Definite clause grammars
Implementing context-free grammars

L545
Dept. of Linguistics, Indiana University
Spring 2013

---

Representing context-free grammars

Towards a basic setup:
- What needs to be represented?
- Logic programming: Prolog
- On the relationship between context-free rules and logical implications
- A first Prolog encoding
- Encoding the string coverage of a node: From lists to difference lists
- Adding syntactic sugar:
  - Definite clause grammars (DCGs)
  - Representing simple English grammars as DCGs

What needs to be represented?

We need representations (data types) for:
- terminals, i.e., words
- syntactic rules
- linguistic properties of terminals and their propagation in rules:
  - syntactic category
  - other properties
    - string covered ("phonology")
    - case, agreement, . . .
  - analysis trees, i.e., syntactic structures

Logic programming: Prolog

Logic programming languages are based upon mathematical logic.
- Expressions used in the language are declarative
- Expressions are then proven by a (backwards-reasoning) theorem-prover
  - If A, then B is seen as: to solve B, show A

Prolog is one such logic programming language

Expressions in Prolog

Prolog has two main expressions:
- **facts** state that something is true, e.g.,
  
green(house)
- **rules** state implications:
  
  colored(X) : -
  
green(X).

  This states that an item X is colored only if it is green. (Or: if X is green, it is colored.)

SWI-Prolog

We will use the freely-available SWI-Prolog
- [http://www.swi-prolog.org](http://www.swi-prolog.org)

Notes:
- To open prolog, type `swipl` at a terminal
- Databases of facts and predicates are stored in files ending in `.pl` (e.g., `examples.pl`)
- To load a database, use brackets, followed by a full stop:
  
  `?- [examples].`

  % examples compiled 0.00 sec, 7 clauses true.
Querying the database

You can query the database of facts and rules to see if something is true.

- You can ask if something is true:
  
  ```?- green(house).```
  Yes.

- Or you can ask which things are green:
  
  ```?- green(X).```
  ```X = house```  

Evaluation

Multiple facts

```paint(house,green).
paint(car,green).```  

If I query `paint(X,green)`, Prolog can return 2 answers for X.

- For the 2-argument predicate `paint` (sometimes written `paint/2`), there is a choice point.

First-argument indexing

Prolog actually works by indexing on its first argument

```paint(house,green).
paint(car,blue).```  

It makes a difference as to which argument is not a variable (uninstantiated):

- `paint(house,X)` — Prolog immediately knows that `house` only has a predicate
- `paint(X,green)` — Prolog doesn’t realize that there is only one matching predicate until it has checked all of them.

Lesson: put the most informative item first

Evaluating rules

Prolog does the same evaluation when rules (implications) are uses

```sibling(X,Y) :- parent(Z,X), parent(Z,Y).
parent(john,susan).
parent(john,polly).```  

`sibling(susan,polly)` is true because

parent(Z,susan) and parent(Z,polly) are true when Z = john

Recursion

Evaluations in Prolog involve proving statements by recursing through rules

```parent(john,paul).
parent(paul,tom).
parent(tom,mary).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Y,Z), ancestor(X,Z).```  

The query `ancestor(john,tom)` involves a recursive search through different rules.

http://www.doc.gold.ac.uk/~mas02gw/prolog/tutorial/prologpages/recursion.html
Recursion (2)

What happens when we query `ancestor(john,tom)`?

- Prolog checks the first definition of `ancestor/2` and fails since there is no predicate `parent(john,tom)`.
- Prolog goes back to the choice point and checks the second definition:
  - With `X = john`, the only thing that will work is `Z = paul` (first argument indexing).
  - Prolog checks to see whether `ancestor(paul,tom)` is true.

Lists

Prolog has a list data structure, represented by `[ ... ]`

- `[]` = the empty list
- `[A] = [A|[]]`

Looping

Lists & recursive predicates result in looping:

```
my_length([],0).
% _ is a variable we never use again
my_length([_|T], N) :-
    my_length(T,M),
    N is M + 1.
```

Example of querying:

```
?- my_length([a,b,c],N).
N = 3.
```

Append

We need a way to join, or append, two lists together:

Prolog has such a built-in predicate, `append/3`, which can take lists `L1` and `L2`, and return the joined list `L3`.

```
append([],L2,L2).
append([H1|T1], L2, [H1|L3]) :-
    append(T1,L2,L3).
```

Example call:

```
?- append([a,b,c],[d,e,f],X).
X = [a, b, c, d, e, f].
```

On the relationship between context-free rules and logical implications

- Take the following context-free rewrite rule:
  
  \[ S \rightarrow NP \ VP \]
  
- Nonterminals in such a rule can be understood as predicates holding of the lists of terminals dominated by the nonterminal.
- A context-free rules then corresponds to a logical implication:
  
  \[ \forall X \forall Y \forall Z NP(X) \land VP(Y) \land append(X,Y,Z) \Rightarrow S(Z) \]
  
  where `X`, `Y`, & `Z` refer to string yields
- Context-free rules can thus directly be encoded as logic programs.

Components of a direct Prolog encoding

- terminals: unit clauses (facts)
- syntactic rules: non-unit clauses (rules)
- linguistic properties:
  - syntactic category: predicate name
  - other properties: predicate's arguments, distinguished by position
    - in general: compound terms
    - for strings: list representation
- analysis trees: compound term as predicate argument
A small example grammar $G = (N, \Sigma, S, P)$

$N = \{ S, NP, VP, V_i, V_t, V_s \}$

$\Sigma = \{ a, clown, Mary, laughs, loves, thinks \}$

$S = \{ S \rightarrow NP \rightarrow VP \}
\begin{align*}
S & \rightarrow NP \rightarrow VP \\
VP & \rightarrow V_i \\
VP & \rightarrow V_t NP \\
VP & \rightarrow V_s S \\
V_i & \rightarrow laughs \\
V_t & \rightarrow loves \\
V_s & \rightarrow thinks \\
N & \rightarrow clown \\
\end{align*}$

An encoding in Prolog

s(S) :- np(NP), vp(VP), append(NP,VP,S).

vp(VP) :- vi(VP).

vp(VP) :- vt(VT), np(NP), append(VT,NP,VP).

vp(VP) :- vs(VS), s(S), append(VS,S,VP).

np(NP) :- pn(NP).

np(NP) :- det(Det), n(N), append(Det,N,NP).

pn([mary]). n([clown]). det([a]).

vi([laughs]). vt([loves]). vs([thinks]).

Recognizing a sentence

What happens with $s([mary,laughs],[])$?

Prolog responds with yes because the following predicates are true:

<s([mary,laughs],[]) :-
np([mary,laughs],[laughs]), vp([laughs],[ ]).

vp([laughs],[ ]) :- vi([laughs],[ ]).

np([mary,laughs],[laughs]) :-
np([mary,laughs],[laughs]).

pn([mary],[laughs]).

vi([laughs]), vs([laughs]), vs([laughs]).></s>

Definite clause grammars (DCG)

Basic DCG notation for encoding CFGs

Prolog has a special notation for CFGs

A definite clause grammar (DCG) rule has the form

$LHS \rightarrow RHS.$

$LHS$: a Prolog atom encoding a non-terminal, and

$RHS$: a comma separated sequence of

Prolog atoms encoding non-terminals

Prolog lists encoding terminals

When a DCG rule is read in by Prolog, it is expanded by adding the difference list arguments to each predicate.

Examples for some cfg rules in DCG notation

$S \rightarrow NP VP$

$s \rightarrow np, vp.$

$S \rightarrow NP thinks S$

$s \rightarrow np, [thinks], s.$

$S \rightarrow NP picks up NP$

$s \rightarrow np, [picks, up], np.$

$S \rightarrow NP picks NP up$

$s \rightarrow np, [picks], np, [up].$

$NP \rightarrow \epsilon$

$np \rightarrow [ ].$
An example grammar in definite clause notation

s --> np, vp.
np --> pn.
np --> det, n.
vp --> vi.
vp --> vt, np.
vp --> vs, s.

pn --> [mary]. n --> [clown].
det --> [a]. vi --> [laughs].
vt --> [loves]. vs --> [thinks].

The example expanded by Prolog

?- listing.
np(A, B) :-
  pn(A, B).
np(A, C) :-
  det(A, B),
  n(B, C).

vp(A, B) :-
  vi(A, B).
vp(A, C) :-
  vt(A, B),
  np(B, C).

s(A, C) :-
  np(A, B),
  vp(B, C).

More complex terms in DCGs

Non-terminals can be any Prolog term, e.g.:

s --> np(Per,Num),
    vp(Per,Num).

This is translated by Prolog to

s(A, B) :-
  np(C, D, A, E),
  vp(C, D, E, B).

Restriction:

- The LHS has to be a non-variable, single term (plus possibly a sequence of terminals).

Using compound terms to store an analysis tree

s(s_node(NP,VP)) --> np(NP), vp(VP).

np(np_node(PN)) --> pn(PN).
np(np_node(Det,N)) --> det(Det), n(N).

vp(vp_node(VI)) --> vi(VI).
vp(vp_node(VT,NP)) --> vt(VI), np(NP).
vp(vp_node(VS,S)) --> vs(VI), s(S).

Adding more linguistic properties

s --> np(Per,Num), vp(Per,Num).

np(Per,Num) --> vi(Per,Num).

vp(Per,Num) --> vt(Per,Num), np(_,_).

Example call

?- s(Tree,[mary,laughs],[]).
Tree = s_node(np_node(mary_node), vp_node(laugh_node))
Definite clause grammars
Representation
Prolog
CFGs
CFGs in Prolog
Difference lists
DCGs
Tracing agreement properties

?- trace.
true.

[trace] ?- s([mary, laugh], []).
Call: (6) s([mary, laugh], []) ? creep
Fail: (6) s([mary, laugh], []) ? creep
Call: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Exit: (7) np(3, sg, [mary, laugh], [laugh]) ? creep
Call: (7) vp(3, sg, [laugh], []) ? creep
Fail: (8) vp(3, sg, [laugh], []) ? creep
Fail: (7) np(3, sg, [mary, laugh], [laugh]) ? creep
Fail: (8) vi(3, sg, [laugh], []) ? creep
Fail: (8) vi(3, sg, [laugh], []) ? creep
Redo: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Fail: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Call: (8) det(_G631, [mary, laugh], _G633) ? creep
Fail: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Fail: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Fail: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Fail: (7) np(_G631, _G632, [mary, laugh], _G634) ? creep
Fail: (6) s([mary, laugh], []) ? creep
false.
?- notrace.