Context Free Grammars constituency trees, and CKY

Jonathan May

October 3, 2023

First a little formal language theory

From a formal perspective a language is a (possibly infinite) set of strings (or sentences), where each string is made up of words from an alphabet (or vocabulary). This definition is amenable beyond human languages and has been applied to e.g. computer languages, DNA chains, and mathematical sequences. Much of formal language theory is concerned with the formal devices used to represent these languages.

Finite State Automata and Transducers

A weighted finite-state automaton (wFSA) is one such device. It is a 5-tuple $M = (Q, \Sigma, \lambda, \rho, \delta)$ where $Q = \{q_1, q_2, \ldots, q_m\}$ is a finite set of states, Σ is a finite input alphabet of symbols, $\lambda : Q \to \mathbb{R}$ is an initial weight function, $\rho : Q \to \mathbb{R}$ is a final weight function, and $\delta : Q \times \Sigma \times Q \to \mathbb{R}$ is a transition function. Multiplication of λ for the chosen start state, iterated applications of δ, and ρ for the ending state yield the weight of the string formed by concatenating the series of Σ elements produced in the repeated applications of δ. FSAs and their unweighted counterparts have all kinds of nice properties such as closure under union, concatenation, intersection, and efficient best path, and are the mechanisms driving regular expressions.

Here are two simple wFSAs: the first recognizes any sequence of ‘a’ or ‘b’ that is of even length (for simplicity’s sake, the weights of each member of λ, ρ, and δ can be said to be 1 if the item appears, and 0 if it does not):

$Q_2 = \{q_1, q_2\}; \Sigma = \{a, b\}, \lambda = \{q_1\}, \rho = \{q_1\}, \delta = \{(q_1, a, q_2), (q_1, b, q_2), (q_2, a, q_1), (q_2, b, q_1)\}$

The second recognizes any sequence of ‘a’ that is of length divisible by 3:

$Q_3 = \{q_7, q_8, q_9\}; \Sigma = \{a\}, \lambda = \{q_7\}, \rho = \{q_7\}, \delta = \{(q_7, a, q_8), (q_8, a, q_9), (q_9, a, q_7)\}$

Their intersection recognizes any sequence of ‘a’ that is of length divisible by 6. Intersecting with a simple wFSA that recognizes exactly one string will yield either a wFSA with no transitions, indicating the string is not in the language, or a wFSA that returns that one string, indicating it is in the language. If the wFSA is probabilistic, then the string will be recognized with the probability it occurs in the language.

Finite-state machines can be used to e.g. represent HMMs, though to show this it is better to introduce a generalization, weighted finite state transducers (wFST):

A weighted finite-state transducer is a 6-tuple $M = (Q, \Sigma, \Omega, \lambda, \rho, \delta)$ where Ω is a finite alphabet of output symbols, $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Omega \cup \{\epsilon\} \times Q \to \mathbb{R}$ is the revised transition function, and all other items are as before. ϵ is a special alphabet symbol which indicates no symbol is read/written.

wFSTs are closed under composition, so if you have wFSTs M_1 and M_2, you can build M_3 which behaves as $M_1(M_2)$, i.e. passing a string through the chain of transducers. This allows for the bigram HMM to be fairly cleanly written:

$$\begin{aligned}
M_{\text{trans}} &= \\
Q &= & \{q_x \forall x \in \Sigma \cup \{q_{\text{START}}, q_{\text{END}}\}\} \\
\Sigma &= & \{\text{Penn Treebank POS tags}\} \\
\Delta &= & \Sigma \\
\lambda &= & \{q_{\text{START}} \rightarrow 1\} \\
\rho &= & \{q_{\text{END}} \rightarrow 1\} \\
\delta &= & \{(q_i, j, j, q_j, P(j|i)) \forall i, j \in \Sigma \times \Sigma\} \cup \\
& & \{(q_{\text{START}}, i, i, q_i, P(i|\text{START})) \forall i \in \Sigma\} \cup \\
& & \{(q_i, \epsilon, \epsilon, q_{\text{END}}, P(\text{END}|i)) \forall i \in \Sigma\}
\end{aligned}$$

$$\begin{aligned}
M_{\text{emit}} &= \\
Q &= & \{q\} \\
\Sigma &= & \{\text{Penn Treebank POS tags}\} \\
\Delta &= & \{\text{Vocabulary}\} \\
\lambda = \rho &= & \{q \rightarrow 1\} \\
\delta &= & \{(q, i, j, q, P(j|i)) \forall i, j \in \Sigma \times \Delta\}
\end{aligned}$$

Pushdown automata and context-free grammars

Some languages cannot be recognized by FSAs. For instance, the language $a^n b^n$, i.e. n ‘a’ followed by n ‘b’, for arbitrary n. More practically, the language consisting of strings with balanced parentheses (and possibly other symbols) is also not recognizable by FSAs. The proof of this is quite elegant (it involves something called the ‘pumping lemma’). This is particularly troubling to us because we would like to bracket sentences into hierarchical syntactic chunks.

For example,

[[We/PRP]$_{NP}$ [would/MD [like/VB [[to/TO [bracket/VB [sentences/NNS]$_{NP}$ [into/IN hierarchical/JJ syntactic/JJ chunks/NNS]$_{PP}$]$_{VP}$]$_{VP}$]$_{S}$]$_{VP}$]$_{VP}$ ./.]$_{S}$

This is somewhat more elegantly (but less compactly) written as in Figure 1.

A mechanism that can represent such languages is the *weighted pushdown automaton*, which is like an FSA but with a stack. However it’s far more common to see these languages written in an equivalent formalism, the *weighted context-free grammar* (wCFG). This is a 4-tuple (N, Σ, R, S) where N is a set of non-terminal symbols, σ is a set of terminal symbols, $S \in N$ is a designated start symbol, and $R : N \times (N \cup \Sigma)^* \rightarrow \mathbb{R}$ are the production rules. Here is a CFG (weights of productions shown assumed to be 1) for $a^n b^n$ ($N = \{S\}, \Sigma = \{a, b\}$):

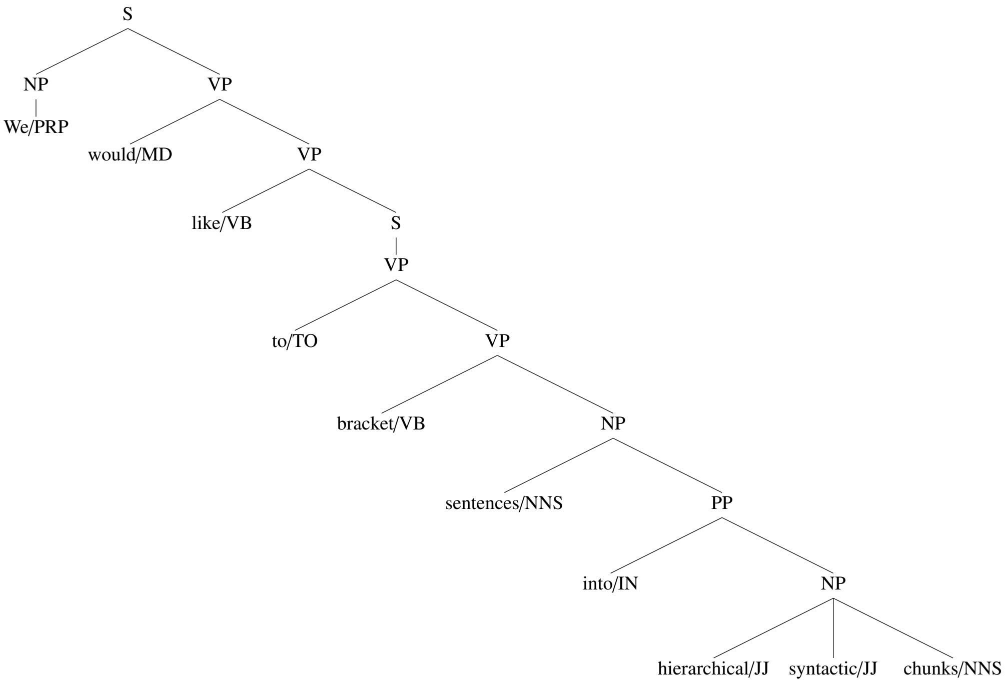


Figure 1: Syntactic Tree

$$
\begin{aligned}
S &\rightarrow ab \\
S &\rightarrow aSb
\end{aligned}
$$

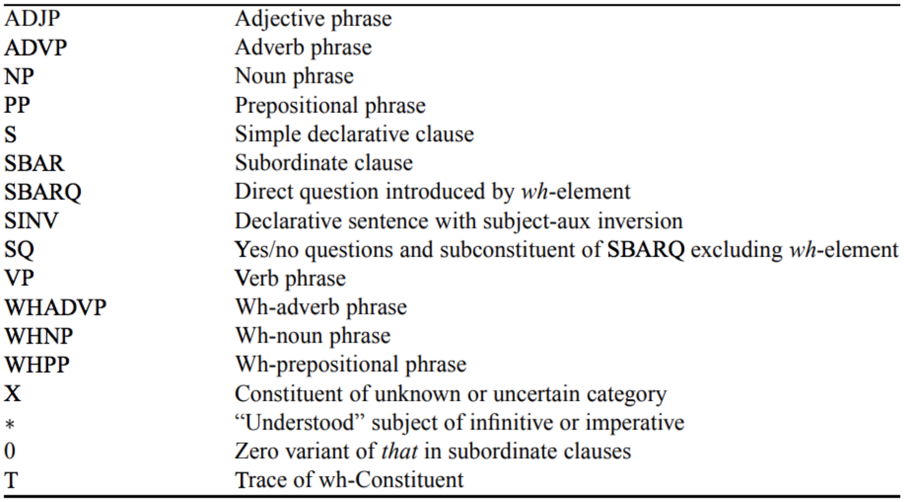
Here are the productions that can be used to form the tree in Figure 1 (excluding tags for POS to word; POS listed in lowercase):

$$
\begin{aligned}
S &\rightarrow NP\,VP\,. \\
S &\rightarrow VP \\
NP &\rightarrow prp \\
NP &\rightarrow nns\,PP \\
NP &\rightarrow jj\,jj\,nn \\
VP &\rightarrow md\,VP \\
VP &\rightarrow vb\,S \\
VP &\rightarrow to\,VP \\
VP &\rightarrow vb\,NP \\
PP &\rightarrow in\,NP
\end{aligned}
$$

Penn Treebank

Built in the 90s at Penn, for about 1 million dollars. 1 million words in about 40,000 sentences of WSJ text. First large scale analysis of naturally occurring syntax (other components include much more POS tagging, speech annotation). Compared to the POS tag set, there are many fewer tree labels:

Table 1.2. The Penn Treebank syntactic tagset



ADJP	Adjective phrase
ADVP	Adverb phrase
NP	Noun phrase
PP	Prepositional phrase
S	Simple declarative clause
SBAR	Subordinate clause
SBARQ	Direct question introduced by *wh*-element
SINV	Declarative sentence with subject-aux inversion
SQ	Yes/no questions and subconstituent of SBARQ excluding *wh*-element
VP	Verb phrase
WHADVP	Wh-adverb phrase
WHNP	Wh-noun phrase
WHPP	Wh-prepositional phrase
X	Constituent of unknown or uncertain category
*	"Understood" subject of infinitive or imperative
0	Zero variant of *that* in subordinate clauses
T	Trace of wh-Constituent

However, the annotation guide for the treebank is 318 pages long (compared to 37 for POS tags).

This was meant to be a documentation of how people really constructed (English, largely news) sentences rather than being told what was and was not done by linguists. Also now we could think, if given a new sentence, can we automatically annotate it in the same way?

BTW, the treebank is divided into sections, and it's common for people to train on sections 2-21, use dev for section 22, and evaluate on section 23. This has enabled comparable results over about 25 years, but there is concern that we've overfit on this data set by now! There are other treebanks that have been constructed, for other languages, and for English, but none is as consistent or large as Penn Treebank.

Evaluation

Parsing scores are typically given as an F1 measure, comparing gold (i.e. reference) to hypothesis brackets. Let's assume the left side of Figure 2 is the gold sentence and the right side is the hypothesis. Then, the brackets for each are:

gold	hyp
(S 0 7)	(S 0 7)
(NP 0 1)	(NP 0 1)
(VP 1 7)	(VP 1 7)
(NP 2 4)	(NP 2 7)
(PP 4 7)	(NP 2 4)
(NP 5 7)	(PP 4 7)
	(NP 5 7)

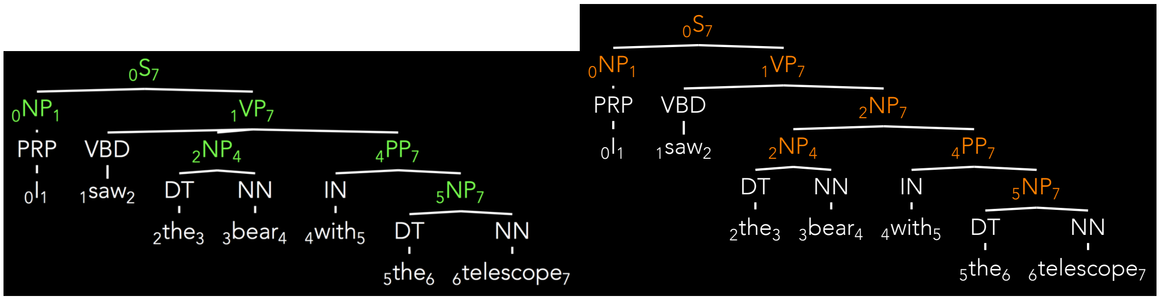


Figure 2: Two parses of a sentence

F1 is the harmonic mean of *recall* and *precision*. In such a scenario we can divide up items in a response into *correct* (items in hypothesis and reference), *missed* (items in reference but not hypothesis), and *spurious* (items in hypothesis but not reference. *Precision* is $\frac{\text{correct}}{\text{correct+spurious}}$. *Recall* is $\frac{\text{correct}}{\text{correct+missed}}$. F1 = $2 \cdot \frac{\text{precision}\cdot\text{recall}}{\text{precision+recall}}$.

In the example above, 6 items are correct, one is spurious, and none are missed. Thus the precision is 6/7 = .857, the recall is 6/6 = 1.0, and the F1 is .923.

Given a grammar and a sentence, how do we efficiently find a parse tree for that sentence? More importantly, how do we find the most likely parse tree? The core of those answers are in the CKY algorithm, a bottom-up dynamic programming algorithm. CKY requires rules be in a special form called ‘Chomsky Normal Form’ (CNF) that only allows rules of the following form:

$$X \rightarrow Y\,Z$$
$$X \rightarrow a$$

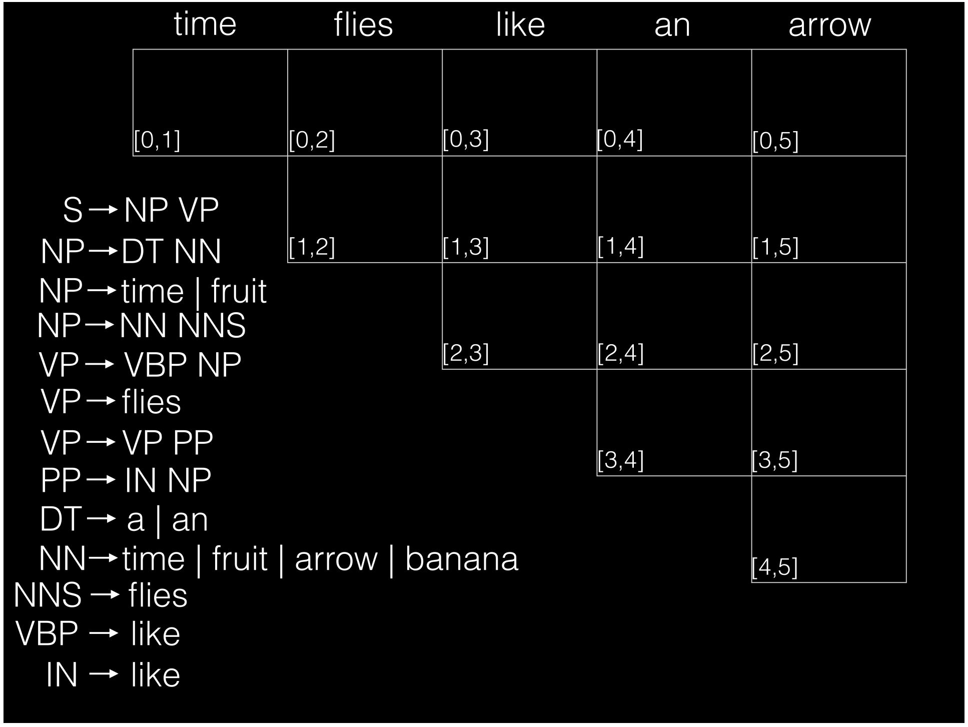
where X, Y, Z are nonterminals and a is terminal. There must be *exactly* two nonterminals on the RHS of a rule or there can be exactly one terminal. Although there are well established techniques for converting, for now let’s assume we already have a grammar in this form.

Here’s the core algorithm (see also alg. 13 on p. 241 of eisenstein):

```
chart = defaultdict(lambda: defaultdict(set)) # start->end->labels
bpt = defaultdict(lambda: defaultdict(set)) # start->end->backpointers
for w in range(1, len(sent)+1): # width
  for start in range(len(sent)-w+1):
    end = start+w
    if w == 1:
      for lhs in terms[words[start]]:
        chart[start][end].add(lhs)
    else:
      for mid in range(start+1, end):
        for rhs1 in chart[start][mid]:
          for rhs2 in chart[mid][end]:
            for lhs in nterms[rhs1][rhs2]:
```

```
        chart[start][end].add(lhs)
        bpt[start][end].add((mid, rhs1, rhs2))
```

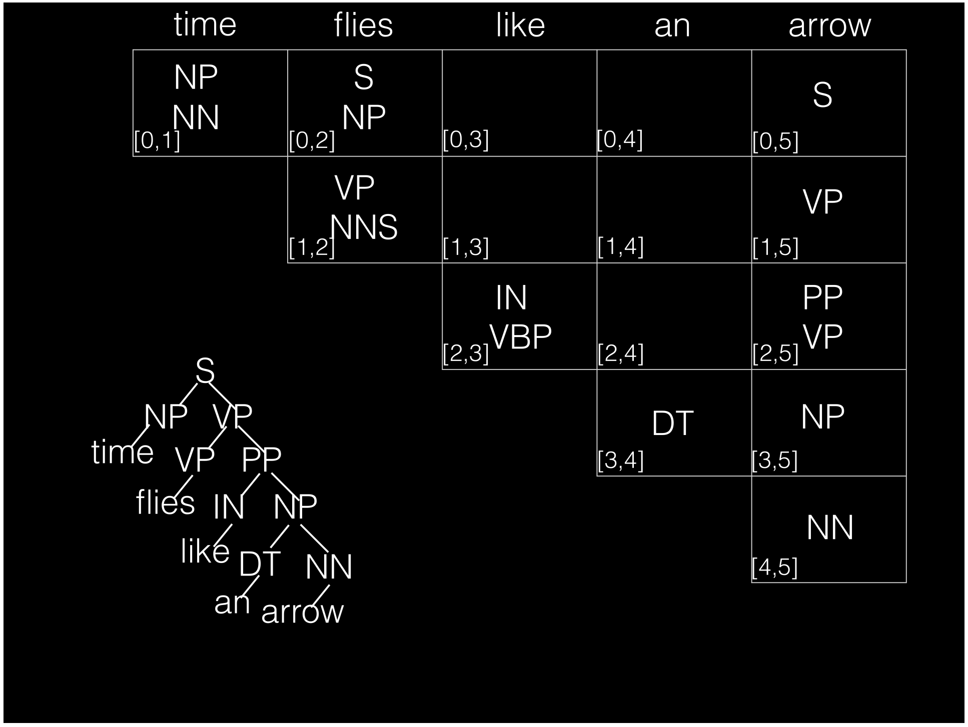
Let's work it out with a small example:



time	flies	like	an	arrow
[0,1]	[0,2]	[0,3]	[0,4]	[0,5]
	[1,2]	[1,3]	[1,4]	[1,5]
		[2,3]	[2,4]	[2,5]
			[3,4]	[3,5]
				[4,5]

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

When done you should have this:



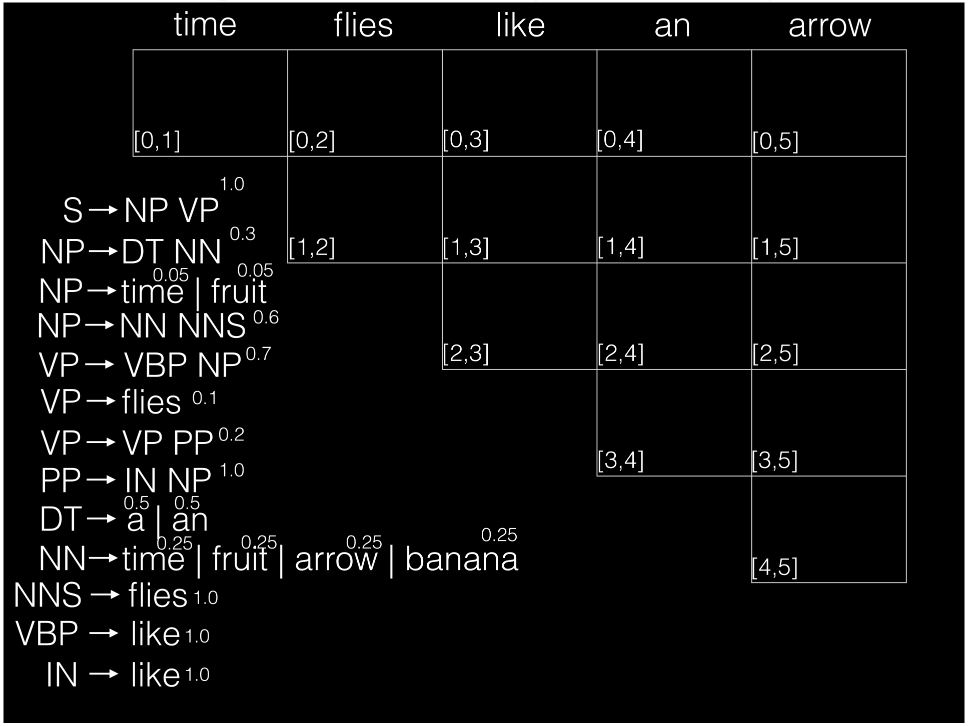
time	flies	like	an	arrow
NP NN [0,1]	S NP [0,2]	[0,3]	[0,4]	S [0,5]
	VP NNS [1,2]	[1,3]	[1,4]	VP [1,5]
		IN VBP [2,3]	[2,4]	PP VP [2,5]
			DT [3,4]	NP [3,5]
				NN [4,5]

Building probabilistic grammars

How do we actually get grammars? We can read them off of the trees in the treebank. As seen above, these are not always in CNF. Simpler than changing the grammar, we can modify the trees themselves. We can do the same with unary chains, collapsing them.

How to choose probabilities for grammar rules? To be probabilistic, the sum of all rules with the same LHS should be 1.0; given that we're still being generative here, this is $P(T|W)P(T) = P(T, W)$ and with the chain rule and markov and independence assumptions as before, we ultimately want a set of $P(RHS|LHS)$ probabilities to multiply. We calculate these empirically from the corpus. We then can modify our CKY algorithm above to calculate weights and choose the maximum for every LHS in a cell.

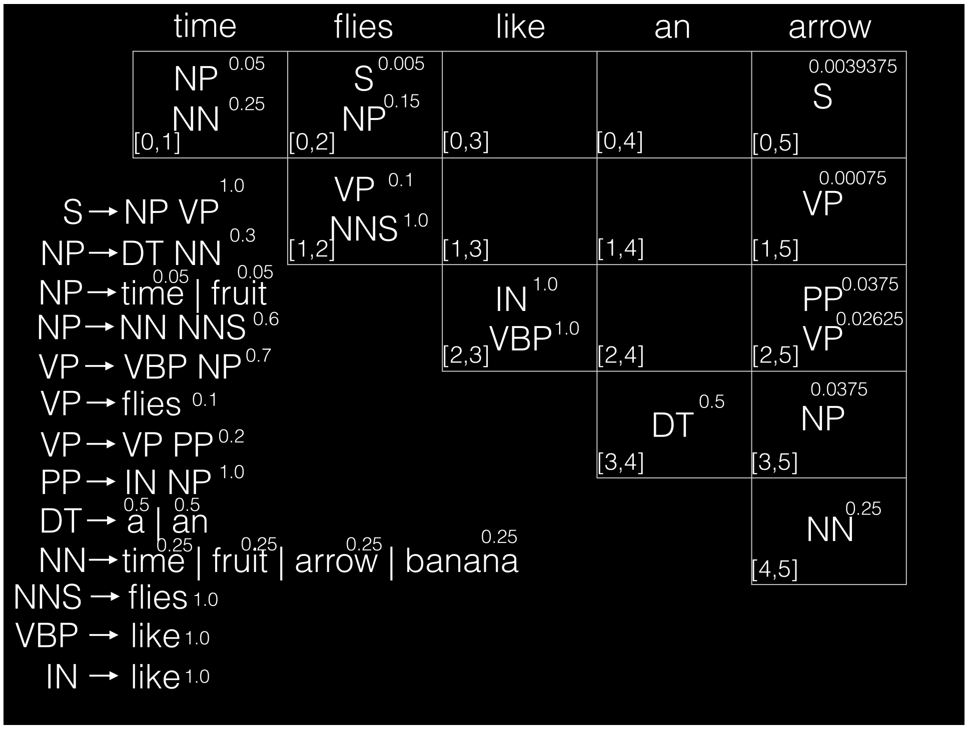
Here is the example from above with weights to work through:



time	flies	like	an	arrow
[0,1]	[0,2]	[0,3]	[0,4]	[0,5]
	[1,2]	[1,3]	[1,4]	[1,5]
		[2,3]	[2,4]	[2,5]
			[3,4]	[3,5]
				[4,5]

S → NP VP 1.0
NP → DT NN 0.3
NP → time 0.05 | fruit 0.05
NP → NN NNS 0.6
VP → VBP NP 0.7
VP → flies 0.1
VP → VP PP 0.2
PP → IN NP 1.0
DT → a 0.5 | an 0.5
NN → time 0.25 | fruit 0.25 | arrow 0.25 | banana 0.25
NNS → flies 1.0
VBP → like 1.0
IN → like 1.0

And here it is filled out:



time	flies	like	an	arrow
NP 0.05 NN 0.25 [0,1]	S 0.005 NP 0.15 [0,2]	[0,3]	[0,4]	S 0.0039375 [0,5]
	VP 0.1 NNS 1.0 [1,2]	[1,3]	[1,4]	VP 0.00075 [1,5]
		IN 1.0 VBP 1.0 [2,3]	[2,4]	PP 0.0375 VP 0.02625 [2,5]
			DT 0.5 [3,4]	NP 0.0375 [3,5]
				NN 0.25 [4,5]

S → NP VP 1.0
NP → DT NN 0.3
NP → time 0.05 | fruit 0.05
NP → NN NNS 0.6
VP → VBP NP 0.7
VP → flies 0.1
VP → VP PP 0.2
PP → IN NP 1.0
DT → a 0.5 | an 0.5
NN → time 0.25 | fruit 0.25 | arrow 0.25 | banana 0.25
NNS → flies 1.0
VBP → like 1.0
IN → like 1.0

The analogue to the forward algorithm, the *inside* algorithm, computes the partition function (sum of probabilities of all trees) by replacing max with add.

It turns out that just using simply extracted rules gets us parse scores of around 73% which is not good; modern pre-neural parsers are in the mid-90s. Why are these rules not good enough on their own? Both because the rules extracted from trees are too specific and because they're not specific enough!

0.1 Too Specific – Markov binarization

Consider the tree:

```
(NP
  (DT the)
  (JJS tallest)
  (NN steel)
  (NN building)
  (NN antenna)
  (PP
    (IN in)
    (NP
      (NNP America))))
```

This yields the rule

```
NP -> DT JJS NN NN NN PP
```

which is totally useless for

```
(NP
  (DT the)
  (JJS tallest)
  (NN building)
  (PP
    (IN in)
    (NP
      (NNP America))))
```

Previously I (hopefully in class, but not in notes) mentioned that changing the trees is an easy way to get into CNF. For perfect reconstruction you can do this by introducing one-time nonterminals, e.g.

```
(NP
  (DT the)
  (T145-1
    (JJS tallest)
    (T145-2
      (NN steel)
      (T145-3
        (NN building)
        (T145-4
          (NN antenna)
          (PP
             (IN in)
             (NP_NNP America)))))))
```

Though in this case that would not generalize to even "The tallest iron building antenna in America" so at least we should introduce terminals for each pattern:

```
(NP
  (DT the)
  (DT^JJS^NN^NN^NN-1
    (JJS tallest)
    (DT^JJS^NN^NN^NN-2
      (NN steel)
      (DT^JJS^NN^NN^NN-3
        (NN building)
        (DT^JJS^NN^NN^NN-4
          (NN antenna)
          (PP
             (IN in)
             (NP_NNP America)))))))
```

But if we introduce reusable nonterminals we can get more flexible rules. For instance, we can remember the nonterminal we're in and the pos tag to our left:

```
(NP
  (DT the)
  (NP^DT
    (JJS tallest)
    (NP^JJS
      (NN steel)
      (NP^NN
        (NN building)
        (NP^NN
          (NN antenna)
          (PP
             (IN in)
             (NP_NNP America)))))))
```

This gives us rules like

```
NP^JJS -> NP^NN
```

which are useful in the smaller sentence.

0.2 Not Specific Enough I: modeling parent-child behavior

Let's say we saw:

```
(NP
  (NP the man)
  (PP in the car))
```

90 times (eliding the unimportant details of the trees)
and

```
(NP
  (NP the man)
  (PP in the car)
  (PP with the dog))
```

10 times. Using MLE, we get

```
 NP -> NP PP 90/200 = .45
 NP -> NP PP PP 10/20 = .05
(NP -> DT NN 100/20 = .5)
```

What happens if we then want to parse 'the man in the car with the dog'? The sentence seen in training scores .05 for the combining rule, while this parse:

```
(NP
  (NP
    (NP the man)
    (PP in the car))
  (PP with the dog))
```

scores .45 × .45 = .2020 for the combining rules! This shouldn't be! A chain (NP NP NP) not seen in training scores higher than the chain seen in training.

What can we do? Annotate with more context – give nonterminals their parent symbols as well:

```
(NP^VP
  (NP^NP the man)
  (PP^NP in the car))
```

and

```
(NP^VP
  (NP^NP the man)
  (PP^NP in the car)
  (PP^NP with the dog))
```

Now the rule

```
NP^VP -> NP^NP PP^NP
```

can't be used twice!

0.3 Not Specific Enough II: modeling lexical behavior

It turns out words do matter! Consider:

```
(S
  (NNS workers)
  (VP
    (VP
      (VBD dumped)
      (NP
        (NNS sacks)))
    (PP
      (IN into)
      (NP
        (DT a)
        (NN bin)))))
```

seems correct. The dumping is into a bin. Consider an alternative:

```
(S
  (NNS workers)
  (VP
    (VP
      (VBD dumped)
      (NP
      (NP
        (NNS sacks))
      (PP
        (IN into)
        (NP
          (DT a)
          (NN bin)))))
```

They dumped a thing called "sacks into a bag." Seems wrong.

Which is more likely? It comes down to the difference between these two rules:

```
VP -> VP PP
NP -> NP PP
```

Both are good. Neither directly compared. This seems arbitrary. But PPs with 'into' have a strong preference to be attached to VPs, not NPs. Consider if it was instead 'sacks of oranges.' What to do? Annotate labels with their lexical 'heads.' What's a head? the most important word in a phrase. How are these determined? Rules, actually. That were written down in 1995.[1] Remember, it's only 21 types so it's not that bad. Here's an example for VP: [VBD VBN MD VBZ TO VB VP VBG VBP ADJP NP]. Use the leftmost of the first of these categories, if it appears. If it doesn't use the left most of the next one and so on. Following the rules you get:

```
(S/dumped
  (NNS/workers workers)
  (VP/dumped
    (VP/dumped
      (VBD dumped)
      (NP/sacks
        (NNS/sacks sacks)))
    (PP/into
      (IN/into into)
      (NP/bin
        (DT/a a)
        (NN/bin bin)))))
```

Now we are comparing these rules

```
VP/dumped -> VP/dumped PP/into
NP/sacks -> NP/sacks PP/into
```

The first seems much more likely.

(In an older version of this course we'd now have to talk about how there aren't likely to be sufficient statistics to estimate these fairly fine-grained rules, so we'll have to add smoothing. The smoothing for parse trees can be quite complicated and would take one lecture at least, but using neural approaches ends up allowing us to skip over all of that.)

Lexicalization proved very important! So important that there isn't a lot of constituent parsing research any more since a new formalism for syntax became dominant...dependency trees!

[1] http://www.cs.columbia.edu/~mcollins/papers/heads