LOGIC PROGRAMMING IN APL

Robert Jernigan
President
Decision Resource Systems
5595 Vantage Point Road
Columbia, MD, 21044, USA

ABSTRACT

The programming of an expert system requires a language for specifying the rules that an expert uses, a data base for storing his knowledge, and a suitable interactive system. Logic programming has been described as a way of implementing expert systems. With logic programming, rules are expressed as assertions of what is true when certain conditions are true. To be true, the assertion has to be based upon fact or upon an inference derived from facts. This paper describes an implementation of a PROLOG-like language in APL. The intent is to achieve logic programming capability while retaining the full facility of APL. PROLOG is both an extension of LISP, thereby satisfying the needs of the Artificial Intelligence community, and a language for relational data bases. This implementation leans toward the relational data base approach.

INTRODUCTION

The programming of an expert system requires a language for specifying the rules that an expert uses, a data base for storing his knowledge, and a suitable interactive system. Logic programming has been described as a way of implementing expert systems [1]. With logic programming, rules are expressed as assertions of what is true when certain conditions are true. To be true, the assertion has to be based upon fact or upon an inference derived from facts. The following example demonstrates an inference based on two facts. 2500 years ago Aristotle used a similar example to demonstrate human reasoning (logic) [2].

```
FACT:     Dice are cubic objects.
FACT:     I am holding some dice.
Inference:  I am holding some cubic
            objects.
```

This simple inference demonstrates the aim of logic programming systems. Such systems must be capable of maintaining facts and the rules for making inferences. The programming task associated with the example above involves both the placement of the two facts in the knowledge base and the statement of the rule that specifies how the inference (derived fact) is to be made. A rule for the example could be "I am holding B if it is true that I am holding A and A is B." It must also be capable of verifying assertions of the form "I am holding some cubic objects." A language for logic programming described by Kowalski [3] is gaining popularity today. That language, PROLOG, has been selected as the language for the inference engine in the Japanese supercomputer project [7].

This paper describes an implementation of a PROLOG-like language in APL. The intent is to achieve logic programming capability while retaining the full facility of APL. PROLOG was selected as the model or base from which to proceed. PROLOG is both an extention of LISP, thereby satisfying the needs of the Artificial Intelligence community, and a language for relational data bases. This implementation leans toward the relational data base approach. It has been built on top of a relational data management and modeling system [4,5] that has seen years of successful service. Existing public and private libraries are also readily incorporated into the APL/PROLOG environment.

© 1984 ACM 0-89791-137-7/84/006/0225 $00.75

APL

The discussion of APL herein is concerned mainly with APL data structures and with control of execution. APL has been used to implement some sophisticated and powerful systems. Most of these systems rely upon the execution sequencing mechanisms intrinsic in the APL system. Order of execution of APL statements is basically as follows:

- expressions are executed from right to left;
- lines of functions are executed from top to bottom;
- the branch operator may be used to direct execution sequencing to other than the next line;
- called functions are pushed on top of the execution stack;
- statements may be stored in strings and executed following some selection process, perhaps a Dykstra-like "guard" statement.

The principal APL data structure is the array, which may have two types: numeric or character. Using the capabilities inherent in the language the user can easily construct higher order structures such as nested arrays, arrays with mixed types, relations, and list structures. Packages of APL functions that implement these structures are used to provide the user with specialized capabilities. One such package, APLDOT, developed by the author and Alan Eddy in 1976-77, provides for a high-level language that was originally used for financial and strategic modelling during the northeastern United States rail crisis of the 1970's [4].

APLDOT

APL *D*ata *O*rganization *T*echnique (APLDOT) implements structures such as simplified nested arrays, mixed data types, and relations. APLDOT is a high-level language extension of APL that facilitates implementation of mathematical models. Formulas describing a problem are stored in relational data sets, which can be accessed by the report writers and editors. The relational organization, automatic control of execution, and lack of assignment statements provide a natural base for implementing a language like PROLOG. Order of execution in APLDOT is controlled entirely by a reference function, which forces evaluation of a formula only when its result is needed by another formula or a report generator.

APLDOT includes:

- functions implementing a relational organization of APL variables and functions;
- a simple but powerful relational calculus;
- a direct-effect retrieval function, called the reference function, that has as its value the value of the referent objects;
- context control and switching calculus within the relational calculus;
- a dynamic and accessible data dependency tree;
- incorporation of formulas within the data base in a form similar to Iverson's direct-definition notation;
- automatic control of execution sequence of the formulas of a model;
- report generators;
- a powerful function and data editor. (This document was produced using the editor).

APL/PROLOG

PROLOG statements, also known as rules or Horn clauses, contain a HEAD and a TAIL separated by the "⊃" symbol. For example,

$$\text{HEAD}\supset\text{TAIL} \qquad (1)$$

is a PROLOG statement that may be read "HEAD is true if TAIL is true." The statement may also contain constants and variables, which are enclosed in parentheses. Variables are distinguished by the use of the "_" symbol as the first character of the name.

Horn clauses contain basic units that are evaluated and are either true or false. These units, called GOALs, have a name, or predicate, and a parameter list, which may be null. The head goal contains the assertion; the tail contains the conditions that must be met for the assertion in the head to be true. Goals in the tail are connected by logical AND (∧) and OR (∨) operators. Execution of the tail proceeds from left to right and parentheses may be used to establish subclauses, which act as a single goal in the clause. Any variables that occur within a clause are bound to the clause itself and are not available outside the clause. Variables and constants within the parameter list are separated by the semicolon, ";".

```
    FATHER(JOHN;JOE)⊃
       SON(JOE;MARY)∧SPOUSE(JOHN;MARY) (2)
```

is an assertion that may be read as follows, "John is the father of Joe if Joe is the son of Mary and John is Mary's spouse." All of the names in the above clause are constants. "SON", which predicates the relation Joe has to Mary, must be true if the clause is to be true.

If the rule were stated in a general form it would appear as

```
      FATHER(_A;_B)⊃
         SON(_B;_C)∧SPOUSE(_A;_C). (3)
```

Execution of this clause would find all cases in the knowledge base that satisfy the rule. There are two variables associated with the HEAD, "_A" and "_B". During execution these variables would have to be set to some value if the statement were to be true. Failure to do so would cause execution to be false, which it would be if the knowledge base could not satisfy the conditions specified in the TAIL. Notice also that "_C" is a variable that is local to the TAIL and will be discarded after execution of the TAIL.

The above clause may be read as "A" is the father of "B" if there is a "C" such that "B" is the son of "C" and "A" is the spouse of "C". Of course, it may be the case that the knowledge base contains the fact that "B" is the son of "A". The following restatement of the clause would attempt to make that determination first.

```
FATHER(_A;_B)                              (4)
    ⊃MALE(_A) ∧ (SON(_B;_A)
    ∨SON(_B;_C)
    ∧(SPOUSE(_A;_C)∨SPOUSE(_C;_A))
    ∧(ASSERT(SON(_B;_A))∨TRUE)).
```

The rule now not only searches the SON relation first but will add (ASSERT) the fact discovered by the subclause containing the SPOUSE reference to the SON relation if that subclause was used to resolve the clause. The use of the AND and OR symbols is consistent with their use in APL. Note that if two subclauses or goals are connected by an OR symbol, the second clause is not executed if the first is true.

Every PROLOG goal or clause returns a boolean scalar result, i.e., it is either true or false. In the example, the use of the TRUE subgoal, which is always true, is used to force the statement to be true if the assert command should fail in its attempt to add a discovered fact to the knowledge base. In this context, that addition is not a necessary action and does not change any discovered facts.

During execution of the TAIL a relational data base is built using the variables in the tail as fields. This relation is temporary and will be discarded when execution of the clause has completed. Before control is returned to the calling goal, any fields mentioned in the HEAD that are fields of the temporary relation will be returned as values for the corresponding variables in the calling goal. It is important to note that this can cause a change in values in the variables of the calling goal. The values of variables in any goal are totally dependent upon any actions taken during execution of the goal.

In the example above "_A" and "_B" would be set to fields from the "SON" relation. "_B" would then serve as a selection qualified for the "SPOUSE" relation. A relational JOIN operation is then performed on the two relations. The TAIL may have numerous goals, each with its associated variables, and automatically causes a JOIN operation over the variables returned from goal execution. Goals mentioned in the TAIL are not restricted to relations defined in the knowledge base, but may be references to other clauses, APL statements, commands, or APLDOT formulas.

INTERACTIONS

PROLOG mode is entered by invoking the "PROLOG" function. This can be done by typing "PROLOG" once the APL/PROLOG workspace is loaded. A vertical bar, "|", is issued as the prompt character, indicating it is ready for the user to enter a statement. There are five types of PROLOG statements:

- commands;
- assertions;
- questions;
- APL statements;
- rules or "Horn clauses".

Fact and rule assertions are terminated by a period, and commands and questions are terminated with a question mark. A statement that has degenerated to garbage due to typing errors may be terminated by a right arrow, "→", which discards current input and issues a fresh prompt. When an input line has been entered without a terminator, it can be continued on the next line so that several lines may be used for a single entry. If any typing errors are made during input, the editor may be entered by typing "EDIT." as the last five characters of a line.

Rules of Syntax.

The following example of a Horn clause is referred to by the rules below:

```
| HEAD(_A;_B) ⊃
|    GOAL1(_A)∧GOAL2(_A;_B)
|    ∨GOAL3(_B;_A).                        (5)
```

1. The name of a goal precedes its arguments.
2. Arguments are enclosed within parentheses.
3. Arguments are separated by semi-colons, ";".
4. Range specifications with goal names are separated with the APL jot character, "∘".
5. Horn clauses use the APL right shoe, "⊃", to separate the head from the tail.
6. Horn clauses are terminated with periods.
7. The APL logical-and symbol, "∧", is used to link two goals when the second is to be executed only if the first succeeds.
8. The APL logical-or symbol, "∨", is used to link two goals when the second is to be executed only if the first fails.
9. Variables use the APL underbar, "_", as the first character.

Commands.

With the exception of the "QUIT" command, all commands are implemented as APL functions. There are only three restrictions placed on the user in writing command functions:

- the command must parse the PROLOG argument list;
- the command must be asserted as fact; Example:
  COMMAND(NEWCMD;THIS IS THE DESCRIPTION).
- the command must return a boolean scalar.

The SHOW command will produce a list of the defined commands:

```
| SHOW(COMMAND)?                          (6)
APL/PROLOG COMMAND TABLE
 NAMES LINE         DESC
====== ==== ==============================
ASSERT [1]  STANDARD PROLOG ASSERT COMMAND
DENY   [2]  INVERSE OF ASSERT-DELETES FACTS
TRUE   [3]  COMMAND THAT IS ALWAYS TRUE
FALSE  [4]  COMMAND THAT IS ALWAYS FALSE
SWITCH [5]  TOGGLE A SWITCH
QUIT   [6]  EXIT PROLOG MONITOR TO APL
PEDIT  [7]  EDIT PROLOG HORN CLAUSES
SHOW   [8]  DISPLAY FUNCTION
```

Assertions.
Either a fact or a rule may be asserted. Fact assertions are the PROLOG way of building a knowledge base. For example:

```
| FATHER(SAM;JOHN).                        (7)
```

asserts that Sam is the father of John. The choice of the word "FATHER" is arbitrary, but makes the statement easier to understand. "FOO" could have been used to assert the same fact, but leads to difficulties in understanding what the assertion means. Since it is not always possible to choose words that convey the meaning of the fact in a single name, a description or title to a set can be given. When a new set of facts is being asserted, PROLOG will ask for a title to further clarify the meaning of a set of facts. In the above example of the SHOW command, the titles are displayed under the DESC heading.

Each set of facts is kept in a relational data base. Associated with the data base are its name, its title or description, and the names of the fields. Field names may be specified along with the data base name. For example,

```
| KINSHIP∘FATHER∘SON(SAM;JOHN).          (8)
```

would assert a fact for a data base named "KINSHIP" that has two fields, "FATHER" and "SON". If no field name is used, PROLOG will default to the use of numbers as field names. Specifying the names in a goal, question, or an assertion establishes the RANGE of fields within the data base over which that goal or assertion will operate.

Questions.

A question contains the name of a goal, a set of parameters, and is terminated by a question mark, "?". For example,

```
| FATHER(_A;JOHN)?                          (9)
```

is a question asking "Who is the father of John?". "_A" is a variable and "JOHN" is a constant.

APL statements.

The APL statement, "APL", is reserved as a special goal type. It is the primary interface to the APL language from PROLOG. A single parameter is required, which must be an APL expression. The expression must conform to APL rules of syntax with one exception - PROLOG variables may be used. If any results of execution of the APL expression are to be returned, they must be assigned to a PROLOG variable within the expression. If any PROLOG variables are to be referenced within the expression, they must have been assigned values by PROLOG before execution of the expression.

Examples:

```
|  APL(_A←1 + 8 10 10 12 ÷100).          (10)
   ...APL(_C←ASK'ENTER NAMES TO APPEAR
       ON REPORT')∧...
|  APL(⎕←HOWEDIT).
|  APL(⎕PW←132).
```

The first statement assigns the vector 1.08 1.10 1.10 1.12 to the variable "_A". The second form, which would be imbedded within a Horn clause, is useful for setting variables to be used by subsequent goals within the clause. The third form is a way of viewing the content of a variable. These three forms are only representative of what may be done. The result of an APL statement is FALSE if the statement fails during execution, if a boolean scalar ZERO (0) obtains, or if the result is null.

Horn clauses.

Horn clauses, which were discussed in detail in the previous section, are used to assert the rules that constitute PROLOG programs. They are executed either as the result of a question or as a referenced goal in another Horn clause. The goal that does the referencing, whether in a question or a clause, is known as the parent goal. If the parameters of the parent goal have values and the values correspond to variables in the head of the clause, the variables are set to the corresponding values in the parent goal. When no values are passed from the parent goal to the head goal of a clause, it is expected that the subsequent execution of the clause tail will result in the setting of the variables of the head. Those values are then passed back to the parent setting corresponding variables in the parent.

When a goal is being processed, a search is made of the knowledge base to find the appropriate data base or Horn clause to use for its evaluation. The search is based both on the name, if specified, of the parent goal and on the number and values of the parameters. It is possible for the search to find several clauses that could be used for evaluation of the parent goal. When this happens, the user has the option of having the first goal automatically selected or of reviewing the goals and manually selecting which is to be used first. Currently, clauses are executed in turn until a result is found that can be passed back to the parent goal.

As clauses are executed, the variables are passed from goal to goal within the clause. Keeping in mind that variables may represent an array of values, goal evaluation may result in the elimination of some of the values. In example 3 above, execution of the goal SON may obtain a set of values for the variable _C that are not in the SPOUSE relation. When this happens, a null result obtains for the clause and the head goal fails. Successful execution results in all values satisfying the rule being returned to the parent.

APLLOGIC

APLLOGIC is the version of PROLOG implemented in APL by the Applied Physics Laboratory. At this writing it is running on an IBM 3033 under APL/MVS. Shortly it will be installed on The APL Machine, which the Laboratory is acquiring from the Analogic Corporation, Wakefield, Massachusetts. With that implementation several changes are planned in the way execution control is managed. The most important of these is concurrent execution of all clauses that are selected during the search process. The array-based relational data base logic of this PROLOG, the reliance upon APL for data structuring functions, and the array processing of the APL Machine should yield a prototype PROLOG machine well suited for implementing expert systems [6].

REFERENCES

1. Clark, K. L. and McCabe, F. G., "PROLOG: a language for implementing expert systems," *MACHINE INTELLIGENCE*, Edinburgh University Press.

2. Aristotle, *LOGIC*, from *THE PHILOSOPHY OF ARISTOTLE*, The New American Library, New York, 1963.

3. Kowalski, R., *LOGIC FOR PROBLEM SOLVING*, New York: Elsevier - North-Holland Publishing Co., 1979.

4. Kruba, Stephan B., "APLDOT: an APL programmer's modeling language," *APL*83 *CONFERENCE PROCEEDINGS*, Wash, D.C., 1983.

5. Eisner, A., Jernigan, R., Yionoulis, S. M., Platt, J. A., "Satellite Reliability Information Management System," paper submitted to APL84.

6. Warren, David, "A view of the Fifth Generation and its impact", *THE AI MAGAZINE*, Vol. 3, No. 4, pp. 34-35, Fall 1982.

7. Moto-oka, T., Editor, *FIFTH GENERATION COMPUTER SYSTEMS*, New York: North-Holland Publishing Co., 1982.