Faculty of Computer Science, Dalhousie University *4-Dec-2025*

CSCI 4152/6509 — Natural Language Processing

Lecture 19: Other Types of Grammars: PCFG, DCG

Location: Studley LSC-Psychology P5260 Instructor: Vlado Keselj
Time: 14:35 – 15:55

Previous Lecture

- Natural language syntax:
  - phrase structure, clauses, sentences
  - Parsing, parse tree examples
- Context-Free Grammars review:
  - formal definition
  - inducing a grammar from parse trees
  - derivations, and other notions
- Bracket representation of a parse tree
- Typical phrase structure rules in English:
  - S, NP, VP, PP, ADJP, ADVP

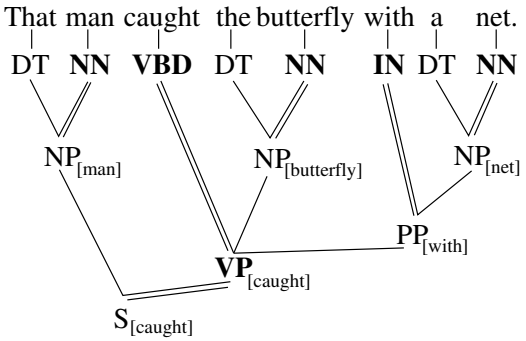
22 Heads and Dependency

Heads and Dependency

- a phrase typically has a central word called head, while other words are direct or indirect dependents
- a head is also called a governor, although sometimes these concepts are considered somewhat different
- phases are usually called by their head; e.g., the head of a noun phrase is a noun

Example with Heads and Dependencies

- the parse tree of "That man caught the butterfly with a net."
- annotate dependencies, head words



Head Feature Principle

The **Head Feature Principle** is the principle that a constituent in a parse tree will have the same set of some characteristic features as its head child. The features that are shared in this way between parent and head child are called **head features**.

The head feature principle is particularly emphasized in some grammar formalisms, such as the Head-driven Phrase Structure Grammar (HPSG).

If we use a context-free grammar, or a similar grammar formalism, to parse natural language, an interesting question is how to detect head-dependent relations. We can achieve it by annotating head child in each context-free rule. Sometimes in literature, these heads are labeled on the right-hand side of each rule using a subscript H, as in:

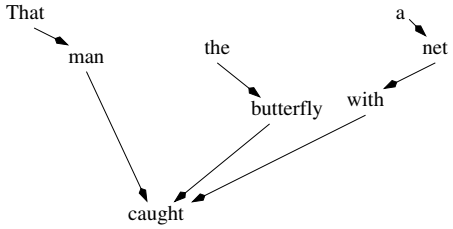
$$NP \rightarrow DT\ NN_H$$

or, using a Head-driven Phrase Structure Grammar (HPSG) style, they are labeled using a letter H before the child constituent, as in:

$$[NP] \rightarrow [DT]\ H[NN]$$

Dependency Tree

- dependency grammar
- example with "That man caught the butterfly with a net."



A dependency tree consists of direct dependence relations between words in a sentences, and the words are the nodes of the tree. A typed dependency tree also includes labels of the edges, describing the types of dependencies. A dependency tree can be easily produced from a context-free parse tree in which the heads of the phrases are labelled.

A dependency grammar is a grammar that defines rules that are used to form a dependency tree.

Arguments and Adjuncts

- There ar two kinds of dependents:
  1. **arguments,** which are required dependents, e.g.,
     We deprived him of food.
  2. **adjuncts,** which are not required;
     - they have a "less tight" link to the head, and
     - can be moved around more easily

     Example:
     We deprived him of food yesterday in the restaurant.

23 Probabilistic Context-Free Grammar (PCFG)

Reading: Chapters 13 and 14

The main issue with the CFG model when parsing natural languages is ambiguity. A typical natural grammar that we can create to describe a natural language is ambiguous; which means that for many sentences it will give two or

more parse trees and we will have a problem of choosing one of those trees, likely one that does not represent an intended meaning of the sentence. A way to solve this problem is the use of **Probabilistic Context-Free Grammar (PCFG)**, also known as the **Stochastic Context-Free Grammar (SCFG)**.

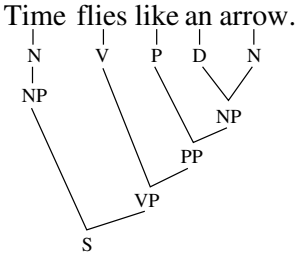
Both, n-gram model and HMM are linear models, which may not be most suitable to model the structured nature of natural language syntax. While Bayesian Networks could be one way of capturing structured nature of language in a probabilistic way, PCFGs represent another way, which is directly derived from the Context-Free Grammar formalism.

There are arguments that the use of PCFGs could also improve language modeling. For example, if we consider the words that may continue the sentence

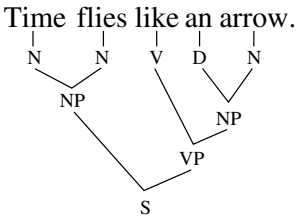
The velocity of the seismic waves rises to...

a linear model will likely assign a higher probability to the word "rise" after the plural "waves" than to the word "rises," which actually correctly appears in the sentence and agrees with the head "velocity" of the noun phrase.

As previously described, context-free grammars represent a structural model for describing syntax. For example, the syntax of the sentence "Time flies like an arrow." could be represented as the following context-free parse tree:



There are known efficient parsing algorithms for context-free grammars in the theory of of formal languages, and applications such as design of compilers and interpreters for programming languages. Two examples of such parsing approaches are recursive descent parsing and shift-reduce LR parsing. A large obstacle in applying these parsers to the problem of NL parsing is in the requirement that the language is unambiguous. Natural languages are inherently ambiguous and a parser for natural language must handle ambiguous grammars and ambiguous input. For example, if we assume a different meaning of the above sentence, we obtain a different parse tree, like the following one:



The above two trees induce the following CFG:

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	N	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

To have a complete CFG specification, we need to add that the set of terminals is {'time', 'arrow', 'flies', 'an', 'like'}, the set of non-terminals is { S, NP, VP, D, N, PP, P, V}, and the start symbol is S.

If we parse the same sentence using this grammar, then we will obtain at least two different parse trees. To make parsing more usable, we need a way of assigning a score or probability to each tree, so we can always choose the "best" parse tree in a certain sense.

23.1 PCFG as a Probabilistic Model

To transform a CFG into a probabilistic model we model derivations as stochastic process in a generative way. For example, the left-most derivation corresponding to the first parse tree described above is:

$$\begin{aligned} \text{S} &\Rightarrow \text{NP VP} \Rightarrow \text{N VP} \Rightarrow \text{time VP} \Rightarrow \text{time V PP} \Rightarrow \text{time flies PP} \Rightarrow \text{time flies P NP} \\ &\Rightarrow \text{time flies like NP} \Rightarrow \text{time flies like D N} \Rightarrow \text{time flies like an N} \Rightarrow \text{time flies like an arrow} \end{aligned}$$

At each step of the derivation, given a non-terminal that needs to be re-written, we usually have several options, corresponding to several rules that have this non-terminal on the left-hand side.

Hence, we calculate the probability of the tree by multiplying probabilities of all rules occurring in the tree:

$$\begin{aligned} \mathrm{P}(\text{first tree}) &= \mathrm{P}(N \to time)\mathrm{P}(V \to flies)\mathrm{P}(P \to like)\mathrm{P}(D \to an) \\ &\quad \mathrm{P}(N \to arrow)\mathrm{P}(NP \to N)\mathrm{P}(NP \to DN)\ldots\mathrm{P}(S \to NPVP) \end{aligned}$$

If we assign the following probabilities to the rules:

S	→	NP VP	/1	VP	→	V NP	/.5	N	→	time	/.5
NP	→	N	/.4	VP	→	V PP	/.5	N	→	arrow	/.3
NP	→	N N	/.2	PP	→	P NP	/1	N	→	flies	/.2
NP	→	D N	/.4					D	→	an	/1
V	→	like	/.3								
V	→	flies	/.7								
P	→	like	/1								

then the probability of the first tree is 0.0084, and the probability of the second tree is 0.00036. We can conclude that the first tree is more likely, which should correspond to our intuition.

The probability assigned to a rule $N \to \alpha$ is the probability $\mathrm{P}(N \to \alpha|N)$, so if $N \to \alpha_1$, $N \to \alpha_2$, …, $N \to \alpha_n$ are all rules with the nonterminal N on its left hand side, then

$$\sum_{i=1}^{n} \mathrm{P}(N \to \alpha_i) = 1$$

These probabilities are easily learned from a set of parse trees, usually called parse treebank, by counting the number of occurrences of distinct rules.

This model is a language model, since the sum of probabilities of all sentences in the language is 1. Actually, in order to be a language model, we also require that the grammar is proper, i.e., that all infinite trees have probability 0, which is not always the case. We will not go into further details regarding this question here, except noting that it has been proved that any PCFG with probabilities induced from a treebank is proper.

Computational Tasks for PCFG Model

- Evaluation

$$P(\text{tree}) = ?$$

- Generation
- Learning
- Inference
  - Marginalization

$$\mathrm{P}(\text{sentence}) = ?$$

– Conditioning

$$\mathrm{P}(\text{tree}|\text{sentence}) = ?$$

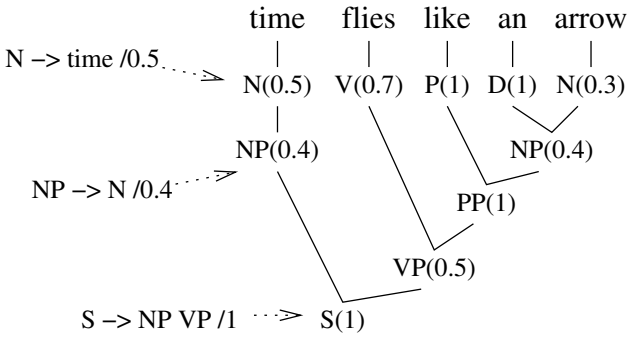
– Completion

$$\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$$

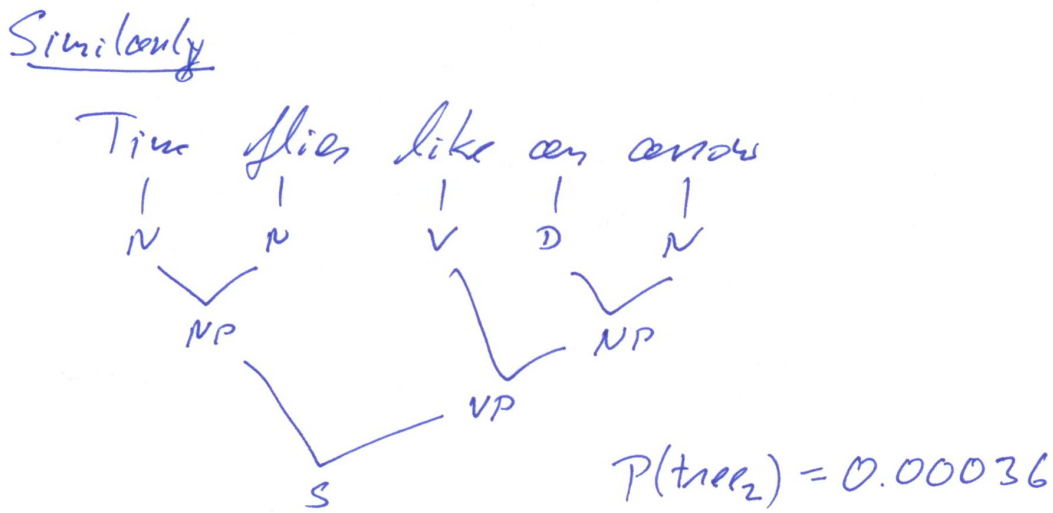
Evaluation

Given a tree t, what is $\mathrm{P}(t)$? As seen in the example, we simply multiply probabilities associated with each node of the tree.

The following tree illustrate the evaluation example using our example:



P(tree) = 0.5x0.7x1x1x0.3x0.4x0.4x1x0.5x1=0.0084



Generation

We can generate (sample) sentences by starting with the initial nonterminal S, and by rewriting it using a rule $S \to \alpha$ according to the probabilities assigned to the rules that have S on their left hand side. We obtain a sentential

form—a string of terminals and nonterminals. Take the first nonterminal in this form, and rewrite it in the same way; and so on. The loop stops when there are no nonterminals, i.e., when we obtain a sample sentence.

An interesting question is whether this process stops. If the grammar is proper, it stops with probability 1. If the grammar is not proper, we might easily be trapped in an infinite derivation.

Generation (sampling)

S => NP VP => N VP => flies VP => ...

S → NP VP / 1

NP → N / 0.4
NP → N N / 0.2
NP → D N / 0.4

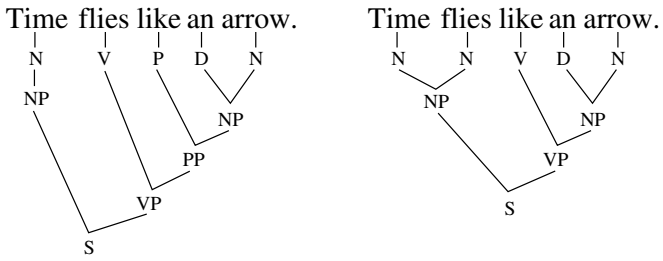
N → time / 0.5
N → arrow / 0.3
N → flies / 0.2

- choose rule randomly according to the given distribution

Question: Is the process going to stop?
A: Stops with probability 1 if the grammar is proper.
Good News: A grammar learned from a corpus is always proper.

Learning

An direct approach to learning from a parse treebank is by counting the number of rule occurrences. For example, let us revisit the two parse trees of the sentence "Time flies like an arrow.", given at the beginning of this chapter:



If we extract all the rules from the non-leaf vertices of these trees, and sort them by the left hand side, we obtain the

following list:

D	→	an	N	→	arrow	S	→	NP VP
D	→	an	N	→	arrow	S	→	NP VP
NP	→	D N	N	→	flies	VP	→	V NP
NP	→	D N	N	→	time	VP	→	V PP
NP	→	N N	N	→	time	V	→	flies
NP	→	N	PP	→	P NP	V	→	like
			P	→	like			

Now, instead of repeating the same rules, we can only include them once and record their counts like this:

D	→	an	$\times 2$	N	→	arrow	$\times 2$	S	→	NP VP	$\times 2$
NP	→	D N	$\times 2$	N	→	flies	$\times 1$	VP	→	V NP	$\times 1$
NP	→	N N	$\times 1$	N	→	time	$\times 2$	VP	→	V PP	$\times 1$
NP	→	N	$\times 1$	PP	→	P NP	$\times 1$	V	→	flies	$\times 1$
				P	→	like	$\times 1$	V	→	like	$\times 1$

We can not calculate total counts for all rules with the same left-hand side and obtain totals: 2 for D, 4 for NP, 5 for N, 1 for PP, 1 for P, 2 for S, 2 for VP, and 2 for V. Finally, we calculate probabilities of all rules by dividing their counts with corresponding total counts for the corresponding left-hand side non-terminal:

D	→	an	$/1$	N	→	arrow	$/0.4$	S	→	NP VP	$/1$
NP	→	D N	$/0.5$	N	→	flies	$/0.2$	VP	→	V NP	$/0.5$
NP	→	N N	$/0.25$	N	→	time	$/0.4$	VP	→	V PP	$/0.5$
NP	→	N	$/0.25$	PP	→	P NP	$/1$	V	→	flies	$/0.5$
				P	→	like	$/1$	V	→	like	$/0.5$

This final list of rules with their probabilities is the PCFG *learned* from the two parse trees given earlier. A set of parse trees is called a *treebank*, particularly if it is large and manually curated, so the two given parse trees can be called a small treebank. We can also say that this final PCFG is *induced* or *generated* by the given treebank.

Inference

Marginalization: $\mathrm{P}(\text{sentence}) = ?$

Conditioning: $\mathrm{P}(\text{tree}|\text{sentence}) = ?$

Completion: $\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$

24 Parsing Natural Language in Prolog

24.1 Parsing Natural Language in Prolog using Difference Lists

Prolog's unique built-in features of backtracking and unification are very convenient for building certain types of parsers for natural languages. To illustrate how this can be done, let us start with a minimal toy example of two sentecnes in English.

Example: Let us consider these twe following two sentences in English: "the dog runs" and "the dogs run". We will use the following simple CFG to parse these sentences:

```
S  -> NP VP          N  -> dog
NP -> D N            N  -> dogs
D  -> the            VP -> run
                     VP -> runs
```

The grammar is not really correct for English since it allows some non-grammatical sentences to be accepted, such as “the dog run”, but we are going to ignore this for now.

The problem is: How could we build a parser with this grammar to parse a simple sentence such as: “the dog runs”

Using Difference Lists: Idea

Consider rule: `S -> NP VP` and sentence [the,dog,runs]

Using Difference Lists to Parse CFG

The problem of parsing using this grammar can be expressed in the following way in Prolog:

```
s(S,R)  :- np(S,I), vp(I, R).
np(S,R) :- d(S,I), n(I,R).
d([the|R], R).
n([dog|R], R).
n([dogs|R], R).
vp([run|R], R).
vp([runs|R], R).
```

Additional Material

Parsing using Difference Lists

Save this in file `parse.prolog`. On Prolog prompt we type:

```
?- ['parse.prolog'].
% parse.prolog compiled 0.00 sec, 1,888 bytes

Yes
?- s([the,dog,runs],[]).

Yes
?- s([runs,the,dog],[]).

No
```

24.2 Definite Clause Grammar (DCG)

Basic Definite Clause Grammar (DCG)

- DCG — Prolog built-in mechanism for parsing

Example

```
s  --> np, vp.
np --> d, n.
```

```
d  --> [the].
n  --> [dog].
n  --> [dogs].
vp --> [run].
vp --> [runs].
```

This is a simplified Prolog notation for predicates described previously. The variables S, R, and I that were used for difference lists before, are now implicitly used in the DCG grammar. We can use more Prolog variables. These variables are added at the front of the predicate argument list in the implicit representation.

24.3 Building a Parse Tree in DCG

Building a Parse Tree

A parse tree can be built in the following way:

```
s(s(Tn,Tv))    --> np(Tn), vp(Tv).
np(np(Td,Tn)) --> d(Td), n(Tn).
d(d(the))      --> [the].
n(n(dog))      --> [dog].
n(n(dogs))     --> [dogs].
vp(vp(run))    --> [run].
vp(vp(runs))   --> [runs].
```

At Prolog prompt we type and obtain:

```
?- s(X, [the, dog, runs], []).
 X = s(np(d(the),n(dog)),vp(runs));
```

24.4 Example of Handling Agreement in DCG

Handling Agreement

```
s(s(Tn,Tv))       --> np(Tn,A), vp(Tv,A).
np(np(Td,Tn),A)   --> d(Td), n(Tn,A).
d(d(the))         --> [the].
n(n(dog),sg)      --> [dog].
n(n(dogs),pl)     --> [dogs].
vp(vp(run),pl)    --> [run].
vp(vp(runs),sg)   --> [runs].
```

This grammar will accept sentences “the dog runs” and “the dogs run” but not “the dog run” and “the dogs runs”. Other phenomena can be modeled in a similar fashion.

24.5 Embedded Code in DCG

Embedded Code

We can embed additional Prolog code using braces, e.g.:

```
s(T)   --> np(Tn), vp(Tv), {T = s(Tn,Tv)}.
```

and so on, is another way of building the parse tree.

24.6 Expressing PCFGs in DCGs

Expressing PCFGs in DCGs

Let us consider the previous example of a PCFG:

S	→	NP VP	/1	VP	→	V NP	/.5	N	→	time	/.5
NP	→	N	/.4	VP	→	V PP	/.5	N	→	arrow	/.3
NP	→	N N	/.2	PP	→	P NP	/1	N	→	flies	/.2
NP	→	D N	/.4					D	→	an	/1
V	→	like	/.3								
V	→	flies	/.7								
P	→	like	/1								

The probabilities can be calculated as an additional argument:

```
s(T,P) --> np(T1,P1), vp(T2,P2),
           {T = s(T1,T2), P is P1 * P2 * 1}.
np(T,P) --> n(T1,P1), {T = n(T1), P is P1 * 0.4}.
```

and so on.

A full DCG code for the above PCFG could be:

```
s(s(Tn,Tv),P) --> np(Tn,P1), vp(Tv,P2), {P is P1 * P2}.
np(np(T),P) --> n(T,P1), {P is P1 * 0.4}.
np(np(T1,T2),P) --> n(T1,P1), n(T2,P2), {P is P1 * P2 * 0.2}.
np(np(Td,Tn),P) --> d(Td,P1), n(Tn,P2), {P is P1 * P2 * 0.4}.
v(v(like), 0.3) --> [like].
v(v(flies), 0.7) --> [flies].
p(p(like), 1.0) --> [like].
vp(vp(Tv,Tn), P) --> v(Tv, P1), np(Tn, P2), {P is P1 * P2 * 0.5}.
vp(vp(Tv,Tp), P) --> v(Tv, P1), pp(Tp, P2), {P is P1 * P2 * 0.5}.
pp(pp(Tp,Tn), P) --> p(Tp, P1), np(Tn, P2), {P is P1 * P2}.
n(n(time), 0.5) --> [time].
n(n(arrow), 0.3) --> [arrow].
n(n(flies), 0.2) --> [flies].
d(d(an), 1.0) --> [an].
```

After loading this grammar into Prolog interpreter, and then after issuing the query:
`?- s(T,P,[time,flies,like,an,arrow],[]).`
the interpreter would reply with:

```
T = s(np(n(time)), vp(v(flies), pp(p(like), np(d(an), n(arrow)))))
P = 0.0084
```

and after typing `;` (semi-colon), we get:

```
T = s(np(n(time), n(flies)), vp(v(like), np(d(an), n(arrow))))
P = 0.00036
```

After typing second '`;`', the interpreter reports 'No' since there are no more parse trees.

25 CYK Chart Parsing Algorithm

Slide notes:

Efficient Inference in PCFG Model

- Using backtracking is not efficient approach
- Chart parsing is an efficient approach
- We will take a look at the CYK chart parsing algorithm

CYK Chart Parsing Algorithm

- When parsing NLP, there are generally two approaches:
  1. Backtracking to find all parse trees
  2. Chart parsing
- CYK algorithm: a simple chart parsing algorithm
- CYK: Cocke-Younger-Kasami algorithm
- CYK can be applied only to a CNF grammar

The CYK algorithm (Cocke-Younger-Kasami) is a well-known efficient parsing algorithm. The algorithm has a running-time complexity of $O(n^3)$ for a sentence of length n.

CYK can be applied only to a CNF (Chomsky Normal Form) grammar, so if the grammar is not already in CNF, we would have to convert it to CNF. A Context-Free Grammar is in CNF if all its rules are either of the form $A \rightarrow B\,C$, where A, B, and C are nonterminals, or $A \rightarrow w$, where A is a nonterminal and w is a terminal. If a CFG is not in CNF, it can be converted into CNF.

Chomsky Normal Form

- all rules are in one of the forms:
  1. $A \rightarrow B\,C$, where A, B, and C are nonterminals, or
  2. $A \rightarrow w$, where A is a nonterminal and w is a terminal
- If a grammar is not in CNF, it can be converted to it

Is the following grammar in CNF?

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	N	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

How about this grammar? (Is it in CNF?)

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	time	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

Note: What if the grammar is not in CNF

There is a standard algorithm for converting arbitrary CFG into CNF. The problem is: How to calculate probabilities of the rules in the new grammar? One way is to sample from the old grammar, and to estimate probabilities in the new grammar by parsing the sample sentences and counting. The probabilities can also be calculated directly, but it is not a straightforward task.

Here are the steps needed for conversion of an arbitrary CFG into CNF. This is just a partial sketch of the algorithm: calculating probabilities of the new rules in the first two steps is not trivial and it is not given.

Eliminate empty rules $N \to \epsilon$

Find all "nullable" nonterminals, i.e., terminals N such that $N \Rightarrow^* \epsilon$.

From each rule $A \to X_1 \dots X_n$ create new rules by striking out some nullable nonterminals. This is done for all combination of nonterminals in the rule, except for striking out all $X_1 \dots X_n$ if they are all nullable.

Remove empty rules.

If the start symbol is nullable, add $S \to \epsilon$, and treat that as a special case.

Eliminate unit rules $N \to M$

For any two variables A and B, such that $A \Rightarrow^* B$, for all non-unit rules $B \to \zeta$, we add $A \to \zeta$. Remove unit rules.

All possible derivations $A \Rightarrow^* B$ are easy to find since the empty rules are already eliminated.

Eliminate terminals in rules, except $A \to w$

For each terminal w that appears on the right hand side of some rule with some other symbols, we introduce a new nonterminal N_w, and a rule $N_w \to w$ with probability 1. Then we replace w in all other rules with N_w.

Eliminate rules $A \to B_1 B_2 \dots B_n$ **(**$n > 2$**)**

For each rule $A \to B_1 B_2 \dots B_n$ ($n > 2$), we introduce $n-2$ new nonterminals $X_1, \dots, X_{n-2}$, and replace this rule with the following rules: $A \to B_1 X_1$, $X_1 \to B_2 X_2$, $\dots X_{n-2} \to B_{n-1} B_n$, and assign the following probabilities to them: $\mathrm{P}(A \to B_1 X_1) = \mathrm{P}(A \to B_1 B_2 \dots B_n)$, $\mathrm{P}(X_1 \to B_2 X_2) = 1, \dots \mathrm{P}(X_{n-2} \to B_{n-1} B_n) = 1$.

CYK Example

Let us first show the CYK algorithm on an example, before presenting it in detail. We assume that the following grammar in CNF is given:

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	time	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

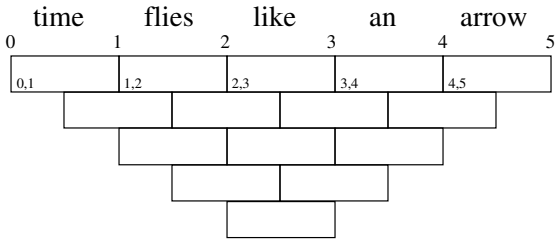
CYK Example (continued): time flies like an arrow

Using this grammar, we want to parse the sentence 'time flies like an arrow'. We first tokenize the sentence by breaking into the words, and we will enumerate all boundaries between words, starting with the start of the first

word in the following way:

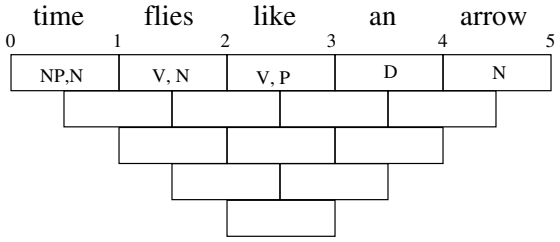
$_0$ time $_1$ flies $_2$ like $_3$ an $_4$ arrow $_5$

We organize parsing around a table, called **parsing chart** or simply **chart**, as follows:

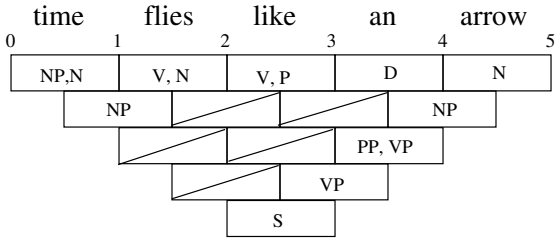


Each entry of the parsing chart is called a **chart entry**. In the empty chart above, we added numbers in the first row of chart entries, which denote spans that are associated with the entries. The first row of chart entries have all spans of lenght 1.

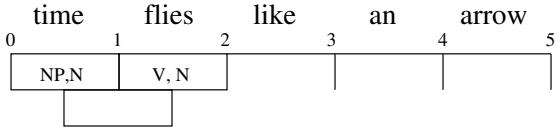
To fill the first row of the table we look at the words and all rules of the form $N \rightarrow w$, where w is the appropriate word, and we add the corresponding non-terminal N to the appropriate chart entry. This is how we fill the first row of the chart and obtain the following:



Completely filled chart looks as follows:

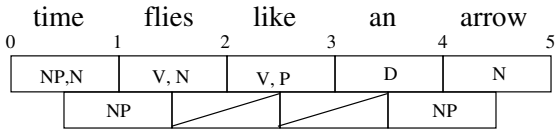


To fill the second row, we look at the span covered by the chart entry. For example, the first chart entry in the second row, covers the span 0–2, as shown below:

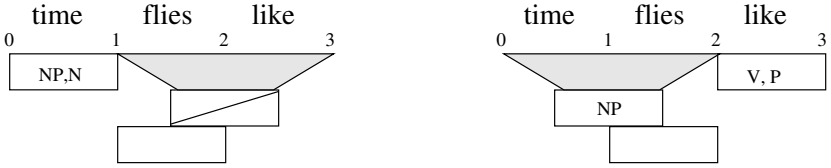


By finding all pairs of non-terminals from the two chart entries that cover sub-spans 0–1 and 1–2, we find pairs:
NP V
NP N
N V
N N

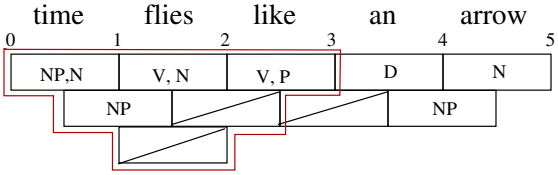
and then we look for the rules, for which one of these sequences is on the right-hand side. We can find the following rule:
NP $\rightarrow$ N N
and we add the non-terminal NP to the entry for the span 0–2. In some cases, we will not find any matching rules and we obtain empty chart entries, which are marked by one crossing diagonal:



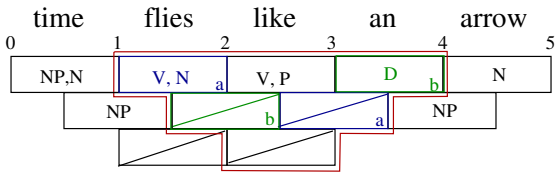
We fill out similarly rows after 2, but it gets a bit more complicated since we will have more than one pair of sub-spans that correspond to a chart entry. For example, the first entry of the third row covers the span 0–3, and then we need to look at the pairs of entries for span 0–1 and 1–3, and 0–2 and 2–3, as shown:



In the first case, when we look at entries ^{0}[NP, N]^{1} and ^{1}[ ]^{3} we do not get any pairs of non terminals. In the second case, we look at the entries ^{0}[NP]^{2} and ^{2}[V, P]^{3} and get the following pairs of non-terminals:
NP V
NP P
for which we cannot find any appropriate rules, whose right-hand sides correspond to these pairs. This is why the entry ^{0}[ ]^{3} remains empty, as follows:



Similarly, we fill the next entry covering the span 1–4 by looking at the pairs of entries covering spans 1–2 and 2–4 (labeled with 'a' in the figure below), and the pair 1–3 and 3–4 (labeled with 'b'):

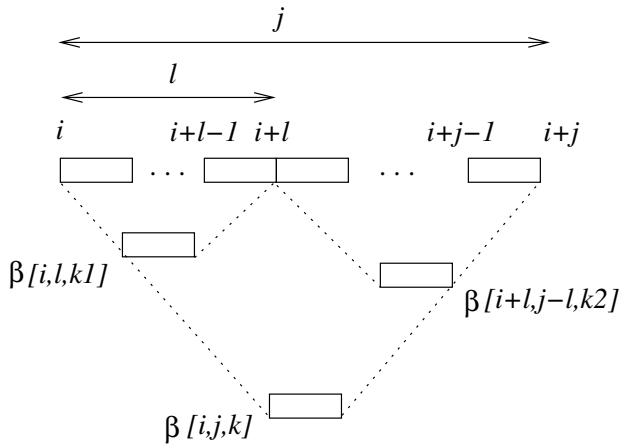


And we continue the process in this way.

Implementation

The example implies that we need to use a two-dimensional table to store chart entries. Using a two-dimensional table is a possible solution, but in that case the table entries would be quite complex since each of them needs to store a set of non-terminals. To make the solution simpler, we can use a three-dimensional table, such that the third dimension corresponds to all different non-terminals.

Explanation of Index Use in CYK



CYK Algorithm

Let all nonterminals be: $N^1, \ldots N^m$.

In the standard CYK algorithm, we have a two dimensional table β in which only the entries $\beta_{ij}, 1 \leq i \leq i+j-1 \leq n$, are used. Each entry β_{ij} contains a set of nonterminals that can produce substring $w_i \ldots w_{i+j-1}$ using the grammar rules, i.e., $\beta_{ij} = \{N | N \Rightarrow^* w_i \ldots w_{i+j-1}\}$.

If we enumerate all nonterminals: N^1, N^2, …, N^m, then each set of nonterminals β_{ij} can be represented by extending β to be a 3-dimensional table β_{ijk}, in which $\beta_{ijk} = 1$ means that N^k can produce substring $w_i \ldots w_{i+j-1}$, and $\beta_{ijk} = 0$ that it cannot.

Algorithm 1 CYK Parsing Algorithm

Require: sentence $= w_1 \ldots w_n$, and a CFG in CNF with nonterminals $N^1 \ldots N^m$,
N^1 is the start symbol
Ensure: parsed sentence

```
allocate matrix β ∈ {0,1}^{n×n×m} and initialize all entries to 0
for i ← 1 to n do
    for all rules N^k → w_i do
        β[i, 1, k] ← 1
for j ← 2 to n do
    for i ← 1 to n − j + 1 do
        for l ← 1 to j − 1 do
            for all rules N^k → N^{k1} N^{k2} do
                β[i, j, k] ← β[i, j, k] OR (β[i, l, k1] AND β[i + l, j − l, k2])
return β[1, n, 1]
```

The line $\beta[i,j,k] \leftarrow \beta[i,j,k]$ OR $(\beta[i,l,k_1]$ AND $\beta[i+l, j-l, k_2])$ in the algorithm is essentially is a shorthand expression for:

```
if β[i, l, k1] AND β[i + l, j − l, k2] then
    β[i, j, k] ← 1
```

26 Efficient Inference in PCFG Model

Efficient Inference in PCFG Model

Let us consider the marginalization task:

$\mathrm{P}(\text{sentence}) = ?$

If 'sentence' is the following sequence of words: $w_1 w_2 \dots w_n$, then $\mathrm{P}(\text{sentence})$ is the following conditional probability:

$$\mathrm{P}(\text{sentence}) = \mathrm{P}(w_1 w_2 \dots w_n | S)$$

i.e., it is the probability of generating the sentence given that we start from S, i.e, it is $\mathrm{P}(S \Rightarrow^* w_1 \dots w_n)$.

An obvious way to calculate this marginal probability is to find all parse trees of a sentence and sum their probabilities, i.e:

$$\mathrm{P}(\text{sentence}) = \sum_{t \in T} \mathrm{P}(t),$$

where T is the set of all parse trees of the sentence 'sentence'. However, this may be very inefficient. We also need a way to find all parse trees.

As an example illustrating that the above direct approach may lead to an exponential algorithm, consider a CFG with only two rules $S \Rightarrow S\ S$ and $S \Rightarrow a$. The sentences a^n have as many parse trees as there are binary trees with n leaves, which is a well-known Catalan number, $\approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$ as $n \to \infty$.

An algorithm for efficient marginalization can be derived from the CYK algorithm.

PCFG Marginalization

The CKY algorithms is adapted to solve the problem of efficient PCFG marginalization, but replacing entries of the table β with numbers between 0 and 1. These numbers are called inside probabilities, and they represent the following probabilities:

$$\beta[i, j, k] = \mathrm{P}(w_i \dots w_{i+j-1} | N^k)$$

So, $\beta[i, j, k]$ is the probability that the string $w_i \dots w_{i+j-1}$ is generated in a derivation where the starting non-terminal is N^k. Algorithm 2 is the probabilistic CYK algorithm for calculating $\mathrm{P}(\text{sentence})$.

Algorithm 2 Probabilistic CYK for $\mathrm{P}(\text{sentence})$

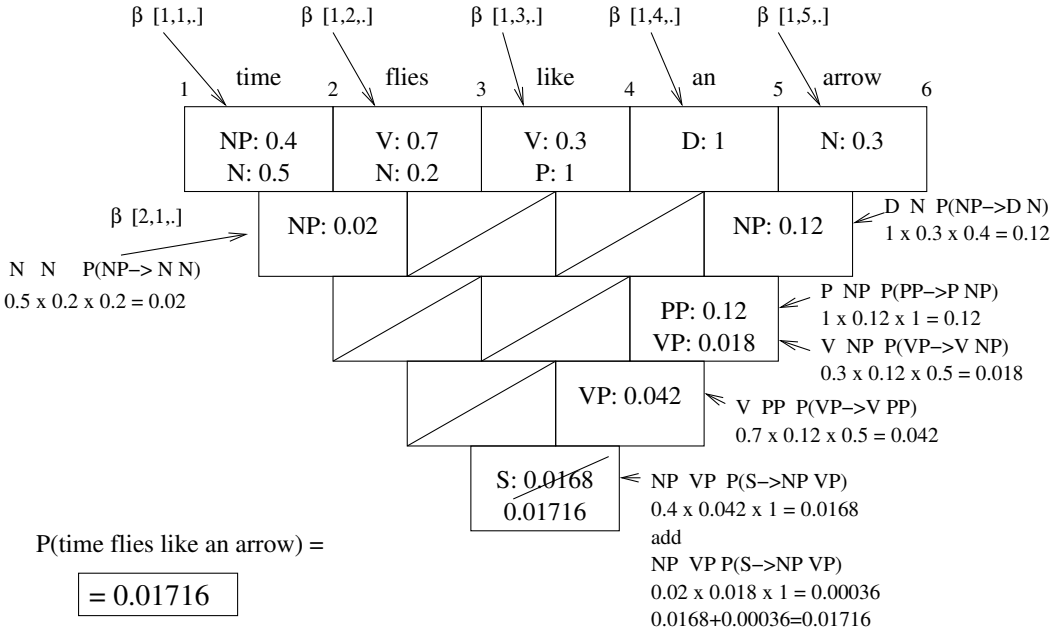
Require: sentence $= w_1 \dots w_n$, and a PCFG in CNF with nonterminals $N^1 \dots N^m$, N^1 is the start symbol
Ensure: P(sentence) is returned

```
allocate β ∈ ℝ^{n×n×m} and initialize all entries to 0
for i ← 1 to n do
    for all rules N^k → w_i do
        β[i, 1, k] ← P(N^k → w_i)
for j ← 2 to n do
    for i ← 1 to n − j + 1 do
        for l ← 1 to j − 1 do
            for all rules N^k → N^{k1} N^{k2} do
                β[i, j, k] ← β[i, j, k] + P(N^k → N^{k1} N^{k2}) · β[i, l, k1] · β[i + l, j − l, k2]
return β[1, n, 1]
```

PCFG Marginalization Example (grammar)

S	$\rightarrow$	NP VP	/1	VP	$\rightarrow$	V NP	/.5	N	$\rightarrow$	time	/.5
NP	$\rightarrow$	time	/.4	VP	$\rightarrow$	V PP	/.5	N	$\rightarrow$	arrow	/.3
NP	$\rightarrow$	N N	/.2	PP	$\rightarrow$	P NP	/1	N	$\rightarrow$	flies	/.2
NP	$\rightarrow$	D N	/.4					D	$\rightarrow$	an	/1
V	$\rightarrow$	like	/.3								
V	$\rightarrow$	flies	/.7								
P	$\rightarrow$	like	/1								

PCFG Marginalization Example (chart)



Conditioning

The conditioning computational problem in the PCFG model becomes the task of finding the conditional probability $\mathrm{P}(\text{tree}|\text{sentence})$, for a particular sentence and a particular parse three of the given sentence. Using the definition of the conditional probability, we have:

$$\mathrm{P}(\text{tree}|\text{sentence}) = \frac{\mathrm{P}(\text{tree}, \text{sentence})}{\mathrm{P}(\text{sentence})}$$

and since the sentence is a part of the parse tree, we can further write:

$$\mathrm{P}(\text{tree}|\text{sentence}) = \frac{\mathrm{P}(\text{tree}, \text{sentence})}{\mathrm{P}(\text{sentence})} = \frac{\mathrm{P}(\text{tree})}{\mathrm{P}(\text{sentence})}$$

Slide notes:

Conditioning

- Conditioning in the PCFG model: $\mathrm{P}(\text{tree}|\text{sentence})$
- Use the formula:

$$\mathrm{P}(\text{tree}|\text{sentence}) = \frac{\mathrm{P}(\text{tree}, \text{sentence})}{\mathrm{P}(\text{sentence})} = \frac{\mathrm{P}(\text{tree})}{\mathrm{P}(\text{sentence})}$$

- $\mathrm{P}(\text{tree})$ — directly evaluated
- $\mathrm{P}(\text{sentence})$ — marginalization

$\mathrm{P}(\text{tree})$ is calculated by multiplying probabilities of all rules in the tree, and $\mathrm{P}(\text{sentence})$ is calculated by the Algorithm 2 used for marginalization.

Completion

The completion task becomes the parsing problem; i.e., the problem of finding the most probably parse tree give the sentence, which can be expressed as:

$$\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$$

Slide notes:

Completion

- Finding the most likely parse tree of a sentence:

$$\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$$

- Use the CYK algorithm in which line 9 is replaced with:

9: $\beta[i,j,k] \leftarrow \max(\beta[i,j,k], \mathrm{P}(N^k \rightarrow N^{k_1}N^{k_2}) \cdot \beta[i,l,k_1] \cdot \beta[i+l, j-l, k_2])$

- Return the most likely tree

The most probable completion is computed in a similar way to the marginalization algorithm (Algorithm 2). The difference is that the line 9 is replaced by the line

9: $\beta[i,j,k] \leftarrow \max(\beta[i,j,k], \mathrm{P}(N^k \rightarrow N^{k_1}N^{k_2}) \cdot \beta[i,l,k_1] \cdot \beta[i+l, j-l, k_2])$

Additionally in step 10, we are not just interested in $\beta[1,n,1]$, which is the probability of the most probable tree, but we also want to obtain the actual tree.

Algorithm 3 CYK-based Completion Algorithm for $\arg\max_t \mathrm{P}(t|\text{sentence})$

Require: sentence $= w_1 \dots w_n$, and a PCFG in CNF with nonterminals $N^1 \dots N^m$, N^1 is the start symbol
Ensure: The most likely parse tree is returned

1: allocate $\beta \in \mathbb{R}^{n \times n \times m}$ and initialize all entries to 0
2: **for** $i \leftarrow 1$ to n **do**
3: **for all** rules $N^k \rightarrow w_i$ **do**
4: $\beta[i,1,k] \leftarrow \mathrm{P}(N^k \rightarrow w_i)$
5: **for** $j \leftarrow 2$ to n **do**
6: **for** $i \leftarrow 1$ to $n-j+1$ **do**
7: **for** $l \leftarrow 1$ to $j-1$ **do**
8: **for all** rules $N^k \rightarrow N^{k_1}N^{k_2}$ **do**
9: $\beta[i,j,k] \leftarrow \max(\beta[i,j,k], \mathrm{P}(N^k \rightarrow N^{k_1}N^{k_2}) \cdot \beta[i,l,k_1] \cdot \beta[i+l, j-l, k_2])$
10: **return** Reconstruct$(1, n, 1, \beta)$

The tree can be reconstructed from the table using algorithm 4:

10: **return** Reconstruct$(1, n, 1, \beta)$

Algorithm 4 Reconstruct(i, j, k, β)

Require: β — table from CYK, i — index of the first word, j — length of sub-string sentence, k — index of non-terminal

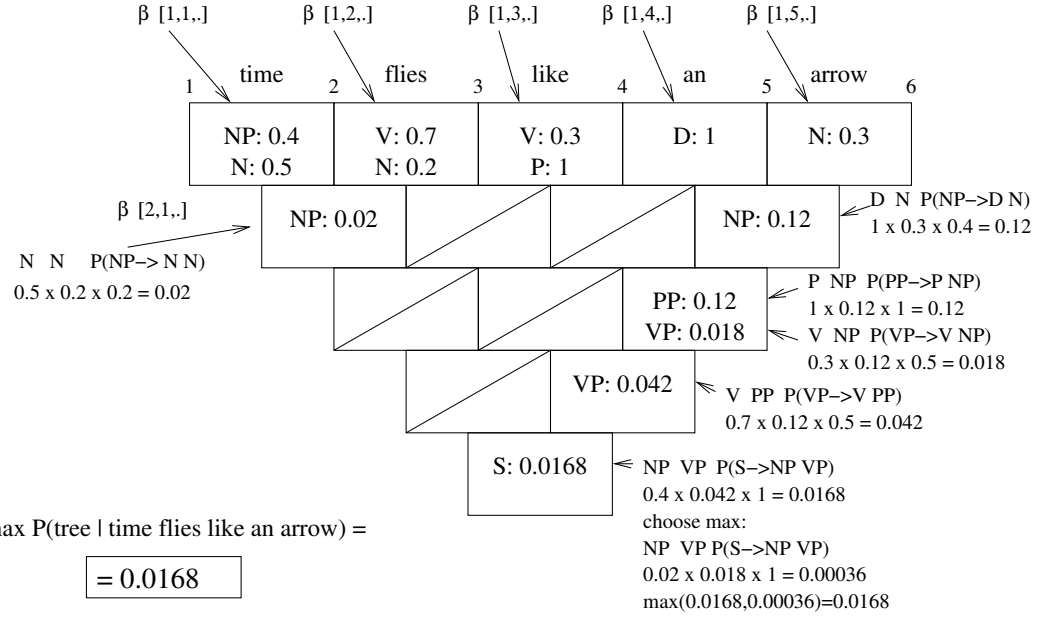
Ensure: a most probable tree with root N^k and leaves $w_i \dots w_{i+j-1}$ is returned

1: **if** $j = 1$ **then**
2: **return** tree with root N^k and child w_i
3: **for** $l \leftarrow 1$ to $j - 1$ **do**
4: **for all** rules $N^k \rightarrow N^{k_1} N^{k_2}$ **do**
5: **if** $\beta[i, j, k] = \mathrm{P}(N^k \rightarrow N^{k_1} N^{k_2}) \cdot \beta[i, l, k_1] \cdot \beta[i + l, j - l, k_2]$ **then**
6: create a tree t with root N^k
7: $t.\mathit{left\_child} \leftarrow$ Reconstruct(i, l, k_1, β)
8: $t.\mathit{right\_child} \leftarrow$ Reconstruct$(i + l, j - l, k_2, \beta)$
9: **return** t

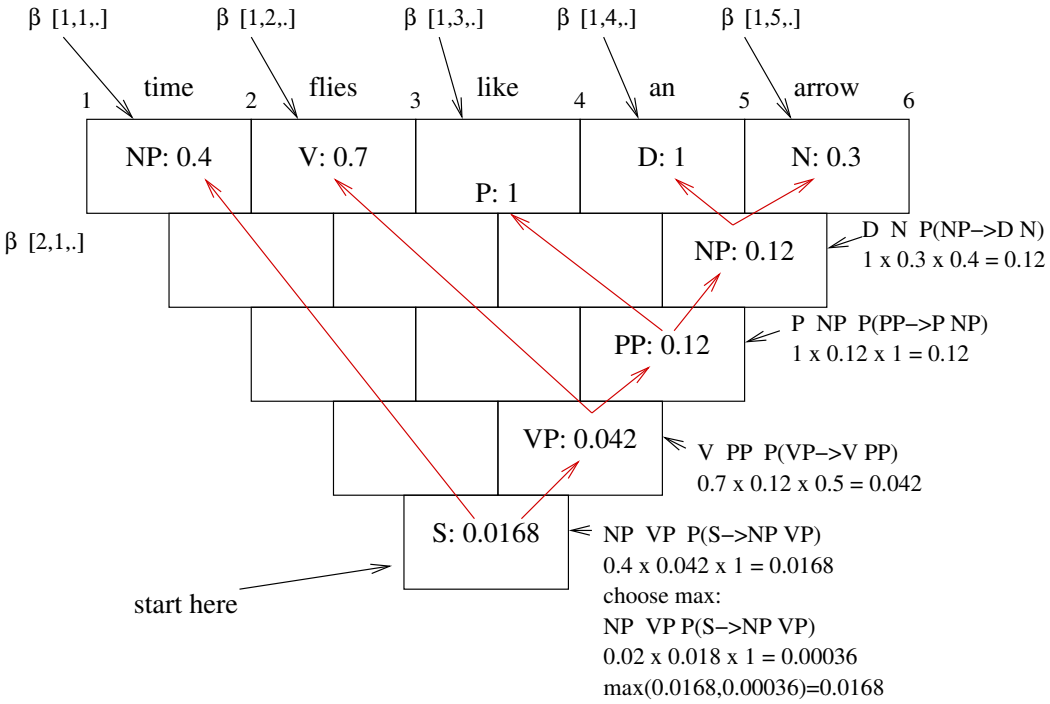
PCFG Completion Example (grammar)

S	→	NP VP	/1	VP	→	V NP	/.5	N	→	time	/.5
NP	→	time	/.4	VP	→	V PP	/.5	N	→	arrow	/.3
NP	→	N N	/.2	PP	→	P NP	/1	N	→	flies	/.2
NP	→	D N	/.4					D	→	an	/1
V	→	like	/.3								
V	→	flies	/.7								
P	→	like	/1								

PCFG Completion Example (chart)

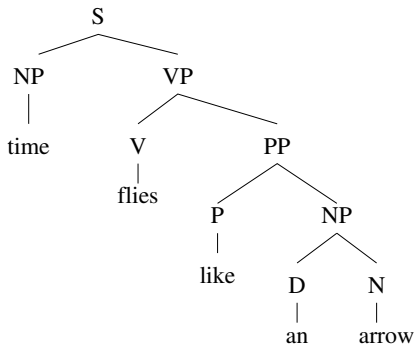


PCFG Completion Example (tree reconstruction)



PCFG Completion Example (final tree)

The most probable three:



Topics related to PCFGs

– An interesting open problem is whether the inference in PCFGs can be reduced to a message-passing-style algorithm as used in Bayesian Networks?

Issues with PCFGs

The Probabilistic Context-Free Grammars were shown to perform quite well in parsing English, but usually with some additional mechanisms to address certain issues. Two most prominent issues in using PCFGs to parse nature languages are the inability of PCFGs to capture structural and lexical dependencies.

Structural dependencies are rule dependencies on the position in a parse tree. For example, pronouns occur more frequently as subjects than objects in sentences, so the rule choice between NP $\rightarrow$ PRP and NP $\rightarrow$ DT NN should depend on the position of a noun phrase in a tree. Generally, NL parse trees are usually deeper at their right side than the left side, and this propertly is typically not modeled well with PCFGs.

Lexical dependencies are rule dependencies on the words that are eventually derived from those rules, particularly phrase head words. As an example, the PP-attachment problem is resolved based on the rule probabilities of the rules applied higher in the parse tree, such as NP $\rightarrow$ NP PP and VP $\rightarrow$ VB NP PP, while they truly frequently depend on the verb being used and other word, particularly head words.

Slide notes:

PP-Attachment Example

- Consider sentences:
  - "Workers dumped sacks into a bin." and
  - "Workers dumped sacks of fish."
- and rules:
  - NP $\rightarrow$ NP PP
  - VP $\rightarrow$ VBD NP
  - VP $\rightarrow$ VBD NP PP

As an example, let us consider simple sentences:

- "Workers dumped sacks into a bin." (from [JM]), and
- "Workers dumped sacks of fish."

At some level of parsing, we can see both of these sentences as:

- NP VBD NP PP

and now the question is whether the "NP PP" should be combined to make an NP, or the sequence "VBD NP PP" should be combined to make a VP. In a PCFG this will depend only on the probability of the rules: NP $\rightarrow$ NP PP, VP $\rightarrow$ VBD NP, and VP $\rightarrow$ VBD NP PP. However, we can see that the probailities should actually depend on affinity of the verb 'dump' and preposition 'into' on one side, and the noun 'sacks' and the preposition 'of' on other side.

A Solution: Probabilistic Lexicalized CFGs

- use heads of phrases
- expanded set of rules, e.g.:

$$\text{VP(dumped)} \rightarrow \text{VBD(dumped) NP(sacks) PP(into)}$$

- large number of new rules
- sparse data problem
- solution: new independence assumptions
- proposed solutions by Charniak, Collins, etc. around 1999