TCPW/CW/K.INT.

BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

How to obtain a Prolog System with your Preferred Syntax

by
Bart DEMOEN *

Internal Report
BIM-prolog IR2

May 1984

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

How to obtain a prolog system with your preferred syntax.(1)

Abstract.

We present the Prolog system, curently under development at B.I.M. and K.U.Leuven. It offers the possibility to redefine in some sense the syntax, so that in fact it is feasable to program in your preferred Prolog dialect.

Section 1 Towards a clear-cut flexible prolog system.

All around, people have felt the need for a Prolog system and thus have defined a Prolog language [1,2,3,4,6]. Different people have different taste, but Prolog as proposed (rather informally) in [7] seems to be the de facto standard Prolog for later developed systems. Nevertheless, these systems suffer from some drawbacks which should be avoided. We will come back to this point later.
It is a fact that there is a need for a clean definition of a Prolog system, syntactically as well as semantically, that is not cluttered with too many exceptions and in which programming in Prolog is by all means attractive.
We are building a new system, which we call Professional Prolog. It will be composed of a structured editor, a translator, an interpreter etc. It will be modular in the sense that modules can be translated separately.[5]
Its syntax is compatible to a high extent with existing Prolog systems.[2,3,4,5]
It will however offer the user a flexibility, which he could use for example to define his own Prolog syntax (see section 8).

---

(1) this work is sponsored by the 'Diensten voor de Programmatie van het Wetenschapsbeleid' under contract nr K8AR/SOFT/1

We will present the concrete and abstract syntax of Professional Prolog.

After that, some remarks on the difference with existing systems will be made.

Then, the flexibility is introduced in the system.

Section 2 The concrete syntax.

The central object of the syntax is the term. Its definition is very much like the one of term in [3]. A term has a precedence, which is inherited from the operators appearing in it (if any). We have adopted the convention of DEC-10 prolog, i.e. the precedence is an integer between 0 and 1200, with higher precedence meaning less binding.

A term can be an unstructured object, i.e. an atom or a number.

A term can also be a structered object, in which case it has a name - the name of the functor or the operator appearing in the term - and one or more arguments, which are terms again. We give a formal syntax in which nonterminals are surrounded by < >; alternatives appear on different lines and any symbol to the right of ==> and not between < > is an endsymbol.

```
 <term N> ==> <op(N,fx)> <subterm N-1>

              <op(N,fy)> <subterm N>

              <subterm N-1> <op(N,xfx)> <subterm N-1>

              <subterm N-1> <op(N,xfy)> <subterm N>

              <subterm N> <op(N,yfx)> <subterm N-1>

              <subterm N-1> <op(N,xf)>

              <subterm N> <op(N,xf)>


 <term 0> ==> <functor> ( <arglist> )

              (<subterm 1200>)

              <constant>

              <variable>

 <subterm N> ==> <term M>     where M =< N

 <op(N,T)> ==> <name>  different from  '<string of char>'
                       and defined as an operator of type T and precedence N
 <functor> ==> <name>  NOT defined as an operator


 <arglist> ==> <subterm 999>

               <subterm 999> , <arglist>

                the reason for the restriction to 999 is that the predefined
                operator , of precedence 1000 would cause ambiguity


 <constant> ==> <atom>

                <integer>

                <real>


 <atom> ==> <name>   NOT defined as an operator
```

For the moment, further (lexical) details are unimportant.

Some operators are predefined, like aritmetic operators and the operators

`,   ;   :-   ?-`   that have to do with the semantics.

The type of an operator can be prefix (fx,fy), infix (xfx,xfy,yfx) or postfix (xf,yf): the x and the y in the types denote the associativity of the operator, which should be clear from the syntax of term. No action is associated with an operator: i.e. if + is an operator, then

```
                4 + 5 stands for the structured object    +
                                                         / \
                                                        4   5

                which is certainly different from 9
```

Section 3 A Prolog program.

A prolog program consists of a set of sentences: a sentence is a term that is followed by a definite sequence of characters. We choose the following:

```
<program> ==> <sentence>

              <sentence> <program>

<sentence> ==> <subterm 1200> . <eoln>
```

where <eoln> denotes the end of line control character.

Up to now we have completely ignored semantics and in fact the semantics puts some restrictions on the terms that are legal prolog sentences.

We give now another definition of <sentence> but it should be understood that the former is still valid:

```
<sentence> ==> <clause> . <eoln>
               <query> . <eoln>
               <directive> . <eoln>

<directive> ==> :- <goals>

<query> ==> ?- <goals>

<clause> ==> <head> :- <goals>
             <head>

<head> ==> <term 1200>  different from <integer>, <real> and <variable>

<goals> ==> <andlist>
            <orlist>
            <subterm 1200> different from <integer>
<andlist> ==> <goals> , <goals>
<orlist> ==> <goals> ; <goals>
```

The meaning of this piece of syntax is the following:

any <subterm 1200> will be treated according to its structure:

```
1/      ?-
        |
        |       it will be treated as a query
        |
     <goals>


2/      :-
        |
        |       it will be treated as a directive
        |
     <goals>


3/      :-
      /    \  . it will be treated as a clause
     /      \
  <head>   <goals>
```

4/ any other <sentence> will be treated as

```
                  :-
                 /  \
                /    \
        <sentence>   true
```

Section 4 The abstract syntax.
==============================

Almost any action on a prolog sentence needs the structure of the term: pretty printing, translation, elaborate editing actions (for example substituting one subterm for another). Since this structure is not apparent from he prolog text one writes, we decided not to keep the text, but rather an abstract form of it. This abstract form is of course the tree structure associated with the terms of the sentences in the text.
The abstract syntax looks like:

```
 <sentence> ==> <compound>

 <compound> ==> <functor> <arglist>

 <arglist> ==> <argument>

               <argument> <arglist>

 <argument> ==> <atom>

                <integer>

                <real>

                <compound>

                <variable>
```

A <compound> can be represented by the name of its <functor> and its arity (its number of arguments).
<atom>, <integer>, <real> and <variable> are represented by a keyword and their value or name.
This abstract form is very general: almost any existing Prolog can be put in this form, so that large parts of the tools we are developing, are independent of the particular Prolog one works with.

Section 5 The programs.

At the moment, all programs -like parsers, editors, interpreters ... - are written in Prolog. We have good reasons for that:

1/ since we are still in the design phase, the writing of programs should not take to much time and effort: programming in Prolog is fast; in fact, from earlier experience, we can say that Prolog is a very good language for rapid prototyping, which is what we need [9]

2/ we have a prolog system available,[1] which does not offer all the features of the future system, (for instance, it has no <orlist>, nor operators) but which is strong enough to simulate most of it

3/ once a translator was written, we could already write programs in the new syntax and have them executed by the existing system and so gain some insight in the working of the future system

Right now, a parser-translator has been written, which transforms a prolog program into its abstract form. In building the parser, we found it useful to put a mild restriction on the generality of the definition of operators:

a postfix operator is not allowed to be infix nor prefix.

This makes the parsing more efficient: one needs a larger look ahead when not making this restriction. Also, it prevents the writing of terms which are difficult to parse for human programmers.

A structured editor is under development. It is also written in Prolog and it benifits from some ideas of [8]: since in the existing system, there is tail-recursion optimisation as well as garbage collection, this is quite feasible.

Section 6 Some remarks.

Our syntax deviates from the de facto standard at some points:

-variables always begin with an underscore: the uppercase convention is a nuisance to a lot of people (some claim it better be a lowercase letter a variable has to begin with); in any case, when querying a database, one would like to write

```
?- capital_city_of( _x , Belgium) .
```

instead of having to write 'Belgium', to indicate that we mean the atom Belgium and not the variable.

-this brings us to a more general statement about quotes:

> whenever you quote a sequence of characters, you are denoting an atom with that sequence of characters as value (or name); i.e. '+' means the atom with value + and NOT the operator + ; quotes are thus used to strip a sequence of characters from all properties, except of beeing an atom

this is useful in writing parsers, where the selection of a particular procedure can be achieved by having a quoted operator as an argument in the head of a clause; another place where one can use this, is when one wants to output an operator, e.g. the multiplication operator; this is achieved simply by the term

```
?- write( '*' ) .
```

this is a much cleaner way, then to allow operators not to have any operands;

lastly, 'xyz' is the same atom as xyz , of course on condition that xyz is not declared as an operator

-a last remark about operators: we stated already that an operator has no action associated with it; this means that operators are just syntactic sugar, but it should be used rigourously: once you have declared a name as an operator, every (non-quoted) occurence of this name is treated as an operator; in particular, it should have the correct number of operands;

on the other hand

```
'+'(4,5)  is the same term as   4 + 5
```

except that the former has precedence 0, the latter has the precedence of the infix operator +

Section 7 Some flexibility introduced in the system.

Part of the flexibility in a Prolog system comes from the fact that operators can be defined and deleted by the user by means of appropriate directives: this seems to be standard now.

We have gone a little further, so that in our system the user has almost complete control over the syntax he wants to write his programs in. An illustration of this statement appears in Section 8.

Of course, not all features are meant for the naive prolog programmer, but more experienced people can certainly take advantage of it.

a/ new_name.

First of all, we incorporated a renaming facility: for example by the directive

```
                :- new_name(cut,!) .
```

the atom cut becomes a new name for !, so that wherever one used to write the atom !, one can write the atom cut.

This seems very innocent, but in combination with operator declarations like

```
                :- new_name(!,fac) .

                :- store_op(fac,xf,150) .
```

it allows you to use ! as the faculty operator.

This renaming occurs at the lexical level.

Now one can freely choose alternatives for existing operators, such as

```
                and  for  ,
                or   for  ;
                if   for  :-  etc.
```

but also one can give a name to frequently occuring numbers and structures.

Even the point . denoting the end of a sentence, can be renamed, so that later on, one could use . as an infix list constructor.

This feature can be seen as an editing facility.

b/ attribute.

In questioning a relational database, it is useful to be able to denote the different components of a relation by their attribute. Also, one does not like to write down long lists of void variables in which one is not interested.

A query as

```
                ?- student_info(_,_,321,_,_name) .
```

is easier to write as   ?- student_info(stud_name = _name , stud_nr = 321) .

To achieve this, we have introduced a new directive with name attribute, of which we give an example:

```
     :- attribute(student_info(stud_adr,home_adr,stud_nr,birthday,stud_name)) .
```

This can be useful in writing programs which interact with databases, or in database queries.

c/ new structure.

Certainly   _x supplies _y to _z is more readable then supplies_to(_x,_y,_z).

To be able to write the first form, you need to give the directives

```
        :- store_op('supplies',xfy,950) .
        :- store_op('to',xfy,950) .
```

but then still the two forms are not equivalent.

To achieve the equivalence, we have introduced a new directive

```
        :- new_struct(supplies_to(_x,_y,_z), _x supplies _y to _z) .
```

Roughly speaking, the effect is that any occurence of _x supplies _y to _z will be replaced by supplies_to(_x,_y,_z).

In fact, it is a pattern matching substitution so that

```
        _supplier supplies motorcars to _customer  will be replaced by
        supplies_to(_supplier,motorcars,_customer)
```

There is a small problem about the order in which different new_struct directives are executed (it can make a hell of a difference !!).

Also if this substitution is applied recusively, one has to be careful.

But good aggreements solve this.

Also, in contrast to new_name and attribute, the new_struct directive has no effect on other directives.

The main difference between new_name and new_struct is in the moment of the execution of the directive:

new_name is executed between the lexical and syntactic analysis, while

new_struct is executed after the syntactic analysis

The new_struct feature can be seen as a forced partial evaluation, but it is different in the sense that its goal is not to optimise code and that the substitution occurs not only at the call level, but also on the argument level. This feature makes it possible to include the query language DATATRIEVE [10] as part of prolog.

Section 8 How the flexibility can be used.

We claim that with the new_name and new_struct directives, we have in fact extended our syntax, so that it becomes a host for other (prolog) dialects which can be defined as a sequence of terms. We give an example:

```
        +append(nil,_l,_l)-/;
        +append(.(_a,_l),_m,.(_a,_n))-append(_l,_m,_n);
```

is definition for an append procedure in the syntax of [1].

Clauses begin with + , the body of a clause begins with the leftmost - , different goals are separated by - , and the end of a sentence is ;

A query is written as

```
        -append(_x,.(a,nil),.(b,.(a,nil)))-write(_x);
```

/ is the cut.

Any program written with this syntax, can be analysed by our parser-translator by first giving the directives:

```
        :- delete_op('+',fx) .

        :- delete_op('+',yfx) .

        :- delete_op('-',fx) .

        :- delete_op('-',yfx) .

        :- store_op(+,fx,1200) .

        :- store_op(-,fx,1200) .

        :- store_op('-',xfy,1000) .

        :- new_name( / , !) .

        :- new_name( ; , .) .

        :- new_struct( (_x :- true) , ( + _x)) .

        :- new_struct(  ( _x , _y) ,  ( _x - _y)) .

        :- new_struct( (_x :- _y) , ( + _x - _y)) .

        :- new_struct(  ( ?- _x) , ( - _x) ) .
```

In this way, the syntax of [1] is embedded in the more general syntax of terms. It should be stressed however that this gives no protection against errors against the syntax of [1]: some terms which mean nothing to [1], still have a meaning in the more general syntax. One can freely mix sentences in the syntax of [1] with more general terms: the general syntax is really a host.

Conclusion

We presented a Prolog system, which allows for the incorporation of almost any syntax that is based on the definition of <term>. Also we pointed out some features that make other systems obscure and that we have tried to avoid.

References.

[1] 'Description of Prolog' M. Bruynooghe
Departement Computerwetenschappen K.U.Leuven 1979

[2] MProlog Language Refence Manual March 1983
SzKI, Budapest, Hungary

[3] DECsystem-10 PROLOG USER'S MANUAL D.L.Bowen
11-11-1982 University of Edingburgh, Dept of Artificial Intelligence

[4] CProlog User's Manual Version 1.2 F. Pereira
SRI International, Menlo Park, California

[5] 'Modules in Prolog, once again' G. Janssens
Departement Computerwetenschappen K.U.Leuven jan. 1984

[6] Prolog II Manuel d'utilisation Mars 1982
M. Van Caneghem
Groupe Intelligence Artificielle ERA CNRS 363
Fac des Sciences de Luminy Marseille

[7] 'Programming in Prolog' W.F. Clocksin, C.S. Mellish
Springer-Verlag Berlin Heidelberg New York 1981

[8] D. Warren
talk at the Prolog Programming Environments Workshop
Linkoping, Sweden march 1982

[9] 'Prolog as a language for prototyping of Information Systems'
R. Venken, M. Bruynooghe
oct. 1983 Namur, Belgium

[10] Introduction to VAX-11 DATATRIEVE Digital Equipment Corporation
Maynard, Massachusettes

concrete syntax

alternatieven in 1 syntaxregel staan onder elkaar ofwel gescheiden door |

blanco's in de regels dienen enkel om de leesbaarheid te verhogen

om termen van elkaar te scheiden zijn meestal blanco's nodig: zo is

ab45 1 <name>; ab 45 , 45ab en **qw zijn er telkens twee

```
<sentence> ==> <clause> . <eoln>

               <query> . <eoln>

               <directive> . <eoln>

<directive> ==> :- <subterm 1199>


<clause> ==> <head>

             <head> :- <goals>

<head> ==> <term 1199>  verschillend van <integer> of <variable>

<query> ==> ?- <goals>

<goals> ==> <subterm 1199>

<subterm N> ==> <term M>     where M =< N

<term N> ==> <op(N,fx)> <subterm N-1>

             <op(N,fy)> <subterm N>

             <subterm N-1> <op(N,xfx)> <subterm N-1>

             <subterm N-1> <op(N,xfy)> <subterm N>

             <subterm N> <op(N,yfx)> <subterm N-1>

             <subterm N-1> <op(N,xf)>

             <subterm N> <op(N,xf)>


<term 0> ==> <functor> ( <arglist> )

             (<subterm 1200>)
             <constant>

             <variable>


<op(N,T)> ==> <name>  verschillend van '<string of char>'
                      die als operator van type T en
                      precedence N gedefinieerd werd



<functor> ==> <name>  die NIET als operator gedefinieerd werd


<arglist> ==> <subterm 999>

              <subterm 999> , <arglist>

           de , betekent hier niet de pregedefinieerde infix-operator met
           precedence 1000
           als voor <arglist> ook <subterm 1000> toegelaten werd dan zou de ,
           voor dubbelzinnigheid zorgen


<constant> ==> <atom>

               <integer>


<variable> ==> <underscore> <restname>


<atom> ==> <name>   die NIET als operator gedefinieerd werd


<integer> ==> <number>

              - <number>

<number> ==> <digit>

             <digit> <number>


<name> ==> '<string of char>'

           <letter> <restname>

           1 symbool van de lijst ;!,

           <special sequence>


<special sequence> ==> <special char>

                       <special char> <special sequence>


<special char> ==> +-*/ ^<>=`~:.?%   $&


<string of char> ==> <char>
```

```
                        <char> <string of char>


<alfanum> ==> <letter>

              <digit>

              <underscore>


<restname> ==> <empty>

               <alfanum> <restname>


<char> ==> een willekeurig printable teken verschillend van '
           ..


<letter> ==> A | B | ... | Z | a | b | ... | z


<digit> ==> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9


<underscore> ==> _


<eoln> ==>  end of line

<empty> ==>

<comment> ==> { <tekst zonder }> }     mag willekeurig waar voorkomen:
                                        scheidt eventueel nonterminals
```

Er moet nog een lijst met built-in's komen en een lijst directieven (niet alleen operatordefinities, maar ook I/O patroon en ...?).

Het punt . kan onbeperkt gebruikt worden als atoom, functor, operator met de conventie dat een . gevolgd door <eoln> het einde van een <sentence> betekent.

```
                 abstracte syntax
                 ----------------

<sentence> ==> <compound>

<compound> ==> <functor> <arglist>

<arglist> ==> <argument>

              <argument> <arglist>

<argument> ==> <atom>

               <integer>

               <compound>

               <variable>
```

```
Voorstelling van de abstracte syntax
------------------------------------

<atom>                         0      atom_name

<integer>                      int    integer_value

<variable>                     var    variable_name

                        of     void

<compound>                     arity  functor_name   <arglist>

<arglist>                      <argument>

                        of     <argument>   <arglist>
```