PROLOG PROLOGUE

Upton, Oxfordshire, UNITED KINGDOM - Graham Thwaites

This issue of IQLR sees the launch of GT-Prolog, a new implementation of the Prolog. language widely used within industry and academia for the development of Artificial Intelligence (Al) applications and for symbolic programming generally. Although GT-Prolog isn't the first Prolog system for the QL (that honour goes to Hans Lub's QL Prolog, available from the Quanta library) it does seem that information on Prolog within the QL community is somewhat lacking. This article will attempt to redress that situation.

Prolog has its origins in attempts made in the early 1970's to use mathematical logic as the basis of a new programming language. These efforts were focussed particularly on the universities of Edinburgh and Marseilles (the name Prolog is derived from "Programmation en Logique") and culminated around 1977 in the DEC-10 implementation which first demonstrated the use of compilation techniques to achieve acceptable efficiency. The DEC-10 version also established the conventions of the 'Edinburgh' dialect, for many years the de-facto language standard, and laid the foundations for a development methodology and toolset oriented towards high programmer productivity and based on incremental program evolution. Almost all subsequent Prolog implementations for mainframes, Unix-based workstations and personal computers have followed this same route to the extent that a programmer familiar with one implementation expects to be able to move to any other with relatively little effort.

Within Europe the availability of a practical logic programming language led to the rapid takeup of Prolog, especially for problems of an experimental nature, and the language has featured prominently in advanced R&D programmes sponsored at national and European Community levels. Prolog has also occupied a crucial role in the Japanese Fifth Generation efforts. In the USA, however, Lisp had already established a dominant position in term of both hardware availability and programmer expertise and takeup has been somewhat slower. During the early 1980's there was even something of a language schism within the AI community although this has now been replaced by a greater appreciation of the benefits of each language and a more tolerant acceptance of "horses for courses".

PROLOG PROLOGUE - (CONTD)

For the newcomer to Prolog there are two pleasant surprises in store - the first concerns how simply programs are written in the language itself and the second is the ease in which Prolog programs are constructed using the development environment provided by the GT-Prolog Workbench. It should be stressed that neither of these aspects requires the user to be a logician nor to have any greater level of mathematical ability than is required to use any other programming language.

The Language:

Prolog is based on a small set of concepts which include pattern matching, tree-structured data and backtracking. A program is written as a set of clauses which are stored within the Prolog database. A simple example representing a portion of a family tree covering four generations is given by the following twelve clauses:

```
parent(walter,ken).      parent(edith,ken).
parent(ken,graham).      parent(kath,graham).
parent(ken,glenn).       parent(kath,glenn).
parent(graham,kate).     parent(angela,kate).
female(edith).           female(kath).
female(angela).          female(kate).
```

Each of the clauses for the predicate parent/2 (parent with 2 arguments) represents a relationship between its arguments. Similarly the female/1 clauses identify the female members of the family. In these cases each argument is a symbolic constant (an atom) identifying the person concerned. Prolog also supports numeric values, strings, characters, variables and compound data items which combine the properties of records, lists and arrays.

We can interrogate the database using a query as follows:

```
?- parent(ken,glenn).
```

This causes the database to be scanned for a parent/2 clause whose first first argument matches ken and whose second argument matches glenn. Each of the first four clauses fails but on the fifth attempt we get a response that the goal was proven unconditionally. The user is then asked whether any alternative solutions should be sought but, in this case, there are no more potential matches and the query completes.

We can also issue a query involving a variable (an identifier starting with an uppercase letter):

```
?- parent(kath,Who).
```

This time the first three clauses fail as the first arguments do not match but the fourth attempt succeeds so we get told that the query is successful with Who matched to graham. However, this time the search will also succeed with Who matched to glenn and so on.

We can also interrogate in an inverse fashion:

```
?- parent(Who,kate).
```

This will give alternative solutions with Who matched to graham and angela successively.

The availability of further solutions is achieved by creating a choice point to remember any further clauses after a particular match has been made. Backtracking to a choice point undoes the effects of any successful matches before trying again. We can extend the usefulness of the database by adding some rules to facilitate extra queries:

```
mother(X,Y):-parent(X,Y),female(X).
father(X,Y):-parent(X,Y),not female(X).
grandparent(X,Y):-parent(X,Someone),parent(Someone,Y).
```

The mother /2 clause should be read as “X is the mother of Y if X is a parent of Y and X is female”. Similarly the father /2 clause uses negative information “X is the father ... and X is not female”. Finally grandparent /2 uses a purely local variable Someone to identify a person who is both the child of X and the parent of Y. Again these rules can be invoked with any combination of arguments. Note that the following query to identify all grandmothers will give two solutions for edith since she is a granny twice over:

```
?- grandparent(X,Y),female(X).
```

In this case multiple nested choice points will be created. When all choice points are exhausted the query completes.

Readers who have been involved in the construction of expert systems will recognise that the Prolog rule structure is naturally backward chaining. However, it is also relatively easy to construct forward chaining or mixed strategy systems suitable for constructing blackboard and other knowledge-based architectures and these are well documented in the multitude of textbooks which are now available. Similarly, examples of using Prolog to build effective game playing and problem solving programs are widely and freely available. Many of these provide user interfaces based on natural language processing, a facility that is supported directly by the Prolog grammar notation.

The examples given here only hint at the power of Prolog and there is obviously no space to cover the full range of data types, control constructs, library procedures and so on. For anybody wanting to look at the language and its usage in more detail I would recommend starting with "Prolog - Programming for Artificial Intelligence" by Ivan Bratko (Addison-Wesley), ISBN 0-201-41606-9. Hans Lub also recommends "The Art of Prolog" by Leon Sterling and Ehud Shapiro (MIT Press), ISBN 0-262-19250-0, although I find this book somewhat biased towards a logicians viewpoint as well as having some minor syntactic differences to the Edinburgh standard. Both of these are available in paperback.

The Environment:

Like its Lisp and Smalltalk cousins Prolog has an inherent self-modification ability which allows an executing program to extend its own facilities dynamically. In Prolog's case this is done by asserting new rules and facts into the Prolog database which can then be invoked by appropriate goals. Existing clauses can also be deleted (retracted in Prolog terminology). GT-Prolog has exploited this facility to produce an integrated compiler, debugger, editor and error handler which together provide a highly efficient environment for the rapid design, coding and debugging of complete or partial programs. The compiler can also be used to produce modules of compiled code which can be loaded several times faster than recompiling from source and provides a mechanism for hiding the implementation details of code which is to be distributed to other users.

The debugger is an enhanced version of the ubiquitous 'Byrd Model' originally developed at Edinburgh University which allows the setting of multiple spypoints. When a spypoint is encountered the execution of the running program is interrupted and an interactive dialogue is entered allowing the program to be examined, modified if necessary and continued. GT-Prolog provides six ports at which user interaction is available. If desired execution between breakpoints can be traced and logged to a window, file or printer. If the code being debugged is free of side effects then goals which fail unexpectedly can be retried with additional spypoints or at a greater level of detail. Runaway programs can be halted by a break mechanism.

As the environment is itself written in Prolog its facilities are also available to user programs so that menus, dialogue boxes and multiple windows can all form part of a user interface. QDOS graphics facilities are also accessible via library procedures and all Prolog I/O maps onto QDOS channels. Potentially dire effects of modifying the Workbench code by changing the database are prevented by marking it as static.

QL Limitations:

The only real limit on GT-Prolog is the amount of QL memory available since the inherent implementation limits are to allow up to 64k instances of any particular sort of object (distinct atoms, functors, clauses, I/O streams etc) and up to 16Mb of data space. To put this in context CHAT80, a geographical database program with a sophisticated natural language query interface and which is probably the largest Prolog program in the public domain (over 128kb in source form), comprises 2965 clauses, 514 functors, 1222 atoms and compiles into 77kb of code.

Now available on a QL near you..

GT-Prolog/QL

The easy way to explore the world of Artificial Intelligence and Symbolic Processing

Edinburgh Syntax - as adopted for the International Prolog Standard
Incremental Compiler - provides complete/partial recompilation in situ
Automatic Garbage Collectors - reclaim code and data spaces
Tail-Recursion Optimisation - improves speed and memory usage
First Argument Indexing/Hashing - automatic performance gains
Extended data types - 32bit integers / 48bit reals / 255 byte atoms / 32kb strings / 8bit characters
Functors/Predicates - up to 255 arguments + specially optimised lists
Interactive Workbench (needs 512k)
  * **Windows, Menus, Dialogues** (pointer environment not required)
  * **Editor** - directly coupled to the compiler for faster development
  * **Debugger** - dynamic breakpoints, tracing, database interrogation
  * **Break mechanism** - halt runaway programs, modify and continue
  * **Error handler** - user and system errors handled by catch and throw
User Guide, Reference Manual - over 150 pages of documentation
Full QDOS compatibility - 130+ library procedures
Configurable Memory - up to 16Mb data, 2Gb code
Turbo Performance - Naive Reverse exceeds 5k LIPS (16Mhz Gold Card)

Price including postage/packing £89.95

Grange
Technology
Limited