Definite clause grammars
Implementing context-free grammars

L545
Dept. of Linguistics, Indiana University
Spring 2016

What we want to accomplish

▶ See how to encode context-free grammars (CFGs)
▶ Begin to see how features fit in
  ▶ First step towards grammar engineering
▶ Learn just enough Prolog to support this

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

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

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.)
We will use the freely-available SWI-Prolog

- 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:
  ```prolog
  ?- [examples].
  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:
  ```prolog
  ?- green(house).
  Yes.
  ```
- Or you can ask which things are green:
  ```prolog
  ?- green(X).
  X = house
  ```

Use a semi-colon to see multiple answers

Querying different arguments

If we have:
```prolog
height(house,20).
```
Then we can query for either the height of `house` or items which have height 20
```prolog
?- height(house,X).
X = 20
?- height(X,20).
X = house
?- height(car,X).
false.
```

First-argument indexing

Prolog actually works by indexing on its first argument
```prolog
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 one 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

Evaluation

Multiple facts
```prolog
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.

Evaluating rules

Prolog does the same evaluation when rules (implications) are uses
```prolog
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(X,Z), ancestor(Z,Y).`  

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

[http://www.doc.gold.ac.uk/~mas02gw/prologtutorial/prologpages/recursion.html](http://www.doc.gold.ac.uk/~mas02gw/prologtutorial/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.

To see this, run `trace` before making the call.
- Tracing can be shut off with `notrace`.

Lists

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

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

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([H|T1], L2, [H|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 rule 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

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).
```

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]) :- pn([mary,laughs],[laughs]).
```

A small example grammar $G = (N, \Sigma, S, P)$

```
N = {S, NP, VP, V_i, V_t, V_s, Det}
\Sigma = {a, clown, Mary, laughs, loves, thinks}
S = S
P =
```

Difference list encoding

```
s(X0,Xn) :- np(X0,X1), vp(X1,Xn).
vp(X0,Xn) :- vi(X0,Xn).
vp(X0,Xn) :- vt(X0,X1), np(X1,Xn).
vp(X0,Xn) :- vs(X0,X1), s(X1,Xn).
np(X0,Xn) :- pn(X0,Xn).
np(X0,Xn) :- det(X0,X1), n(X1,Xn).
```

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 → NP VP**
  \[ s \rightarrow np, vp. \]
- **S → NP thinks S**
  \[ s \rightarrow np, [thinks], s. \]
- **S → NP picks up NP**
  \[ s \rightarrow np, [picks, up], np. \]
- **S → NP picks NP up**
  \[ s \rightarrow np, [picks], np, [up]. \]
- **NP → ε**
  \[ np \rightarrow []. \]

An example grammar in definite clause notation

\[ s \rightarrow np, vp. \]
\[ np \rightarrow pn. \]
\[ np \rightarrow det, n. \]
\[ vp \rightarrow vi. \]
\[ vp \rightarrow vt, np. \]
\[ vp \rightarrow vs, s. \]
\[ np \rightarrow [mary]. \]
\[ n \rightarrow [clown]. \]
\[ det \rightarrow [a]. \]
\[ vi \rightarrow [laughs]. \]
\[ vt \rightarrow [loves]. \]
\[ vs \rightarrow [thinks]. \]

The example expanded by Prolog

?- listing.
\[
\begin{align*}
\text{np}(A, B) & :- \\
\text{pn}(A, B).
\text{np}(A, C) & :- \\
\text{det}(A, B), \\
\text{n}(B, C).
\text{vp}(A, B) & :- \\
\text{vi}(A, B).
\text{vp}(A, C) & :- \\
\text{vt}(A, B), \\
\text{np}(B, C).
\text{vp}(A, C) & :- \\
\text{vs}(A, B), \\
\text{s}(B, C).
\text{vi}(laughs, A). \\
\end{align*}
\]

More complex terms in DCGs

Non-terminals can be any Prolog term, e.g.:
\[ s \rightarrow np(\text{Per}, \text{Num}), \\
vp(\text{Per}, \text{Num}). \]

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

\[ s(s\_node(NP, VP)) \rightarrow np(NP), vp(VP). \]
\[ np(np\_node(PN)) \rightarrow pn(PN). \]
\[ np(np\_node(Det, N)) \rightarrow det(Det), n(N). \]
\[ vp(vp\_node(VI)) \rightarrow vi(VI). \]
\[ vp(vp\_node(VT, NP)) \rightarrow vt(VT), np(NP). \]
\[ vp(vp\_node(VS, S)) \rightarrow vs(VS), s(S). \]
\[ pn(mary\_node) \rightarrow [mary]. \]
\[ n(clown\_node) \rightarrow [clown]. \]
\[ det(a\_node) \rightarrow [a]. \]
\[ vi(laughs\_node) \rightarrow [laughs]. \]
\[ vt(loves\_node) \rightarrow [loves]. \]
\[ vs(thinks\_node) \rightarrow [thinks]. \]

Example call

?- s(Tree, [mary, loves], []).
Tree = s\_node(np\_node(mary\_node), vp\_node(laugh\_node))
Adding more linguistic properties

\[
\text{s} \rightarrow \text{np}(\text{Per}, \text{Num}), \text{vp}(\text{Per}, \text{Num}). \\
\text{vp}(\text{Per}, \text{Num}) \rightarrow \text{vi}(\text{Per}, \text{Num}). \\
\text{vp}(\text{Per}, \text{Num}) \rightarrow \text{vt}(\text{Per}, \text{Num}), \text{np}(\_\_, \_). \\
\text{vp}(\text{Per}, \text{Num}) \rightarrow \text{vs}(\text{Per}, \text{Num}), \text{s}. \\
\text{np}(3, \text{sg}) \rightarrow \text{pn}. \\
\text{np}(3, \text{Num}) \rightarrow \text{det}(\text{Num}), \text{n}(\text{Num}). \\
\text{pn} \rightarrow [\text{mary}]. \\
\text{det}(\text{sg}) \rightarrow [\text{a}]. \\
\text{det}(\_\_) \rightarrow [\text{the}]. \\
\text{n}(\text{sg}) \rightarrow [\text{clown}]. \\
\text{n}(\text{pl}) \rightarrow [\text{clowns}]. \\
\text{vi}(3, \text{sg}) \rightarrow [\text{laughs}]. \\
\text{vi}(\_, \text{pl}) \rightarrow [\text{laugh}]. \\
\text{vt}(3, \text{sg}) \rightarrow [\text{loves}]. \\
\text{vt}(\_, \text{pl}) \rightarrow [\text{love}]. \\
\text{vs}(3, \text{sg}) \rightarrow [\text{thinks}]. \\
\text{vs}(\_, \text{pl}) \rightarrow [\text{think}].
\]

Tracing agreement properties

?- trace.
true.

\[
\text{[trace]} \text{?- s([mary, laugh], []).} \\
\text{Call: (6) s([mary, laugh], []) ? creep} \\
\text{Fail: (6) s([mary, laugh], []) ? creep} \\
\text{Call: (7) np(\_G631, \_G632, [mary, laugh], \_G634) ? creep} \\
\text{Fail: (7) np(\_G631, \_G632, [mary, laugh], \_G634) ? creep} \\
\text{Exit: (8) np([mary, laugh], [laugh]) ? creep} \\
\text{Exit: (7) np(3, sg, [mary, laugh], [laugh]) ? creep} \\
\text{Call: (7) vp(3, sg, [laugh], []) ? creep} \\
\text{Fail: (8) vi(3, sg, [laugh], []) ? creep} \\
\text{Fail: (8) vi(3, sg, [laugh], []) ? creep} \\
\text{Redo: (7) vp(3, sg, [laugh], []) ? creep} \\
\text{Call: (7) vt(3, sg, [laugh], \_G634) ? creep} \\
\text{Fail: (8) vt(3, sg, [laugh], \_G634) ? creep} \\
\text{Fail: (8) vt(3, sg, [laugh], \_G634) ? creep} \\
\text{Redo: (7) vp(3, sg, [laugh], []) ? creep}
\]

?- notrace.
false.

?- trace.
true.