CS 533
Natural Language Processing
Lecture 7 – March 24, 2003

Matthew Stone & David DeVault

THE VILLAGE

Department of Computer Science
Center for Cognitive Science
Rutgers University

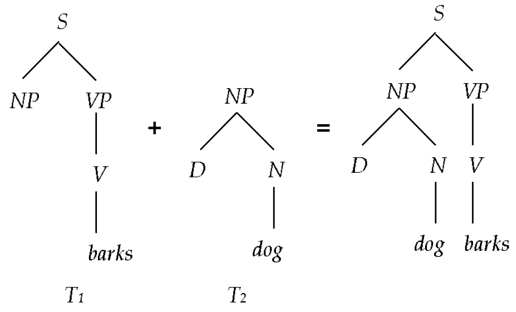
Parsing

Outline
- The periphery – right and left frontier
- Stack and search
- Charts and efficiency
- Feature structures
- Semantics and interpretation
- Adding statistical information

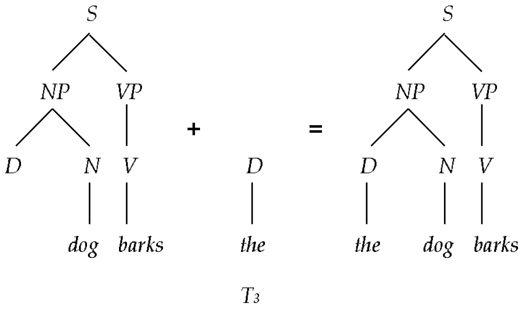
Parsing problem

Use the grammar to derive an explicit representation of the constituency of an input sentence.

Recall: Grammatical derivations Tree Substitution Grammar

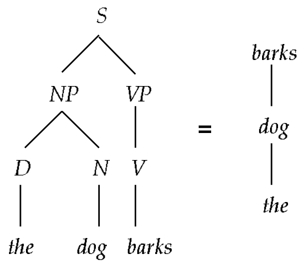


Recall: Grammatical derivations Tree Substitution Grammar



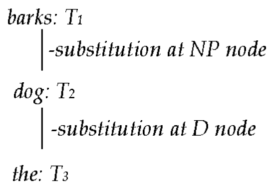
Dependency

Sentences built from words by operations



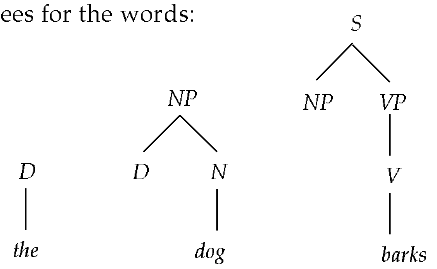
Dependency formalisms

Tree Substitution Grammar


barks: T1
-substitution at NP node
dog: T2
-substitution at D node
the: T3

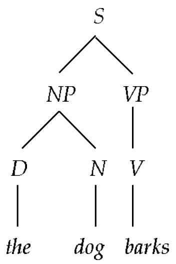
Parsing problem

Start from words and possible elementary trees for the words:



Parsing problem

Determine a derived tree whose leaves are the input words in order and which is derived from possible trees for the words



Key idea of parsing

Incremental structure building

Adapt steps of grammatical derivation to keep track of the ***order*** of consituents.

Incremental structure building

You have two things next to each other that you can combine:

an incomplete constituent with room for an X next | *a complete new X you can add in*

and you combine them:

a more complete constituent that combines the two together

Incremental structure building – symmetric case

You have two things next to each other that you can combine:

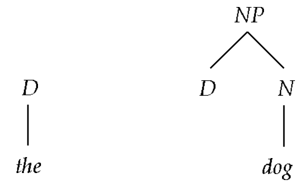
a complete new X you can add in | *an incomplete constituent with room for an X just before*

and you combine them:

a more complete constituent that combines the two together

Key idea of parsing

Incremental structure building

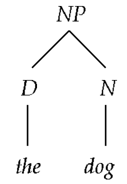


a complete new D you can add in

an incomplete NP with room for a D just before

Key idea of parsing

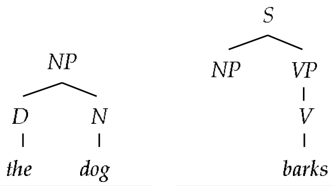
Incremental structure building



a new complete NP that combines the two together

Key idea of parsing

Incremental structure building

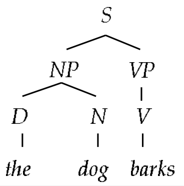


a complete new NP you can add in

an incomplete S with room for an NP just before

Key idea of parsing

Incremental structure building



a new complete S that combines the two together

Adding things to trees – the periphery

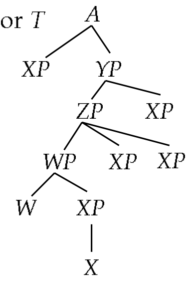
Where is the *XP* in that first tree *T*?

an incomplete constituent with room for an XP next

a complete new XP you can add in

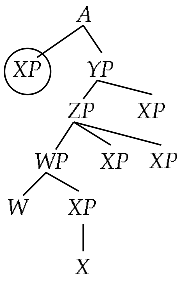
The periphery

Consider this example for *T*



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

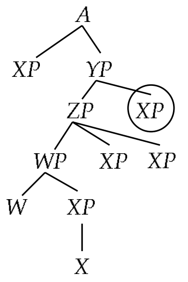
Consider this sample ***T***



Can new XP go here?

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

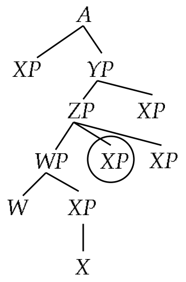
Consider this sample ***T***



Can new XP go here?

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

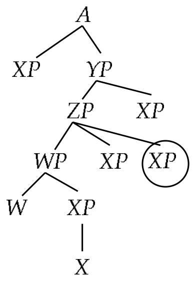
Consider this sample ***T***



Can new XP go here?

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

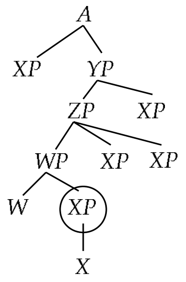
Consider this sample ***T***



Can new XP go here?

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

Consider this sample ***T***



Can new XP go here?

Periphery

A completed constituent on the ***right*** can only be added to an incomplete tree on its ***right periphery*** – after the last word already accounted for in the incomplete tree and up to the first gap not yet accounted for.

Periphery

A completed constituent on the ***left*** can only be added to an incomplete tree on its ***left periphery*** – before the first word already accounted for in the incomplete tree and up to the last gap not yet accounted for.

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(leaf(_)).
% fthr(gap(_)) :- fail.
```

FTHR ct’d

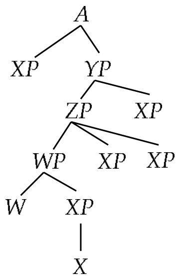
Recursive case: make sure there are no gaps in the right list and the head subtree is fthr.

```
fthr(node(_,_,H,R)) :-
  nogaps(R), fthr(H).

nogaps([]).
nogaps([leaf(_)|L]) :- nogaps(L).
nogaps([node(_,_,_,_)|L]) :- nogaps(L).
```

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

Consider this sample ***T***



Substitute at next place

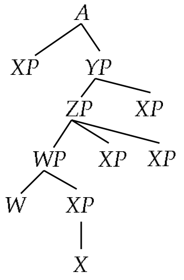
Replace next gap with node

```
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***



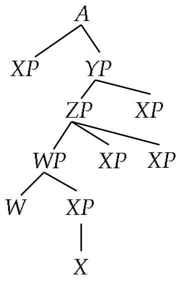
Substitute at next place

```
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).
```

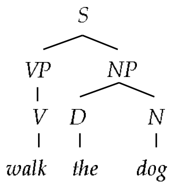
Parsing

Our basic strategy will be to use this parsing combination operation to add complete constituents that we find into incomplete constituents that we are still building.

What kind of data structures do we need to keep track of all these constituents?

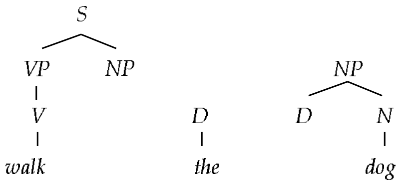
Parsing – stack and search

Consider this example



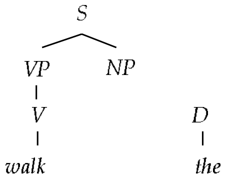
Parsing – stack and search

Consider this example during parsing



Stack

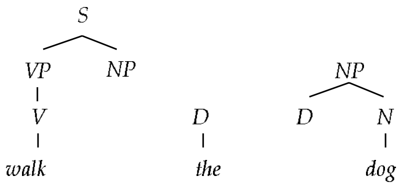
Walk does not combine directly with *the*



We have to ***push walk*** and see what we can do with *the.*

Stack

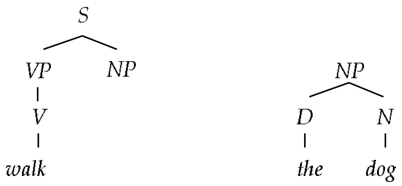
We now consider *the* and *dog*



(*Walk* is still on the stack.) We simplify.

Stack

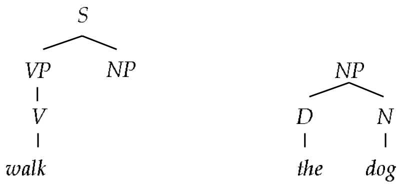
We now consider *the* and *dog*



(*Walk* is still on the stack.) We simplify.

Stack

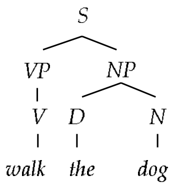
Now look at combinations using the stack:



We can simplify again.

Stack

Now look at combinations using the stack:



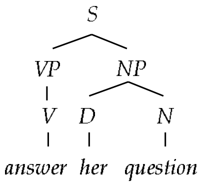
We can simplify again.

Stack

In general, we have to postpone consideration of larger, earlier incomplete constituents while we assemble smaller, later constituents

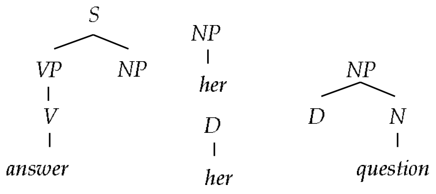
Parsing – stack and search

Consider this example



Parsing – stack and search

Consider this example during parsing



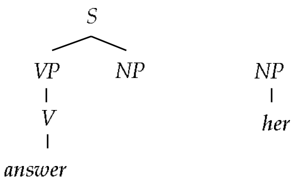
Language is ambiguous

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

We somehow need to try ***all*** the alternatives.

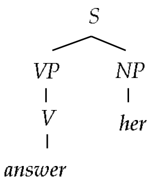
Parsing – search

Try this combination first, perhaps:



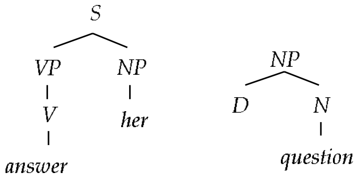
Parsing – search

Try this combination first, perhaps:



Parsing – search

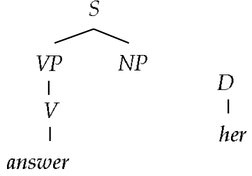
Now need this combination, somehow:



Crash!

Parsing – search

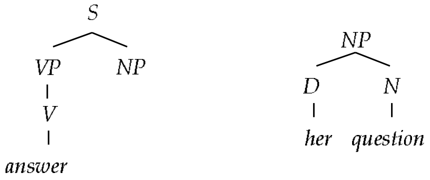
Backtrack to the last choice, and consider another possibility:



Delay *answer* – no combo possible here.

Parsing – search

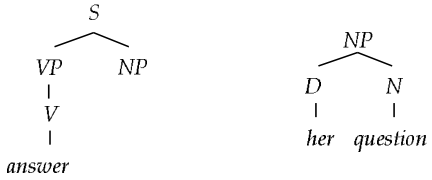
Consider the next word:



Make the combination.

Parsing – search

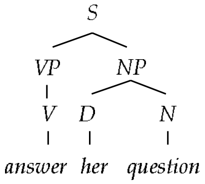
Consider the next word:



Reconsider elements from the stack.

Parsing – search

Consider the next word:



Reconsider pending open constituents.

Parsing – chart and efficiency

Backtracking search is expensive, but in fact it makes the parser do the ***same work*** over and over.

In backtracking, if you change your mind about word *N*, you have to ***reanalyze*** everything after *N*.

But in this reanalysis, you'll just find the ***same*** complete constituents starting after *N*.

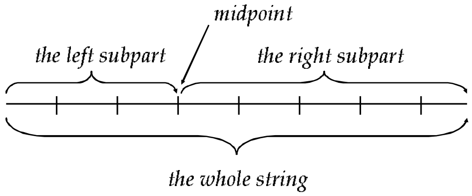
Parsing – chart and efficiency

What you can do instead is ***store*** all the smaller constituents you find, in a structure called the ***chart***.

Then when you go to build larger constituents, you look in the chart for smaller constituents, rather than searching to derive them.

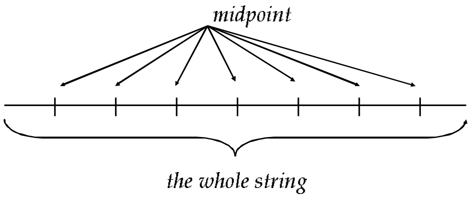
Exploring possible paths

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



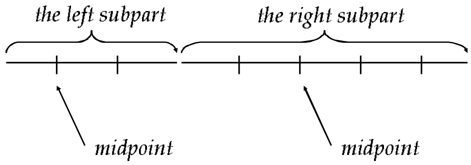
Exploring possible paths

You can put the ***midpoint*** after any word.



Exploring possible paths

After you guess, you break the ***smaller*** segments up, ***recursively***.

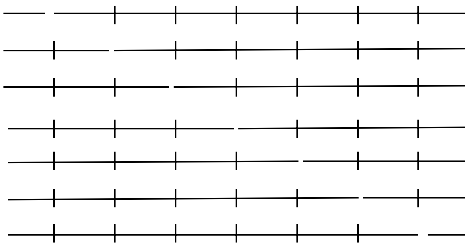


Now imagine putting those pieces together, *bottom up*

You have all the smaller pieces already.

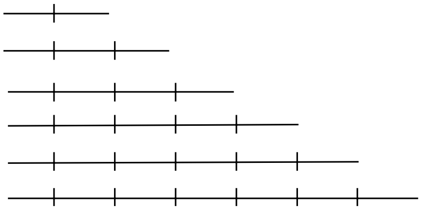
By running through the possible midpoints, you can find all the ways of putting those pieces together.

Visualization



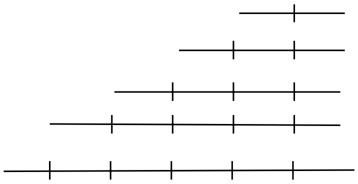
Parsing algorithm *CKY*

Work your way forward through the string



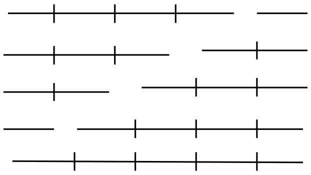
Parsing algorithm *CKY*

At each stage, work backwards to build larger substrings



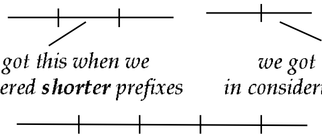
Parsing algorithm *CKY*

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 `assert`

Use predicate `chart(St,End,Tree)` for results

Prolog implementation

Setting up for chart parsing:

`setup_words(Words)` – true if for each word W at position P in Words, all trees T lexicalized to W have been asserted as:

```
chart(P,P+1,T).
```

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 loop – edges beginning at Start or earlier and ending at End.

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

Prolog implementation

Outer loop – edges ending somewhere between End and Max.

```
forward(End, Max) :-
  End > Max, !.
forward(End, Max) :-
  I is End – 2,
  backward(I, End),
  Next is End+1,
  forward(Next, Max).
```

Prolog implementation

Main parse rule:

```
parse(Words,T) :-
  retractall(chart(_,_,_)),
  length(Words,N),
  setup_words(Words),
  forward(2,N),
  chart(0,N,T).
```

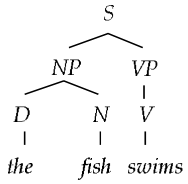
Feature structures

How does parsing interact with linguistic representations? How can you improve linguistic representations for parsing?

Feature structures provide a way of stating linguistic constraints concisely and allowing the parser to collapse together ambiguities.

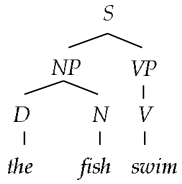
Feature structures

Motivation



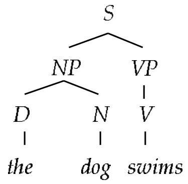
Feature structures

Motivation



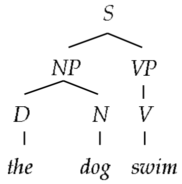
Feature structures

Motivation



Feature structures

But not



Feature structures

English has subject-verb agreement.

- ***number*** is either singular or plural
- ***person*** is either first, second or third
- ***nouns*** are always third person and may be singular or plural
- (first person is *I/we,* second person is *you*)
- ***verbs*** in third person present agree with number of the subject

---

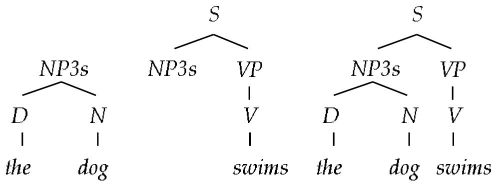
Feature structures

Really English needs different categories of noun phrase for each of these cases.

That means different trees, different parses when different categories are used.

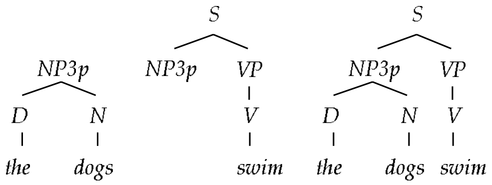
Feature structures

For example



Feature structures

For example



Feature structures – notation

$\begin{pmatrix} \text{number} & \text{sing} \\ \text{person} & \text{third} \end{pmatrix}$ dog

$\begin{pmatrix} \text{number} & \text{plural} \\ \text{person} & \text{third} \end{pmatrix}$ dogs

Feature structures – notation

$\begin{pmatrix} \text{number} & \text{sing} \\ \text{person} & \text{third} \end{pmatrix}$ swims

$\begin{pmatrix} \text{number} & \text{plural} \\ \text{person} & \text{third} \end{pmatrix}$ swim

Feature structures – data structures

Use prolog terms to represent feature structures:

f(Number,Person)

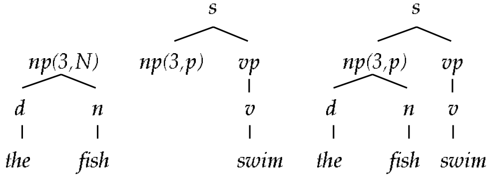
Use unification to enforce feature constraints.

Examples:

f(sing,third)	dog
f(plural,third)	dogs
f(sing,third)	swims

Another idea – categories as complex data structures

use ***terms*** for categories, and allow variables and unification.

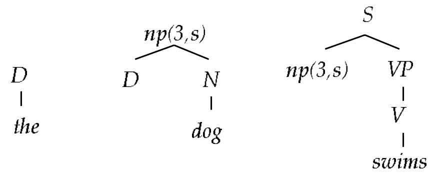


Semantics and interpretation

How does parsing interact with linguistic representations? How can you improve linguistic representations for parsing?

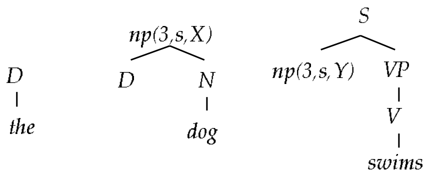
Adding semantic variables to our syntactic representations

Examples:



Adding semantic variables to our syntactic representations

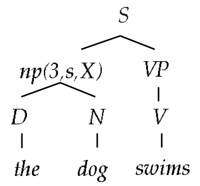
Examples:



dog(X) swims(Y)

Semantic compositionality

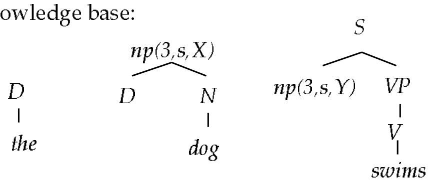
Conjoining formulas and unifying variables:



dog(X) & swims(X)

Incremental interpretation

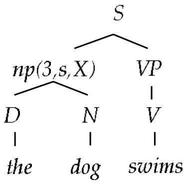
As we parse, we can keep track of which assignments to semantic variables are compatible with a knowledge base:



dog(X) swims(Y)

dog(spot) swims(spot)
dog(rover) swims(lassie)

Incremental interpretation



dog(X) & swims(X)

dog(spot) & swims(spot)

Incremental interpretation – data structures

We can insert semantic constraints and their interpretations into our chart entries:

```
chart(St,End,Tree,Constr,Interp)
```

Example:

```
Constr= [dog(X),swims(X)]
Interp= [[dog(spot),swims(spot)]]
```

Prolog implementation

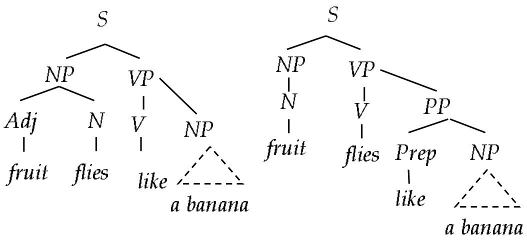
Updated innermost loop:

```
loop(Start,End) :-
  chart(Start, Mid, T1, C1, I1),
  chart(Mid, End, T2, C2, I2),
  combine(T1, T2, T3),
  append(C1,C2,C3),
  findall(C3,(member(C1,I1),
             member(C2,I2)),
         I3),
  assert(chart(Start,End,T3,C3,I3)),
  fail.
loop(Start, End).
```

Adding statistical information

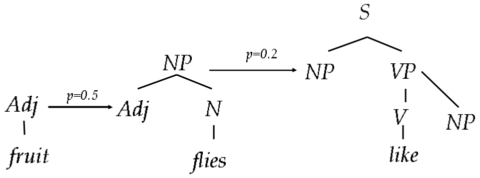
Consider the sentence:

“Fruit flies like a banana.”



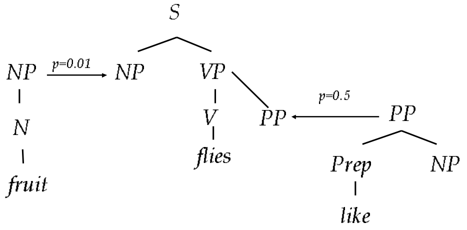
Adding statistical information

We can associate probabilities with each step in a derivation:



Adding statistical information

We can associate probabilities with each step in a derivation:



Adding statistical information

The probability of each step in a derivation can be estimated from a corpus of correct derivations.

A probability score is associated with each chart entry according to the probabilities of the steps needed to derive it.

These probabilities may be used to guide the search for a complete parse.