

Introduction to Algorithms and Data Structures

Lecture 22: Parsing for context-free languages

John Longley

School of Informatics
University of Edinburgh

7 February 2022

The parsing problem

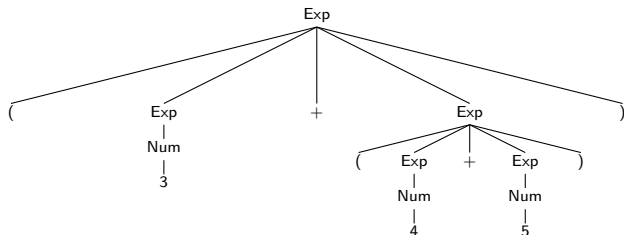
Last time, we saw what a context-free grammar was.

$$\begin{aligned}\text{Exp} &\rightarrow \text{Num} \mid (\text{Exp} + \text{Exp}) \\ \text{Num} &\rightarrow 0 \mid \dots \mid 9\end{aligned}$$

This time, we'll consider the **parsing problem**: how do we get from a **string** of terminals ...

$(3 + (4 + 5))$

... to a **tree**



Often an essential prelude to other tasks (e.g. evaluating an expression!)

The CYK algorithm

We'll describe a general approach that works for *any* CFG, using the **Cocke-Younger-Kasami** (CYK or CKY) algorithm.

(Seemingly first discovered by **Itiroo Sakai** in 1961.)

Another example of **dynamic programming**.

- ▶ First see how this algorithm works on a special class of grammars, those in **Chomsky normal form** (CNF).
- ▶ Then see how *any* context-free grammar can be transformed to an 'equivalent' one in CNF.
- ▶ CYK parses inputs of length n in time $\Theta(n^3)$. Fine for short sentences, but not practical for long computer programs.
Next time, we'll look at parsing algorithms better suited to computer languages: less general, but faster.

What's Chomsky normal form?

Recall that in a CFG, the right-hand side of each production is a (possibly empty) string of **terminals** and **non-terminals**. E.g.

$$\text{Exp} \rightarrow (\text{Exp} + \text{Exp})$$

A grammar in **Chomsky normal form** is one in which each RHS consists of

- ▶ *either* just two non-terminals (e.g. $X \rightarrow YZ$)
- ▶ *or* just one terminal (e.g. $X \rightarrow +$).

We'll see soon what this curious restriction buys us.

Most important point is that RHSs with ≥ 3 symbols are forbidden.

Chomsky normal form: example

The following grammar is in CNF.

Terminals: book, orange, heavy, my, very

Non-terminals: NP, Nom, AP, A, Det, Adv

Start symbol: NP

NP \rightarrow Det Nom

Nom \rightarrow book | orange | AP Nom

AP \rightarrow heavy | orange | Adv A

A \rightarrow heavy | orange

Det \rightarrow my

Adv \rightarrow very

Generates noun phrases like:

my very heavy orange

my very heavy orange book

(N.B. CNF grammars often involve some duplication!

Writing $AP \rightarrow A$ would be simpler, but not CNF.)

CYK parsing: the idea

Let's insert 'position markers' in the input string we wish to parse:

0 my 1 very 2 heavy 3 orange 4 book 5

We can then talk about **substrings** of the input: e.g. the pair (2,4) indicates the substring 'heavy orange'.

Primary question: Can the **entire string** (0,5) be derived from the start symbol **NP**? If so, how?

As is common in Dynamic Programming, we approach this by generalizing our objective slightly: **Which substrings** can be derived from **which non-terminals**?

We store the solutions to these 'subproblems' in a 2-dim array: entry for (i,j) (where $i < j$) records possible analyses of the substring indicated by (i,j) .

Broadly speaking, we work our way from shorter to longer substrings (some flexibility re precise ordering of subproblems).

Filling out the CYK chart: example

NP → Det Nom A → heavy | orange
Nom → book | orange | AP Nom Det → my
AP → heavy | orange | Adv A Adv → very

0 my 1 very 2 heavy 3 orange 4 book 5

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

CYK: The general algorithm

```
CYK (s,G):           # s=input string, G=CNF grammar
  n = length(s)
  allocate table[0,...,n-1][1,...,n]
  for j = 1 to n      # columns
    for  $(X \rightarrow t) \in G$ 
      if  $t = s[j-1]$ 
        add X to table[j-1,j]  # diagonal cell
    for i = j-2 downto 0      # rows
      for k = i+1 to j-1     # possible splits
        for  $(X \rightarrow YZ) \in G$ 
          if  $Y \in \text{table}[i,k]$  and  $Z \in \text{table}[k,j]$ 
            add X to table[i,j]  # non-diagonal cell
  return table
```


From recognizer to parser

- ▶ So far, we just have a **recognizer**: a way of determining whether a string belongs to the given language.
- ▶ Changing this to a **parser** requires recording which existing constituents were combined to make each new constituent.

	j	1	2	3	4	5
i		my	very	heavy	orange	book
0	my	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

- ▶ The algorithm identifies **all possible parses**.
There may also be **phantom constituents** that don't form part of any complete syntax tree (e.g. 'my very heavy orange').

Runtime of CYK

Looking at the pseudocode for CYK, we have three nested for-loops, each of which we go round $\leq n$ times.

And within them, some iteration over the grammar rules.

So for any fixed grammar G , the algorithm runs in time $O(n^3)$.

(If we allow grammar to vary, runtime is $O(mn^3)$, where m is 'size' of grammar.)

What would happen if we allowed **ternary rules**, e.g. $A \rightarrow BCD$?

To fill a cell (i, j) , we'd need to consider all possible three-way splits $(i, k), (k, l), (l, j)$ where $i < k < l < j$.

Number of these is quadratic in $j - i$.

So our overall runtime would go up to $\Theta(n^4)$.

That's the main reason we like Chomsky normal form (there are other minor benefits).

More on Chomsky normal form

Recall: a context-free grammar $G = (\Sigma, N, S, P)$ is in **Chomsky normal form (CNF)** if all productions are of the form

$$A \rightarrow BC \quad \text{or} \quad A \rightarrow a \quad (A, B, C \in N, a \in \Sigma)$$

Theorem: Disregarding the empty string, every CFG G is equivalent to a grammar G' in Chomsky normal form. ($\mathbf{L}(G') = \mathbf{L}(G) - \{\epsilon\}$)
And there's an algorithm which, given G , finds a suitable G' .

Key idea: To eliminate rules with ≥ 3 symbols on the RHS, we could replace e.g.

$$X \rightarrow ABCD \quad \text{by} \quad X \rightarrow AY, Y \rightarrow BZ, Z \rightarrow CD$$

where Y, Z are **newly added** nonterminals.

Converting to Chomsky Normal Form

Consider for example the grammar

$$S \rightarrow TT \mid [S] \qquad T \rightarrow \epsilon \mid (T)$$

Step 1: Apply trick on last slide to rules with ≥ 3 symbols on RHS.
In this case, apply it to $S \rightarrow [S]$ and $T \rightarrow (T)$:

$$\begin{aligned} S &\rightarrow TT \mid [W] & T &\rightarrow \epsilon \mid (V \\ W &\rightarrow S] & V &\rightarrow T) \end{aligned}$$

Step 2: Identify the set E of all non-terminals X such that ϵ can be derived from X (**nullable** non-terminals).

In this case, $T \rightarrow \epsilon$ tells us $T \in E$. Then $S \rightarrow TT$ tells us $S \in E$. And that's all. So $E = \{S, T\}$.

In general, E is the smallest set such that if $X \rightarrow Y_1 \dots Y_r \in P$ and $Y_1, \dots, Y_r \in E$ then $X \in E$ (allowing $r = 0$ here).

Converting to Chomsky Normal Form, ctd.

$$\begin{array}{ll} S \rightarrow TT \mid [W & T \rightarrow \epsilon \mid (V \\ W \rightarrow S] & V \rightarrow T) \end{array}$$

Step 3: Delete all ϵ -productions.

To compensate, for each rule $X \rightarrow Y\alpha$ or $X \rightarrow \alpha Y$, where $Y \in E$ and $\alpha \neq \epsilon$, add a new rule $X \rightarrow \alpha$.

In this case, since $E = \{S, T\}$, we get:

$$\begin{array}{ll} S \rightarrow TT \mid T \mid [W & T \rightarrow (V \\ W \rightarrow S] \mid] & V \rightarrow T) \mid) \end{array}$$

Step 4: Remove **unit productions** $X \rightarrow Y$.

To compensate, for every rule $Y \rightarrow \alpha$, add in $X \rightarrow \alpha$.

In this case, do this for $S \rightarrow T$:

$$\begin{array}{ll} S \rightarrow TT \mid (V \mid [W & T \rightarrow (V \\ W \rightarrow S] \mid] & V \rightarrow T) \mid) \end{array}$$

Converting to Chomsky Normal Form, ctd., ctd.

$$\begin{array}{ll} S \rightarrow TT \mid (V \mid [W & T \rightarrow (V \\ W \rightarrow S] \mid] & V \rightarrow T) \mid) \end{array}$$

By this stage, all RHSs consist of 1 terminal or 2 symbols. So just need to get rid of terminals from the 'binary' rules.

Step 5: For each terminal a , add a fresh nonterminal Z_a and a production $Z_a \rightarrow a$, then replace a by Z_a in all binary rules.

In this case, we add four rules:

$$\begin{array}{ll} Z_{(} \rightarrow (& Z_{)} \rightarrow) \\ Z_{[} \rightarrow [& Z_{]} \rightarrow] \end{array}$$

And rewrite the existing rules to:

$$\begin{array}{ll} S \rightarrow TT \mid Z_{(}V \mid Z_{[}W & T \rightarrow Z_{(}V \\ W \rightarrow SZ_{]} \mid] & V \rightarrow TZ_{)} \mid) \end{array}$$

The grammar is now in Chomsky Normal Form, and we're done.

Assorted remarks

- ▶ Given a CFG \mathcal{G} , we can do the above (once for all) to convert it to a CNF grammar \mathcal{G}' , then run CYK for \mathcal{G}' (many times).
- ▶ This will give us a syntax tree w.r.t. \mathcal{G}' . Bit of work to translate back to a tree w.r.t. \mathcal{G} — not very hard/interesting.
- ▶ If \mathcal{G} has m rules, our algorithm gives a \mathcal{G}' with $O(m^2)$ rules. Quadratic blow-up possible, but not a problem in practice.
- ▶ Since we expect to run CYK for \mathcal{G}' on many inputs, wouldn't it be nice to find the **smallest** CNF grammar equivalent to \mathcal{G} ? (Where 'size' of a grammar is e.g. total length of all RHSs.)
- ▶ Unfortunately, this is an example of an **NP-hard** problem. No known **polynomial-time** algorithm (i.e. one with runtime $O(n^d)$ for some d).
- ▶ Versions of CYK are quite widely used in Natural Language context (where sentences typically have < 100 words). But $\Theta(n^3)$ parsing not good enough for computer languages.

Reading

Recommended: D. Jurafsky and J.H. Martin,
Speech and Language Processing, 3rd ed. (draft).
Chapter 13 (Constituency parsing), Sections 1 and 2.
Available at <https://web.stanford.edu/~jurafsky/slp3>

Today's music: Antonín Dvořák, *Humoresque*,
arranged by the jazz pianist Art Tatum.