CFG parsing

Computational Linguistics (LING 455)

Rutgers University Nov 16 & 19, 2021

Parsing

CNF

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

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

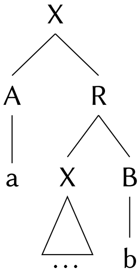
Any CFG can be converted to CNF, so there's no loss in expressive power. How can we convert the $\mathrm{a}^n\mathrm{b}^n$ grammar?

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

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.[1]

```
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.

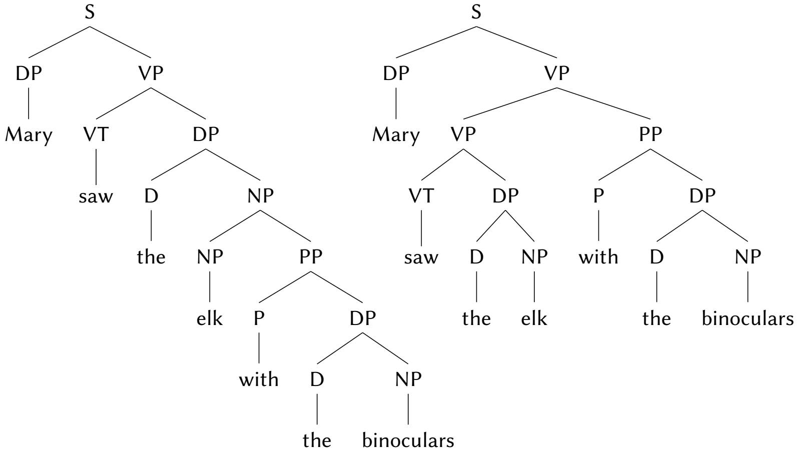
[1] The reason to allow this flexibility that different languages may have different categories (`cat`), and different vocabularies (`term`).

A sample toy grammar

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

eng :: CFG 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



The task of a parser

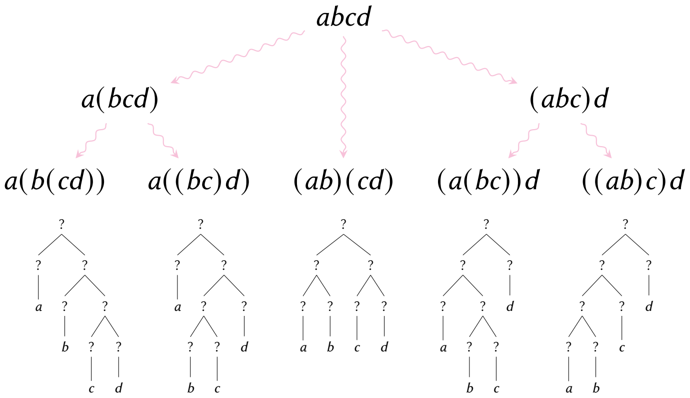
Given input `xs :: [term]`, a **parser** determines whether `xs` has **any** analysis in `g`, reporting the category that `xs` can be assigned.

- 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 ]
```

```
*W12> 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]]
```

```
*W12> 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
```

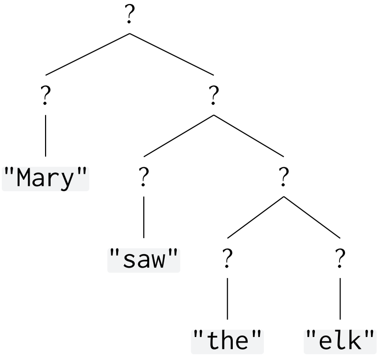
The breaks

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

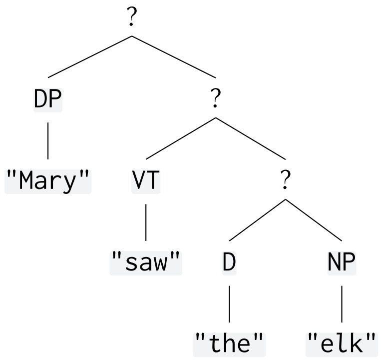
```
*W12> breaks ["Mary","saw","the","elk"]
[ (["Mary"],["saw","the","elk"]),  -- keep going
  (["Mary","saw"],["the","elk"]),  -- won't work
  (["Mary","saw","the"],["elk"]) ] -- won't work
```

```
*W12> breaks ["saw","the","elk"]
[ (["saw"],["the","elk"]),         -- keep going
  (["saw","the"],["elk"]) ]        -- won't work
```

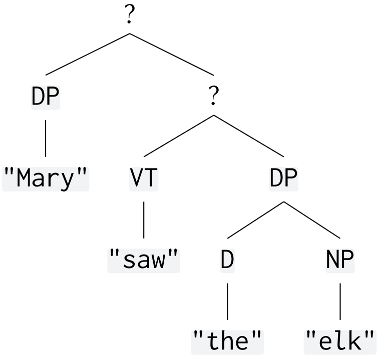
How a successful parse is filled in



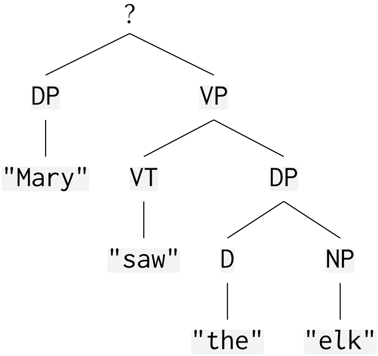
How a successful parse is filled in



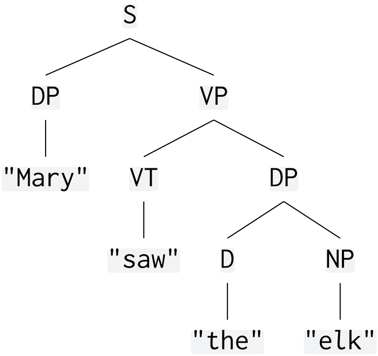
How a successful parse is filled in



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) =>
     CFG 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
```

```
*W12> parse eng (words "Mary saw the elk")
[S] -- there's a successful parse, yielding S
*W12> parse eng (words "Mary saw the")
[]  -- no possible parses
```

Ambiguity happens

When multiple breaks work out in the end:

```
*W12> breaks ["saw","the","elk","with","Mary"]
...
(["saw"],["the","elk","with","Mary"]) -- V  + DP ~> VP
...
(["saw","the","elk"],["with","Mary"]) -- VP + PP ~> VP
...
```

```
*W12> parse eng (words "saw the elk with Mary")
[VP,VP] -- 2 successful VP parses
*W12> 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

```
*W12> length $ parse eng $ words "Mary saw the elk with
the elk with the elk with the elk with the elk"
42
(9.69 secs, 4,648,676,120 bytes)
```

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

CYK parsing (Cocke–Younger–Kasami)

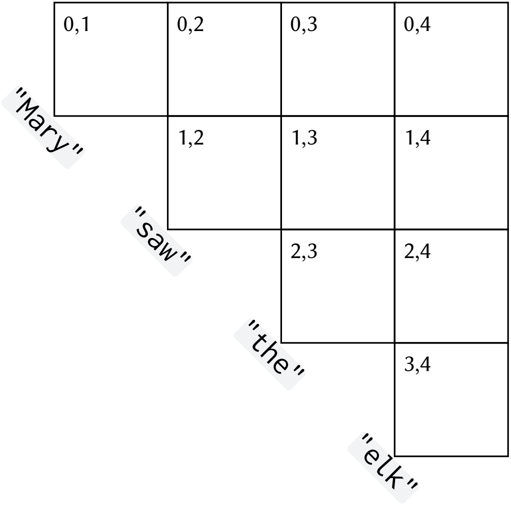
Each substring is identified by a **span**, a pair of numbers (i, j):

| `"Mary"` | `"saw"` | `"the"` | `"elk"` |
0 1 2 3 4

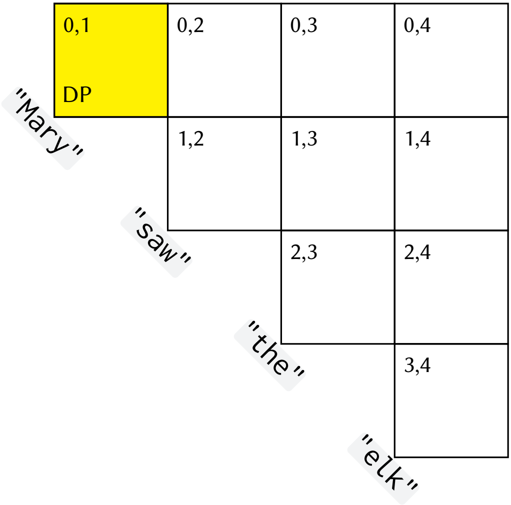
Spans can arranged in a table. Then parsing amounts to filling the table in. The key to efficiency: each span occurs exactly **once!**

0,1	0,2	0,3	0,4
`"Mary"`	1,2	1,3	1,4
	`"saw"`	2,3	2,4
		`"the"`	3,4
			`"elk"`

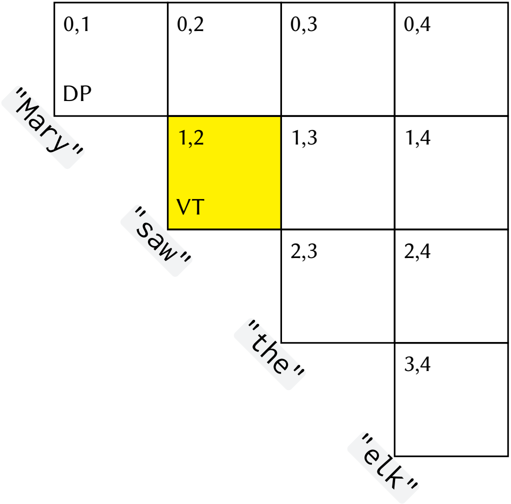
CYK algorithm: left-to-right, bottom-up



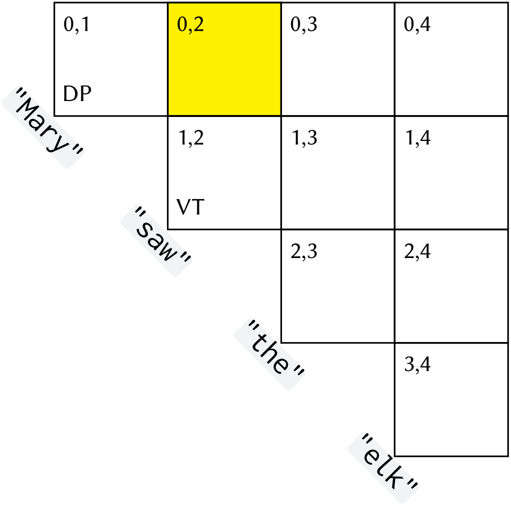
CYK algorithm: left-to-right, bottom-up



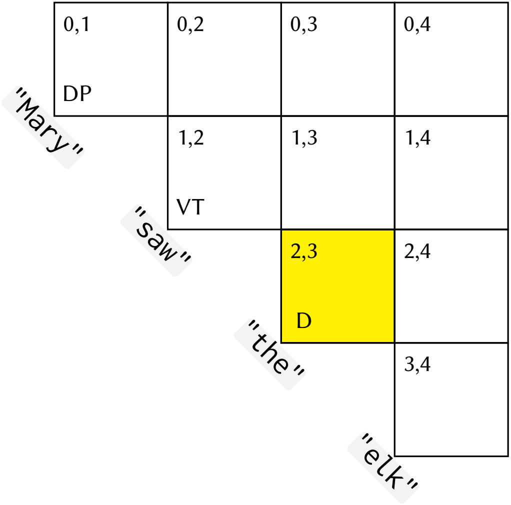
CYK algorithm: left-to-right, bottom-up



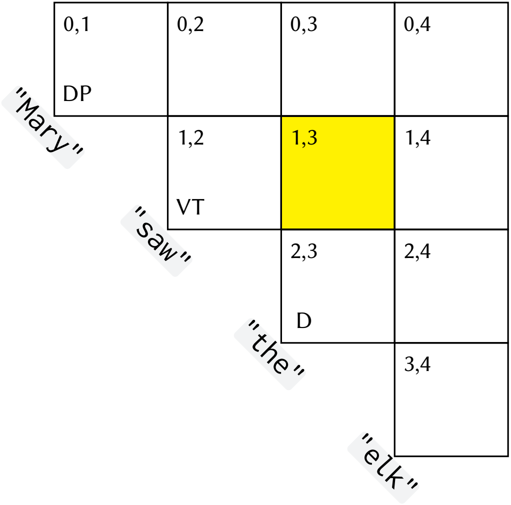
CYK algorithm: left-to-right, bottom-up



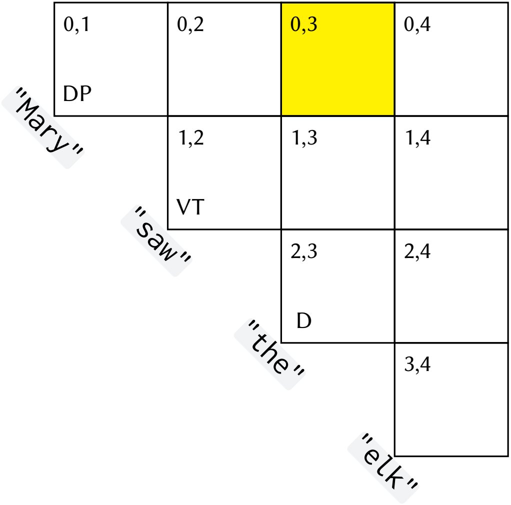
CYK algorithm: left-to-right, bottom-up



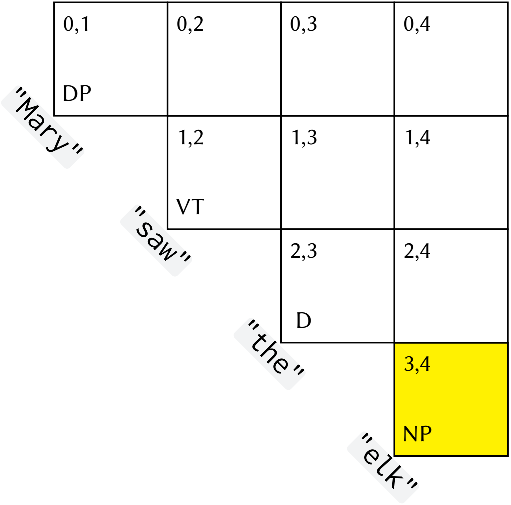
CYK algorithm: left-to-right, bottom-up



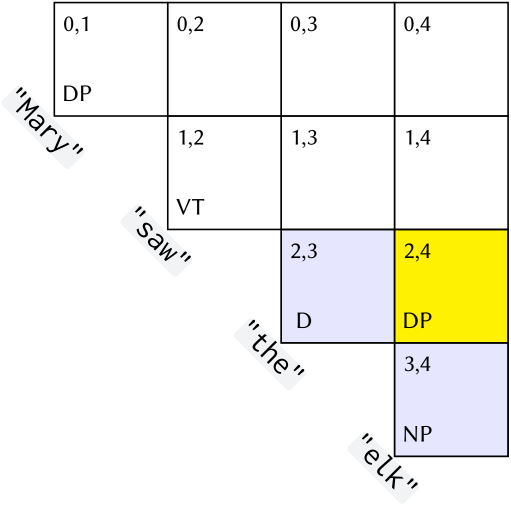
CYK algorithm: left-to-right, bottom-up



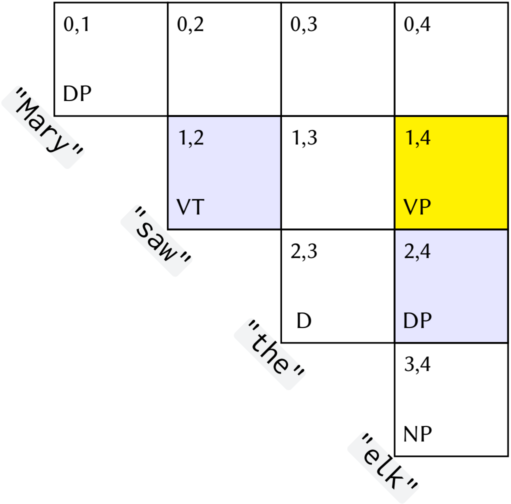
CYK algorithm: left-to-right, bottom-up



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



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



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

"Mary"	0,1 DP	0,2	0,3	0,4 S
"saw"		1,2 VT	1,3	1,4 VP
"the"			2,3 D	2,4 DP
"elk"				3,4 NP

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

0,1 DP	0,2	0,3	0,4 S
	1,2 VT	1,3	1,4 VP
		2,3 D	2,4 DP
			3,4 NP

"Mary" "saw" "the" "elk"

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

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

Ambiguity in a CYK chart

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

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

The actual algorithm

```haskell
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<0         = 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]
```

Parsing to trees

Both `parse` and `parseCYK` return a list of `cat`'s that a string of `term`'s can be assigned according to a `CFG`. What if we want **trees**?

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

Parsing to LBT's

Shown here for `parse`. Analogous changes work for `parseCYK`.

```
parse :: (Eq cat, Eq term) =>
     CFG cat term -> [term] -> [cat]
parse g [x] = [ n | n :- y <- g, y==x ]
parse g xs  =
  [ n | (ls,rs) <- breaks xs,
        nl <- parse g ls, nr <- parse g rs,
        n :> (l,r) <- g, l==nl, r==nr ]
```

```
parseToLBT :: (Eq cat, Eq term) =>
     CFG cat term -> [term] -> [LBT cat term]
parseToLBT g [x] = [ Leaf n x | n :- y <- g, y==x ]
parseToLBT g xs  =
  [ Branch n tl tr | (ls, rs) <- breaks xs,
        tl <- parseToLBT g ls, tr <- parseToLBT g rs,
        n :> (l, r) <- g, label tl == l, label tr == r ]
```

Examples

```
*W12> parseToLBT eng $ words "the elk"
[Branch DP
  (Leaf D "the")
  (Leaf NP "elk")]

*W12> parseToLBT eng $ words "saw the elk"
[Branch VP
  (Leaf VT "saw")
  (Branch DP
    (Leaf D "the")
    (Leaf NP "elk"))]

*W12> parseToLBT eng $ words "Mary saw the elk"
[Branch S
  (Leaf DP "Mary")
  (Branch VP
    (Leaf VT "saw")
    (Branch DP
      (Leaf D "the")
      (Leaf NP "elk")))]
```

Pretty output

Actually, the `LBT` output is not broken across lines like this. If we want to display our trees in a more readable way, there's a helpful library:

```
import Data.Tree
```

```
toTree :: (Show cat, Show term) =>
          LBT cat term -> Tree String -- convert LBT to Tree
toTree (Leaf   n x)   = Node (show n) [Node (show x) []]
toTree (Branch n l r) = Node (show n) [toTree l, toTree r]

displayForest :: (Show cat, Show term) =>
                 [LBT cat term] -> IO ()
displayForest = putStrLn . drawForest . map toTree
```

```
*W12> parsed = parseToLBT eng $ words "Mary saw the elk"
*W12> displayForest parsed
```

The output

```
S
|
+- DP
|  |
|  `- "Mary"
|
`- VP
   |
   +- VT
   |  |
   |  `- "saw"
   |
   `- DP
      |
      +- D
      |  |
      |  `- "the"
      |
      `- NP
         |
         `- "elk"
```

Enriched parsing

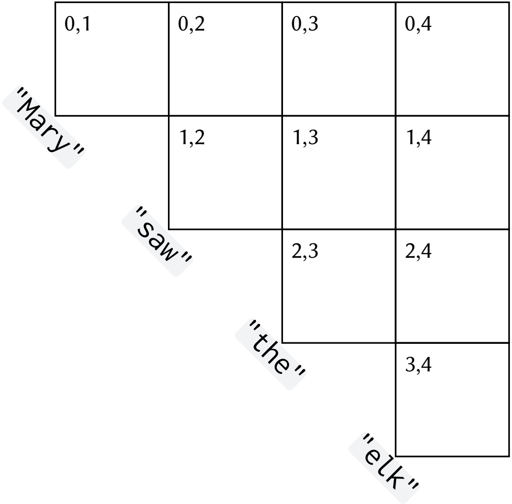
The strategy for parsing to `LBT`'s is quite reminiscent of how we generalized FSA parsing to FST parsing:

```
step delta x (q,m) = [ (s, m<>n) | (r,y,n,s) <- delta,
                                   q==r, y==x ]
```

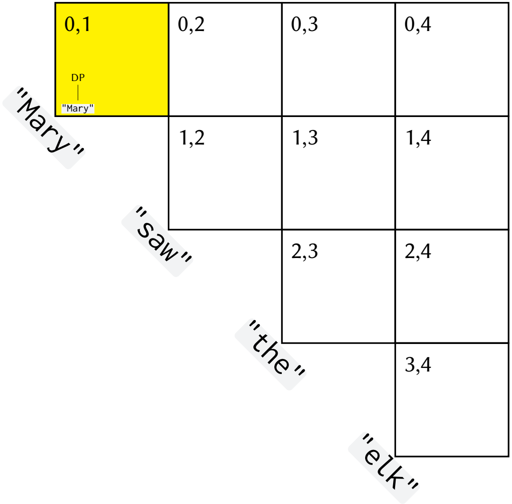
```
parseToLBT g [x] = [ Leaf n x | n :- y <- g, y==x ]
parseToLBT g xs  =
  [ Branch n tl tr | (ls, rs) <- breaks xs,
        tl <- parseToLBT g ls, tr <- parseToLBT g rs,
        n :> (l, r) <- g, label tl == l, label tr == r ]
```

The old result of a parse (previously, a state; now, a category) is **enriched** with some extra info (previously, a string; now, a tree).

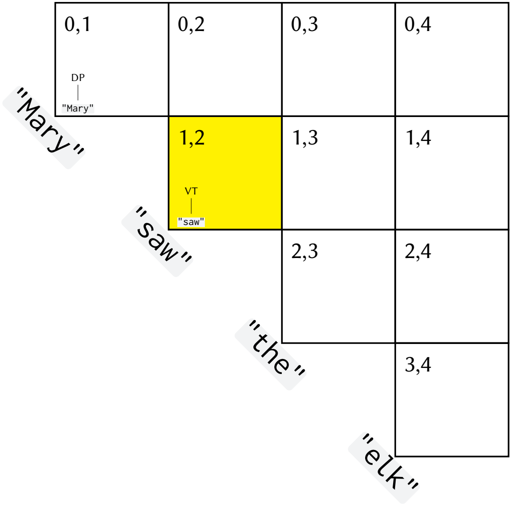
Generalized to CYK



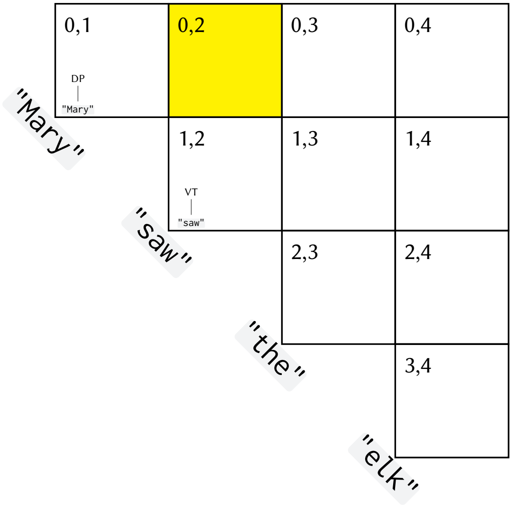
Generalized to CYK



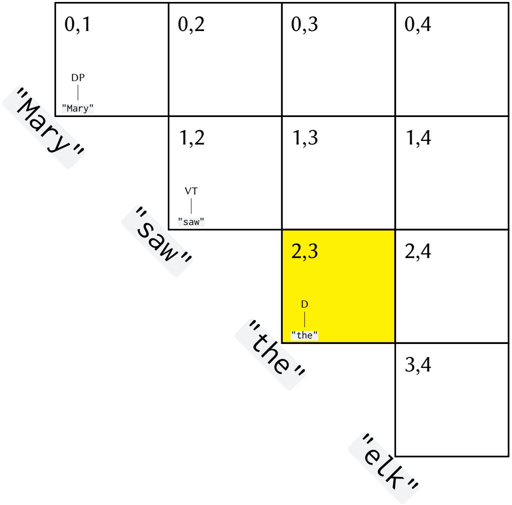
Generalized to CYK



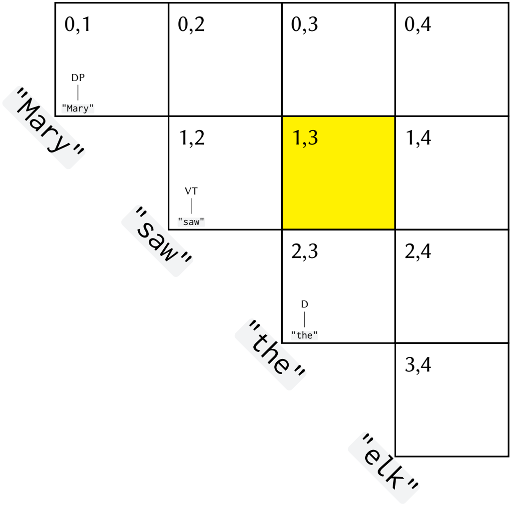
Generalized to CYK



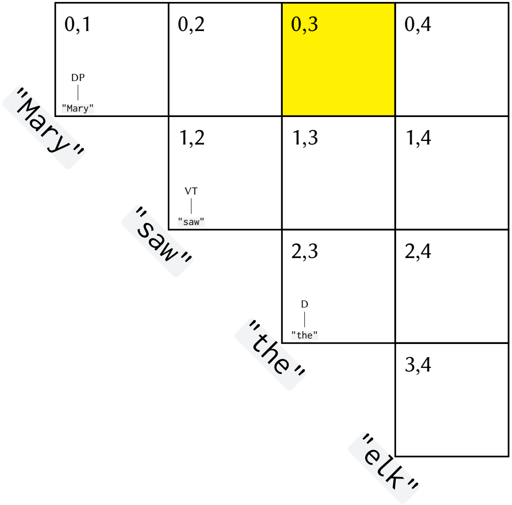
Generalized to CYK



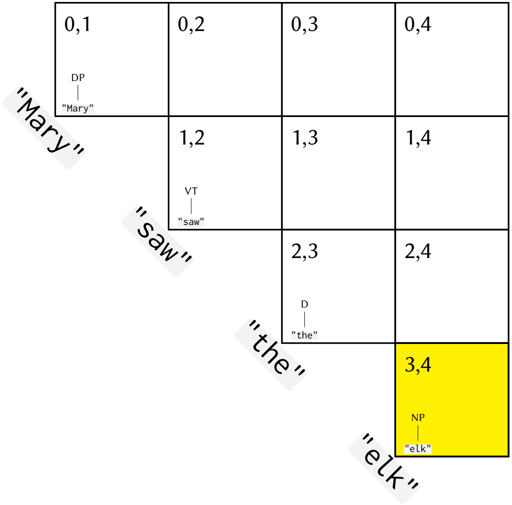
Generalized to CYK



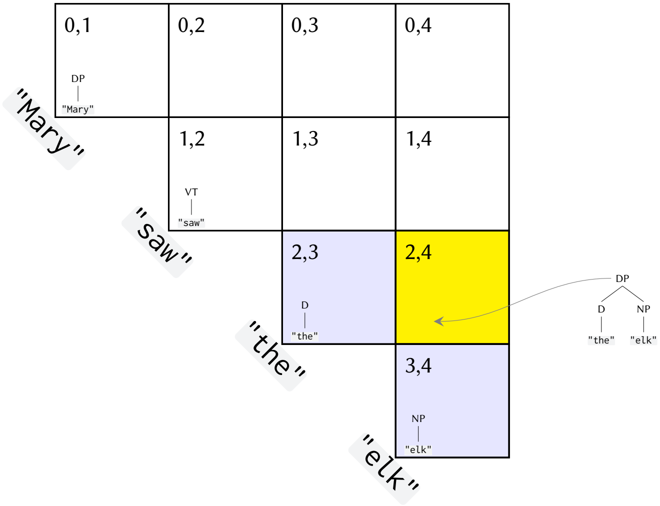
Generalized to CYK



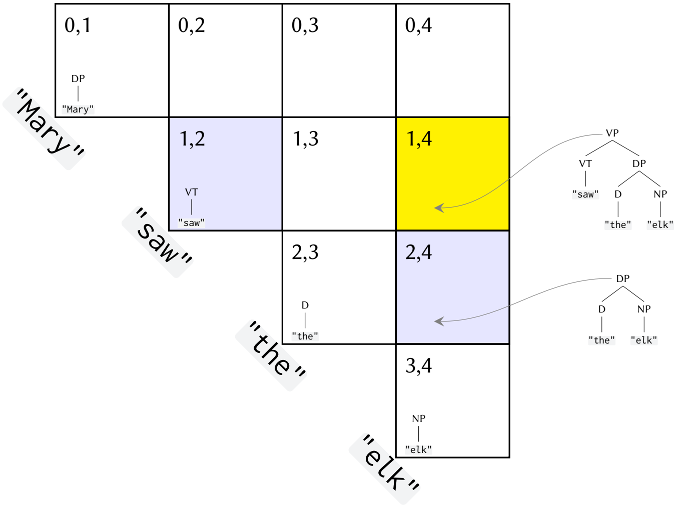
Generalized to CYK



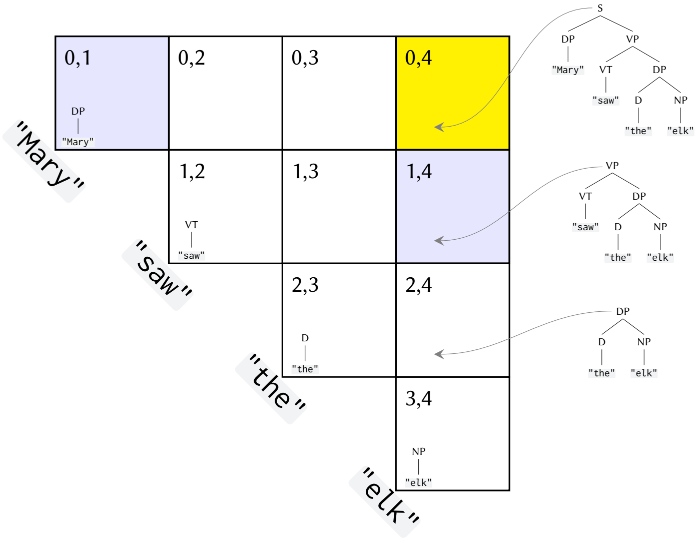
Generalized to CYK



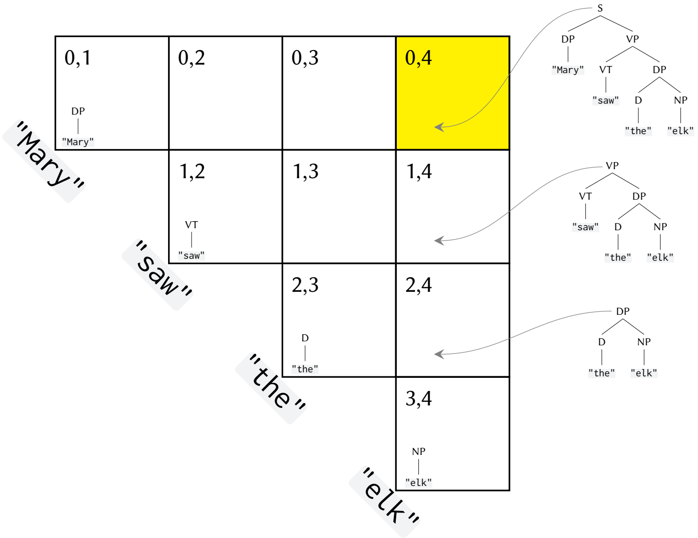
Generalized to CYK



Generalized to CYK



Generalized to CYK



Weighted or probabilistic CFGs

A straightforward extension of CFGs pairs rules with **weights**, **probabilities**, or **costs**.

```
type WCFG cat term weight = [(Rule cat term, weight)]
type PCFG cat term = WCFG cat term Double
```

Computing the weight of a whole analysis means **accumulating** the weights, via a monoid! (Exercise: change CYK.)

```
parseW g [x] = [ (n, w) | (n :- y, w) <- g, y==x ]
parseW g xs  =
  [ (n, w<>wl<>wr) | (ls, rs) <- breaks xs,
                     (nl, wl) <- parseW g ls,
                     (nr, wr) <- parseW g rs,
                     (n :> (l,r), w) <- g, l==nl, r==nr ]
```

Transition-based parsing

Parsing as state transitions

Another way to conceptualize parsing is via transitions:

- Specify what a **starting stage** is
- Specify what a **goal stage** is
- Specify a **transition relation on stages**

Example: finite-state parsing

- A starting stage is a pair (A, xs), where $A \in I$ and xs is the input
- A goal stage is a pair (A, ε), where $A \in F$
- $(A, x_i x_{i+1} \ldots x_n) \Rightarrow (B, x_{i+1} \ldots x_n)$ iff $(A, x_i, B) \in \Delta$

Shift-reduce parsing

CFG parsing can work in a similar way. Given a set of rules G:

- A starting stage is (ε, xs), where xs is the input
- A goal stage is (A, ε), where A is a nonterminal
- Transitions either read a terminal or reduce 2 nonterminals:
  1. SHIFT: $(\Phi, x_i x_{i+1} \ldots x_n) \Rightarrow (A\Phi, x_{i+1} \ldots x_n)$, where $A \rightarrow x_i \in G$
  2. REDUCE: $(RL\Phi, xs) \Rightarrow (A\Phi, xs)$, where $A \rightarrow LR \in G$

An example

	Type	Rule	Configuration
0			(ε, Mary saw the elk)
1	SHIFT	DP $\longrightarrow$ Mary	(DP, saw the elk)
2	SHIFT	VT $\longrightarrow$ saw	(VT DP, the elk)
3	SHIFT	D $\longrightarrow$ the	(D VT DP, elk)
4	SHIFT	NP $\longrightarrow$ elk	(NP D VT DP, ε)
5	REDUCE	DP $\longrightarrow$ D NP	(DP VT DP, ε)
6	REDUCE	VP $\longrightarrow$ VT DP	(VP DP, ε)
7	REDUCE	S $\longrightarrow$ DP VP	(S, ε)

In effect, shift-reduce parsing involves constructing something known as a **pushdown automaton**, in which a pushdown stack of nonterminals functions as an auxiliary memory source.

In Haskell

```haskell
type Stage cat term = ([cat], [term])

shift
  :: Eq term =>
     CFG cat term -> Stage cat term -> [Stage cat term]
shift g (cs, t:ts) = [(n:cs, ts) | n:-x <- g, x==t]

reduce
  :: Eq cat =>
     CFG cat term -> Stage cat term -> [Stage cat term]
reduce g (r:l:cs, ts) = [(n:cs, ts) | n:>(l',r') <- g,
                                      l'==l, r'==r]


step
  :: (Eq cat, Eq term) =>
     CFG cat term -> Stage cat term -> [Stage cat term]
step g st@(r:l:cs, t:ts) = shift g st ++ reduce g st
step g st@(cs    , t:ts) = shift g st
step g st@(r:l:cs, ts)   = reduce g st
step g st                = [st]
```

Keep taking steps till nothing changes

```
parseSR g ws = helper g $ [([], ws)]
  where
    oneStep sts = sts >>= step g
    helper g sts
      | oneStep sts == sts = sts
      | otherwise = helper g (oneStep sts)
```

```
*W12> s = words "Mary saw the elk with the binoculars"
*W12> parseSR eng s
[([S],[]),([S],[])]
```