Logisch Programmeren en Zoektechnieken: Herfst 2007

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

Operators in Prolog

Operators provide a more convenient way of writing certain expressions in Prolog that could otherwise be difficult to read for humans. For example, we can write `3 * 155` instead of `*(3, 155)` or `N is M + 1` instead of `is(N, +(M, 1))`.

Both forms of notation are considered to be equivalent, i.e. matching works:

```
?- +(1000, 1) = 1000 + 1.
Yes
```

The objective of this lecture is to show you how you can define your own operators in Prolog.

Operator Precedence

Some operators bind stronger than others. In mathematics, for example, `*` binds stronger than `+`. The degree to which an operator is binding is called its *precedence*.

In Prolog operator precedences are numbers (in SWI-Prolog between 0 and 1200). The arithmetic operator `*`, for example, has precedence 400, `+` has precedence 500. That is, the lower an operator's precedence value, the stronger it is binding.

This is why Prolog is able to compute the correct result in the following example (i.e. not 25):

```
?- X is 2 + 3 * 5.
X = 17
Yes
```

Precedence of Terms

The precedence of a term is defined as the precedence of its *principal operator*. If the principal functor isn't (written as) an operator or the term is enclosed in parentheses then the precedence is defined as 0.

Examples:

- The precedence of `3 + 5` is 500.
- The precedence of `3 * 3 + 5 * 5` is also 500.
- The precedence of `sqrt(3 + 5)` is 0.
- The precedence of `elephant` is 0.
- The precedence of `(3 + 5)` is 0.
- The precedence of `3 * +(5, 6)` is 400.

Operator Types

Operators can be divided into three groups:

- *infix operators*, like + in Prolog
- *prefix operators*, like ¬ in logic or - for negative numbers
- *postfix operators*, like ! in mathematics (factorial)

Is giving the type of an operator and its precedence already enough for Prolog to fully "understand" the structure of a term containing that operator?

Example

Consider the following example:

```
?- X is 25 - 10 - 3.
X = 12
Yes
```

Why not 18?

So, clearly, precedence and type alone are *not* enough to fully specify the structural properties of an operator.

Operator Associativity

We also have to specify the *associativity* of an operator: `-`, for example, is left-associative. This is why `25 - 10 - 3` is interpreted as `(25 - 10) - 3`.

In Prolog, associativity is represented by atoms like `yfx`. Here `f` indicates the position of the operator (i.e. `yfx` denotes an infix operator) and `x` and `y` indicate the positions of the arguments. A `y` should be read as *at this position a term with a precedence lower or equal to that of the operator has to occur*, whereas `x` means that *at this position a term with a precedence strictly lower to that of the operator has to occur.*

Understand how this makes the interpretation of `25 - 10 - 3` unambiguous (note that `-` is defined using the pattern `yfx`)!

Associativity Patterns

Pattern	Associativity		Examples
`yfx`	infix	left-associative	`+, -, *`
`xfy`	infix	right-associative	`,` (for subgoals)
`xfx`	infix	non-associative	`=, is, <` (i.e. no nesting)
`yfy`	makes no sense, structuring would be impossible		
`fy`	prefix	associative	
`fx`	prefix	non-associative	`-` (i.e. `--5` not possible)
`yf`	postfix	associative	
`xf`	postfix	non-associative	

Checking Precedence and Associativity

You can use the built-in predicate `current_op/3` to check precedence and associativity of currently defined operators.

```
?- current_op(Prec, Assoc, *).
Prec = 400
Assoc = yfx
Yes

?- current_op(Prec, Assoc, is).
Prec = 700
Assoc = xfx
Yes
```

Checking Precedence and Associativity (cont.)

The same operator symbol can be used once as a binary and once as a unary operator:

```
?- current_op(Prec, Assoc, -).

Prec = 500
Assoc = fx ;

Prec = 500
Assoc = yfx ;

No
```

Defining Operators

New operators are defined using the `op/3`-predicate. This can be done by submitting the operator definition as a query. Terms using the new operator will then be equivalent to terms using the operator as a normal functor, i.e. predicate definitions will work.

For the following example assume our big animals program has previously been compiled:

```
?- op(400, xfx, is_bigger).
Yes

?- elephant is_bigger dog.
Yes
```

Query Execution at Compilation Time

It is possible to write queries into a program file (using `:-` as a prefix operator). They will be executed whenever the program is compiled.

If for example the file `my-file.pl` contains the line

```
:- write('Hello, have a beautiful day!').
```

this will have the following effect:

```
?- consult('my-file.pl').
Hello, have a beautiful day!
my-file.pl compiled, 0.00 sec, 224 bytes.
Yes
?-
```

Operator Definition at Compilation Time

You can do the same for operator definitions. For example, the line

```
:- op(200, fy, small).
```

inside a program file will cause a prefix operator called `small` to be declared whenever the file is compiled. It can be used inside the program itself, in other programs, and in user queries.

Summary: Operators

- The structural properties of an operator are determined by its precedence (a number) and its associativity pattern (e.g. `yfx`).
- Use `current_op/3` to check operator definitions.
- Use `op/3` to make your own operator definitions.
- Operator definitions are usually included inside a program file as queries (using `:-`, i.e. like a rule without a head).