# Logic Programming: Parsing. Difference Lists, DCGs

Alan Smaill

Oct 15 2015

# Today

- 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
```

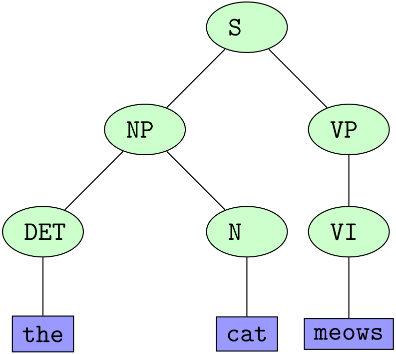
## Recognising grammatical sentences

- Yes:
  - "the cat meows"
  - "the cat bites the dog"
  - "the dog eats the food"
- No
  - "cat the cat cat"
  - "dog bites meows"

## Generation Example

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
```



## A simpler CFG

```
T  -> c
T  -> 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]).
```

## A better way?

- 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  -> c
T  -> 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].
...
```

## Difference Lists

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

- `t` is a list term with the shape `[t1,t2,..tn|X]`, and
- `X` is a variable.

Difference lists correspond to normal lists as follows:

| normal list | difference list |
|---|---|
| $[\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_\mathtt{n}]$ | $([\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_\mathtt{n} \mid \mathtt{X}], \mathtt{X})$ |

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.

# Difference lists cdt

Some examples:

- Empty difference list: `(X,X)`
- n-element difference list: `([a1,a2,..an|X],X)`,
- Appending difference lists `(t,X)` and `(u,Y)`:

  — simply unify `X` and `u`
  — yields `(t[u/X],Y)`

eg, append `([1,2|X],X)` to `([3,4|Y],Y)`;
unify `X=[3,4|Y]`, and obtain `([1,2,3,4|Y],Y)`.

## Difference Lists ctd

Sometimes, people work with just the first part of the difference list (above, `[t1,t2,..tn|X]`); need to be careful that the variable really is a variable when called.

We can write append of difference lists simply by using a different representation. Let's take `Z/X` for a difference list (where `Z=[t1,t2,..tn]`) above; here / is already available as an infix operator. Then difference list append is simply:

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

## Difference Lists ctd

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;
so this will not give all solutions in mode `dl_append(-,-,+)`.

```
?- 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
```

## Difference lists reverse

Compare:

```
%% basic reverse, no optimisation

naive_reverse([],[]).
naive_reverse([X|Xs],Ys) :- naive_reverse(Xs,Rs),
                            append(Rs,[X],Ys).

%% difference lists used in second argument
%% of reverse_dl

reverse_dl([],T\T).
reverse_dl([X|Xs],Rs\T) :- reverse_dl(Xs,Rs\[X|T]).
```

## *An even simpler CFG (again)*

```
T  -> c
T  -> aTb
```

In Prolog, with difference lists:

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

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

# Definite Clause Grammars

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

```
t --> [c].

t --> [a], t, [b].
```

translates to:

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

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

# DCG syntax

- 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`

# Using DCG version

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]
```

## using DCG ctd

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].
```

# DCGs with tests

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).
```

## DCGs with recursion

- 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.
```

## DCGs with parameters

- Non-terminals in DCGs can have parameters:

```
t(0) --> [c].

t(succ(N)) --> [a], t(N), [b]
```

  - Can keep track of depth of nesting in terms.

With parameters, we go outside the expressiveness of CFGs.

```
u(N) --> n(N,a),
         n(N,b),
         n(N,c).
n(0,X) --> [].

n(succ(N),X) --> [X], n(N,X).
```

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

# Parsing with parse trees

- "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 of DCGs to Prolog

Next time:

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