Language, Logic and Computation
Lecture 3

Matthew Stone

Department of Computer Science
Center for Cognitive Science
Rutgers University
Parsing

Two key ideas

*Incremental structure-building*

*Exploring possible paths*
Suppose you have an *incomplete tree* $T$ up to a certain point $M$ in the word string.

Suppose you have just found a *new constituent* $C$ that picks up at $M$ where $T$ leaves off.

How do you add $C$ to $T$?
Suppose $C$ is a complement of category $XP$

Consider this sample $T$
Suppose $C$ is a complement of category $XP$

Consider this sample $T$

Can $C$ go here?
Suppose \( C \) is a complement of category \( XP \)

Consider this sample \( T \)

**Can C go here?**
Suppose \( C \) is a complement of category \( XP \)

Consider this sample \( T \)

Can \( C \) go here?
Suppose \( C \) is a complement of category \( XP \)

Consider this sample \( T \)

Can \( C \) go here?
Suppose \( C \) is a complement of category \( XP \)

Consider this sample \( T \)

Can \( C \) go here?
Coding this up

New structure

node (category,
    left,
    head,
    right)

Assume that lexical material in structure is a substring including the head.
FTHR – full to head’s right

Mutual recursion again!

Base cases – lexical nodes

\[ fthr(\text{leaf}(\_)). \]
\[ \% fthr(\text{gap}(\_)) :- \text{fail}. \]
FTHR ct’d

Recursive case: make sure there are no gaps in the right list and the head subtree is fthr.

\[
fthr(node(_,_,H,R)) :-
    \text{nogaps}(R), \ fthr(H).
\]

\[
\text{nogaps}([],\).
\]
\[
\text{nogaps}([\text{leaf}(_)|L]) :- \ \text{nogaps}(L).
\]
\[
\text{nogaps}([\text{node}(_,_,_,_,_)|L]) :- \ \text{nogaps}(L).
\]
Suppose $C$ is a complement of category $XP$

Consider this sample $T$
Substitute at next place

Replace next gap with node

replace_next([[gap(C)|Rest]],
             node(C,L,H,R),
             [[node(C,L,H,R)|Rest]]).

replace_next([[node(C,L,H,R)|Rest]], N, 
              [[node(C,L,H,R)|Result]]) :-
             replace_next(Rest, N, Result).


Suppose $C$ is a complement of category $XP$

Consider this sample $T$
Substitute at next place

subst_next(node(C,L,H,R), N, node(C,L,H,X)) :-
    fthr(H),
    replace_next(R, N, X).

subst_next(node(C,L,H,R), N, node(C,L,X,R)) :-
    subst_next(H, N, X).

Suppose \( C \) is a complement of category \( XP \)

Consider this sample \( T \)
Summary

combine(T1, T2, T3) :-
    fthr(T2),
    fthl(T2),
    subst_next(T1, T2, T3).
Parsing

Two key ideas

_Incremental structure-building_

_Exploring possible paths_
Language is ambiguous

There’s no way to just keep track of the right tree incrementally, as we parse through the string.

What we can do instead is keep track of all the alternatives.

To see how, we need to know what these alternatives look like.
Exploring possible paths

Imagine *guessing* the structure of a derivation, *top-down*.

![Diagram showing the structure of a derivation with a midpoint between the left and right subparts, and the whole string.]
Exploring possible paths

You can put the *midpoint* after any word.
Exploring possible paths

After you guess, you break the smaller segments up, recursively.
Now imagine putting those pieces together, *bottom up*

You have all the smaller pieces already.

By running through the possible midpoints, you can find all the ways of putting those pieces together.
Visualization
Parsing algorithm $CKY$

Work your way forward through the string
Parsing algorithm *CKY*

At each stage, work backwards to build larger substrings
 Parsing algorithm \textit{CKY} \\

By exploring alternative midpoints
This builds a complete table

At this stage, we have all the elements we need from earlier rounds

we got this when we considered shorter prefixes

we got this earlier in considering this prefix
Summary

For end = 2 up to n
   For start = end-2 down to 0
      For mid = start+1 up to end-1
         Combine (start-mid) (mid-end)
Prolog implementation

Store facts in the knowledge base for constituents, using \texttt{assert}

Use predicate \texttt{chart(St, End, Tree)} for results
Prolog implementation

Innermost loop

loop(Start, End) :-
    chart(Start, Mid, T1),
    chart(Mid, End, T2),
    combine(T1, T2, T3),
    assert(chart(Start, End, T3)),
    fail.
loop(Start, End).
Prolog implementation

Outer loops

backward(Start, End) :-
   Start < 0, !.
backward(Start, End) :-
   loop(Start, End),
   Next is Start - 1,
   backward(Next, End).
Prolog implementation

Outer loops

\[
\text{forward}(\text{End}, \text{Max}) :\text{-} \\
\text{End} > \text{Max}, !. \\
\text{forward}(\text{End}, \text{Max}) :\text{-} \\
I \text{ is } \text{End} - 2, \\
\text{backward}(I, \text{End}), \\
\text{Next is } \text{End} + 1, \\
\text{forward}(\text{Next}, \text{Max}).
\]