. PROLOG does not have an assignment operator but rather uses the unification process. To force unification within an arithmetic rule, 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.

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.

The system has a number of built-in arithmetic functions. The following list gives a complete catalog: +, -, *, /, sqrt, esp, ln, truc, abs, sin, cos, tan, arctan. Example: if the expression (= x (exp 2.2)) occurs on the right-hand side of a rule, x will be unified with 9.0250.

System Defined Functions

This section lists the system-defined functions available to the user. These functions when encountered on the right-hand side of a rule execute automatically and return answers like functions in a normal programming-language. If the programmer wants to retain an answer that has been returned from a function, the = function described below must be used. The programmer may re-define a system-defined function if desired by asserting rules with the same name as the system function; a warning message is issued to the user when a function gets re-defined. In general, it is unwise to declare variable names that conflict with function names.

At the top level of the interpreter and in a BREAK the interpreter uses a simple LISP evaluator to maintain values of variables between calls to IF or ?. Certain LISP functions, CAR, CDR, COND, CONS, EVAL, LIST, QUOTE, SET, and SETQ are therefore available to the user. In general, the programmer should not use these functions in rules. If the documentation below mentions "evaluation" or "being evaluated" it is referring to a LISP evaluation, and the normal LISP functions can be used. At

command or in a BREAK, functions marked by [fexpr] do not evaluate their arguments before execution (that is, before EVALuation). During proofs, no functions EVAL their arguments and variables are passed according to their most recent unification.

(&& <arg1> <arg2> ...) [fexpr]

The arguments are evaluated from left to right. The first nil argument terminates the evaluation and nil is returned, else the value of the last argument is returned. This is essentially a compound "and" statement.

(|| <arg1> <arg2> ...) [fexpr]

The arguments are evaluated from left to right. The first non-nil argument terminates the evaluation and its value is returned. If none of the arguments can be evaluated to a non-nil value, nil is returned for the entire relation. Essentially, this is a compound "or" statement.

(assert <as1> <as2> ...) [fexpr]

New assertions <as1> <as2> ... are added to the list of current rules. t is returned.

(atom <arg>)

Returns t if <arg> is an atom, else nil.

(axioms '(<as1> <as2> ...))

New axioms <as1> <as2> ... are added to the current list of rules. The list of assertions added is returned. If no arguments are specified, (axioms) will return a list of current assertion/predicate names.

(boundp <arg>)

Returns t if <arg> is a bound atom, else nil.

(break <prompt>)

Causes the interpreter to interrupt execution until the user enters t or nil. If t is entered, the BREAK relation returns a non-nil value to the prover in the rule where it was invoked; entering nil causes nil to be returned. A prompt character other than * is recommended to differentiate a "break" from the command level.

(car <arg>)

The first element in the list specified by <arg> is returned. nil is returned when <arg> is atomic.

(cdr <arg>)

The list specified by <arg> is returned minus the first element. nil is returned when <arg> is atomic.

(delete <as1> <as2> ...) [fexpr]

Delete removes <as1> <as2> ... from the current list of assertions and returns t.

(cond (<tst1> <fun1> ... <ret1>) ... (<tstn> <funn> ... <retn>) [fexpr]

Cond evaluates <tst1> through <tstn> until a non-nil value is reached. Then <funm> through <retm> are evaluated and cond returns <retm>. If <funm> or <retm> are missing, cond returns <tstm>.

(cons <arg1> <arg2>)

Cons constructs a new list with <arg1> as the first element (car) and <arg2> as the remainder (cdr). The new list is returned.

(delete <as1> <as2> ...)

Delete removes <as1> <as2> ... from the current list of assertions and returns t.

(dump "filename" [as1 as2 ...])

Dump takes writes assertions out to a file specified by "filename". Note that the quotes are necessary. Assertion names are optional (as indicated by the brackets) but if no assertions are mentioned, then all current assertions are written to the file. If any assertion names are mentioned then only the current assertions mentioned with that name are dumped to the file. A new version of the file is created if possible and T is returned. If HC cannot create a new version of the file, NIL is returned.

(eval <arg>)

<arg> is evaluated.

(gensym) [fexpr]

A new, unique atom is generated and returned; gensym stands for generate symbol. Use (= x (gensym)) where x is a variable.

(if <as1> <as2> ...) [fexpr]

HC attempts to prove the validity of all of its arguments according to facts and assertions previously entered via (assert ...) and (axioms ...). <ax1> <ax2> are and-ed together so if any <axi> cannot be proven, HC returns nil. The ? function is a synonym for the if function.

(index <arg>)

The first atom in the list <arg> is returned.

(list <arg1> <arg2> ...)

<arg1> <arg2> ... are concatenated to form a new list which is returned.

(load <file1> <file2> ...) [fexpr]

<file1> <file2> ... are read in as if they were entered from the terminal. Note that file names containing [.()] must be quoted, i.e., (load "prob.hc"). LOAD will not allow the user to load the same file more than once; NIL will be returned. Loading a file that itself contains a LOAD command will cause HC to load both files.

(member <arg1> <arg2>)

Returns t if <arg1> is in the list <arg2>, else nil.

(! <arg>)

At the command level t is returned if <arg> is nil, else nil. During proofs t is returned if <arg> cannot be proven by HC, else nil.

(print <arg1> <arg2> ...)

The arguments are printed one per line on the output.

(quit <arg1> <arg2> ...) [fexpr]

The arguments are printed one per line and HC exits.

(quote <arg>)

'<arg>

These two forms are equivalent and cause <arg> to be returned without evaluation.

(rand [n])

This function returns a pseudo random number between 0. and 1. not including 0. or 1. The seed, an integer n, is optional (as indicated by the brackets); the system automatically supplies a default seed.

(readvar <arg> <promptstring>)

If <arg> is a variable, the interpreter reads an expression from the user and unifies that expression with the variable. The <promptstring> is useful to let the user know that the interpreter is waiting for input.

(set <arg1> <val1> <arg2> <val2> ...)

The value of <arg1> is set to <val1>, <arg2> to <val2>, etc. The last <val> argument is returned.

(setq <arg1> <val1> <arg2> <val2> ...)

This is shorthand for (set '<arg1> <val1> '<arg2> <val2> ...).

(= <var1> <val1> <var2> <val2> ...) [fexpr]

<var1> is unified with <val1>, <var2> is unified with <val2>, etc. during proofs. t is returned when <var1> <var2> ... are bona fide variables. At the command level, = is a [fexpr] implementation of the set function.

(stat [acfgluvz])

Stat allows the internal state of HC to be displayed. The flag arguments determine the manner of display.

- a all bound atoms are listed
- c all constants are listed
- f all built-in functions are listed
- g garbage collection is performed
- l the number of in-core list cells is displayed
- u all unbound atoms are listed
- v all variables are listed
- z all flags but [gl] are set

The default flags for stat are [a].

(stop <arg1> <arg2> ...)

The arguments are printed one per line, HC terminates the current proof, and returns to the command level. nil is always returned.

(trace <relation1> <relation2> ...)

The relation names specified are marked for tracing by AFLAG. If XFLAG is set, these assertions will be listed during proofs.

(try <as1> <as2> ...)

HC attempts to prove the validity of each argument according to facts and assertions previously entered via (assert ...) and (axioms ...). nil is returned when <asn> cannot be proved.

(untrace <relation1> <relation2> ...)

Unmarks the relations specified; see the trace function description.

(variable <arg1> <arg2> ...) [fexpr]

The arguments are declared variables and will receive special treatment during proofs. The name of the function may be abbreviated to var. If no arguments are specified, (variable) evaluates to a list of all declared variables. Do not declare t or nil as variables|

(? <as1> <as2> ...) [fexpr]

This is shorthand for (try '<as1> '<as2> ...).

The following built-in predicates can be used to test relations between arguments.

(== <exp1> <exp2>)

Tests for equality of two expressions. If there are any unbound variables in either expression, they will be automatically unified with the corresponding argument of the other expression to force equality.

(!= <exp1> <exp2>)

Tests for the inequality of the two expressions. Will force unification of unbound variables.

(> <atom1> <atom2>)

Tests whether <atom1> is greater than <atom2>. If both arguments are numbers, a numeric comparison will be done; if both arguments are symbolic, a lexical comparison will take place. A number is always less than a symbol.

(>= <atom1> <atom2>)

Tests whether <atom1> is greater than or equal to <atom2>.

(< <atom1> <atom2>)

Tests whether <atom1> is less than <atom2>.

(<= <atom1> <atom2>)

Tests whether <atom1> is less than or equal to <atom2>.

The arithmetic functions are:

(+ n1 n2 ...)

The sum of the numbers n1 n2 ... is returned.

(- n1)

(* n1 n2 ...)

The product of the numbers n1 n2 ... is returned.

(/ n1 n2)

n1 divided by n2 is returned.

(abs n1)

The absolute value of n1 is returned.

(exp n1)

e to the n1th power is returned.

(ln n1)

The natural logarithm of n1 is returned.

(sqrt n1)

The square root of n1 is returned.

(trunc n1)

The integal portion of n1 is returned.

All trigonometric functions deal with arguments or answers specified in radians.

(cos angle)

The cosine of an angle specified in radians is calculated and the value returned.

(sin angle)

The sine of an angle specified in radians is calculated and the value returned.

(tan angle)

The tangent of an angle specified in radians is calculated and the value returned.

(arctan x)

The arctangent of x is computed and the value in radians is returned.

User Modifications to the Interpreter

Occasionally, the user may wish to extend the power of the HC interpreter directly by compiling a function to perform a special task. The arithmetic functions were implemented by adding user-defined functions, for example. This section describes how such a user-defined function can be added to the prover. The function must be programmed in PASCAL, and a separate compilation of the new source-listing must be made. This section may be skipped until needed.

Suppose that a function "NEW" has been programmed, the following rules must be followed:

1. The string FNEW must be added to the structure definitions at the top of the file (end of first page of code), in particular, the string must be added along with all other Ffunctions to the fmode list.

2. The actual PASCAL code should be added to the code at the position where all the other Ffunctions are defined (well down in the listing). The function call should look like this:

```
FUNCTION F_NEW(ARGS: LIST):LIST;
```

where the first LIST is the argument list and the second LIST is the output list. The underscore must be included.

3. Inside of F_EVAL() a new case including "FNEW" must be added.

4. In the GLOBINIT() function at the end of the code a new call must be added:

```
MK_FUN('new\0              ', FNEW);
```

The spacing depends on the variable IDSIZ which controls the length of strings that the interpreter can accept. The current value of IDSIZ is sixteen so thirteen blanks follow new\0. The \0 is used as a terminator.

5. If NEW is an FEXPR, that is, if it does not evaluate its arguments, then FNEW should be added to the set of FEXPRs in GLOBINIT().

6. The version number in the constant section at the beginning of the program, as well as the version

comments should be updated to keep documentation informative.

In general, if these instructions seem unclear, an examination of the code of the prover will clarify any questions. The F_functions provide good examples of code similar to what the user is programming and may be examined as models of code that correctly interface with the HC interpreter.

User-written PASCAL code should only refer to local variables; references to global variables will not succeed. In general, PASCAL uses assignment to give variables a value, assigned values corresponding to a memory location. Unification is not compatible with the assignment operator of algebraic or even functional programming-languages (LISP). Reference to variables bound in HC under unification from a PASCAL routine, therefore, does not work.

The programmer must be sensitive to returning values from the newly defined function. Failure is signified by returning a nil value. Anything else will cause the predicate in the HC rule that called the function to succeed.

If the user wishes to add an input/output function additional care must be taken due to local system-interface difficulties. The source-code listing should be examined at the point toward the top where system-dependent routines are specified.

All user input is converted to lower case automatically by the prover, so if the programmer desires to have a variable protected from the HC user, it should be coded in upper case.

The user may also wish to extend the maximum string-length accepted by the prover. The following modifications are necessary:

1. the variable IDSIZ must be set to the desired length plus one. IDSIZ can be found on the first page of code in the constants-definition area.
2. the GLOBINIT procedure at the end of the file must be modified so that each string has the correct number of padding blanks after it (PASCAL does not allow variable-length strings). For example, "t" in the current prover is declared as a string in GLOBINIT by typing a t followed by \0 (terminator) followed by fifteen blanks.

These changes will allow the interpreter to accept strings between quote marks up to IDSIZ - 1 characters.

CHAPTER 3

Program Development

This chapter describes the actual method of using the HC theorem-prover, including how to call the prover, how to load files, the internal working environment, and methods of debugging programs. Practical details and advice are included to facilitate program development. HC interprets programs, so the user will find that programming can actually be accomplished interactively as opposed to programming with compiled languages, which essentially amounts to batch job submissions. Normally, programs are more easily debugged in such an interactive environment.

Calling HC and Loading Files

The following information pertains to VAX/VMS on which the HC implementation of PROLOG is available. The access path to the interpreter on VAX3 is given by dba0:[roachjw]hc.exe and can be run by typing r dba0:[roachjw]hc.exe. This gets rather tedious after a while, and adding the following line to your login.com file will prove helpful:

```
hc :== $dba0:[roachjw]hc
```

After the login.com file has been executed, the user may simply type hc to invoke the prover. Often the user will wish to load a file when calling the interpreter. This is achieved by simply typing

```
hc vax_file_pathname
```

If the vax_file_pathname does not lead to a file that can be loaded, the interpreter will return nil and wait at top level of the interpreter for user commands. The user may also wish to set flags (explained below) on entry to HC. This is easily accomplished by typing

```
hc -ta vax_file_pathname
```

where -ta sets the tflag and aflag on.

Once inside the prover, the user may wish to read in a file from a directory. This can be achieved by using the load command: for example, (load "vax_path_name") will read in any file specified by the path name. The path name specification is a string and, in accordance with string limitations imposed by the interpreter, must be sixteen characters or less in length. The value returned by load command depends on the last evaluated expression. For a file with a simple assert command, t is returned. If the path name does not lead to a file, nil is returned. The load function is callable from within HC by including it in a rule like any normal predicate. The following is therefore possible:

```
( (moreaxioms *file) if (readvar *file) (load "*file") )
```

a rule which would first read in the path to a file from the terminal and then attempt to load a file using the path name unified to *file.

The command-level prompt is *. A (QUIT) or command exits the prover and returns control to VAX/VMS. The parentheses for the (QUIT) command are necessary. All built-in prover functions, such as arithmetic operations, are computable from the top level. Names of some other such functions include: variable, assert, if, trace, and untrace. A complete list of system functions can be printed on the terminal by typing (stat f).

The Programming Environment

Once inside the interpreter, the user will want to perform certain actions interactively that make program development easier: for example, printing a list of all rules with the "solves" relation on the left hand side, inserting a break point in a rule, and tracing the execution of a program. This section describes how to use such amenities plus some other features to help the user.

Printing a list of all axioms with a particular relation on the left hand side of the rule is easy; the user need only type the name of relation followed by a return. For example, in the quick-sort program shown in detail below, typing "qsort" after loading the file will result in,

```
(((qsort nil nil))
 ((qsort (x . y) z) if (partition x y u v)
                       (qsort u u1)
                       (qsort v v1)
                       (concatenate u1 (x . v1) z)))
```

(although it will not print so neatly). The user can perform this function at command level (in response to the * prompt), at a break point, or in reponse to the continue? prompt.

A number of options controlled by flags can be invoked by the user. The most useful flags are the eflag, tflag, aflag, and vflag. Together with the break function and limit variable described below, these options allow considerable debugging help. Flags can be enabled in two different ways: on the VAX/VMS command-line invoking the prover the user can type

```
hc -eta
```

which automatically enables the eflag, tflag, and aflag. Also, by typing (= <flagname> t) from within the prover, the user can enable the flag. Any flag can be disabled by typing (= <flagname> nil).

The eflag, or echo flag, can help the programmer find an unbalanced parenthesis mistake, one of the most common errors. For example, if the user tries to read in a file and the error message "?Input list munged" is printed (sometimes the message, " ?"argument" is non-atomic " is printed), then a mistake in balancing parentheses has been made. By trying "hc -e file.spec" the prover spots such parenthesis errors most of the time. With the eflag enabled, the system shows the file as it is being loaded and stops at the point where it can no longer continue. Sometimes, the no continuation point pinpoints the mistake;

often, the error occurs close to the stopping point. There are parenthesis errors, however, for which the eflag cannot help, usually when a right parenthesis is omitted at the end of a rule.

The tflag, or trace flag, allows program execution to be monitored. The sequence of goals solved and the answers computed are printed as the program executes. As each new goal enters the goal stack, (nextto robot box1), say, the system prints "(Q-n) (nextto robot box1)" where n is an integer. If (nextto robot box1) matches the left hand side of some rule, then the right hand side of that rule replaces (nextto robot box1) on the goal stack as explained in the control section earlier. When the prover next tries to solve the goal at the front of the stack (generated from the right-hand side of the rule matching (nextto robot box1)), it prints "(Q-n+1) goal"; n, therefore, can be seen as a counter that shows which level the prover is trying to solve a problem. Until an answer is returned to level n, all goals will have a higher number than n and will be working to solve the n level goal, either directly or indirectly. When an answer is found, the system returns it and prints "(R-m) answer" where m is an integer that can be paired with an earlier "(Q-m) goal" printout. Thus, the user can, with patience, follow the sequence of goals and answers that the system computes to solve a problem. If a program contains a bug, it can be traced down using the tflag feature. As an example, consider now the following natural-language parsing-program:

```
;
; syntax parser using lists
```

```
;
; grammar
;
(var p q r u v w x y z x1)
(assert
        ((nf (x y z . w) ((det x) (adj y) (noun z)) w) ->
                (det x) (adj y) (noun z))
        ((nf (x y . z) ((det x) (noun y)) z) ->
                (det x) (noun y))
        ((nf (x y . z) ((adj x) (noun y)) z) ->
                (adj x) (noun y))
        ((nf (x . y) ((noun x)) y) ->
                (noun x))
        ((vp (x . y) ((verb x)) y) ->
                (verb x))
        ((pp (x . y) ((prep x) (np p)) z) ->
                (prep x) (nf y p z))
        ((np x ((nf p) (con w) (nf q)) z) ->
                (nf x p (w . y)) (con w) (nf y q z))
        ((np x ((nf p)) z) ->
                (nf x p z))
        ((s x ((np p) (vp q))) ->
                (np x p y) (vp y q nil))
        ((s x ((np p) (vp q) (pp r))) ->
                (np x p y) (vp y q z) (pp z r nil))
        ((s x ((np p) (vp q) (np r))) ->
                (np x p y) (vp y q z) (np z r nil))
        ((s x ((pp p) (np q) (vp r) (np x1))) ->
                (pp x p y) (np y q z) (vp z r v) (np v x1 nil))
        ((s x ((np p) (pp q) (vp r) (np x1) (pp u))) ->
                (np x p y) (pp y q z) (vp z r v)
                        (np v x1 w) (pp w u nil))
)
;
; dictionary
;
(assert
        ((noun alcohol))
        ((noun cat))
        ((noun fuel))
        ((noun it))
        ((noun mat))
        ((noun oxygen))
        ((noun rocket))
        ((noun tons))

        ((det a))
        ((det the))

        ((adj eight))
        ((adj giant))
        ((adj liquid))
        ((adj red))
```

```
        ((prep for))
        ((prep of))
        ((prep on))

        ((verb carried))
        ((verb is))
        ((verb rose))
        ((verb was))

        ((con and))
        ((con or))
)
;
; sentences to be parsed
;
(= p1 (s (the cat is on the mat) x))
(= p2 (s (the giant rocket rose) x))
(= p3 (s (for fuel it carried alcohol and liquid oxygen) x))
```

If the user types (= tflag t) and (try p1), the following trace will result:

```
*(= tflag t)
t
*(try p1)
(Q-1) (s (the cat is on the mat) x)
(Q-2) (np (the cat is on the mat) p y)
(Q-3) (nf (the cat is on the mat) p (w . y))
(Q-4) (det the)
(R-4) (det the)
(Q-4) (adj cat)
(R-4) nil
(Q-4) (det the)
(R-4) (det the)
(Q-4) (noun cat)
(R-4) (noun cat)
(R-3) (nf (the cat is on the mat) ((det the) (noun cat))
        (is on the mat))
(Q-3) (con is)
(R-3) nil
(Q-3) (nf (the cat is on the mat) p (w . y))
(Q-4) (adj the)
(R-4) nil
(Q-4) (noun the)
(R-4) nil
(R-3) nil
(Q-3) (nf (the cat is on the mat) p y)
(Q-4) (det the)
(R-4) (det the)
(Q-4) (adj cat)
(R-4) nil
```

```
(Q-4) (det the)
(R-4) (det the)
(Q-4) (noun cat)
(R-4) (noun cat)
(R-3) (nf (the cat is on the mat) ((det the) (noun cat))
        (is on the mat))
(R-2) (np (the cat is on the mat)
        ((nf ((det the) (noun cat))))
        (is on the mat))
(Q-2) (vp (is on the mat) q nil)
(R-2) nil
(Q-2) (np (the cat is on the mat) p y)
(Q-3) (nf (the cat is on the mat) p (w . y))
(Q-4) (det the)
(R-4) (det the)
(Q-4) (adj cat)
(R-4) nil
(Q-4) (det the)
(R-4) (det the)
(Q-4) (noun cat)
(R-4) (noun cat)
(R-3) (nf (the cat is on the mat) ((det the) (noun cat))
        (is on the mat))
(Q-3) (con is)
(R-3) nil
(Q-3) (nf (the cat is on the mat) p (w . y))
(Q-4) (adj the)
(R-4) nil
(Q-4) (noun the)
(R-4) nil
(R-3) nil
(Q-3) (nf (the cat is on the mat) p y)
(Q-4) (det the)
(R-4) (det the)
(Q-4) (adj cat)
(R-4) nil
(Q-4) (det the)
(R-4) (det the)
(Q-4) (noun cat)
(R-4) (noun cat)
(R-3) (nf (the cat is on the mat) ((det the) (noun cat))
        (is on the mat))
(R-2) (np (the cat is on the mat)
          ((nf ((det the) (noun cat))))
        (is on the mat))
(Q-2) (vp (is on the mat) q z)
(Q-3) (verb is)
(R-3) (verb is)
(R-2) (vp (is on the mat) ((verb is)) (on the mat))
(Q-2) (pp (on the mat) r nil)
(Q-3) (prep on)
(R-3) (prep on)
(Q-3) (nf (the mat) p nil)
(Q-4) (det the)
```

```
(R-4) (det the)
(Q-4) (noun mat)
(R-4) (noun mat)
(R-3) (nf (the mat) ((det the) (noun mat)) nil)
(R-2) (pp (on the mat)
          ((prep on) (np ((det the) (noun mat)))) nil)
(R-1) (s (the cat is on the mat)
         ((np ((nf ((det the) (noun cat)))))
          (vp ((verb is)))
          (pp ((prep on) (np ((det the) (noun mat)))))))
(s (the cat is on the mat)
   ((np ((nf ((det the) (noun cat)))))
    (vp ((verb is)))
    (pp ((prep on) (np ((det the) (noun mat)))))))
*
```

This trace shows the entire derivation path for this particular parse of the sentence.

The trace can be modified to give more information by enabling the aflag, or axiom flag. When the user enables aflag, the axioms being used in the computation are printed. The tflag and aflag together constitute a very useful tool for program debugging since the user can determine exactly what the program is doing. As a short example, consider the first part of the trace above with the aflag enabled:

```
(setq tflag t)
t
*(setq aflag t)
t
*(try p1)
(Q-1) (s (the cat is on the mat) x)
(X-1) ((s (the cat is on the mat) ((np p) (vp q))) ->
        (np (the cat is on the mat) p y) (vp y q nil))
(Q-2) (np (the cat is on the mat) p y)
(X-2) ((np (the cat is on the mat)
           ((nf p) (con w) (nf q)) y) ->
        (nf (the cat is on the mat) p (w . y))
        (con w) (nf y q y))
(Q-3) (nf (the cat is on the mat) p (w . y))
(X-3) ((nf (the cat is on the mat)
       ((det the) (adj cat) (noun is)) (on the mat)) ->
          (det the) (adj cat) (noun is))
```

```
(Q-4) (det the)
(X-4) ((det the))
(R-4) (det the)
(Q-4) (adj cat)
(R-4) nil
(X-3) ((nf (the cat is on the mat)
           ((det the) (noun cat)) (is on the mat)) ->
                (det the) (noun cat))
(Q-4) (det the)
(X-4) ((det the))
(R-4) (det the)
(Q-4) (noun cat)
(X-4) ((noun cat))
(R-4) (noun cat)
(R-3) (nf (the cat is on the mat)
          ((det the) (noun cat)) (is on the mat))
 and so on
```

These examples show that a full trace produces a considerable amount of output. Although a programmer may need considerable time to follow such a trace, such a person can obtain a complete understanding of a program and the backup behavior of the interpreter.

The xflag in conjunction with the TRACE function allows the user to turn on a selective trace of the rules being applied. If the xflag is enabled and the user has typed (trace np), then (try p1) will print out each np axiom as it is tried. This allows the programmer to concentrate on the execution of a group of rules that may contain an error. The selective trace can be turned off by calling (untrace np).

The vflag is normally used for batch submission jobs. When the vflag (verify flag) is disabled, and an error occurs during processing, the prover exits without any attempt to recover. Other flags, such as the uflag, nflag, and dflag, are debugging

tools for the interpreter builders; they are not especially recommended for casual users.

The break function can also assist the programmer to debug HC code. On the right-hand side of any rule, the programmer can introduce the relation (break arg) where arg is some prompt character that will print on the screen when the interpreter hits the break. Break effectively stops the interpreter in mid-execution. The user can query the unification value of any variable in the rule that contains the break by typing the name of the variable. If that variable has a value, it will be printed. If the variable has no unification value, the name of the variable will be echoed. The user can force a variable to take a value by calling the forcevar function: (forcevar x (the cat is on the giant mat)) will cause x to be unified with the list (the cat is on the giant mat), for example. The user may wish to initiate a new computation by calling the HC interpreter from the break to determine some intermediate answer. The interpreter does not continue from a break unless the programmer releases it. Computation can continue only if the programmer types t in response to the break prompt; if nil is typed instead of t, the rule containing the BREAK fails.

Another interactive feature is the stat function. By calling (stat v), for example, the user can determine all variables that have been declared. One of the three most common programming mistakes involves forgetting to declare a variable

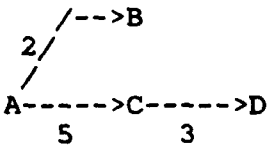
and then using the "variable" in a rule. Of course, such a "variable" is really a symbolic constant and will not match according to the programmer's expectations. The (stat v) help function can assist the programmer to find this kind of error. The stat function has some other arguments. For example, (stat f) will list all *prover-function names* and predicates that are available to the programmer. If (stat g) is enabled, a garbage collection is forced and information about the amount of available space is printed.

The Most Dastardly HC Bug: Infinite Looping

Programs with infinite loops can be easily created. A bug known as "left recursion" often crops up in all kinds of programs. To help the programmer, a limit on the number of rules that can be applied before the system automatically breaks has been instituted. The default limit is set to 100 rules. Essentially, if 100 rules have been applied, the interpreter suspends execution awaiting a command from the user. When this automatic break occurs, the user is prompted with "continue?". If the user types a t and returns, computation will resume; if the user types nil and returns, the interpreter will return to the top level. If something other than t or nil is typed, the interpreter tries to evaluate it. If the user types a procedure call, the procedure is executed; if the user types a variable name, its value is printed; if the user sets the tflag, trace is enabled; rules can be added with the ASSERT function or deleted with the DELETE function. The continue? break allows the user to take stock of the situation and to decide whether an infinite loop has occurred. Most computations inherently require many rule applications so if the continue? prompt appears, there is usually no cause for alarm. The limit can be changed two ways: inside the prover (including at any break point), the user can type (= limit 350), for example; a different method allows the user to change limit when the interpreter is invoked. For example, hc -ta -l:350 parse.x will call HC, enable the tflag and aflag, set the limit to three hundred fifty, and load the parse.x

file. Basically, the limit variable has been included to keep programmers from sitting in front of terminals wondering whether the system is slow or whether left recursion has struck again.

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))  )
)
(variable ans mid point1 point2 x y)
```

but (if (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 HC does not know that. HC 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 HC is hung in a loop. If this process is unclear, simply type the rules into HC, turn on the tflag and aflag and watch this problem run. 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 give the user a really good idea of the problems inherent in left recursive looping, so give it a try.

CHAPTER 4

Conclusions

This report has described a computational method that is new in the sense that few programmers have ever written code this way and old in the sense that computer scientists have classically known that rule-based programming, if correctly implemented, can compute any computable function. Interest in rule-based programming has increased because of the simplicity, elegance, and ease of programming symbolic-manipulation problems in this style. Unlike conventional languages, assignment, global variables, and explicit control-statements are absent. Instead, certain features such as storage management (garbage collection, for example) and an automatic tree-search algorithm are supplied. Whether these unusual features will find favor depends in part on whether difficult programs can easily be developed. PROLOG and other rule-based languages have an additional feature, built-in pattern matching, that allows a very natural programming style. Programs written in rule-based languages thus tend to be much more concise and appear more like the definition of a solution to a programming problem. Advocates of this programming style and of cognitive psychology (see Newell and Simon [10]) contend that rule-based programming is very similar to human thought processes; we may surmise, therefore, that programming with rules should be considerably easier.

Looking at PROLOG programming in a different way, database and knowledgebase programs are easily created because of the relational nature of program statement and data definition. The great power of rule-based languages derives in part from their relational nature. Representing large amounts of information is simplified in relational languages and thus symbol manipulation problems are more easily programmed. To program relationallly means that control states, what gets executed next, is less important than determining the information states that a program must go through to compute an answer. Thinking in terms of information transformations revolutionizes prior concepts of programming. Whether people will find programming in terms of information states rather than control more congenial can only be tested with time.

In the final analysis, programming in PROLOG means adjusting to a different, non-algorithmic way of thinking about computation. New programmers will not have any more difficulty with learning this style of thinking than with learning conventional programming-languages. Old programmers on the other hand may have trouble adjusting to such a different mode of program creation.

APPENDIX

HC Syntax

In this section we define a formal syntax for HC using Backus Naur Form (BNF). The rules given here are slightly modified from a text written by Daniel Chester [3] for his version of PROLOG, HCPRVR. The syntax for the language defined in this paper are given by,

```
<term> ::= <constant> | <variable> | ( <list of terms> ) |
           <string> | * | ?
<constant> ::= <literal atom>
<variable> ::= <literal atom> | *<literal atom>
<string> ::= "<constant>" | "<variable>"
<list of terms> ::= | <term> | <term><list of terms> |
                    <term> . <term>

<atomic formula> ::= ( <predicate name><list of terms> ) |
                     *
<predicate name> ::= <constant>

<rule> ::= ( <fact> ) |
           ( <conclusion> if <list of premises> )
<fact> ::= <atomic formula>
<conclusion> ::= <atomic formula>
<list of premises> ::= <premise> |
                       <premise><list of premises>
<premise> ::= | <variable> | <atomic formula> | /
```

A literal atom consists of as many as sixteen characters. If the first character is numeric, then the literal atom is considered to be a constant; if the first character is alphabetic, then the literal atom can be either a constant or a variable. The * wild card matches anything between parentheses, variables, or constants, and the ? wild card will match any constant or variable. The BNF for <axiom> is a bit imprecise in that almost any literal atom including "if" will work. For example,

((reverse (x . y) z w) -> (reverse y z (x . w)))

is perfectly legal.

References

1. Backus, J. W. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM, 1978, 21 (8), 613-641.

2. Bowen, K., & Kowalski, R. Amalgamating language and metalanguage and logic programming, Tech Rep. Syracuse, NY: Syracuse University, June 1981.

3. Chester, D. Using HCPRVR Int Rep. Austin, TX: University of Texas at Austin, June 1980.

4. Clocksin, W., & Mellish, C. Programming in PROLOG. New York: Springer-Verlag, 1981.

5. Coelho, H., Cotta, J., & Pereira, L., How to solve it with PROLOG, 2nd. ed. Lisboa: Laboratorio Nacional de Engenharia Civil, 1980.

6. Floyd, R. Non-deterministic algorithms. JACM, 1967, 14 (4), 636-644.

7. Gallaire, H., & Minker, J. Logic and data bases. New York: Plenum Press, 1978.

8. Gödel, K. Über Formal Unentscheidbare Sätze Der Principia Mathematica Und Verwandter Systems. Monatshefte der Mathematik und Physik, 1931, 38, 173-198.

9. Kowalski, R. Logic for problem solving. New York: North Holland, 1979.

10. Newell, A., & Simon, H. Human problem solving. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

11. Post, E. Formal reductions of the general combinatorial decision problem. American Journal of Mathematics, 1943, 65, 197-268.

12. Robinson, J. A. A machine-oriented logic based on the resolution principle. JACM, 1965, 12, 23-41.

13. Shortliffe, E. Computer-based medical consultations: MYCIN. New York: Elsevier, 1976.

14. Stefik, M. Planning with constraints. Art. Int. Journal, 1981, 16, (2), 111-169.

15. Waltz, D. Understanding line drawings of scenes with shadows. In Patrick A. Winston (ed.). The psychology of computer vision. New York: McGraw-Hill, 1975.

16. Weyrauch, R. Prolegomena to a theory of mechanized formal reasoning. Art. Int. Journal, 1980, 13, (1, 2), 133-170.

17. Winston, P., & Horn, B. LISP. Reading, Mass: Addison-Wesley, 1980.

442:MAT:716:maf
81u442-390

OFFICE OF NAVAL RESEARCH

Code 442

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CDR Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Department of the Navy

Leader
Engineering Psychology Programs
Code 442
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Leader
Communication & Computer Technology
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Leader
Manpower, Personnel and Training
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. A. Meyrowitz
Information Systems Program
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Commanding Officer
ONR Eastern/Central Regional Office
ATTN: Dr. J. Lester
Barnes Bldg.
495 Summer Street
Boston, MA 02210

Commanding Officer
ONR Branch Office
ATTN: Dr. C. Davis
1030 East Green Street
Pasadena, CA 91106

Commanding Officer
ONR Western Regional Office
ATTN: Dr. E. Gayle
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco, CA 96503

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. L. Chmura
Code 7503
Naval Research Laboratory
Washington, D.C. 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

442:MAT:716:maf
81u442-390

Department of the Navy

Dr. Jerry C. Lamb
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N215
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code N-OOT
Orlando, FL 32813

Dr. R. Neetz
Code 1226
Naval Missile Test Center
Pt. Mugu, CA 93042

Dr. Albert Colella
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps.
Code RD-1
Washington, D.C. 20380

Dr. Thomas McAndrew
Code 32
NUSC-New London
New London, CT 06320

Department of the Navy

HQS, U.S. Marine Corps.
ATTN: CCA40 (MAJOR Pennell)
Washington, D.C. 20380

Commanding Officer
MCTSSA
Marine Corps. Base
Camp Pendleton, CA 92055

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design,
NAVAIR 5313
Washington, D.C. 20361

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 4701
Washington, D.C. 20360

CAPT Darrell D. Dempster, SC, USN (Ret)
System Management American Corporation
1745 Jefferson Davis Highway
Arlington, VA 22202

Dr. Mel C. Moy
Code 302
NPRDC
San Diego, CA 92152

Mr. Ramon L. Hershman
Code 302
NPRDC
San Diego, CA 92152

Navy Personnel Research and Development Center
Planning & Appraisal
Code 04
San Diego, CA 92152

442:MAT:716:maf
81u442-390

Department of the Navy

Navy Personnel Research and
Development Center
Management Systems, Code 303
San Diego, CA 92152

Navy Personnel Research and
Development Center
Performance Measurement &
Enhancement
Code 309
San Diego, CA 92152

LCDR Stephen D. Harris
Code 6021
Naval Air Development Center
Warminster, PA 18974

Dr. Julie Hopson
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Mr. John Impagliazzo
Code 101
NUSC - Newport
Newport, RI 02840

Walter P. Warner
Code K02
Strategic Systems Dept.
Naval Surface Weapons Center
Dahlgren, VA 22448

Dr. Thomas Fitzgerald
Code 101
NUSC - Newport
Newport, RI 02840

Department of the Air Force

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Department of the Air Force

Air University Library
Maxwell Air Force Base, AL 36112

Foreign Addresses

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LJ
ENGLAND

Dr. Kenneth Gardner
Applied Psychology Unit
Admiralty Marine Technology
Establishment
Teddington, Middlesex TW11 OLN
ENGLAND

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Dr. A. D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, CB2 2EF
ENGLAND

Professor B. Shackel
Department of Human Sciences
University of Technology
Loughborough, LEICS. LE11 3TU
ENGLAND

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314

Dr. Craig Fields
Director, Cybernetics Technology
Office
Defense Advanced Research Projects
Agency
1400 Wilson Blvd.
Arlington, VA 22209


DAT
FILM