Logic Programming

Lecture 6:
Parsing, Difference Lists and Definite Clause Grammars

Outline for today

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

Context Free Grammars

- A simple context free grammar

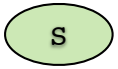
```
S ⟶ NP VP
NP ⟶ DET N
VP ⟶ VI | VT NP
DET ⟶ the
N ⟶ cat | dog | food
VI ⟶ meows | barks
VT ⟶ bites | eats
```


Recognizing grammatical sentences

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

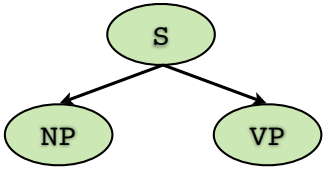
Generation example

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



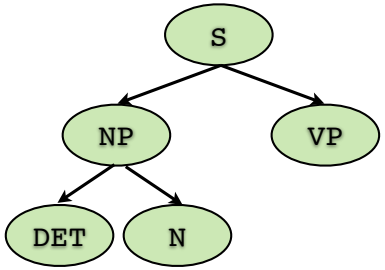
Generation example

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



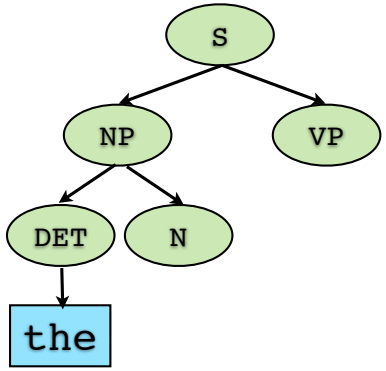
Generation example

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



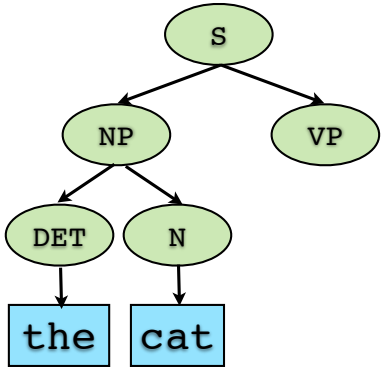
Generation example

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



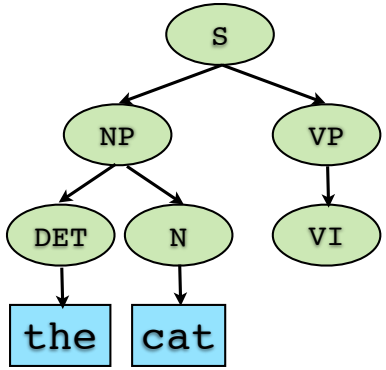
Generation example

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



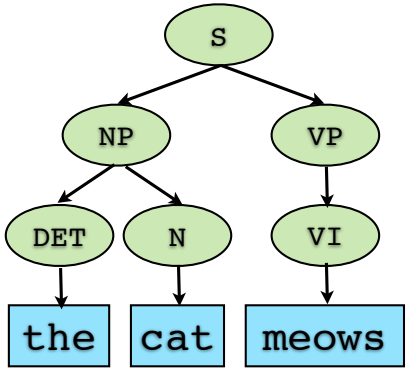
Generation example

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



Generation example

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



An even simpler context-free grammar

T → c T → a T b

- In Prolog:

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

Using append to parse

```
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?

- Obviously, need to guess when we're generating
  - But we **also** guess when we're parsing **known** sequence
- This is inefficient (lots of backtracking!)
  - Reordering goals doesn't help much.

An idea: Accumulators

- Want to use input data to guide search
- Parse using two parameters `L,M`
  - such that `M` is a suffix of `L`
- `L` = list "before" parsing item
- `M` = "remainder" of list after parsing item

An even simpler context-free grammar (mark II)

`T` ⟶ `c` `T` ⟶ `a T b`

- In Prolog, using accumulators:

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

Using accumulator version

```
?- 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 $[t_1,...,t_n|X]$
  - `X` is a variable
- Empty difference list: `(X,X)`
- Constant difference list: $([a_1,...,a_n|X],X)$
- Appending difference lists `(t,X)` and `(u,Y)`:
  - Simply unify `X` and `u`! Yields `(t[u/X],Y)`

An even simpler context-free grammar (mark III)

`T` ⟶ `c` `T` ⟶ `a T b`

- In Prolog, using 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 difference lists is so useful that Prolog has built-in syntax for it:

```
s --> [c].
s --> [a], S, [b].
```

- translates to:

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

DCG syntax

- Rules of form *nonterm* `-->` *body*
- Body terms are:
  - **terminal lists** `[`t_1`,...,`t_n`]` (may be `[]`)
  - **nonterminals** *s*,*t*,*u*,....
  - **sequential composition** $body_1$`,` $body_2$
  - **alternative choice** $body_1$ `;` $body_2$

Using DCG version

- DCGs translated to difference list implementation
- So they are used the same way:

```
?- t(L,[]).
L = [c].
L = [a,c,b].
L = [a,a,c,b,b].
...
```


Using DCG version (II)

- Can also use built-ins `phrase/2, phrase/3`

```
?- phrase(t,L,M).
L = [c|M].
L = [a,c,b|M].
?- phrase(t,L).
L = [c].
L = [a,c,b].
```


Large example revisited

```
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**:
  - *{pred}* where *pred* is an arbitrary Prolog goal
- Example:

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


DCGs and recursion

- Left recursion (as usual) leads to nontermination:

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

- Avoid by using right recursion & fallthrough:

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


DCGs with parameters

- Nonterminals in DCGs can have parameters.

```
t(0) --> [c].
t(succ(N)) --> [a], t(N), [b].
```

- Keeps track of depth of nesting.

DCGs + parameters > CFGs

- With parameters we can parse non-CFGs:

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

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 in a tutorial exercise...

- Further reading:
  - LPN, ch. 7-8 - more difference list examples and translation to Prolog
- Next time:
  - Search techniques
  - depth-first, iterative deepening, breadth-first, best-first