IMPLEMENTING PASCAL-LIKE CONSTRUCTS: AN EXERCISE IN PROLONG PROGRAMMING

INFORMATICA 2/87

Bogdan Filipič
Department of Computer Science and Informatics
»Jožef Stefan« Institute, Ljubljana

UDK 681.3.06:519.682 PROLOG

ABSTRACT. Due to the declarative meaning of programs, Prolog is a powerful programming language. However, in practice it turns out that numerous tasks within a certain kind of programs are quite procedural. The paper describes a simple implementation of some Pascal-like construots in Prolog as a practical solution to this problem.

IMPLEMENTACIJA PASCALSKIH KONSTRUKTOV KOT VAJA IZ PROGRAMIRANJA V PROLOGU. Prolog je zaradi deklarativnega pomena programov močan programski jezik, vendar se v praksi izkaže, da so mnoga opravila znotraj določenih programov povsem proceduralnega značaja. V članku je kot praktična rešitev tega problema prikazana implementacija nekaterih pascalskih konstruktov v Prologu.

INTRODUCTION

Due to the declarative meaning of programs, Prolog is a powerful programming language [1,3]. It is especially well suited for solving non-numerical problems. From the programmer's point of view, programming in Prolog is very efficient. On the other hand, Prolog implementations suffer from the lack of supporting the procedural programming approach. In practice it namely turns out that numerous tasks within a certain kind of programs are quite procedural. One may, for example, wish to perform an action conditionally, repeat a procedure until a certain condition is satisfied or execute a sequence of actions a given number of times. To do this in Prolog, we usually use cuts and repeat-fail loops. Unfortunatelly, improving efficiency through these mechanisms often weakens the clarity and the conciseness of a Prolog program.

The problem can be solved neater by using explicit definitions for the control of execution. Including Pascal-like constructs into Prolog may be seen as a practical solution to this problem.

IF-THEN AND *IF-THEN-ELSE* PROCEDURES

Some Prolog implementations support conditional by default. Remember, for example, *P -> Q ; R* from DECsystem-10 Prolog [2], C-Prolog [5] or Quintus Prolog [6]. Arity Prolog [7] even supports *ifthen/2* and *ifthenelse/3* predicates. Here are common definitions of these procedures:

```
if_then(P,Q) :- call(P), !, call(Q).
if_then(P,Q).

if_then_else(P,Q,R) :- call(P), !, call(Q).
if_then_else(P,Q,R) :- call(R).
```

Using operators, we may simply define an appropriate predicate *if/1* that is determined as a principal functor. Its definition includes both above alternatives (see implementation in the appendix). We may now code conditionals of both forms:

```
if P then Q.
if P then Q else R.
```

where *P*, *Q* and *R* denote single Prolog goals or sequences (i.e. conjunctions or disjunctions) of goals.

CASE PROCEDURE

Once the *if* predicate has been implemented, we may define *case* as a sequence of *if* procedures trying to match a certain condition value and to satisfy related goal(s). We introduce the Prolog *case* construct

```
    case X of [ X1:Q1, X2:Q2, ..., Xn:Qn ].
```

that is interpreted as

```
            if X = X1 then Q1
            else if X = X2 then Q2
            else if   .
                      .
                      .
            else if X = Xn then Qn.
```

and

```
 case X of
    [ X1:Q1, X2:Q2, ..., Xn:Qn otherwise R ].
```

that stands for

```
            if X = X1 then Q1
            else if X = X2 then Q2
            else if   .
                      .
                      .
            else if X = Xn then Qn
            else R.
```

Note that our implementation does not support more than one value being assigned with each alternative. This can be done by joining possible values into a list and substituting condition *X = Xi* with testing the list membership. The following is an example of using the *case* construct:

```
  menu_selection :-
     write( 'Selection? ' ),
     read_line( Answer ),
     case Answer of
        [ a : option1( ... ),
          b : option2( ... ),
          c : option3( ... ),
          h : ( help, menu_selection),
          x : exit_menu
          otherwise
             ( beep,
               write( 'Illegal answer' ), nl,
               menu_selection
             )
        ].

     read_line( Answer ) :-
        get( ASCII ),
        name( Answer, [ASCII] ).

     beep :- put(7).
```

REPEAT-UNTIL PROCEDURE

Here we introduce the *repeat/1* predicate that will have similar effect as the built-in *repeat/0*. You must keep in mind that proposed

```
                  repeat Q until P.
```

is just a Pascal-like notation for the procedure that will be executed in the Prolog sense, i.e. failure of *Q* will result in backtracking. The execution may be viewed as Pascal-like when backtracking is not possible, for example:

```
   repeat ( write('Filename? '), read(F) )
   until exists(F).
```

Note that, when *Q* stands for a sequence of goals, they must be put into brackets and the left bracket must be separated from the principal functor repeat by blank. Otherwise the entire expression denotes a predicate of an arity greater than 1 which will fail, of course. Similarly, this syntax restriction must be considered when using *if*.

FOR PROCEDURE

In order to implement the *for* loop, let us first introduce a utility for managing global counters [4]. Suppose we have certain values in our program that can be passed to or modified by any part of the program. Modifying these values may be understood as assigning values to global variables in procedural languages. Such variables are particularly suitable as counters. Each counter is specified by its name, a key, and the related integer value. The counter managing predicates below simply use *retract* and *assert* to assure the current value of a counter to be recorded in the database.

Based on previously defined *repeat* procedure and the global counters, the *for* loop has the following two forms:

```
     for [Count,I] := I1 to I2 do Q.
     for [Count,I] := I1 downto I2 do Q.
```

Count identifies the procedure and should be instantiated to a Prolog constant. Furthermore, the global counter with the key *Count* is activated when executing the procedure. Its current value is instantiated to *I*. The following example illustrates how to use the loop:

```
          for [i,I] := 1 to 10 do
             ( write(I), nl ).
```

Again, the execution is Pascal-like only when the goal(s) appearing in the loop can be satisfied in no more than one way.

CONCLUSION

The paper presents the implementation of some Pascal-like constructs in Prolog. We found them useful for writing procedural segments of programs, such as input and output procedures. As already stated, our purpose is not to reduce the importance of standard Prolog concepts, but just to add some convenient procedural features. In our opinion, combination of both declarative and procedural approaches is the right solution when wondering about how to code a complex task effectively.

And finally, the reader might have noticed that we said nothing about the *repeat* procedure when discussing Pascal-like features. The reader is invited to implement it himself as an exercise.

REFERENCES

[1] Bratko I.: Prolog Programming for Artificial Intelligence, Addison-Wesley, 1986

[2] Byrd L., Pereira F., Warren D.: A Guide to Version 3 of DEC-10 Prolog, Department of Artificial Intelligence, University of Edinburgh, 1983

[3] Clocksin W.F., Mellish C.S.: Programming in Prolog, Springer-Verlag, 1984

[4] Filipič B., Mozetič I.: A Library of Prolog Utilities, Report IJS DP-4466, Jožef Stefan Institute, Ljubljana, 1986

[5] Pereira F.: C-Prolog User's Manual, University of Edinburgh, Department of Computer Aided Architectural Design, 1984

[6] Quintus Prolog User's Guide and Reference Manual, Quintus Computer Systems Inc., Palo Alto, 1985

[7] The Arity/Prolog Programming Language, Arity Corporation, Concord, 1986

APPENDIX: THE PROGRAM

```
%--------------------------------------
% File PROCED :
% Implementing Pascal-like constructs
%--------------------------------------

:- op( 900, fx,  if   ).
:- op( 850, xfx, then ).
:- op( 800, xfx, else ).

:- op( 900, fx,  case      ).
:- op( 850, xfx, of        ).
:- op( 800, xfx, otherwise ).
:- op( 750, xfx, ':'       ).

:- op( 900, fx,  repeat ).
:- op( 850, xfx, until  ).

:- op( 900, fx,  for    ).
:- op( 850, xfx, to     ).
:- op( 850, xfx, downto ).
:- op( 800, xfx, do     ).
:- op( 750, xfx, ':='   ).

:- [counts].   % Counter management.

if P then Q :-
   if_then_else(
      Q = (R else S),        % If 'else' found
      if_then_else(P,R,S),   % then if_then_else
      if_then(P,Q)           % else if_then.
   ).

   if_then(P,Q) :- call(P), !, call(Q).
   if_then(_,_).

   if_then_else(P,Q,_) :- call(P), !, call(Q).
   if_then_else(_,_,R) :- call(R).

case X of [ Xn:Q otherwise R ] :-
   if X=Xn then Q else R, !.
case X of [ Xn:Q ] :-
   if X=Xn then Q, !.
case X of [ Xi:Q | Others ] :-
   if X=Xi then Q else case X of Others.

repeat Q until P :-
   repeat, call( (Q, !) ), call(P).

for [_,_]:=I1 to I2 do _ :-
   I1>I2, !.
for [Count,I]:=I1 to I2 do Q :-
   ctr_set(Count,I1),
   repeat ( ctr_inc(Count,I), Q )
   until I=I2,
   ctr_remove(Count), !.
for [_,_]:=I1 downto I2 do _ :-
   I1<I2, !.
for [Count,I]:=I1 downto I2 do Q :-
   ctr_set(Count,I1),
   repeat ( ctr_dec(Count,I), Q )
   until I=I2,
   ctr_remove(Count), !.
```

```
%--------------------------
%  File COUNTS :
%  Managing global counters
%--------------------------

% ctr_set/2 sets a counter to the
% desired number.

ctr_set(Key,N) :-
   integer(N),
   ctr_remove(Key),
   assert( counter(Key,N) ), !.

% ctr_inc/2 increments a counter and
% returns its previous value.

ctr_inc(Key,N) :-
   retract( counter(Key,N) ),
   N1 is N + 1,
   assert( counter(Key,N1) ), !.

% ctr_dec/2 decrements a counter and
% returns its previous value.

ctr_dec(Key,N) :-
   retract( counter(Key,N) ),
   N1 is N - 1,
   assert( counter(Key,N1) ), !.

% ctr_is/2 returns the current value
% of a counter.

ctr_is(Key,N) :-
   counter(Key,N), !.

% ctr_remove/1 removes a counter
% from the database.

ctr_remove(Key) :-
   retract( counter(Key,_) ), fail.
ctr_remove(_).
```