**PROLOG**

Prolog is a logic language based on a subset of first order logic. Some constructions of first order logic have been removed to allow performance to be competitive, and some extra-logical features (mainly related to flow control, input/output, and meta-programming) have been added. High performance Prolog compilers are available, and integration with other languages is not difficult.

One of the main differences with Logic Programming is the restriction of formulae to a special subclass, called Horn clauses, for which fast resolution procedures are known. Also, the way programs are executed has been fixed by a rule establishing a left-to-right, depth-first search. This has the drawback of being incomplete (there might be correct problem models which do not lead to solutions, but there are always alternative models which do result in the finding of a solution), but in turn allows efficient implementations.

Most CLP systems are built extending Prolog, and their internal machinery is full of implementation techniques developed for Prolog. Thus it is not strange that there is a good deal of programming techniques, built-ins, and miscellaneous facilities which are shared among Prolog and other CLP languages. *Prolog IV* itself can be put in a ``ISO Prolog mode'', in which only Prolog programs are accepted and executed--no constraints are available.

**Language Features**

1. You can declare facts

This makes it simple to build a knowledge base for an application.

1. You can declare rules

This makes it simple to build connections between facts or 'reasoning' into the system.

1. Facts and Rules can be added as the program is running

This makes it possible for the system to 'learn' as it is being used.

1. It has sophisticated processing algorithms

The language will process the knowledgebase looking for patterns or successes and backtracking when it fails.

1. It has list processing capabilities

Although not seen in the adventure game the language can process lists of items, this makes it useful for natural language processing where a sentence may be considered as a list of words.

The main characteristics/notions of the Visual Prolog programming language are:

* based on logical programming with Horn clauses
* fully object oriented
* object predicate values (delegates)
* strongly typed
* algebraic data types
* pattern matching and unification
* controlled non-determinism
* fully integrated fact databases
* supports parametric polymorphism
* automatic memory management
* supports direct linkage with C/C++
* supports direct calling of Win32 API functions

The object system is constructed for loose coupling between a provider and a user.  Objects can only be accessed through interfaces, and interfaces are only loosely coupled to implementations.  Any class can implement any interface with or without inheriting implementation from other classes.

The combination of strong type check, no-need-for-pointer-arithmetic, and automatic memory management practically removed access violations. Visual Prolog has a long reputation for not producing access violations. As one of our good users once said: *That kind of errors is just not an option in Visual Prolog*. We intend to keep it that way and Visual Prolog 7 is no exception from this principle.  Our goal is that you will have to interface to foreign code or insist on making pointer arithmetic to provoke access violations.

Algebraic data types, fact databases and pattern matching combined with non-deterministic search makes Visual Prolog very well suited for dealing with complex structured knowledge.

All Visual Prolog data except predicate values and objects have a human readable textual representation that can be written and read back into programs.

**Graphical Integrated Development Environment (IDE)**

Visual Prolog Integrated Development Environment (IDE)  is designed to make it easy, convenient and fast to develop, test, and modify applications written in Visual Prolog.

It might be especially useful in developing of large projects.

* Tree representation of modules, include files, and resources in the Project window helps to group project items into packages and thus gives an extra level of abstraction.
* The Text Editor supports convenient text editing and browsing to declarations and implementations.
* The Dialog Editor provides standard controls to design dialogs.
* The Menu Editor allows to create both pull-down and pop-up menus.
* The Toolbar Editor allows to create various kinds of toolbars.
* The Graphics Editor is a convenient tool for creating, viewing and editing icons, cursors and small bitmaps.
* The Build Facility supports inserting of necessary packages and include directives.
* The **Browse Facilities** supports search for specific entities, "go to definition" and "go to declaration"

**Compiler**

The Visual Prolog compiler is a successor of the Turbo Prolog compiler created in 1980s that was the first Prolog compiler. Since that time the Prolog Development Center has been developing and improving its compiler. Currently Visual Prolog compiler is a powerful and efficient compiler that can:

* create object files for creating standalone executables or DLLs;
* resolve cross references among declarations;
* validate predicate mode;
* perform powerful type checking;
* validate facts initialization in constructors;
* perform predicate resolution.

**Linker**

Visual Prolog contains a powerful linker that can:

* create EXE and DLL files;
* use LIB files, generated by the latest Microsoft Visual C compilers.

**Debugger**

The Visual Prolog Development Environment contains a built-in graphical debugger that can:

* show usual debugger views: memory, stack, variables;
* show class and object facts with their values;
* perform step into, step over, etc;
* including extra stepping: step out, run to prolog code;
* visualize fail and exit.

**Control Annotation**

Control annotation allows the programmer to have some command on the execution flow of the program. There are three ways in which a given execution path can be forced by the programmer:

* Ordering of goals in a clause.
* Ordering of clauses in a predicate.
* Pruning operators.

We will describe the use of pruning operators later, as they are really additions external to a logical language, while the ordering of goals and clauses stem from decisions regarding mainly performance matters, and which happen to be also useful for controlling the execution flow.

**Goal Ordering**

Execution of goals is always performed left-to-right. This allows the programmer to know the behavior of the program and the order in which variables will be instantiated. It also impacts the performance of the program--precisely by instantiating some variables to some values after or before some points. Consider the following piece of code:

p(a):- <something really big>.

p(b).

q(b).

which is called using these two different queries:

?- p(X), q(X). %% (1)

?- q(X), p(X). %% (2)

(1) will execute the big chunk of code in the first clause of p/1 to fail afterwards, and succeed with the second clause of q/1. (2) will instantiate X to b, and will not even try to execute the first clause of p/1--so the effect is more profound than just reorganizing the order of the clauses of p/1. Moreover, the optimal ordering of goals depends ultimately on the query mode, i.e., the values the variables have at runtime.

**Clause Ordering**

The order of clauses in a predicate determine the order in which alternative branches for a computation are taken. Therefore, in the case of several solutions, it determines the order in which these solutions are returned. Compare the code

p(a).

p(b).

p(c).

with the code

p(b).

p(a).

p(c).

In the former case, a query ?- p(X). will return the solutions X = a, X = b, X = c, and in the latter, the solutions will be returned as X = b, X = a, X = c. Therefore, putting the clauses more likely to lead to a solution in the first place is sensible, because this would shorten the computation needed to reach this solution. In fact, a wrong order of clauses can lead to non termination: the following code

n(s(X)):- n(X).

n(z).

loops ad infinitum when the query ?- p(X). is issued. Switching the clauses

n(z).

n(s(X)):- n(X).

returns an infinite number of solutions (of course, there **is** an infinite number of solutions). When all solutions are required, the whole search tree is explored, so if it is infinite, the search will never end, yielding either an infinite number of solutions, or falling into the so-called ``infinite failure'': a branch which does not lead to a solution, but with a pattern which repeats itself.

**Input/Output**

The easiest way of doing input/output in Prolog is using the so-called DEC-10 predicates. They are based on the idea of having a current input and output, which can be redirected to write to and read from files.

|  |
| --- |
|  |
| |  |  | | --- | --- | | **Predicate** | **Explanation** | | write(X) | Write the term X on the current output stream. | | Nl | Start a new line on the current output stream. | | read(X) | Read a term (finished by a full stop) from the current input stream and unify it with X. | | put(N) | Write the ASCII character code N. N can be a string of length one. | | get(N) | Read the next character code and unify its ASCII code with N. | | see(File) | File becomes the current input stream. | | seeing(File) | The current input stream is File. | | seen | Close the current input stream. | | tell(File) | File becomes the current output stream. | | telling(File) | The current output stream is File. | | told | Close the current output stream. | |

**Name : Rohit Aggarwal**

**Roll No. : 7CS-097**