

# Chapter 4

## Parsing

### 4.1 Operator Declarations

A very simple way to treat new syntax in Prolog is via operator declarations. These allow one to declare new operators, along with priority and whether they are infix, prefix or postfix.

The advantage is that no parser has to be written: the standard Prolog parser is then able to read in files using these new operators. The drawback is that the syntax has to build upon the core syntax of Prolog.

Let's say we want to extend Prolog with an equivalence operator `<=>`. First, we have to declare the operator:

```
:- op(990,xfy,'<=>').
```

The 1120 is the precedence of the operator; as a reference the built-in operator `+` has the priority 500, `*` the priority 400, and exponentiation the priority 200. The operator `:-` to build clauses has priority 1200, disjunction `;` has priority 1100, and the conjunction `,` has priority 1000. The `xfy` tells Prolog that the new operator is infix and associates to the right. You can now use the operator in this manner in your program. Note that you can still safely use the prefix version using quotes as well (i.e., `'<=>'(X,Y)`).

We can check that it indeed associates to the right as follows. As you can see by looking at the solution for `L`, the `B` associates with the `<=>` on its right:

```
| ?- E = (A <=> B <=> C) , E=..L.  
E = A<=>B<=>C,  
L = [<=>,A,B<=>C] ?  
yes
```

We can now define this operator for example as follows:

```
X <=> Y :- if(call(X),call(Y),\+ call(Y)).
```

Here is a small program that now uses the operator:

```

h(a). h(b). h(c).
i(b). i(c).
j(c). j(d).
hij(X) :- h(X), i(X) <=> j(X).

```

The program behaves as expected:

```

| ?- hij(X).
X = a ? ;
X = c ? ;
no

```

Another approach to extend Prolog is term expansion. This does not change the syntax of Prolog programs, but it can adapt its meaning. It can be used in conjunction with operator declarations to extend the syntax. The term expander is run automatically by the Prolog parser after parsing every clause. The term expander can modify the clause or individual literals in the clause, e.g., by adding new arguments or even by completely replacing the calls. The definite clause grammars which will examine in the next section are internally implemented this way: the Prolog syntax is extended with the new `-->` operator and the term expander inserts new arguments to calls.

## 4.2 DCGs

DCG stands for Definite Clause Grammar. It is a mechanism to write context-free grammars conveniently as a Prolog program and to use it for parsing. DCGs automatically make use of a “programming trick” called difference lists, which we explain first.

### 4.2.1 Difference Lists

Lists in Prolog are represented using the constant `[]/0`, representing the empty list, and the functor `./2` representing non-empty lists. For example, the list consisting of two elements 1 and 2, would be represented by the Prolog term `.(1, .(2, []))`. This is illustrated on the top-left in Figure 4.1. Prolog provides several more convenient ways of writing a list. For example, `[1,2]` is equivalent to `.(1, .(2, []))`, and so is `[1| [2]]` and `[1 | [2 | []]]`.

We will sometimes call a term composed of the `./2` functor as a cons-cell.<sup>1</sup>

**Exercise 4.2.1** Find more equivalent ways of writing the term `.(1, .(2, []))`. Check that they are indeed equivalent using your Prolog system. How many equivalent ways are there?

Concatenation of lists is a common operation in Prolog, particularly when developing parsers. Concatenation of two lists can be performed using the built-in predicate `append/3`. By hand, `append/3` can be defined as follows.

<sup>1</sup>In Lisp the functor `cons/2` is used instead of `./2`.

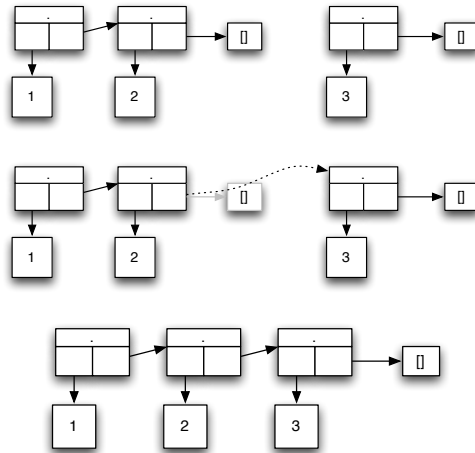


Figure 4.1: Illustrating appending the ordinary list [3] to [1,2]

```
append([],R,R).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

While this is convenient, the drawback is that `append` completely traverses its first argument every time it is called until it finds the empty list, and then “redirects” this “pointer” to the start of the second list (see Figure 4.1).

There are different ways to view a difference list. One is that when we pass around an ordinary list, we only pass a “pointer” to the first cons-cell of the list. Thus, when we append we need to traverse the entire list to find the last cons-cell. The idea of the difference list is to pass around two “pointers”: one to the first cons-cell (as before) and one to the end of the list. To group these two pointers into one datastructure, we use an arbitrary binary functor, e.g., `-/2`.<sup>2</sup> When we concatenate, we can simply “jump” straight to the end of the list, without having to traverse it. However, this would not be much use if we did not change one more aspect: namely that instead of terminating our lists by the empty list, we will terminate them by a free Prolog variable. This means, that we can concatenate two difference lists in constant time, by one unification!

This process is illustrated in Figure 4.2. The code for appending two difference lists is simply:

```
append_dl(FX-X,FY-Y,FX-Y) :- X=FY.
```

or even more succinctly

```
append_dl(FX-X,X-Y,FX-Y).
```

We pay a price, however:

<sup>2</sup>Of course we need to be consistent and always use the same functor.

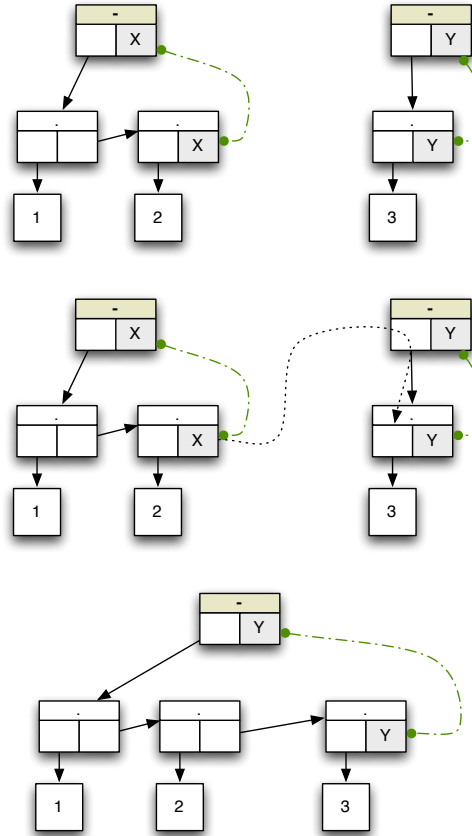


Figure 4.2: Illustrating appending the difference list  $[3|Y]-Y$  to  $[1,2|X]-X$

- traversing a list becomes a bit more complicated: rather than checking for the empty list, we need to compare with the pointer to the end of the list.
- we can only concatenate a list “once” in this way (as first argument). Indeed, after appending the first list in Figure 4.2 using `append_dl` it is no longer  $[1,2|X]-X$  but  $[1,2,3]-[3]$ . This also explains the term *difference list*:  $[1,2,3]-[3]$  represents the list  $[1,2]$  obtained by taking the difference between  $[1,2,3]$  and  $[3]$ .

Thus, one should use difference lists only in appropriate circumstances. Luckily, parsing is one of them.

**Exercise 4.2.2** Write a predicate which translates a difference list into an ordinary list.

### 4.3 Simple DCGs

Let us first examine a very simple parsing problem. Let us try and parse simple arithmetic expressions using addition (+). For simplicity, we restrict ourselves to three identifiers  $x$ ,  $y$ , and  $z$ .

This could be encoded using the following context-free grammar with terminals  $\{x, y, z, +\}$  and non-terminals  $\{P, P', N\}$  and start symbol  $P$ :

$$\begin{aligned} P &\rightarrow NP' \\ P' &\rightarrow \epsilon \mid +P \\ N &\rightarrow x \mid y \mid z \end{aligned}$$

This grammar was obtained by left-factoring the following simpler grammar (not suitable for LL(1) parsing):

$$\begin{aligned} P &\rightarrow N \mid N + P \\ N &\rightarrow x \mid y \mid z \end{aligned}$$

Let us first translate this grammar into Prolog without difference-lists. The idea is to translate every non-terminal symbol into one predicate. Each predicate has one argument, and it should be true if the corresponding non-terminal can generate the string encoded in the argument (as an ordinary list).

```
n([x]).
n([y]).
n([z]).

pc([]).
pc([+|Rest]) :- p(Rest).

p(String) :- n(S1), append(S1,S2,String), pc(S2).
```

Let us examine whether the Prolog code works as expected:

```
| ?- p(X).
X = [x] ? ;
X = [x,+,x] ? ;
X = [x,+,x,+,x] ? ;
X = [x,+,x,+,x,+,x] ?
yes
| ?- p([x,+,y,+,z]).
yes
| ?- p([x,0,y]).
0 = + ? ;
no
| ?- p([x,x]).
no
```

**Exercise 4.3.1** What happens if we move the `append` call in the clause of `p` to the end:

```
p(String) :- n(S1), pc(S2), append(S1,S2,String).
```

Let us now try and rewrite the above Prolog code using difference-lists:

```
dl_n([x|T]-T).
dl_n([y|T]-T).
dl_n([z|T]-T).
dl_pc(X-X).
dl_pc([+|T]-R) :- dl_p(T-R).
dl_p(X-R) :- dl_n(X-X1), dl_pc(X1-R).
```

How do we call it:

```
| ?- dl_p(X).
X = [x|_A]-_A ? ;
X = [x,+,x|_A]-_A ? ;
X = [x,+,x,+,x|_A]-_A ? ;
```

We can simply force the difference to be the empty list, thus obtaining an ordinary list in the first component of the difference list:

```
| ?- dl_p(X-[]).
X = [x] ? ;
X = [x,+,x] ? ;
X = [x,+,x,+,x] ? ;
X = [x,+,x,+,x,+,x] ?
```

In the above code the functor `-` just keeps the two components of a difference list together. We can simply use two arguments instead, resulting probably in slightly more efficient code:

```
n([x|T],T).
n([y|T],T).
n([z|T],T).
pc(X,X).
pc([+|T],R) :- p(T,R).
p(X,R) :- n(X,X1), pc(X1,R).
```

This version also clarifies an alternate (but equally valid) view of difference lists:

- One can view the first component of a difference list as the string that still needs to be processed.
- In the second argument, every predicate returns the remaining part of the input string, which it has not consumed.

The translation of a grammar to a Prolog program is thus pretty straightforward:

- every non-terminal  $N$  is translated into a Prolog predicate  $p_N$  with two arguments,
- a grammar rule  $A \rightarrow B_1, \dots, B_n$  is translated into  $p_N(I_0, I_n) : -R_1, \dots, R_n$ , where  $R_i$  is  $p_i(I_{i-1}, I_i)$  if  $B_i$  is a non-terminal symbol and  $R_i$  is  $I_{i-1} = [B_i|I_i]$  if  $B_i$  is a terminal symbol and where  $I_i$  are fresh variables.

Luckily, Prolog systems can do this translation for us if we use the DCG syntax. Our grammar in DCG syntax can be written as follows:

```
n --> [x] ; [y] ; [z].
pc --> [] ; [+],p.
p --> n , pc.
```

This code is translated internally<sup>3</sup> into almost<sup>4</sup> *exactly* the same code as the previous version using difference lists.

```
| ?- p([x,+,y,+,z], []).
yes
```

**Exercise 4.3.2** Examine the internal representation of the above DCG using the Prolog `listing` command.

Some more details about DCGs:

- all predicate calls on the RHS get two extra arguments; these are threaded.
- one can put calls in to brackets `{ }` in order to prevent the addition of the extra arguments. This is useful for actions to be executed while parsing (sometimes called semantic actions or parser actions).
- one can provide further input to the non-terminal symbols in the form of extra arguments
- similarly, a non-terminal can return values (called semantic values) in the form of extra arguments.

symbol are written as lists...

Let us translate the DCG into a calculator. We will pass an environment containing the values for  $x$ ,  $y$ , and  $z$  and return the value of the expression.

```
n(env(X,Y,Z),R) --> [x],{R=X} ; [y],{R=Y} ; [z], {R=Z}.
pc(E,LR,R) --> [], {R=LR} ; [+],p(E,PR),{R is LR+PR}.
p(E,R) --> n(E,NR) , pc(E,NR,R).
```

<sup>3</sup>Using a mechanism called term expansion.

<sup>4</sup>The only difference stems from the use of the disjunction.

```
| ?- p(env(1,2,3),R,[x,+,y,+,z],[ ]).
R = 6 ?
yes
```

**Exercise 4.3.3** What happens if you remove the curly braces around `R` in `LR+PR`. Explain.

## 4.4 Treating Precedences and Associativites with DCGs

A general scheme to treat operator precedences and associativity using context free grammars is

- to create one layer of non-terminals per operator precedence level, where
- every layer calls the layer below, and
- where the bottom layer can produce atomic expressions (such as identifiers or literals) and call the top layer while generating parentheses.

Let us try and parse simple arithmetic expressions using addition (+), exponentiation (\*\*) and parentheses. For simplicity, we restrict ourselves to three identifiers  $x$ ,  $y$ , and  $z$ .

This could be encoded using the following context-free grammar, with three layers:

$$\begin{aligned} P &\rightarrow E \mid E + P \\ E &\rightarrow N \mid N ** E \\ N &\rightarrow (P) \mid x \mid y \mid z \end{aligned}$$

Unfortunately, this grammar is not suitable for recursive descent parsing (it is not LL(1)). For this we need to perform left-factoring, resulting in the following grammar:

$$\begin{aligned} P &\rightarrow EP' \\ P' &\rightarrow \epsilon \mid +P \\ E &\rightarrow NE' \\ E' &\rightarrow \epsilon \mid ** E \\ N &\rightarrow (P) \mid x \mid y \mid z \end{aligned}$$

This grammar is now LL(1). Unfortunately, it still has one problem: it treats both `+` and `**` as a right-associative operator. While this is correct for exponentiation ( $x**y**z = x**(y**z)$ ) it is not correct for addition ( $x+y+z = (x+y)+z$  and not  $x+(y+z)$ ).

#### 4.4. TREATING PRECEDENCES AND ASSOCIATIVITIES WITH DCGS27

Still, using DCGs, we can build our own parse tree or rather abstract syntax tree. [Explain] For a right-associative operator, we simply build up the parse tree in a natural manner as we recurse. For a left-associative operator we use an accumulator to build up a reversed parse tree.

Below we also use a new feature of Prolog: a string between double quotes represents a list of its ASCII codes. For example, "abAB0" represents the list [97,98,65,66,48] and "(" represents [40], and can thus be used to represent ASCII terminal symbols. This feature is useful if we want to develop parsers that work directly on ASCII files.

##### Program 4.4.1

```
plus(R) --> exp(A), plusc(A,R).

plusc(Acc,Res) --> "+",!, exp(B), plusc(plus(Acc,B),Res).
  /* left associative: B associates to + on left */
plusc(A,A) --> [].

exp(R) --> num(A), expc(A,R).

expc(Acc,exp(Acc,R2)) --> "**",!, exp(R2). /* right associative */
expc(A,A) --> [].

num(X) --> "(",!,plus(X), ")".
num(id(x)) --> "x".
num(id(y)) --> "y".
num(id(z)) --> "z".

parse(S,T) :- plus(T,S,[]).
```

This parser can be called as follows:

```
| ?- parse("x+y+z",R).
R = plus(plus(id(x),id(y)),id(z)) ?
yes
| ?- parse("x**y**z",R).
R = exp(id(x),exp(id(y),id(z))) ?
yes
| ?- parse("x**y+x**y**z+y",R).
R = plus(plus(exp(id(x),id(y)),exp(id(x),exp(id(y),id(z))))),id(y)) ?
yes
```

For the last call, we illustrate the parsing process in Figure 4.3 and the (abstract) syntax tree constructed by our parser (and returned in R) in Figure 4.4.

**Exercise 4.4.2** Extend the parser from Program 4.4.1 to handle the operators -, \*, /.

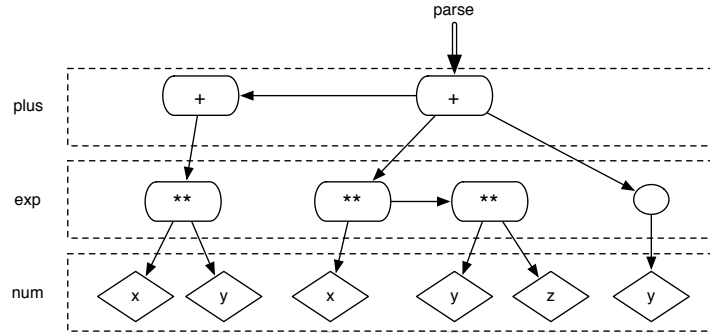


Figure 4.3: Illustrating the parse hierarchy and tree for "x\*\*y+x\*\*y\*\*z+y"  
 (Note: error in arrows of +)

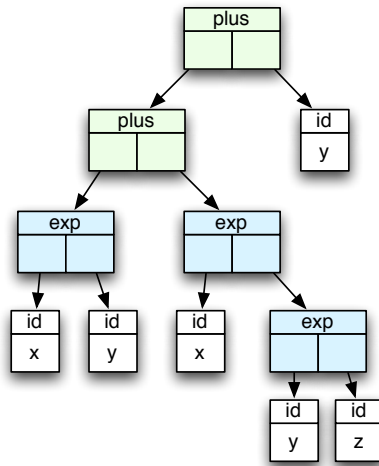


Figure 4.4: The abstract syntax tree constructed for "x\*\*y+x\*\*y\*\*z+y"

## 4.5 Tokenisers

**Exercise 4.5.1** Extend the parser from Program 4.4.1 and Exercise 4.4.2 to link up with a tokenizer that recognises numbers and identifiers.

## 4.6 What have we learned

- Grammars can be naturally translated into Prolog (every non-terminal becomes a Prolog predicate).
- Prolog execution yields a top-down recursive descent parser.
- Difference lists yield concatenation in constant time and are especially useful when implementing parsers.
- The DCG notation allows one to use difference-lists with ease, and enables one to translate grammars very conveniently into Prolog. Semantic actions can be integrated using curly braces, semantic values can be passed and returned as extra arguments.