Context-free grammars

Computational Linguistics (LING 455)

Rutgers University　　　Nov 9 & 12

Introducing CFGs

What CFGs make possible

```
*W11> s0 = words "Mary saw John with the binoculars"

*W11> parsed = parseToTree eng s0

*W11> displayForest parsed
```

```
S
+- DP
|  `- Mary
`- VP
   +- VP
   |  +- VT
   |  |  `- saw
   |  `- DP
   |     `- John
   `- PP
      +- P
      |  `- with
      `- DP
         +- D
         |  `- the
         `- NP
            `- binoculars
```

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$X \longrightarrow ab \qquad (1)$$
$$X \longrightarrow aXb \qquad (2)$$

Here is an example derivation using this CFG:

$$X$$

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$X \longrightarrow ab \qquad (1)$$
$$X \longrightarrow aXb \qquad (2)$$

Here is an example derivation using this CFG:

$$X$$
$$aXb \qquad (2)$$

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$
\begin{aligned}
X &\longrightarrow ab && (1)\\
X &\longrightarrow aXb && (2)
\end{aligned}
$$

Here is an example derivation using this CFG:

$$
\begin{array}{cc}
X & \\
aXb & (2)\\
aaXbb & (2)
\end{array}
$$

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$X \longrightarrow ab \qquad (1)$$
$$X \longrightarrow aXb \qquad (2)$$

Here is an example derivation using this CFG:

$$\begin{array}{cc} X & \\ aXb & (2) \\ aaXbb & (2) \\ aaaXbbb & (2) \end{array}$$

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$\mathrm{X} \longrightarrow \mathrm{ab} \qquad (1)$$
$$\mathrm{X} \longrightarrow \mathrm{aXb} \qquad (2)$$

Here is an example derivation using this CFG:

$$\begin{array}{cc} \mathrm{X} & \\ \mathrm{aXb} & (2) \\ \mathrm{aaXbb} & (2) \\ \mathrm{aaaXbbb} & (2) \\ \mathrm{aaa}\color{red}{\mathrm{ab}}\mathrm{bbb} & (1) \end{array}$$

This is a grammar for…

Our first CFG

This CFG says that the symbol X can be **rewritten** as ab, or as aXb.

$$
\begin{aligned}
X &\to ab && (1)\\
X &\to aXb && (2)
\end{aligned}
$$

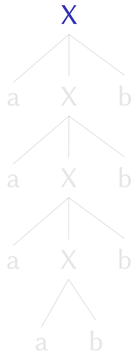
Here is an example derivation using this CFG:

$$
\begin{array}{cc}
X & \\
aXb & (2)\\
aaXbb & (2)\\
aaaXbbb & (2)\\
aaaabbbb & (1)
\end{array}
$$

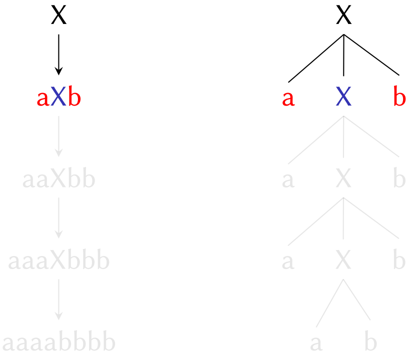
This is a grammar for…our old friend $\{a^n b^n \mid n \geqslant 1\}$!

Viewing derivations as trees

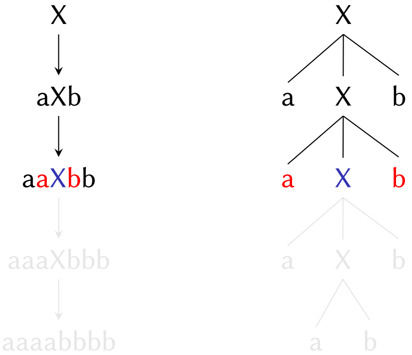
$$X \to aXb \to aaXbb \to aaaXbbb \to aaaabbbb$$



Viewing derivations as trees

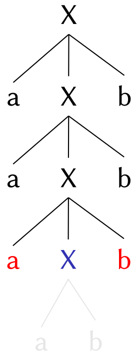


Viewing derivations as trees



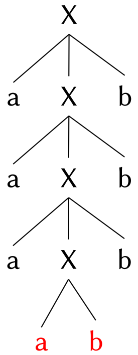
Viewing derivations as trees

$$X \rightarrow aXb \rightarrow aaXbb \rightarrow aaaXbbb \rightarrow aaaabbbb$$

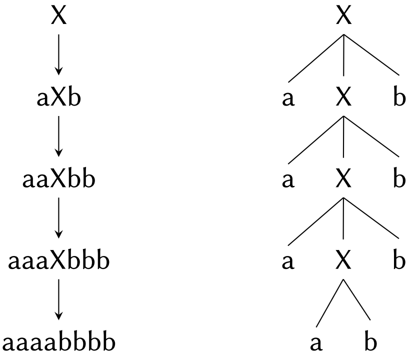


Viewing derivations as trees

X
↓
aXb
↓
aaXbb
↓
aaaXbbb
↓
aaaabbbb



Viewing derivations as trees



A tree constitutes **a proof** that a grammar generates a string.

- Trees can be read bottom-up too: “if I have ab, I have X, . . . ”

Formal definition

A context-free grammar (CFG) G is a 4-tuple (N, Σ, I, R) where:

- Σ is a set of **terminal** symbols — the alphabet
- N is a set of **nonterminal** symbols — the categories
- $I \subseteq N$ is the set of **initial** non-terminals
- R is a set of **rules**

Formal definition

A context-free grammar (CFG) G is a 4-tuple (N, Σ, I, R) where:

- Σ is a set of **terminal** symbols — the alphabet
- N is a set of **nonterminal** symbols — the categories
- $I \subseteq N$ is the set of **initial** non-terminals
- R is a set of **rules**

And what's a rule? Just a pair (A, ω) (which we write '$A \to \omega$') where:

- $A \in N$ — LHS is a non-terminal
- $\omega \in (N \cup \Sigma)^*$ — RHS is a sequence of stuff in N or Σ

Then we say '$u \overset{G}{\Longrightarrow} v$' to mean that we can derive v by rewriting symbols in u according to the rules of G. ⊣

Our previous example, mathematized

The CFG for balanced strings of a's and b's…

$$\begin{aligned} X &\longrightarrow ab & (1) \\ X &\longrightarrow aXb & (2) \end{aligned}$$

…Is really the mathematical object (N, Σ, I, R) where:

- $\Sigma = \{a, b\}$ — terminals
- $N = \{X\}$ — non-terminals/categories
- $I = \{X\}$ — initial non-terminals/categories
- $R = \{(X, ab), (X, aXb)\}$ — rules

Some terminology

Why do we call such grammars **"context-free"**?

- The LHS of a rewrite rule $A \to \omega$ does not consider the context in which A might be situated. It can apply regardless.
- If we had allowed rules like $\varphi A \psi \to \varphi \omega \psi$, our grammar would be context-**sensitive**.

Some terminology

Why do we call such grammars “**context-free**”?

- The LHS of a rewrite rule $A \longrightarrow \omega$ does not consider the context in which A might be situated. It can apply regardless.
- If we had allowed rules like $\varphi A \psi \longrightarrow \varphi \omega \psi$, our grammar would be context-**sensitive**.

When does a grammar display **recursion**?

- When the LHS of a rule occurs in the RHS too (**direct** recursion).
  - X $\longrightarrow$ aXb
  - Compare: `fac n = n * fac (n-1)`
- When rewriting the RHS produces the LHS (**indirect** recursion).
  - X $\longrightarrow$ aY; Y $\longrightarrow$ Xb
  - Compare: `even n = odd (n-1); odd n = even (n-1)`

Equivalent derivations

Here is a simple, somewhat more linguistic, CFG:

$$
\begin{aligned}
&\mathrm{NP} \rightarrow \mathrm{NP} \text{ and } \mathrm{NP}\\
&\mathrm{NP} \rightarrow \mathrm{NP} \text{ or } \mathrm{NP}\\
&\mathrm{NP} \rightarrow \text{eggs}\\
&\mathrm{NP} \rightarrow \text{toast}\\
&\mathrm{NP} \rightarrow \text{bacon}
\end{aligned}
$$

We have 2 ways of deriving *eggs and toast*:

Equivalent derivations

Here is a simple, somewhat more linguistic, CFG:

$$\begin{aligned} &\text{NP} \longrightarrow \text{NP and NP} \\ &\text{NP} \longrightarrow \text{NP or NP} \\ &\text{NP} \longrightarrow \text{eggs} \\ &\text{NP} \longrightarrow \text{toast} \\ &\text{NP} \longrightarrow \text{bacon} \end{aligned}$$

We have 2 ways of deriving *eggs and toast*:

$\underline{\text{NP}}$	$\underline{\text{NP}}$
$\underline{\text{NP}}$ and NP	NP and $\underline{\text{NP}}$
eggs and $\underline{\text{NP}}$	$\underline{\text{NP}}$ and toast
eggs and toast	eggs and toast

These derivations seem like different ways of doing the same thing.

Different derivations — ambiguity

There are many ways of deriving *eggs and toast or bacon*. Eg.:

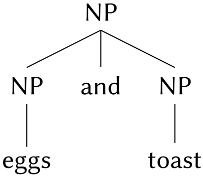
NP	NP
NP and NP	NP or NP
NP and NP or NP	NP and NP or NP
eggs and NP or NP	eggs and NP or NP
eggs and toast or NP	eggs and toast or NP
eggs and toast or bacon	eggs and toast or bacon

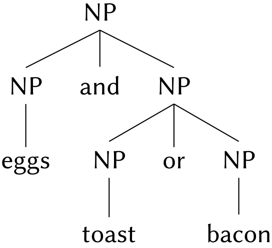
These derivations seem meaningfully different:

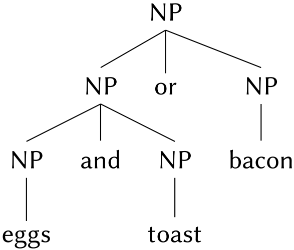
- The one on the L treats *toast or bacon* as an NP
- The one on the R treats *eggs and toast* as an NP

Trees tell us what we need to know

NP → NP and NP
NP → NP or NP
NP → eggs
NP → toast
NP → bacon







Trees

Trees abstract away from the time-course of a derivation, and thereby represent equivalence classes of derivations.

- A tree represents one **analysis** of a string (in a grammar)

When a string has multiple analyses (as distinct from multiple derivations), we say it is **ambiguous** (again, in a grammar).[1]

When a tree treats a string as a unit (as a node of the tree), we call that string a **constituent**. In *eggs and toast or bacon…*

- *toast or bacon* is a constituent in one analysis
- *eggs and toast* is a constituent in the other

[1] Note that syntactic ambiguity need not necessitate *semantic* ambiguity, as in *eggs and toast and bacon.*

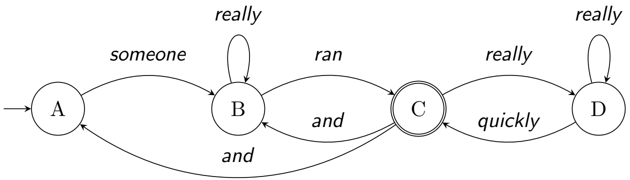
FSAs as rewrite systems

The distinction btw CFGs and FSAs doesn't lie in the use of rewrite systems, nor in the notion of constituency.

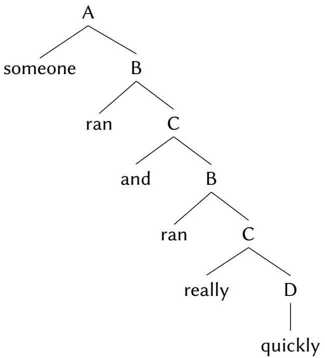
FSAs can be given via rewrite rules restricted to 1 of 2 shapes:

- $A \rightarrow x$ — A a non-terminal, x a terminal
- $A \rightarrow xB$ — A, B non-terminals, x a terminal

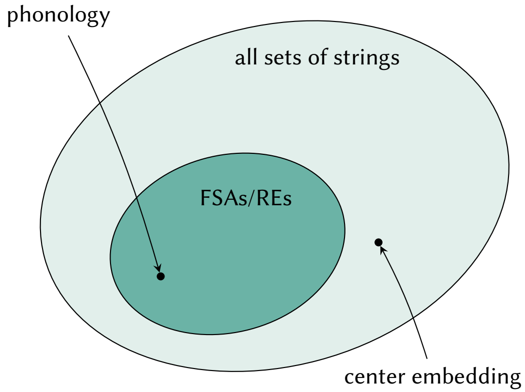
FSAs as rewrite systems (cont)



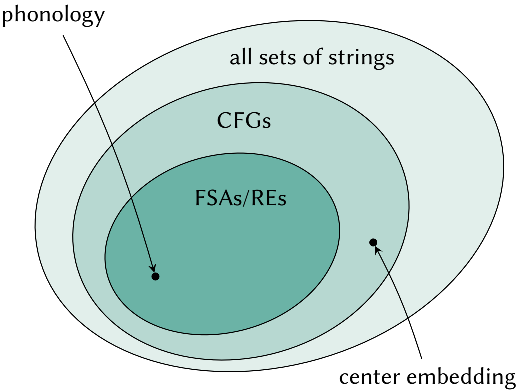
$A \longrightarrow$ someone B
$B \longrightarrow$ really B
$B \longrightarrow$ ran C
$B \longrightarrow$ ran
$C \longrightarrow$ and A
$C \longrightarrow$ and B
$C \longrightarrow$ really D
$D \longrightarrow$ really D
$D \longrightarrow$ quickly C
$D \longrightarrow$ quickly



Hierarchy of language types



Hierarchy of language types



Parsing

CNF

To make things a bit smoother, we will restrict ourselves to CFGs in **Chomsky Normal Form** (CNF):

- $A \rightarrow BC$ — A, B, C are non-terminals
- $A \rightarrow x$ — A a non-terminal, x a terminal

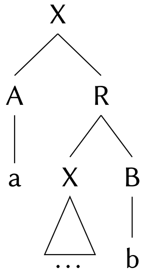
Any CFG can be converted to CNF, so there's no loss in expressive power. How can we convert the a^{n}b^n grammar?

$$\mathrm{X} \rightarrow \mathrm{ab} \quad (1)$$
$$\mathrm{X} \rightarrow \mathrm{aXb} \quad (2)$$

One CNF $a^n b^n$ grammar (can you find another?)

$$
\begin{aligned}
X &\rightarrow AR \\
R &\rightarrow XB \\
X &\rightarrow AB \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}
$$

X	X
AB	AR
aB	AXB
ab	aXB
	aXb



CFGs (in CNF) in Haskell

We'll encode CFGs in Haskell as a list (set) of rules, parameterized by `cat`, the type of non-terminals, and `term`, the type of terminals.[2]

```
type CFG cat term = [Rule cat term]

data Rule cat term = cat :- term | cat :> (cat, cat)
  deriving (Eq, Show) -- infix data constructors! cf.
                      -- A -> x
                      -- A -> BC
```

We represent rules via a datatype `Rule` (also depends on our choice of `cat` and `term`), which encodes the 2 admissible shapes for CNF rules.

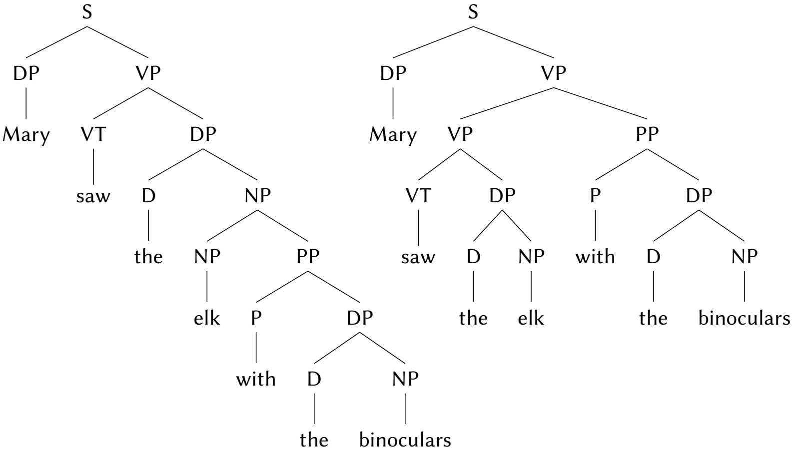
[2] The reason to allow this flexibility that different languages may have different categories (`cat`), and different vocabularies (`term`).

A sample toy grammar

```haskell
data Cat = S | D | DP | NP | VT | VP | P | PP
  deriving (Eq, Read, Show)

eng :: [Rule Cat String]
eng = [ S  :> (DP, VP)      ,
        VP :> (VT, DP)      ,
        DP :> (D , NP)      ,
        NP :> (NP, PP)      ,
        PP :> (P , DP)      ,
        VP :> (VP, PP)      ,
        DP :- "Mary"        ,
        VT :- "saw"         ,
        D  :- "the"         ,
        NP :- "binoculars",
        NP :- "elk"         ,
        P  :- "with"        ]
```

Mary saw the elk with the binoculars, 2 ways



A datatype for trees[3]

```haskell
data LBT cat term = Leaf cat term
                  | Branch cat (LBT cat term)
                               (LBT cat term)
  deriving (Eq, Show) -- Labeled Binary Trees
```

```haskell
t0 :: LBT Cat String
t0 = Branch NP
       (Leaf NP "elk")
       (Branch PP
         (Leaf P "with")
         (Branch DP
           (Leaf D "the")
           (Leaf NP "binoculars")))
```

[3] Compare the datatype for lists: `data List a = Empty | Cons a (List a)`.

Checking that a grammar generates a tree

Given `g :: [Rule cat term]` and `t :: LBT cat term`, can we check whether `t` is generated by `g`?

Checking that a grammar generates a tree

Given `g :: [Rule cat term]` and `t :: LBT cat term`, can we check whether `t` is generated by `g`? You bet! For `t0` we should check:

- `elem (NP :> (NP, PP)) g`
- `elem (NP :- "elk") g`
- `elem (PP :> (P, DP)) g`
- ...
- `elem (NP :- "binoculars") g`

```
t0 = Branch NP
       (Leaf NP "elk")
       (Branch PP
         (Leaf P "with")
         (Branch DP
           (Leaf D "the")
           (Leaf NP "binoculars")))
```

A macro

```haskell
generates :: (Eq n, Eq x) => CFG n x -> LBT n x -> Bool
generates g t = case t of
  Leaf   n x   -> elem (n :- x) g
  Branch n l r -> elem (n :> (label l, label r)) g &&
                  generates g l && generates g r

label :: LBT n x -> n
label (Leaf n _)     = n
label (Branch n _ _) = n
```

```
*W11> generates eng t0
True
```

The task of a parser

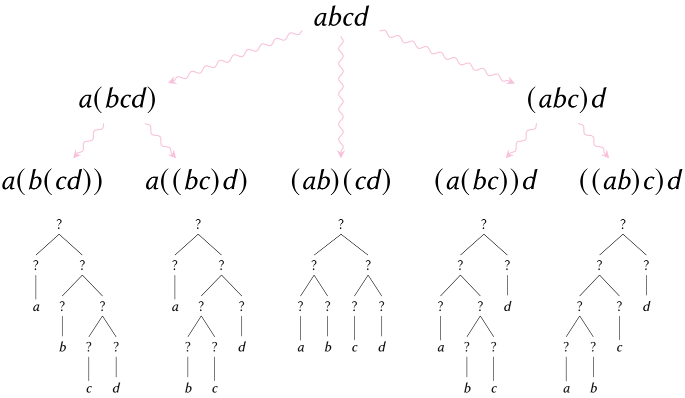
Checking if a given tree is generated by a grammar is comparatively straightforward. It is harder to generate the tree in the first place:

- Given unstructured input `xs :: [term]`, a **parser** determines whether `xs` has **any** analysis in `g`.
- The task is somewhat subtle, because a string of terminals can be bracketed in multiple distinct ways, and each must be checked.

The base case of a single terminal is simple. What about the rest?

```
parse g [x] = [ n | n :- y <- g, y==x ]
```

```
*W11> parse eng ["elk"]
[NP]
```


abcd
a(bcd)
(abc)d
a(b(cd))
a((bc)d)
(ab)(cd)
(a(bc))d
((ab)c)d

The breaks

```
breaks :: [a] -> [([a], [a])]
breaks u = [splitAt i u | i <- [1..length u - 1]]
```

```
*W11> breaks ["Mary","saw","the","elk"]
[ (["Mary"],["saw","the","elk"]),  -- keep going
  (["Mary","saw"],["the","elk"]),  -- won't work
  (["Mary","saw","the"],["elk"]) ] -- won't work
```

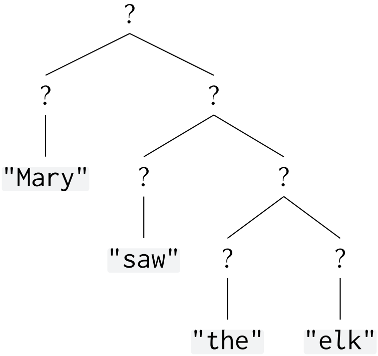
The breaks

```haskell
breaks :: [a] -> [([a], [a])]
breaks u = [splitAt i u | i <- [1..length u - 1]]
```

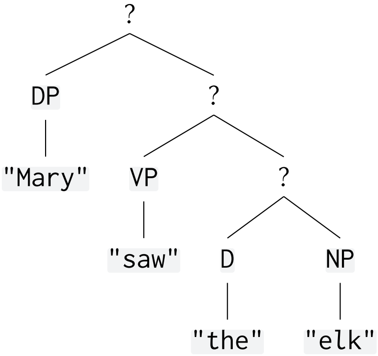
```
*W11> breaks ["Mary","saw","the","elk"]
[ (["Mary"],["saw","the","elk"]),  -- keep going
  (["Mary","saw"],["the","elk"]),  -- won't work
  (["Mary","saw","the"],["elk"]) ] -- won't work
```

```
*W11> breaks ["saw","the","elk"]
[ (["saw"],["the","elk"]),         -- keep going
  (["saw","the"],["elk"]) ]        -- won't work
```

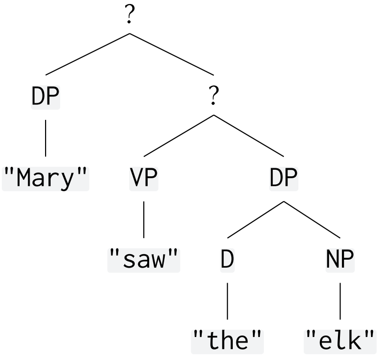
How a successful parse is filled in



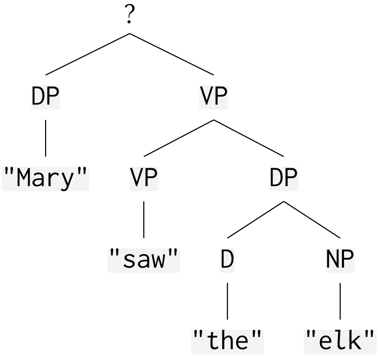
How a successful parse is filled in



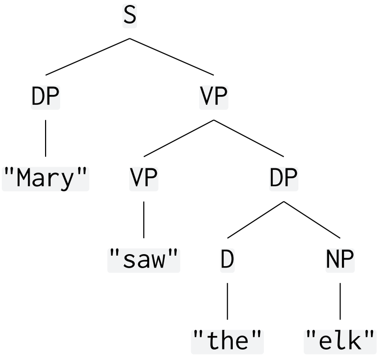
How a successful parse is filled in



How a successful parse is filled in



How a successful parse is filled in



The parser can’t know in advance that this is the right structure. It tries every possible binary tree, and every possible way of assigning category labels to every node in each tree.

Parsing to categories

```
parse
  :: (Eq cat, Eq term) =>
     [Rule cat term]-> [term] -> [cat]
parse g [x] = [ n | n :- y <- g, y==x ]
parse g xs  = [ n | (ls,rs) <- breaks xs,
                    -- break the string
                    nl <- parse g ls, nr <- parse g rs,
                    -- parse the two halves
                    n :> (l,r) <- g, l==nl, r==nr ]
                    -- find a corresponding rule
```

```
*W11> parse eng (words "Mary saw the elk")
[S] -- there's a successful parse, yielding S
*W11> parse eng (words "Mary saw the")
[]  -- no possible parses
```

Ambiguity happens

When multiple breaks work out in the end:

```
*W11> breaks ["saw","the","elk","with","Mary"]
...
(["saw"],["the","elk","with","Mary"]) -- V  + DP ~> VP
...
(["saw","the","elk"],["with","Mary"]) -- VP + PP ~> VP
...
```

```
*W11> parse eng (words "saw the elk with Mary")
[VP,VP] -- 2 successful VP parses
*W11> parse eng (words "Mary saw the elk with Mary")
[S,S]   -- which turn into 2 successful S parses
```

Inefficiencies

Our `parse` function is rather inefficient:

- Each terminal in *abcd* is parsed 5 times (once per potential tree)
- Each pair of symbols is parsed 2 times
- Things will get worse (*much* worse) in longer strings

```
*W11> parse eng (words "Mary saw the elk with the elk
with the elk with the elk with the elk")
[S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,
S,S,S,S,S,S,S,S,S,S,S,S,S,S,S,S] -- 42 parses!
(8.04 secs, 4,137,255,736 bytes)
```

One more `"with the elk"` and it hangs. (132 parses, ~3.5 minutes!)

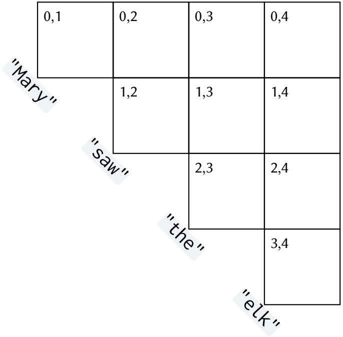
CYK parsing (Cocke–Younger–Kasami)

Each substring is identified by a **span**, a pair of numbers (i, j):

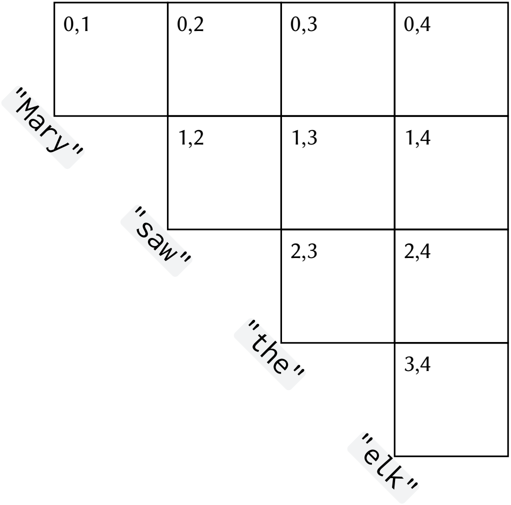
| `"Mary"` | `"saw"` | `"the"` | `"elk"` |

0 1 2 3 4

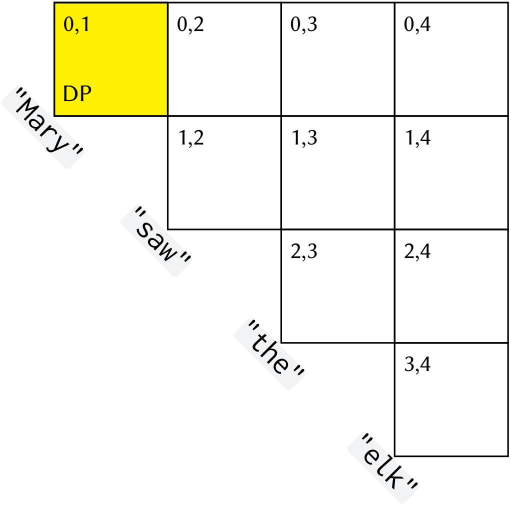
A string's spans can seen as a table; parsing is filling it in. The key to efficiency: each substring occurs exactly **once**!



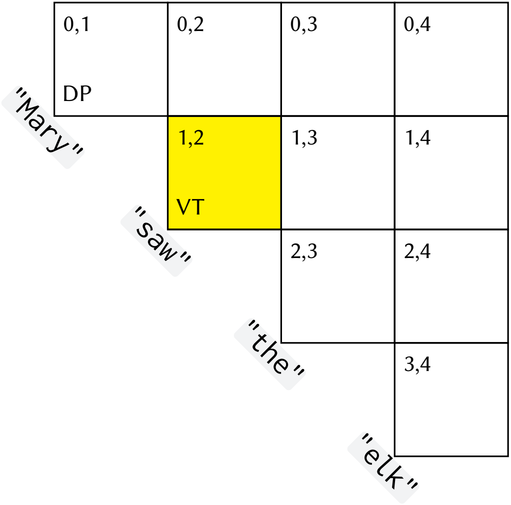
CYK algorithm: left-to-right, bottom-up



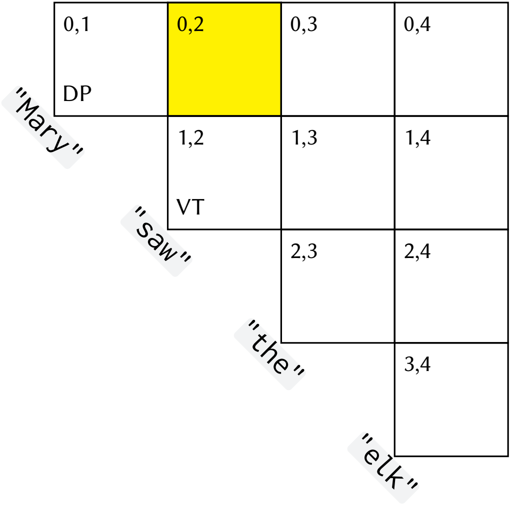
CYK algorithm: left-to-right, bottom-up



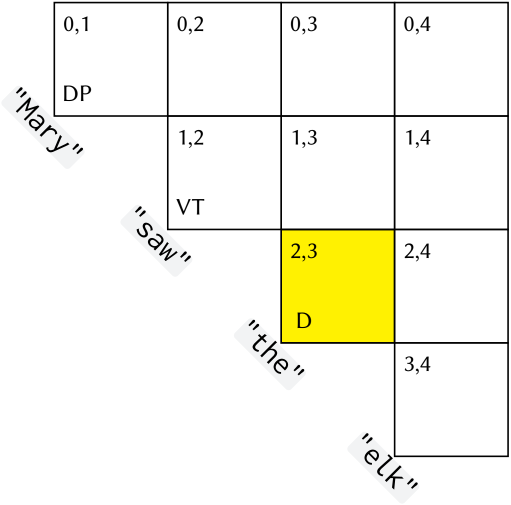
CYK algorithm: left-to-right, bottom-up



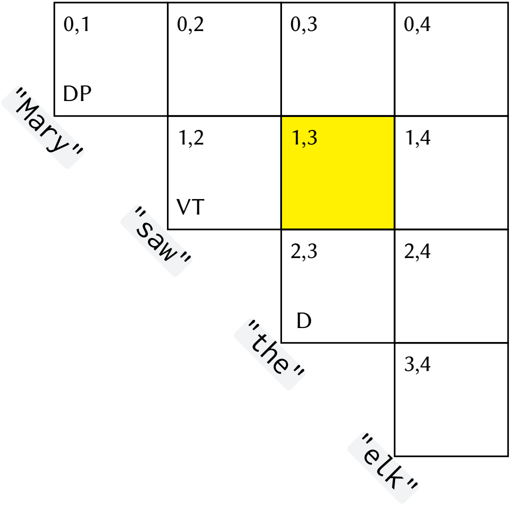
CYK algorithm: left-to-right, bottom-up



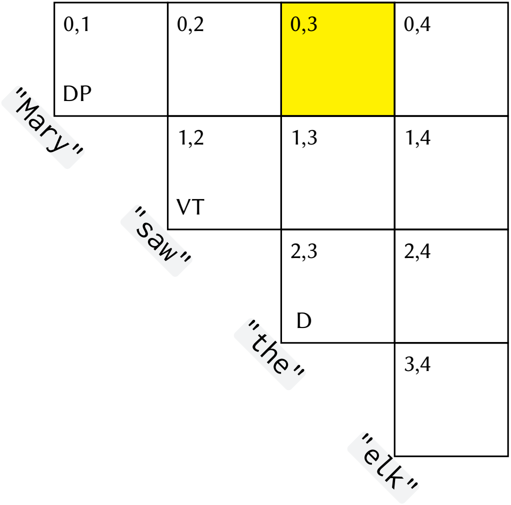
CYK algorithm: left-to-right, bottom-up



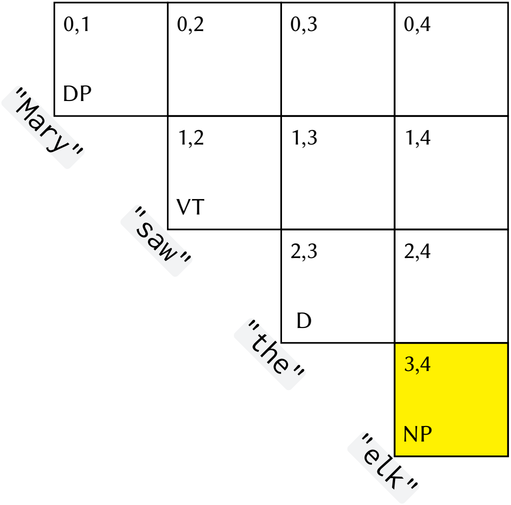
CYK algorithm: left-to-right, bottom-up



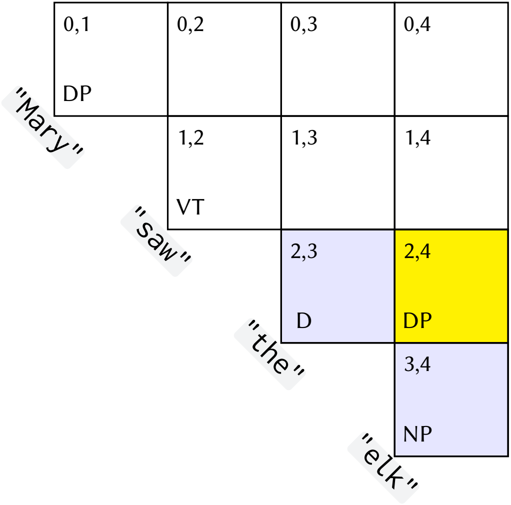
CYK algorithm: left-to-right, bottom-up



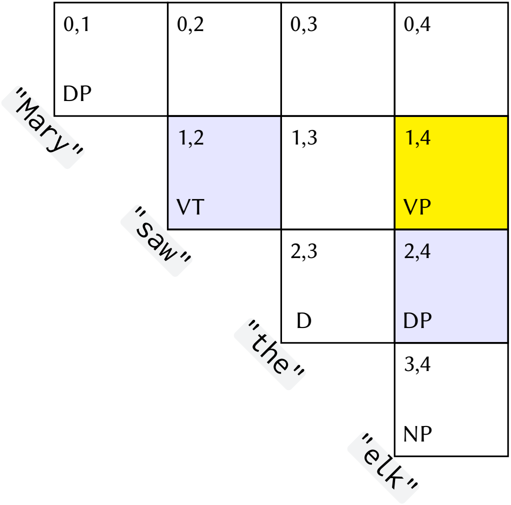
CYK algorithm: left-to-right, bottom-up



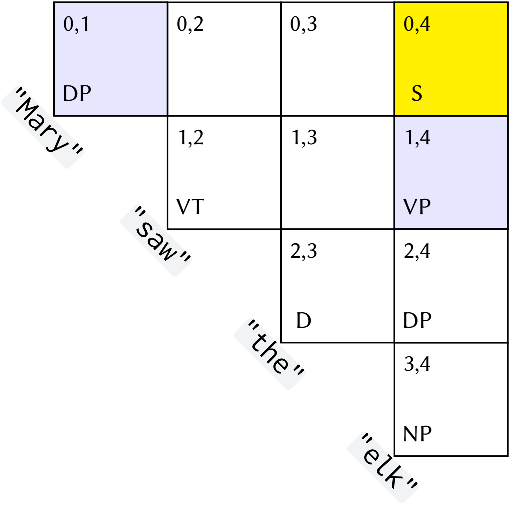
CYK algorithm: left-to-right, bottom-up



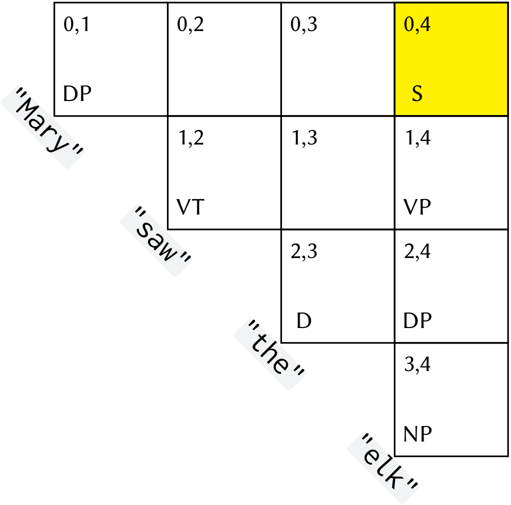
CYK algorithm: left-to-right, bottom-up



CYK algorithm: left-to-right, bottom-up



CYK algorithm: left-to-right, bottom-up



Huge efficiency gains

```
*W12> s = "Mary saw the elk with the elk with the elk
with the elk with the elk with the elk"

*W12> length (parse eng (words s))
132
(207.08 secs, 97,997,855,016 bytes)

*W12> length (parseCYK eng (words s))
132
(0.03 secs, 5,050,824 bytes)
```

Ambiguity in a CYK chart

"saw"	0,1 VT	0,2	0,3 VP	0,4	0,5	0,6 VP, VP
"the"		1,2 D	1,3 DP	1,4	1,5	1,6 DP
"elk"			2,3 NP	2,4	2,5	2,6 NP
"with"				3,4 P	3,5	3,6 PP
"the"					4,5 D	4,6 DP
"binoculars"						5,6 NP

Ambiguity in a CYK chart

"saw"	0,1 VT	0,2	0,3 VP	0,4	0,5	0,6 VP, VP
"the"		1,2 D	1,3 DP	1,4	1,5	1,6 DP
"elk"			2,3 NP	2,4	2,5	2,6 NP
"with"				3,4 P	3,5	3,6 PP
"the"					4,5 D	4,6 DP
"binoculars"						5,6 NP

Ambiguity in a CYK chart

0,1 VT	0,2	0,3 VP	0,4	0,5	0,6 VP, VP
	1,2 D	1,3 DP	1,4	1,5	1,6 DP
		2,3 NP	2,4	2,5	2,6 NP
			3,4 P	3,5	3,6 PP
				4,5 D	4,6 DP
					5,6 NP

"saw"
"the"
"elk"
"with"
"the"
"binoculars"

The actual algo

```
mkChart :: (Eq cat, Eq term) =>
  CFG cat term -> [term] -> [((Int, Int), [cat])]
mkChart g xs = helper (0,1) [] where
  helper p@(i,j) tab
    | j>length xs = tab
    | i==(-1)     = helper (j,j+1) tab
    | i==j-1      = helper (i-1,j) $
                      (p, [n | n:-t <- g, t==xs!!(j-1)]):tab
    | otherwise   = helper (i-1,j) $
                      (p, [n | n:>(l,r) <- g, k <- [i+1..j-1],
                               lc <- fromJust $ lookup (i,k) tab, l==lc,
                               rc <- fromJust $ lookup (k,j) tab, r==rc]):tab

parseCYK :: (Eq cat, Eq term) => CFG cat term -> [term] -> [cat]
parseCYK g xs = snd (head (mkChart g xs))
```

```
*W12> s = words "saw the elk with the binoculars"
*W12> mkChart eng s
[((0,6),[VP,VP]),((1,6),[DP]),((2,6),[NP]),((3,6),[PP]),((4,6),[DP]),
((5,6),[NP]),((0,5),[]),((1,5),[]),((2,5),[]),((3,5),[]),((4,5),[D]),
((0,4),[]),((1,4),[]),((2,4),[]),((3,4),[P]),((0,3),[VP]),((1,3),[DP]),
((2,3),[NP]),((0,2),[]),((1,2),[D]),((0,1),[VT])]
*W12> parse eng s
[VP,VP]
```