



Logic Programming: Parsing. Difference Lists, DCGs

Alan Smaill

Oct 21 2013

- ▶ Context Free Grammars (review)
- ▶ Parsing in Prolog
- ▶ Definite Clause Grammars (DCGs)

A simple CFG:

S → NP VP

NP → DET N

VP → VI | VT NP

DET → the

N → cat | dog | food

VI → meows | barks

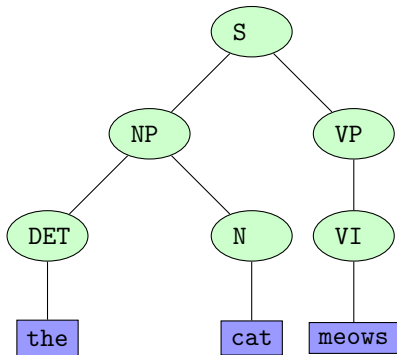
VT → bites | eats

- ▶ Yes:
 - ▶ “the cat meows”
 - ▶ “the cat bites the dog”
 - ▶ “the dog eats the food”

- ▶ No
 - ▶ “cat the cat cat”
 - ▶ “dog bites meows”

This uses a proof tree, rather than a search tree.

S -> NP VP
NP -> DET N
VP -> VI | VT NP
DET -> the
N -> cat | dog | food
VI -> meows | barks
VT -> bites | eats



$T \rightarrow c$

$T \rightarrow aTb$

In Prolog, with lists of characters:

`t([c]).`

```
t(S) :- t(S1),  
        append([a], S1, S2),  
        append(S2, [b], S).
```

```
s(L) :- np(L1), vp(L2), append(L1,L2,L).
```

```
np(L) :- det(L1), n(L2), append(L1,L2,L).
```

```
vp(L) :- vi(L) ;  
         vt(L1), np(L2), append(L1,L2,L).
```

```
det([the]). det([a]).  
n([cat]). n([dog]). n([food]).  
vi([meows]). vi([barks]).  
vt([bites]). vt([eats]).
```

- ▶ Clearly, we need to guess when we're generating –
 - ▶ but **also** guess when we're parsing an **unknown** sequence
- ▶ This is inefficient — lots of backtracking!
 - ▶ Reordering goals doesn't help much

An even simpler CFG (again)

$T \rightarrow c$

$T \rightarrow aTb$

In Prolog, with accumulators:

$t([c|L], L).$

$t([a|L1], M) :- t(L1, [b|M])$

?- t(L, []).

L = [c].

L = [a,c,b].

L = [a,a,c,b,b].

...

A difference list is a pair (t, X) , where

- ▶ t is a list term with the shape $[t_1, t_2, \dots, t_n | X]$, and
- ▶ X is a variable.

Difference lists correspond to normal lists as follows:

$$\frac{\text{normal list}}{[t_1, t_2, \dots, t_n]} \quad \frac{\text{difference list}}{[t_1, t_2, \dots, t_n | X]}$$

Here we need to be careful that different difference lists use different Prolog variables!

Difference lists are important because they allow much more efficient list operations.

Some examples:

- ▶ Empty difference list: (X, X)
- ▶ Constant difference list: $([a_1, a_2, \dots, a_n | X], X)$,
- ▶ Appending difference lists (t, X) and (u, Y) :
 - simply unify X and u
 - yields $(t [u/X], Y)$

Sometimes, people work with just the first part of the difference list (above, $[t_1, t_2, \dots, t_n | X]$); need to be careful that the variable really is a variable when called.

We can write append of difference lists simply using a richer representation. Let's take Z/X for a difference list ($Z = [t_1, t_2, \dots, t_n]$) above; here $/$ is already available as an infix operator. Then difference list append is simply:

`dl_append(X/Y, Y/Z, X/Z).`

This is correct and efficient when we really are dealing with difference lists, and the first and second are inputs. Because there is a single clause, there can only be one solution, if there is any solution.

```
?- dl_append( [1,2|Y]/Y, [3,4|Z]/Z, Ans) .
```

```
Y = [3,4|Z],
```

```
Ans = [1,2,3,4|Z]/Z;
```

```
no
```

```
?- dl_append( A, B, [1,2,3]/Z ).
```

```
A = [1,2,3]/_A,
```

```
B = _A/Z ? ;
```

```
no
```

An even simpler CFG (again)

$T \rightarrow c$

$T \rightarrow aTb$

In Prolog, with difference lists:

$t(L,M) :- L = [c|M].$

$t(L,M) :- L = [a|L1],$
 $t(L1,M1),$
 $M1 = [b|M]$

Parsing using DCGs is so useful that Prolog has built-in syntax for it:

$t \text{ --> } [c].$

$t \text{ --> } [a], t, [b].$

translates to:

$t(L,M) \text{ :- } L = [c|M].$

$t(L,M) \text{ :- } L = [a|L1],$
 $t(L1,M1),$
 $M1 = [b|M].$

- ▶ Rules have the form `nonterm --> body`
- ▶ Body terms are:
 - ▶ **terminal lists** `[t1, ..., tn]` (may be `[]`)
 - ▶ **nonterminals** `s, t, u ...`
 - ▶ **sequential composition** `body1, body2`
 - ▶ **alternative choice** `body1; body2`

DCG is translated to difference lists version, so used in the same way.

```
?- t(L, []).
```

```
L = [c].
```

```
L = [a,c,b].
```

```
L = [a,a,c,b,b]
```

We can also use the built-ins `phrase/2`, `phrase/3`:
when the first argument is a non-terminal from the grammar, this
generates corresponding examples (in difference list form in the
second case).

?- `phrase(t,L)` .

`L = [c]` .

`L = [a,c,b]` .

?- `phrase(t,L,M)` .

`L = [c|M]` .

`L = [a,c,b|M]` .

s --> np, vp.

np --> det, n.

vp --> vi ; vt, np.

det --> [the] ; [a].

n --> [cat] ; [dog] ; [food].

vi --> [meows] ; [barks].

vt --> [bites] ; [eats].

DCG clause bodies can also contain **tests**, written as an arbitrary Prolog goal in curly brackets: `{Goal}`

Example:

```
n --> [Word], {noun(Word)}.
```

```
noun(dog). noun(cat).
```

- ▶ Left recursion, as usual, leads to non-termination:

`exp --> exp, [+], exp`

- ▶ Avoid by using right recursion and fall-through

`exp --> simple_exp, [+], exp.`

`exp --> simple_exp.`

- ▶ Non-terminals in DCGs can have parameters:

$t(0) \text{ --> } [c]$.

$t(\text{succ}(N)) \text{ --> } [a], t(N), [b]$

- ▶ Can keep track of depth of nesting in terms.

With parameters, we go outside the expressiveness of CFGs.

$$\begin{aligned}u(N) \text{ --> } & n(N, a), \\ & n(N, b), \\ & n(N, c).\end{aligned}$$
$$n(0, X) \text{ --> } [] .$$
$$n(\text{succ}(N), X) \text{ --> } [X], n(N, X) .$$

This characterises a set of expressions that has no CFG description.
(what set?)

- ▶ “the cat meows”
 - ▶ S(NP(DET (the), N (cat)),
VP (VI (meows)))
- ▶ “the cat bites the dog”
 - ▶ S(NP (DET (the), N(cat),
VP (VT (bites),
NP(DET(the), N(dog)))
- ▶ Can build parse trees using parameters
– look for this as a tutorial exercise.

- ▶ LPN, chs 7–8: more difference list examples and translation DCGs to Prolog

Next time:

- ▶ search techniques
- ▶ depth-first, iterative deepening, breadth-first, best-first.