A note for a prolog user's group newsletter

Object-Oriented Prolog

Visual Prolog has become the world's most widely used Prolog. Though the dialect differs significantly from the standard set out years ago in Clocksin and Mellish, the many enhancements that the programmers at the Denmark's Prolog Development Centre have created have kept the language current with contemporary programming methodologies.

In this article I want to focus on two of these extensions to the standard, Clocksin and Mellish Prolog: primarily on the object-oriented extensions to incorporate in the latest releases (5.0 and 5.1). I have chosen three common data structures (list, stack and queue) to illustrate how to use Visual Prolog's new object-oriented extensions partly because data structures invite code reuse, and code reuse is among the key benefits of object-oriented programming. But a more important reason for choosing these abstract data types is that they allow me to explain the use to base classes and derived classes; for I shall create first a list as a base class and then use that list class as the base class for two derived classes, implementing the stack and queue ADTs. Implementing the derived classes using the list class as a base alos provides a pretty impressive example of the convenience of a language in which the list is basic, built-in, data structure.

However readers should be aware that using Prolog for building structures has more general advantages than my example illustrates, because Prolog, alone among widely used programming languages, allows the programmer to define recursive data types, without getting involved with pointers; to illustrate Prolog's enormous strength in this area, I also offer a few remarks comparing C++ and Prolog implementations of the tree ADT.

In Visual Prolog, a tree of strings is declared as:

```
DOMAINS

    key = string;
    mytree = tree(string, mytree, mytree)
```

Prolog is explicitly recursive – in our example a tree is key (value) and a left tree and a right tree. C, on the other hand, would use pointers to the left and right trees, and not trees themselves; thus the structure is not evidently recursive. No great intellectual effort is require to deal with pointers, of course, though the convenience of having pointers created and maintained automatically, as Visual Prolog does, is an appreciable advantage. But the slight increment in difficulty is not the main issue; rather, it is an issue of how the data structure is represented in the language's code: C's manner of treating the structure treats it as one comprising a cell containing a datum and two pointers (to two other cells), and not, as a truly recursive language like Prolog does, as a datum and two other cells (each of which contains a datum and two other cells, each of which . . .. Futhermore, C terminates recursion by initializing terminal cells with a special value (usually NULL) – which have to be identified through pointer dereference, a bit a

awkward imposition that is quite out of keeping with idea of recursion. Prolog allows the programmer to terminate recursion in more elegant fashion, by defining a tree as either one with values or an empty tree. To do this, the programmer declares that the data type **mytree** can employ either of two functors, **tree** or **empty**:

```
DOMAINS
        mytree = tree(string,mytree,mytree) ; empty
```

Then instead of deferencing a pointer to discern whether its value is NULL in order to determine whether a branch is empty, we simply check if **TreeToTest = empty**.

Prolog creates and adds data to a binary tree which serves as a container for strings thus – we'll even include a predicate to write the string, which can serve as a model for traversing the tree:

```
DOMAINS
  mytree = tree(string,mytree,mytree) ; empty()

PREDICATES
  create_tree(string, mytree)
  insert(string, mytree, mytree)
  write_tree(mytree).


CLAUSES
          create_tree(A, tree(A,empty,empty)).

         insert(New,end,tree(New,end,end)):-!.

          insert(New,tree(Element,Left,Right),tree(Element,NewLeft,Right)):-
                New<Element,!,
                insert(New,Left,NewLeft).

          insert(New,tree(Element,Left,Right),tree(Element,Left,NewRight)):-
                insert(New,Right,NewRight).

          write_tree(empty).

          write_tree(tree(Item,Left,Right)):-
                write_tree(Left),
                write(Item, " "),
                write_tree(Right).
```

What a breeze!. Compare this code with C++ code for accomplishing the same thing, which is listed in **ctree.c**. The difference is typical – Prolog's features lend it to

modelling domain objects, while C and C++ are more concerned with accurately reflecting what is going in computer memory and with economizing on memory.

Consider even the simplicity with Prolog handles lists.

```
append([ ], List, List ).

append([HeadIList1],List2,[HeadIRest]) :- append(List1,List2,Rest), !.
```

The C code that uses similar (Lisp-like) procedures to accomplish the same goal is more difficult to understand (though, of course, there would be little reason to try to imitate Lisp's list-handling procedures in real-world programming), contained in file clist.c. Ooouuch!!! In contrast with what has to master to create the code to a append on list to another in C, writing an clause in Prolog **append()** demands not much more than a rudimentary understanding of recursion. Prolog programming, like Lisp programming, does demand skill in thinking recursively, but the time taken to master those skills (and most people find acquiring them rather fun) does pay off in the greater simplicity of the coding process.

Another example of the payoff of Prolog data structure being truly recursive. Here's how Prolog, because it is fully recursive, can determines if an given element is a member of the list:

```
member(Token,[TokenI_],1).

member(Token,[_IRest],Position) :- !,
        member(Token,Rest,Cur_position),
        Position = Cur_position + 1.
```

Examine the code in ListList2.h and LispList2.c (which, to run, need to appended to LispList.h and ListList.c) to see how C does it Scary! The examples make it is obvious, I hope, that Prolog allows the programmer to think in terms of the logical structure of the abstract data type, and not on the details of the implementation.

Other features make Visual Prolog an attractive choice for the representation of domain objects and for the implementation of efficient algorithms through the prudent use of some of the more common data structures: Prolog is truly recursive language, and it allows for the creation of lists (arbitrary numbers) of any data-type, whether built-in or user-defined. Furthermore, B+ trees are a standard data structure for external databases in Visual Prolog. The availability of lists and B+ trees as standard data structures and the ability to define truly recursive data structures in Prolog offer real advantages to those working on large object-oriented projects.

Another feature that makes Visual Prolog well suited for object-oriented programming can be discerned in the declarations for the non-member functions in our program listing for the list predicate declaration file. (**list.pre**). The code shows that a feature of Visual Prolog lends the language to object-oriented programming, viz, the

possibility of overloading predicates in Visual Prolog. Visual Prolog allows us to have different predicates with the same name, but with different data types in their parameter lists (or even with different numbers of formal parameters). Thus we can have the following:

```
pop_head1(integerlist, integer, integerlist) -
(i,o,o)
pop_head1(reallist, real, reallist) - (i,o,o)
pop_head1(charlist, char, charlist) - (i,o,o)
pop_head1(stringlist, string, stringlist) -
(i,o,o)
pop_head1(symbollist, symbol, symbollist) -
(i,o,o)
```

The Visual Prolog compiler creates different object code for each of these predicates, recognizing their different signatures. Thus, when we call **pop_head1(Param1,Param2, Param3)** twice, once with parameters of **charlist**, **char**, **charlist** types, and once with **reallist**, **real**, **reallist** types, the Visual Prolog compiler is smart enough to distinguish between the two underlying predicates the compiler generates, that are masked by the programmer's having used a common name for the two.

Another feature to note regarding the contents of that file (**list.pre**). The "**reallist**" and "**charlist**" data types are user-defined domains, and since these domains must have more than file scope, we must declare them as **global domains**. A peculiarity of Visual Prolog is that all source files in a project must incorporate the same domains, and all global domains and global predicates must be declared before any local domains and local predicates are declared, it is convenient to do as I have done in this project, *viz.*, to create a "**glb**" file that contains all global declarations, and to **include** it before any local predicate or local predicates are declared.

We turn now to our example. The code for creating a list is divided into two parts, the class declaration and the class implementation; the implementation cannot be mixed in with the declaration as we ordinarily do in O.O. C++ header files. The declaration takes place between the keywords "**CLASS**" and "**ENDCLASS**"; the keyword, "**CLASS**" is followed by the name of the class, which, according to the Prolog Development Centre's "best practices" guidelines should commence with a small letter "c" followed by a descriptive name that begins with an upper-case letter (guidelines I have followed in calling my classes, "**cIntegerList**," "**cRealLis**t" etc. So our class declarations occur between the "**CLASS cClassName**" statement and **ENDCLASS** keyword:

.

```
CLASS cIntegerList

  PROTECTED PREDICATES
    procedure new(integerlist)
    procedure delete()
```

```
        check_data(integerlist) - (o)
        retrieve_data(integerlist) - (o)
        append(integerlist) - (i)
        pop_head(integer) - (o)
        add_to_front(integer) - (i)
        member(integer,integer) - (i,o), (i,i)
        insert(integer,integer) - (i,i)
        extract(integer,integer) - (i,i)
        cycle()
        reverse()
        list_length(integer)
        delete_all(integer) - (i)
        replace(integer,integer) - (i,i)
        at_end(integer) - (o)
        remove_last_element()
        nth_element(integer,integer) - (i,o)
        pick(integer) - (i)
        is_a_member(integer) - (i)
        eliminate_duplicates()
        sublist(integer,integer,integerlist) - (i,i,o)
        list_minimum(integer) - (o)
        list_maximum(integer) - (o)
        extract_nth_member(integer,integer) - (i,o)          '

    ENDCLASS
```

The second part of the code for creating a class, the class implementation section begins with the keyword "**IMPLEMENT,**" followed by the name of the class, and, following that implementation header, if the class has any member predicates or member facts (as it must if the class is to do anything), the actual clauses for the class's "member predicates," that are essentially like clauses in standard Prolog, as well as, if required, a number of facts that you can add and remove from your program at run time and, to close the class definition, the "**ENDCLASS**" keyword. Predicates are somewhat like functions in C, though unlike functions they can have more than one return value (and unlike functions, which explicitly are processes for taking input values and producing out values, they generate all their results through side-effects, since in fact predicates are just statements that determined to be true, or not to be true). Thus, we get something of the following form:

```
IMPLEMENT clntegerList

    FACTS
      determ classdata(integerlist)
```

```
CLAUSES

  classdata([]).
  new(Data):-
     retractall(classdata(_)),
    assert(classdata(Data)).

  delete():-
                retractall(classdata(_)).

  check_data(DataVal):-
       classdata(DataVal).

  retrieve_data( DataVal ):-
       classdata( DataVal ),
       retractall(classdata(DataVal)),
       assert(classdata([] )).

  append(ToAppendList):-
       classdata(PresentList),
       append1(PresentList,ToAppendList,OutList),
       retractall(classdata(PresentList)),
       assert(classdata(OutList)).

 pop_head(RetVal):-
       classdata(PresentList),
       pop_head1(PresentList,RetVal,NewList),
       retractall(classdata(PresentList)),
       assert(classdata(NewList)).

extract_nth_member(Index,ElementExtracted):-
       classdata(PresentList),
       extract_nth_member1(Index,PresentList,[],ElementExtracted,NewList),
       retractall(classdata(PresentList)),
       assert(classdata(NewList)).

ENDCLASS cIntegerList
```

In Prolog, the collection of facts represent a relational database that comprises facts that you add, remove and modify at run time; and, as in non-object-oriented Prolog, you use the predicates ***assert()***, ***asserta()***, and ***assertz()*** to add facts to the database at run time, and the predicates ***retract()*** and ***retractall()*** to remove predicates at run time,

and a combination of ***retract()*** and one of the ***assert()*** predicates to modify a fact. The predicate ***consult()*** reads facts an external file and makes them internal to the program, while the predicate ***save()*** saves facts in an external file.

A new feature of facts in Visual Prolog is that a fact (whether or not it is a member of a class) can be declared to be "**SINGLE**." To qualify as "SINGLE" an instance of the fact should always exist, that instance is the only instance of the fact that can exist. Another new feature is that facts can be declared with the "**NOCOPY**" keyword, which means that the data are not copied from the heap to the Global Stack when the fact is referenced. New as well is possibility for creating facts as class members. A key difference between facts that are class members and facts that are not (facts in non-object-oriented Prolog, for example) is that member facts are encapsulated, and can be operated on only by the class's member predicates. But this difference is important and affects the style of Prolog programming most Prolog programmers have developed – provided, of course they conform to object-oriented programming guidelines, and write truly object-oriented programs rather that hybrids between the two. Prolog programmers often use database facts to store information that they want a number of predicates to access (analogously to the way that C variables with file and global scope). If one adheres to the principles of object-oriented programming, one does not do this; one instead encapsulates facts in a class. Encapsulating the facts in a object has a certain overhead, for, if done properly, the data becomes hidden from other parts of the program; but that overhead repays with the advantages that data-hiding usually brings.

Further, facts and predicates that are members of a class can be declared to be static: if a fact is static member of a class in Visual Prolog, there will be only one version of the fact however many instances of class are generated, while if the member function is static there will be no pointer to the actual instance of the function will be generated when the member function is called.

The **cXList** (**cIntegerList**, **cRealList**, etc) classes also illustrate the use of Prolog predicate "**new()**,"which serves as a class constructor; objects which are constructed using **new( )** must be deleted explicitly, by calling **delete( )** for the object. The "**new( )**" and **delete( )** predicates are declared as to be **procedures**, another extension to the standard Prolog. A predicate in Visual Prolog is a procedure when, like a "**determ**" predicate in standard Prolog, it has a single solution, but, unlike a "**determ**" predicate in standard, it must not fail. "**new( )**" and "**delete( )**" are declared to be procedures, to preclude the possibility of their failing. Notice too that the "**FACTS**" section of the class – "**FACTS**" is simply a synonym for the standard "**DATABASE**" keyword, but this new nomenclature does serve as a handy reminder that the predicates declared here store the class's data, something useful to bear in mind when doing data-driven programming. In our **cXList** classes, the "**FACTS**" section declares the database predicate "**classdata(listtype)**" to be "**single**." This means that although each instance of the class has its own version of the datum stored with that database predicate (when one wishes all the instances of the class to share a single instance of some datum, one declares the predicate as "**static**," as in C++), there can only be one instance of that datum for each

instance of the class – no instance of the class can store multiple instances of that datum. Hence, like a "**determ**" database predicate, this predicate can return no more than one result, but.unlike "**determ**" predicates, it is constrained to return a result when called, and cannot fail.

The member predicates of the **cXList** classes call upon predicates whose data is not encapsulated within a class domain. Wrapping these non-member predicates within the member predicates of the classes accomplishes essentially two things: first, it protects the data that the class operates on, the **classdata**. Second, it provides greater easy of use – instead of having to carry around a pair of lists as input and output parameters for most functions (the class data before and after the predicate has operated on it), these parameters can be left as implicit. Thus, instead of having a function **pop_head(InputList, Head, OutputList)**, with a **(i,o,o)** flow pattern, we can simply use a function of the form **pop_head(Head)**, with a **(o)** flow pattern, and have the class take care of the overhead of updating its data (the **FACTS** encapsulated in the class).

It must be said, however, that data-hiding is not nearly so strict in Visual Prolog object oriented programming as it is in C++ classes. Whereas in C++ member functions by default are private member functions, In Visual Prolog classes all member predicates and facts are by default public classes; what is more, Visual Prolog does not even a mechanism for making member functions or member data private. However, data can be declared as **protected**, and if no classes are derived from a given class, then the protected members in the base class are effectively private members. I have treated all **cXList** classes as though they were not be used directly, and so have made all their members **protected**.

Subclassing and inheritance in also possible in Visual Prolog. To create a child class of a given class, declares the inheritance by including the name of the parent class in the opening declaration for the class, after a single colon. Thus I have declared the **cRealQueue** with the line **CLASS cRealQueue : cRealList**, to indicate that the **cRealQueue** class inherits from the **cRealList** class. A peculiarity of Visual Prolog is that all class predicates are virtual methods (that is any derived classes can provide a different version of any parent class method). Hence, if a predicate is redefined in the child class, then that version overrides the parent version. There is a limitation on this, however: the redefinition occurs only when the signature of the new predicate is the same as the signature of the predicate in the parent class (that is, it must have not only the same name, but also the same formal parameter list) Thus, when a predicates is redefined in child clas, it ovewrites the parent class's version of the predicate only if is called on an a set of objects of the same number and type. However, if you define the predicate in subclass as having parameters belonging to different domains than parent-class predicate takes, or as having a different number of arguments, then the difference in signatures leads the Visual Prolog compiler to treat the parent-class predicate and the child-class predicate as different predicates, and the child-class predicate will not override the parent-class predicate (that is, the Visual Prolog compiler will not treat the parent-class predicate as a virtual method).

Because all the predicates in a Visual Prolog class are virtual methods, building

abstract classes is almost as simple as creating a class definition without an implementation (for it exists only to have its declared methods inherited by subclasses. But it is almost as simple as that, but not quite. Ordinarily, if the Visual Prolog compiler cannot find an implementation for some method declared in the class definition, it flags an error, and the compilation process fails. Prefacing the class definition with the word "**ABSTRACT**" precludes this. Thus one could have the following:

```
ABSTRACT CLASS integer_datastruct

PREDICATES
        integerlist retrieve_data()
                store_data(integerlist)
                 write_data(integerlist)
ENDCLASS
```

Each classes derived from this base class can implement its own method for storing, retrieving and writing its data; but all the class would present to the world outside the class a similar interface, for all would be invoked by statements which have the same format.

The clauses defining the **cXQueue** class and **cXStack** predicates also point up how object-oriented Visual Prolog handles lexical scope: Predicate names and predicate facts may be redefined in a class hierarchy. To access names in class prior in the class hierarchy, or to access static predicates and static facts, explicit scoping is required. The following clauses from the **cIntegerStack** class gives an example how this is done.

```
push(DataVal):-
        cIntegerList::add_to_front( DataVal).

pop(RetVal):-
        cIntegerList::pop_head(RetVal).
```

The definition of each predicate consists of a clause stated in "class-name::method-name()" format. This format is used for explicit scoping. Objects of type **cIntegerStack** have no database of their own – the **cIntegerStack** methods modify implicitly uses the data associated with its parent, an object of the **cIntegerList** type. Thus the cIntegerStack's methods have to access data belonging to objects that are higher in the class hierarchy, so explicit scoping is required.