A little more CFG parsing

Computational Linguistics (LING 455)

Rutgers University   Nov 23, 2021

Summary

CFGs

CFGs have rewrite rules of the shape $A \longrightarrow \varphi$, where φ is a string of terminals and non-terminals.

- The restriction is on the LHS. This is what makes it context-free.

We can contemplate restrictions on the RHS as well:

- Rules like $A \longrightarrow x$, $A \longrightarrow xB$ only: finite-state/regular
- Rules like $A \longrightarrow x$, $A \longrightarrow BC$ only: Chomsky Normal Form

CNF is a convenience, not a restriction in expressive power.

The parsing task

A **grammar** is a way of finitely and succinctly specifying a (typically infinite) number of objects with a certain shape.

Parsing is the task of structuring a linear representation in a way sanctioned by some grammar.

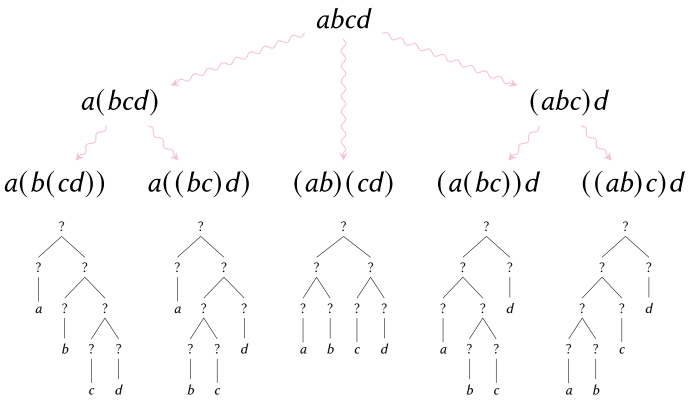
Naive CFG parsing

```
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 ]
```

For non-singleton strings:

- Break the input string into two (every possible way)
- Recurse, parsing the left half and the right half (given `g`)
- Try to combine the resulting yields `nl` and `nr` (given `g`)
- Return the resulting categories

Naive parsing is inefficient: each substring under a given length is parsed multiple (depending on the string length, maybe many) times.


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

CYK

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!**

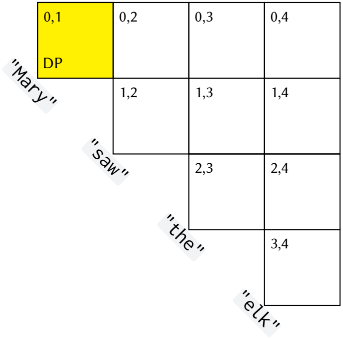
`"Mary"`	0,1	0,2	0,3	0,4
`"saw"`		1,2	1,3	1,4
`"the"`			2,3	2,4
`"elk"`				3,4

CYK

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

$$\underset{0}{|}\ \texttt{"Mary"}\ \underset{1}{|}\ \texttt{"saw"}\ \underset{2}{|}\ \texttt{"the"}\ \underset{3}{|}\ \texttt{"elk"}\ \underset{4}{|}$$

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

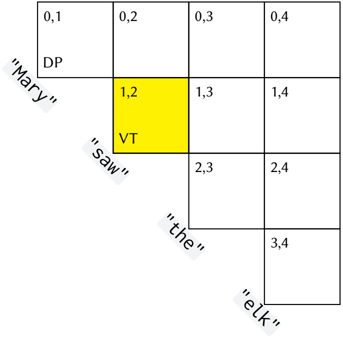


CYK

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!**

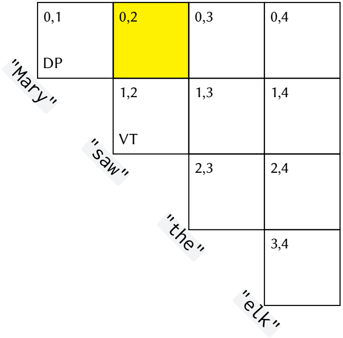


CYK

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!**



CYK

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!**

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

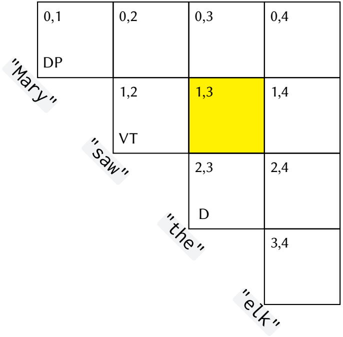
CYK

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!**



CYK

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

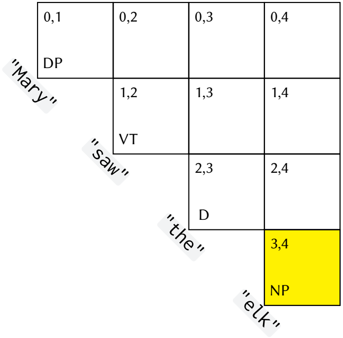
CYK

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!**



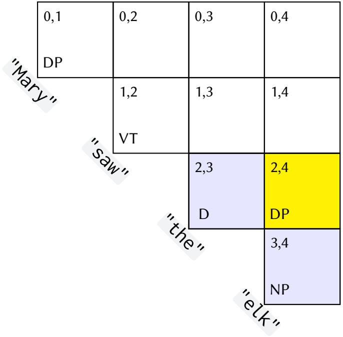
CYK

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!**



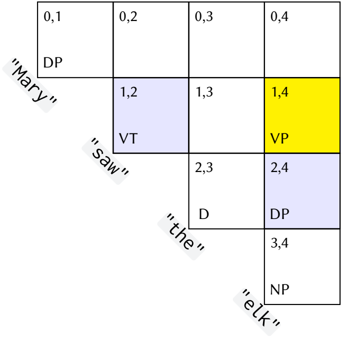
CYK

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!**

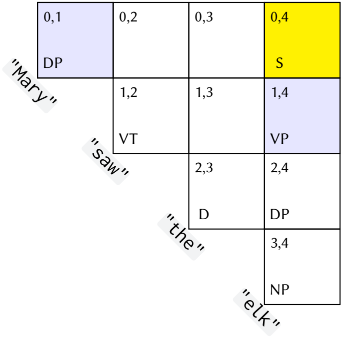


CYK

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!**



CYK

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!**

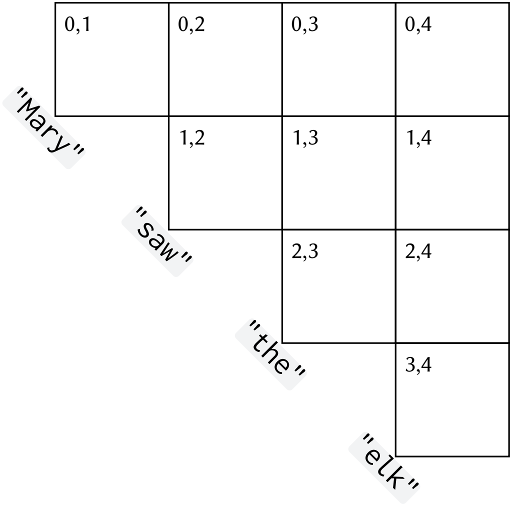
0,1 DP	0,2	0,3	0,4 S
`"Mary"`	1,2 VT	1,3	1,4 VP
	`"saw"`	2,3 D	2,4 DP
		`"the"`	3,4 NP
			`"elk"`

Enriched parsing

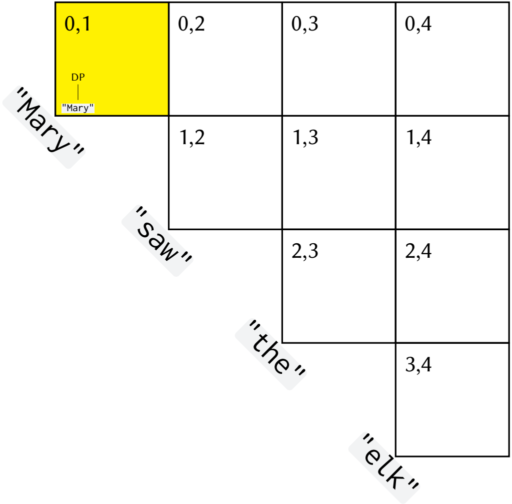
These parsing algorithms treat parse yields as (lists of) categories. We might be interested in other kinds of yields:

- Trees (e.g., `LBT`'s) that encode the structure of the parsed object
- Weights that report (e.g.) how probable a derivation is
- Costs associated with a derivation

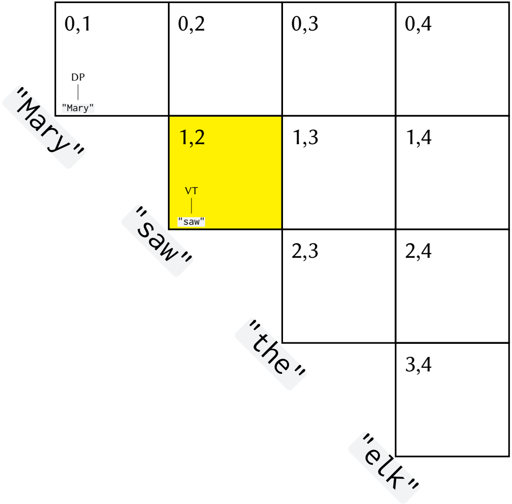
Parsing to trees



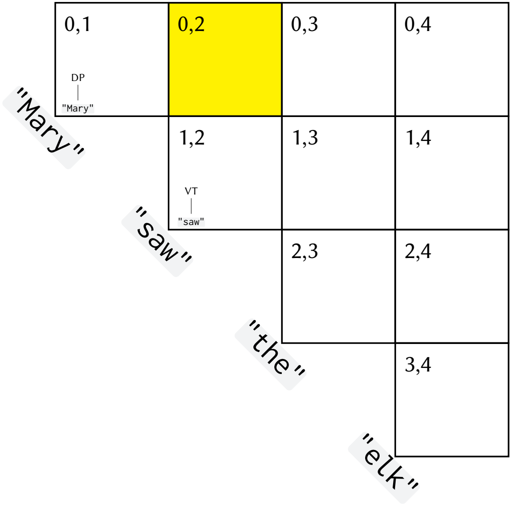
Parsing to trees



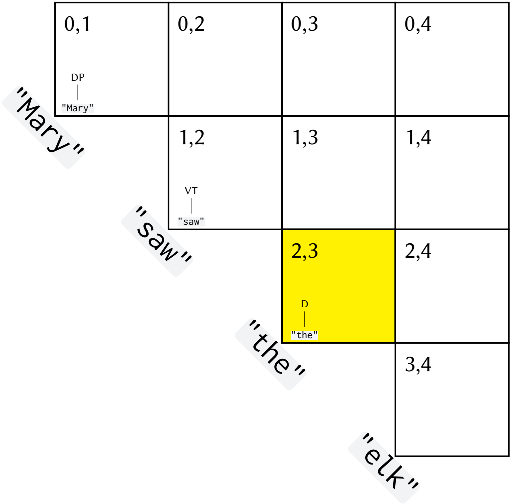
Parsing to trees



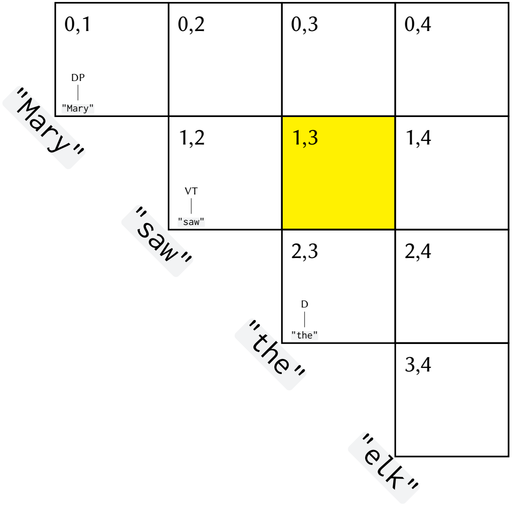
Parsing to trees



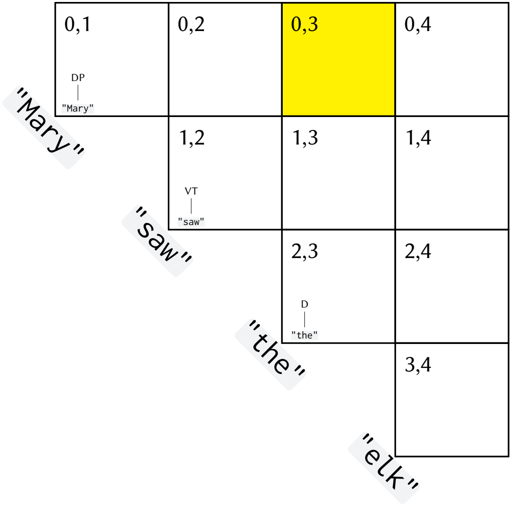
Parsing to trees



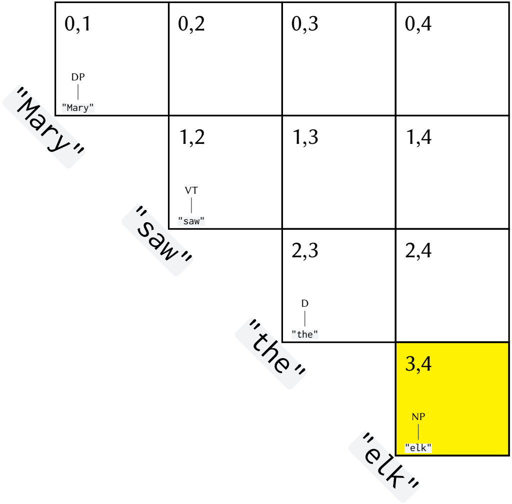
Parsing to trees



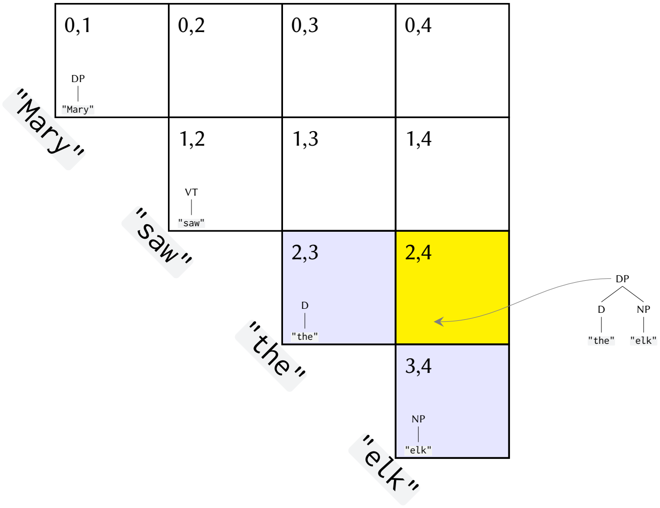
Parsing to trees



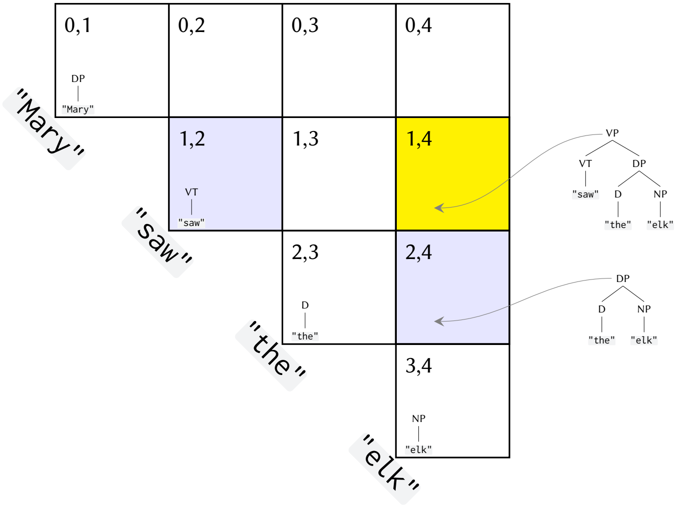
Parsing to trees



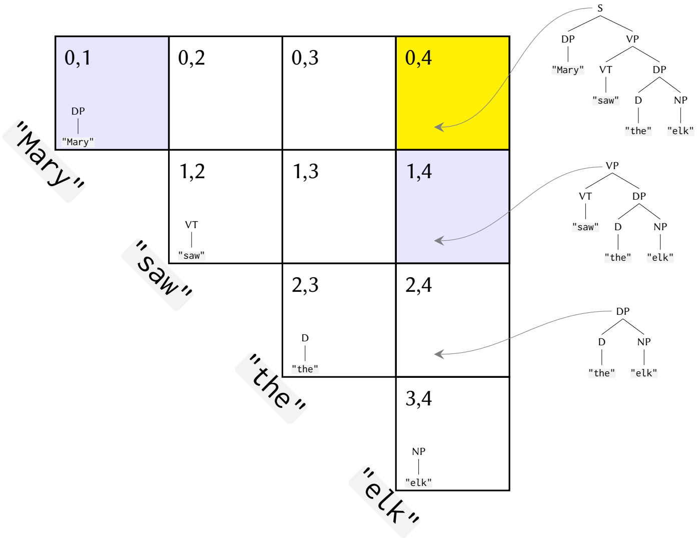
Parsing to trees



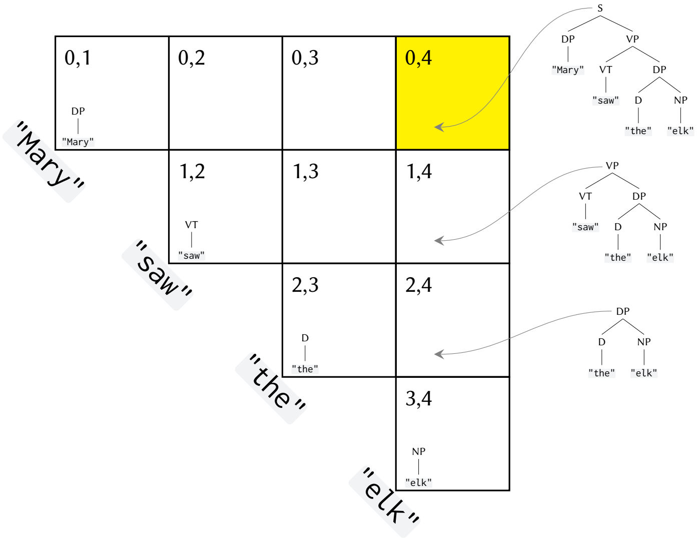
Parsing to trees



Parsing to trees



Parsing to trees



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).

Weighted or probabilistic CFGs

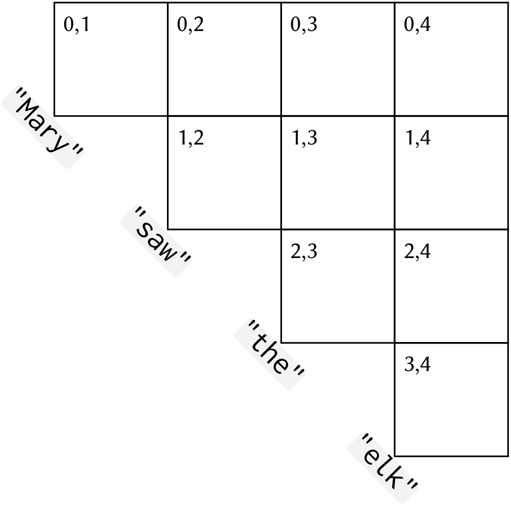
A straightforward extension of CFGs pairs rules with **weights**, **probabilities**, or **costs**.

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

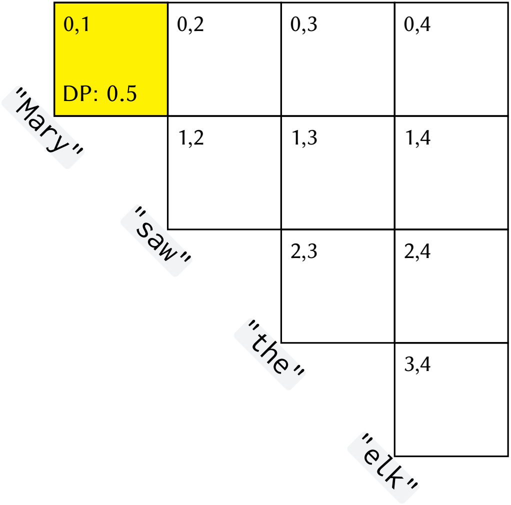
Computing the weight of a whole analysis means **accumulating** the weights, via a monoid! (Exercise: change CYK.)

```
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 ]
```

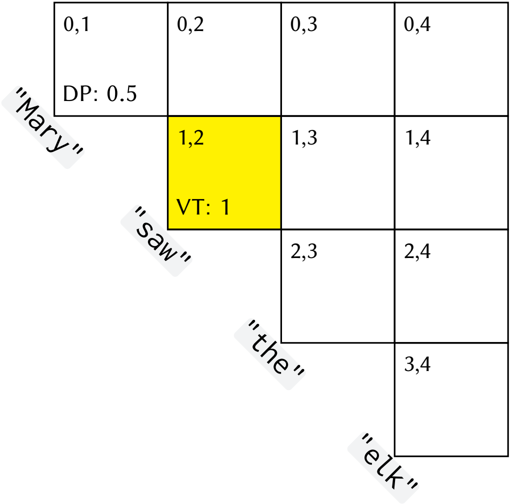
Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



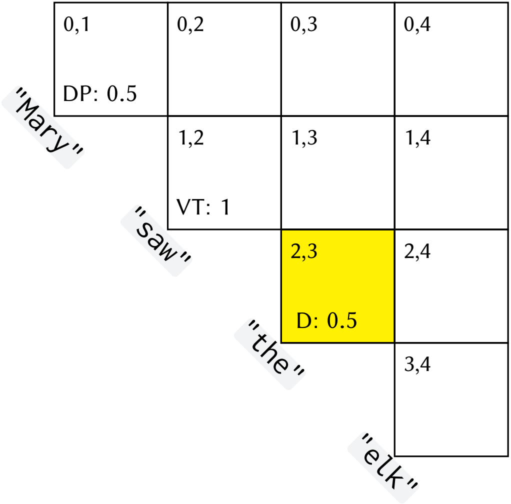
Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



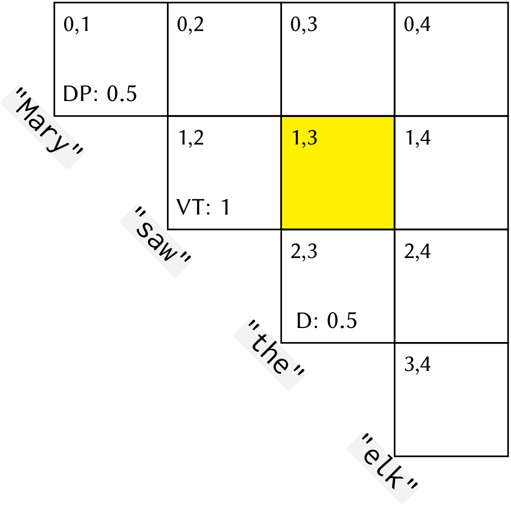
Parsing with weights (assume all $A \to BC$ rules weighted 1)

0,1 DP: 0.5	0,2	0,3	0,4
"Mary"	1,2 VT: 1	1,3	1,4
	"saw"	2,3	2,4
		"the"	3,4
			"elk"

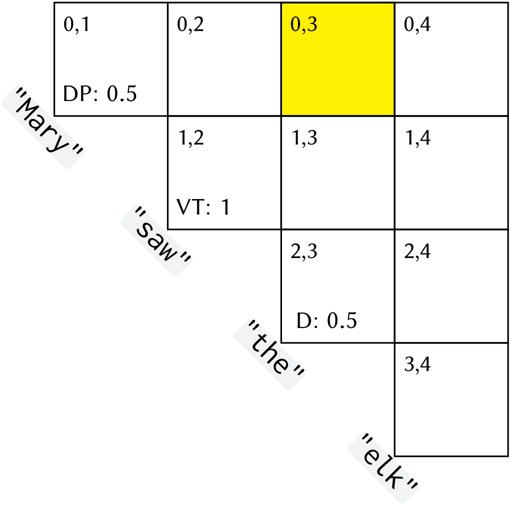
Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



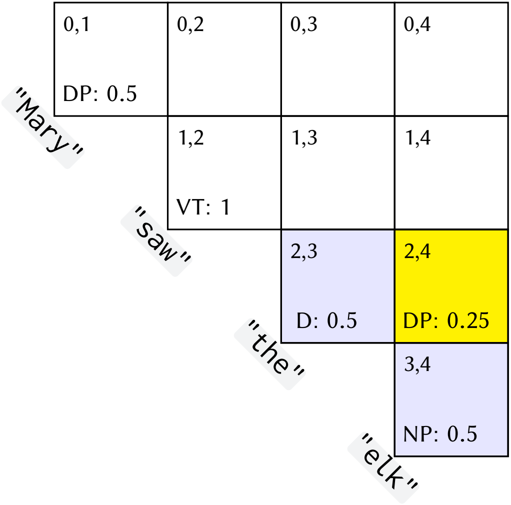
Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)

0,1 DP: 0.5	0,2	0,3	0,4
"Mary"	1,2 VT: 1	1,3	1,4
	"saw"	2,3 D: 0.5	2,4
		"the"	3,4 NP: 0.5
			"elk"

Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)



Parsing with weights (assume all $A \rightarrow BC$ rules weighted 1)

"Mary"	0,1 DP: 0.5	0,2	0,3	0,4
"saw"		1,2 VT: 1	1,3	1,4 VP: 0.25
"the"			2,3 D: 0.5	2,4 DP: 0.25
"elk"				3,4 NP: 0.5

Parsing with weights (assume all $A \longrightarrow BC$ rules weighted 1)

"Mary"	0,1 DP: 0.5	0,2	0,3	0,4 S: 0.125
"saw"		1,2 VT: 1	1,3	1,4 VP: 0.25
"the"			2,3 D: 0.5	2,4 DP: 0.25
"elk"				3,4 NP: 0.5

Parsing with weights (assume all $A \to BC$ rules weighted 1)

"Mary"	0,1 DP: 0.5	0,2	0,3	0,4 S: 0.125
"saw"		1,2 VT: 1	1,3	1,4 VP: 0.25
"the"			2,3 D: 0.5	2,4 DP: 0.25
"elk"				3,4 NP: 0.5

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 $A \in F$
- $(A, x_i x_{i+1} \dots x_n) \Rightarrow (B, x_{i+1} \dots x_n)$ 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)

An example

	Type	Rule	Configuration
0			(ε, Mary saw the elk)
1	SHIFT	DP $\rightarrow$ Mary	(DP, saw the elk)

An example

	Type	Rule	Configuration
0			(ε, Mary saw the elk)
1	SHIFT	DP $\rightarrow$ Mary	(DP, saw the elk)
2	SHIFT	VT $\rightarrow$ saw	(VT DP, the elk)

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)

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)

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)

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)

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)

Another example

	Type	Rule	Configuration
0			(ε, the elk saw Mary)

Another example

	Type	Rule	Configuration
0			(ε, the elk saw Mary)
1	SHIFT	D $\rightarrow$ the	(D, elk saw Mary)

Another example

	Type	Rule	Configuration
0			(ε, the elk saw Mary)
1	SHIFT	D $\longrightarrow$ the	(D, elk saw Mary)
2	SHIFT	NP $\longrightarrow$ elk	(NP D, saw Mary)

Another example

	Type	Rule	Configuration
0			(ε, the elk saw Mary)
1	SHIFT	D $\longrightarrow$ the	(D, elk saw Mary)
2	SHIFT	NP $\longrightarrow$ elk	(NP D, saw Mary)
3	REDUCE	DP $\longrightarrow$ D NP	(DP, saw Mary)

Another example

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

Another example

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

Another example

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

Another example

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

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\longrightarrow$ Mary	(DP P NP D VT, ε)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\rightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\rightarrow$ P DP	(PP NP D VT, ε)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\longrightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\longrightarrow$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\longrightarrow$ NP PP	(NP D VT, ε)
…	…	…	…

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\rightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\rightarrow$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\rightarrow$ NP PP	(NP D VT, ε)
…	…	…	…

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\rightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\rightarrow$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\rightarrow$ NP PP	(NP D VT, ε)
…	…	…	…

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\to$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\to$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\to$ NP PP	(NP D VT, ε)
…	…	…	…

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
3	SHIFT	NP $\to$ elk	(NP D VT, with Mary)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
...	...	...	...
5	SHIFT	DP $\rightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\rightarrow$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\rightarrow$ NP PP	(NP D VT, ε)
...	...	...	...

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
...	...	...	...
3	SHIFT	NP $\rightarrow$ elk	(NP D VT, with Mary)
4	REDUCE	DP $\rightarrow$ D NP	(DP VT, with Mary)

Ambiguity in shift-reduce parsing

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
5	SHIFT	DP $\rightarrow$ Mary	(DP P NP D VT, ε)
6	REDUCE	PP $\rightarrow$ P DP	(PP NP D VT, ε)
7	REDUCE	NP $\rightarrow$ NP PP	(NP D VT, ε)
…	…	…	…

	Type	Rule	Configuration
0			(ε, saw the elk with Mary)
…	…	…	…
3	SHIFT	NP $\rightarrow$ elk	(NP D VT, with Mary)
4	REDUCE	DP $\rightarrow$ D NP	(DP VT, with Mary)
5	REDUCE	VP $\rightarrow$ VT DP	(VP, with Mary)
…	…	…	…

Stage's, shift, and reduce

```
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]
```

Nondeterminism in parsing

At any stage, the next course is not necessarily fully determined.

- This is how ambiguity can arise. After stage 3 in our derivation of *saw the elk with Mary*, do we SHIFT or REDUCE next?

This means that a single `Stage` in general begets a `[Stage]`. We need to try everything we can, to ensure we explore every path.

Trying everything

```haskell
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

`helper` takes steps until we reach a **fixed point**:

```
parseSR :: (Eq cat, Eq term) =>
             CFG cat term -> [term] -> [Stage cat term]
parseSR g ws = helper g [([], ws)] where
  once stages = stages >>= step g     -- taking 1 step
  helper g stages
    | once stages == stages = stages -- fixed point
    | otherwise             = helper g (once stages)
```

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

`parseSR` is really just a recognizer. Could you enrich it to parse to `LBT`'s, or associate parse paths with weights, costs, etc?

Top-down parsing

Shift-reduce parsing works bottom-up: we find out what categories our terminals are, and start building structure.

- Bottom-up is also how CYK and naive parsing algos worked

Top-down parsers **predict** how a derivation will go:

- A starting stage is (A, xs), where $A \in N$ and xs is the input
- A goal stage is $(\varepsilon, \varepsilon)$
- Transitions either expand a non-terminal or match a terminal:
  1. PREDICT: $(A\Phi, x_i \dots x_n) \Rightarrow (BC\Phi, x_i \dots x_n)$, where $A \rightarrow BC \in G$
  2. MATCH: $(A\Phi, x_i x_{i+1} \dots x_n) \Rightarrow (\Phi, x_{i+1} \dots x_n)$, where $A \rightarrow x_i \in G$

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, Mary saw the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, Mary saw the elk)
2	MATCH	DP $\longrightarrow$ Mary	(VP, saw the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S ⟶ DP VP	(DP VP, Mary saw the elk)
2	MATCH	DP ⟶ Mary	(VP, saw the elk)
3	PREDICT	VP ⟶ VT DP	(VT DP, saw the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, Mary saw the elk)
2	MATCH	DP $\longrightarrow$ Mary	(VP, saw the elk)
3	PREDICT	VP $\longrightarrow$ VT DP	(VT DP, saw the elk)
4	MATCH	VT $\longrightarrow$ saw	(DP, the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, Mary saw the elk)
2	MATCH	DP $\longrightarrow$ Mary	(VP, saw the elk)
3	PREDICT	VP $\longrightarrow$ VT DP	(VT DP, saw the elk)
4	MATCH	VT $\longrightarrow$ saw	(DP, the elk)
5	PREDICT	DP $\longrightarrow$ D NP	(D NP, the elk)

An example

	Type	Rule	Configuration
0			(S, Mary saw the elk)
1	PREDICT	S ⟶ DP VP	(DP VP, Mary saw the elk)
2	MATCH	DP ⟶ Mary	(VP, saw the elk)
3	PREDICT	VP ⟶ VT DP	(VT DP, saw the elk)
4	MATCH	VT ⟶ saw	(DP, the elk)
5	PREDICT	DP ⟶ D NP	(D NP, the elk)
6	MATCH	D ⟶ the	(NP, elk)

An example

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

There's something naturally incremental in top-down parsing. We build, or hypothesize, structure, immediately. In bottom-up parsing, REDUCE steps often depend on further-right SHIFT/REDUCE steps.

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S $\rightarrow$ DP VP	(DP VP, the elk saw Mary)

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, the elk saw Mary)
2	PREDICT	DP $\longrightarrow$ D NP	(D NP VP, the elk saw Mary)

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, the elk saw Mary)
2	PREDICT	DP $\longrightarrow$ D NP	(D NP VP, the elk saw Mary)
3	MATCH	D $\longrightarrow$ the	(NP VP, elk saw Mary)

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S ⟶ DP VP	(DP VP, the elk saw Mary)
2	PREDICT	DP ⟶ D NP	(D NP VP, the elk saw Mary)
3	MATCH	D ⟶ the	(NP VP, elk saw Mary)
4	MATCH	NP ⟶ elk	(VP, saw Mary)

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S $\longrightarrow$ DP VP	(DP VP, the elk saw Mary)
2	PREDICT	DP $\longrightarrow$ D NP	(D NP VP, the elk saw Mary)
3	MATCH	D $\longrightarrow$ the	(NP VP, elk saw Mary)
4	MATCH	NP $\longrightarrow$ elk	(VP, saw Mary)
5	PREDICT	VP $\longrightarrow$ VT DP	(VT DP, saw Mary)

Another example

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

Another example

	Type	Rule	Configuration
0			(S, the elk saw Mary)
1	PREDICT	S → DP VP	(DP VP, the elk saw Mary)
2	PREDICT	DP → D NP	(D NP VP, the elk saw Mary)
3	MATCH	D → the	(NP VP, elk saw Mary)
4	MATCH	NP → elk	(VP, saw Mary)
5	PREDICT	VP → VT DP	(VT DP, saw Mary)
6	MATCH	VT → saw	(DP, Mary)
7	MATCH	DP → Mary	(ε, ε)

Non-termination

	Type	Rule	Configuration
…	…	…	…
6	MATCH	D → the	(NP, elk)

Non-termination

	Type	Rule	Configuration
…	…	…	…
6	MATCH	D $\longrightarrow$ the	(NP, elk)
7	PREDICT	NP $\longrightarrow$ NP PP	(NP PP, elk)

Non-termination

	Type	Rule	Configuration
...	...	...	...
6	MATCH	D $\longrightarrow$ the	(NP, elk)
7	PREDICT	NP $\longrightarrow$ NP PP	(NP PP, elk)
8	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP, elk)

Non-termination

	Type	Rule	Configuration
...	...	...	...
6	MATCH	D $\longrightarrow$ the	(NP, elk)
7	PREDICT	NP $\longrightarrow$ NP PP	(NP PP, elk)
8	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP, elk)
9	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP PP, elk)

If we keep PREDICT-ing, the parsing algorithm never terminates.
Which rules are problematic?

Non-termination

	Type	Rule	Configuration
...	...	...	...
6	MATCH	D $\longrightarrow$ the	(NP, elk)
7	PREDICT	NP $\longrightarrow$ NP PP	(NP PP, elk)
8	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP, elk)
9	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP PP, elk)

If we keep PREDICT-ing, the parsing algorithm never terminates.
Which rules are problematic? **Left-recursive** rules, $A \longrightarrow AB$.

Non-termination is **bad**. How might we deal with it here?

Non-termination

	Type	Rule	Configuration
...	...	...	...
6	MATCH	D $\longrightarrow$ the	(NP, elk)
7	PREDICT	NP $\longrightarrow$ NP PP	(NP PP, elk)
8	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP, elk)
9	PREDICT	NP $\longrightarrow$ NP PP	(NP PP PP PP, elk)

If we keep PREDICT-ing, the parsing algorithm never terminates.
Which rules are problematic? **Left-recursive** rules, $A \longrightarrow AB$.

Non-termination is **bad**. How might we deal with it here?

- Look-ahead: curtail the expansions based on length of the input

Earley parsers

Another example of a parser that works top-down, but which does not suffer from non-termination in left-recursive grammars is an **Earley** parser.

We won't discuss Earley parsers in depth in this course, but the way that they avoid non-termination is easy enough to understand:

- Earley parsers have an **agenda** consisting of rules to try next
- A rule like NP $\rightarrow$ NP PP can be added to the agenda
- But adding this rule twice is no different from adding it once: it's already on the agenda!
- Once an agenda is stable (everything **new** that can be added has been), the parser gets to work trying its items

Transition-based parsing

In effect, transition-based parsing constructs something known as a **pushdown automaton** (PDA), in which a pushdown stack of nonterminals functions as an auxiliary memory source.

A pushdown stack is a special kind of memory in which we can only **pop** the top element off the stack, **push** a new element onto the stack, or do composite actions built out of those primitives.

CFG parsing is akin to finite-state parsing, with (pushdown) memory!