Introduction to AI with Prolog

Advanced in terms

Alexey Sery

a.seryj@g.nsu.ru

Department of Information Technologies
Novosibirsk State University

November 17, 2020

Overview

Goals of the lesson

- To introduce operators
- To recall term comparison operators
- To discuss term decomposition operations

Comparison

Prolog contains an important predicate for comparing terms, namely ==/2. It **does not instantiate** variables, thus it is not the same as the unification predicate =/2.

`term1 == term2` is `true` if and only if `term1` and `term2` are identical.

Comparison

Prolog contains an important predicate for comparing terms, namely ==/2. It **does not instantiate** variables, thus it is not the same as the unification predicate =/2.

`term1 == term2` is `true` if and only if `term1` and `term2` are identical.

```
term == term.     true.

term1 == term2.   false.

term == 'term'.   true.
```

Comparison

Prolog contains an important predicate for comparing terms, namely ==/2. It **does not instantiate** variables, thus it is not the same as the unification predicate =/2.

`term1 == term2` is `true` if and only if `term1` and `term2` are identical.

```
X == Y.

X == a.

X = a, X == a.

X = Y, X == Y.
```

Comparison

Prolog contains an important predicate for comparing terms, namely ==/2. It **does not instantiate** variables, thus it is not the same as the unification predicate =/2.

`term1 == term2` is `true` if and only if `term1` and `term2` are identical.

```
X == Y.           false.

X == a.           false.

X = a, X == a.    true.

X = Y, X == Y.    true.
```

Comparison

Prolog contains an important predicate for comparing terms, namely ==/2. It **does not instantiate** variables, thus it is not the same as the unification predicate =/2.

`term1 == term2` is `true` if and only if `term1` and `term2` are identical.

It should now be clear that == can be viewed as a ***stronger*** test for equality between terms than =. That is, if `t1` and `t2` are Prolog terms, and the query `t1 == t2` succeeds, then the query `t1 = t2` will succeed too.

Comparison

A predicate that is reverse of == is \==. This predicate id defined so that it succeeds precisely in those case where == fails.

```
term \== term.

t1 \== t2.

term \== 'term'.

X \== a.

X \== Y.
```

Comparison

A predicate that is reverse of == is \==. This predicate id defined so that it succeeds precisely in those case where == fails.

```
term \== term.     false.

t1 \== t2.         true.

term \== 'term'.   false.

X \== a.           true.

X \== Y.           true.
```

Comparison

Now we have introduced all comparing terms. Here is the summary:

=	The unification predicate. Succeeds if it can unify its arguments, fails otherwise.
\\=	The negation of the unification predicate. Succeeds if = fails, and vice-versa.
==	The identity predicate. Succeeds if its arguments are identical, fails otherwise.
\\==	The negation of the identity predicate. Succeeds if == fails, and vice-versa.
=:=	The arithmetic equality predicate. Succeeds if its arguments evaluate to the same integer.
=\\=	The arithmetic inequality predicate. Succeeds if its arguments evaluate to different integers.

Term notation

Some terms may have multiple notations, and, although they look different to us, Prolog regards them as identical. As we recall, `t` and `'t'` are identical, that is these are two different notations of the same term, as far as Prolog is concerned.

In fact there are many other cases there Prolog treats two strings as being exactly the same term. Most of them are *syntactic sugar* that makes programming more pleasant.

It is very handy to be able to write code in the notation we like, and to let Prolog run our code in the notation it finds natural.

Term notation

A good and most obvious example of this are the arithmetic predicates. As we recall, arithmetic operations are terms, having multiple notations.

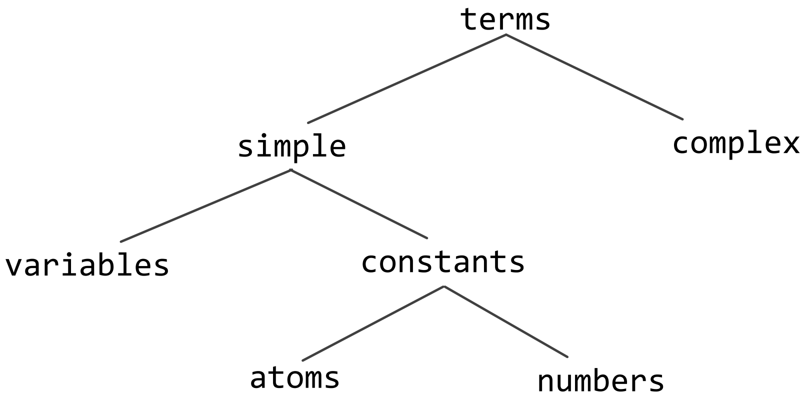
```
2 + 3          ==   +(2,3).
2 - 3          ==   -(2,3).
2 * 3          ==   *(2,3).
3 * (7 + 5)    ==   *(3, +(7,5)).
```

Term notation

A good and most obvious example of this are the arithmetic predicates. As we recall, arithmetic operations are terms, having multiple notations.

```
(2 < 3)     ==   <(2, 3).
(2 =< 3)    ==   =<(2, 3).
(2 =:= 3)   ==   =:=(2, 3).
(2 =\= 3)   ==   =\=(2, 3).
(2 > 3)     ==   >(2, 3).
(2 >= 3)    ==   >=(2, 3).
```

Examining terms



Examining terms

Sometimes it is useful to know of which type a given term is. Prolog provides built-in predicates that test if a given term is of certain type or not.

`atom/1`	Tests whether the argument is an atom.
`integer/1`	Tests whether the argument is an integer.
`float/1`	Tests whether the argument is a floating point number.
`number/1`	Tests whether the argument is a number, i.e. an integer or a float.
`atomic/1`	Tests whether the argument is a constant.
`var/1`	Tests whether the argument is an uninstantiated variable.
`nonvar/1`	Tests whether the argument is not an uninstantiated variable.

Examining terms

Suppose we have a complex term of which we don't know what it looks like. What do we want to know about it?

Probably, what its functor is, what's the arity and what do arguments look like.

Prolog provides built-in predicates that do precisely that.

Examining terms

The first two questions are answered by the predicate `functor/3`. Given a term this predicate will tell us what the functor and the arity are.

```
?- functor(f(x, y), F, A).  F = f, A = 2.

?- functor(f, F, A).        F = f, A = 0.
```

We can use the predicate `functor/3` not only to find the structure of a term, but also to **construct** terms. How? By specifying the second and third arguments and leaving the first uninstantiated.

```
?- functor(T, f, 3).  T = f(_21882, _21884, _21886).
```

Note, that either the first argument or the second and third argument have to be instantiated.

Examining terms

The third question about the structure of an unknown complex term is answered by two predicates.

The first is `arg/3` which tells us about arguments of complex term. It takes a number N and a term `T` and returns the N-th argument of `T` as its last argument. It can be used either to access the value or to instantiate a corresponding argument.

```
?- arg(2, f(x, y), Second).  Second = y.
?- arg(2, f(x, Y), y).       Y = y.
?- arg(3, f(x, y), Third).   false.
```

Note that enumeration on arguments starts at 1.

Examining terms

The second is `'=..'/2`. It takes a complex term and returns a list containing the functor as first element and then all the arguments. This predicate is called **univ**.

```
?- f(x, y, z) =..  Struct.  Struct = [f, x, y, z].
?- f(X) =..  F.             F = [f, X].
?- T =..  [g, x, y, z].     T = g(x, y, z).
```

Operators

Operators are the unary or binary predicates defined as:

1. *Infix* operator. They are written between their arguments.
2. *Prefix* operator that are written before their argument.
3. *Postfix* operator which is written after their argument.

Operators

Prolog standard predicate notations are not ambiguous. For example we wouldn't be confused on how to interpret the expression `is(11, +(2, *(3,3)))`, because it is pretty obvious in which order operations should be performed.

Other notations, however, could be more ambiguous. For example Prolog knows about conventions for disambiguating arithmetic expressions. When we write `2 + 3 * 3`, Prolog knows that it is `2 + (3 * 3)` we meant, not `(2 + 3) * 3`.

Such conventions are implemented with operator *precedence*. Every operator has a certain precedence. The precedence of `+` is greater that the precedence of `*`, for instance. In the expression `2 + 3 * 3` operator `+` has the highest precedence, so it is taken to be the main functor of the expression: `+(2, *(3,3))`.

The precedence of `is/2` is higher than the precedence of any arithmetic operator, so that `11 is 2 + 3 * 3` is interpreted as `is(11, +(2, *(3, 3)))`.

Operators

OK, that is what about precedence. Let us look to the following query, though.

```
?- 6 is 1 + 2 + 3.
```

Prolog correctly answers `true`. But how Prolog parenthesizes the internal representation: `is(6, +(1, +(2, 3)))` or `is(6, +(+(1, 2), 3))`?

Prolog uses information about the **associativity** of operations to disambiguate the expressions.

For example, + is *left associative*. That means the expression to the right of + must have a lower precedence that + itself, whereas the expression to the left may have the same precedence as +.

Some operators might be defined to be non-associative which means that both of their arguments must have a lower precedence.

Operators

The type of an operator (infix, prefix, or postfix), its precedence, and its associativity are the three things that Prolog needs to know to be able to translate the user friendly, but potentially ambiguous operator notation into Prolog's internal representation.

New operator definitions in Prolog look like:

```
:- op(Precedence, Type, Name).
```

Operators

The type of an operator (infix, prefix, or postfix), its precedence, and its associativity are the three things that Prolog needs to know to be able to translate the user friendly, but potentially ambiguous operator notation into Prolog's internal representation.

New operator definitions in Prolog look like:

```
:- op(Precedence, Type, Name).
```

Precedence is a number between 0 and 1200.

Operators

The type of an operator (infix, prefix, or postfix), its precedence, and its associativity are the three things that Prolog needs to know to be able to translate the user friendly, but potentially ambiguous operator notation into Prolog's internal representation.

New operator definitions in Prolog look like:

```
:- op(Precedence, Type, Name).
```

Precedence is a number between 0 and 1200. Type is an atom specifying the type and associativity of the operator: f represents the operator while x and y represent its arguments, x stands for an argument which must have a precedence lower than the precedence of f, and y stands for an argument which could have the precedence lower or equal to the precedence of f.

`infix`	`xfx, xfy, yfx`
`prefix`	`fx, fy`
`postfix`	`xf, yf`

Operators

Here are the definitions for some of the built-in operators. You can see that operators with the same properties can be specified in one statement by giving a list of their names instead of a single name as third argument of op.

```
:- op(1200, xfx, [:-, ->]).
:- op(1200, fx, [:-, ?- ]).
:- op(1200, xfy, [;]).
:- op(1000, xfy, [',']).
:- op(700, xfx, [=, is, =.., ==, \==, =:=, =\=, <, >, =<, >=]).
:- op(500, yfx, [+, -]).
:- op(500, fx, [+, -]).
:- op(300, xfx, [mod]).
:- op(200, xfy, [^]).
```