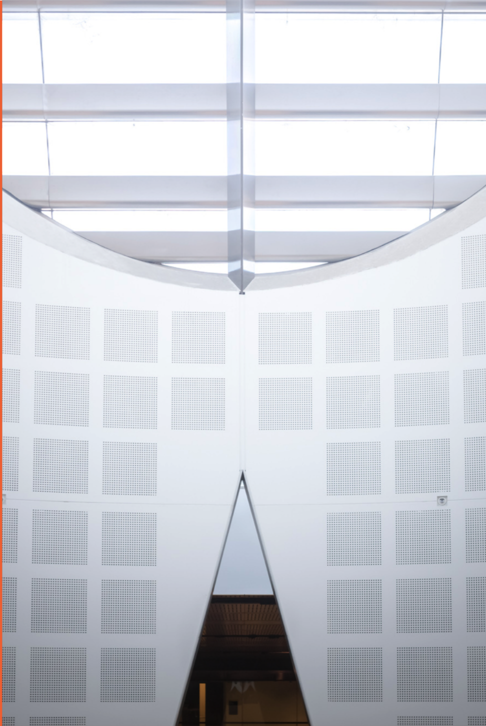
COMP2022|2922
Models of Computation

Chomsky Normal Form and Parsing

Sasha Rubin

October 27, 2022

THE UNIVERSITY OF SYDNEY



Agenda

1. Chomsky Normal Form (CNF) for CFGs
2. CYK Parsing algorithm for CFGs in CNF

- A context-free grammar (CFG) generates strings by rewriting.
- Today we will see how to tell if a given context-free grammar (CFG) generates a given string.
- Here is the decision problem: Given a CFG G and string w decide if G derives w.
- This basic problem is solved by compilers and parsers.

Possible approaches...

1. Systematically search through all derivations (or all parse-trees) until you find one that derives w.
   - Try all i-step derivations for $i = 1, 2, 3, etc.$
   - Problem: When to stop?
   - This problem can be fixed (see Tutorial), but the resulting algorithm takes exponential time in the worst case, i.e., is very slow.
2. Use dynamic programming (aka table-filling, aka tabulation).
   - Similar to divide and conquer.
   - You will study the dynamic programming technique in `COMP3027:Algorithm Design`
   - The parsing algorithm is called the CYK algorithm, and takes polynomial time in the worst case, i.e., is acceptably fast.

Problem
Given a CFG G and string w decide if G generates w.

We will do this for grammars in Chomsky Normal Form because the algorithm is then easier to understand, and one can convert every CFG into this form.

Chomsky Normal Form

Definition

A grammar G is in Chomsky Normal Form (CNF) if every rule is in of one of these forms:

1. $A \to BC$ (A, B, C are any variables, except that neither B nor C is the start variable)
2. $A \to a$ (A is any variable and a is a terminal)
3. In addition, we permit $S \to \varepsilon$ where S is the start variable.

Theorem

Every context-free language is generated by a grammar in CNF.

$$T \to aTb \mid \epsilon$$

$$
\begin{aligned}
S &\to AX \mid \epsilon \\
T &\to AX \\
X &\to TB \mid b \\
A &\to a \\
B &\to b
\end{aligned}
$$

CNF

Theorem
Every context-free language is generated by a grammar in CNF.

In the next slides, we will give a 5-step algorithm to do this:

1. START: Eliminate the start variable from the RHS of all rules
2. TERM: Eliminate rules with terminals, except for rules $A \to a$
3. BIN: Eliminate rules with more than two variables
4. DEL: Eliminate epsilon productions
5. UNIT: Eliminate unit rules

Chomsky Normal Form: algorithm

1. **Eliminate the start variable from the RHS of all rules**
2. Eliminate rules with terminals, except for rules $A \to a$
3. Eliminate rules with more than two variables
4. Eliminate epsilon productions
5. Eliminate unit rules

- Add the new start variable S and the rule $S \to T$ where T was the old start variable.

Chomsky Normal Form: algorithm

1. Eliminate the start variable from the RHS of all rules
2. **Eliminate rules with terminals, except for rules** $A \to a$
3. Eliminate rules with more than two variables
4. Eliminate epsilon productions
5. Eliminate unit rules

- Replace every terminal a on the RHS of a rule (that is not of the form $A \to a$) by the new variable N_a.
- For each such terminal a create the new rule $N_a \to a$.

Chomsky Normal Form: algorithm

1. Eliminate the start variable from the RHS of all rules
2. Eliminate rules with terminals, except for rules $A \to a$
3. **Eliminate rules with more than two variables**
4. Eliminate epsilon productions
5. Eliminate unit rules

For every rule of the form $A \to EFGH$, say, delete it and create new variables A_1, A_2 and add rules:

$$\begin{aligned} A &\to EA_1 \\ A_1 &\to FA_2 \\ A_2 &\to GH \end{aligned}$$

Chomsky Normal Form: algorithm

1. Eliminate the start variable from the RHS of all rules
2. Eliminate rules with terminals, except for rules $A \to a$
3. Eliminate rules with more than two variables
4. **Eliminate epsilon productions**
5. Eliminate unit rules

For every rule of the form $U \to \varepsilon$ (except $S \to \varepsilon$)

1. Remove the rule.
2. For each rule $A \to \alpha$ containing U, add the new rules of the form $A \to \alpha'$ where α' is α with one or more U's removed,
    2.1 but do not add the rule $A \to \epsilon$ if it was removed in an earlier iteration of Step 1.

Chomsky Normal Form: algorithm

1. Eliminate the start variable from the RHS of all rules
2. Eliminate rules with terminals, except for rules $A \to a$
3. Eliminate rules with more than two variables
4. Eliminate epsilon productions
5. **Eliminate unit rules**

For each rule of the form $A \to B$:

1. Remove the rule.
2. For each rule of the form $B \to \alpha$ add the new rule $A \to \alpha$,

   but do not add the rule $A \to A$, and do not add $A \to \alpha$ if it was removed in an earlier iteration of Step 1.

Chomsky Normal Form: example

$$T \to aTb \mid \epsilon$$

Step 1 (START): Eliminate start variable from the RHS of all rules:

$$\begin{aligned} S &\to T \\ T &\to aTb \mid \epsilon \end{aligned}$$

Chomsky Normal Form: example

$$S \to T$$
$$T \to aTb \mid \epsilon$$

Step 2 (TERM): Eliminate rules with terminals, except $A \to a$:

$$S \to T$$
$$T \to ATB \mid \epsilon$$
$$A \to a$$
$$B \to b$$

Chomsky Normal Form: example

$$
\begin{aligned}
S &\to T \\
T &\to ATB \mid \epsilon \\
A &\to a \\
B &\to b
\end{aligned}
$$

Step 3 (BIN): Eliminate rules with more than two variables:

$$
\begin{aligned}
S &\to T \\
T &\to AX \mid \epsilon \\
X &\to TB \\
A &\to a \\
B &\to b
\end{aligned}
$$

Chomsky Normal Form: example

$$S \to T$$
$$T \to AX \mid \epsilon$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

Step 4 (DEL): Eliminate epsilon production $T \to \varepsilon$

$$S \to T \mid \epsilon$$
$$T \to AX$$
$$X \to TB \mid B$$
$$A \to a$$
$$B \to b$$

Chomsky Normal Form: example

$$
\begin{aligned}
S &\to T \mid \epsilon \\
T &\to AX \\
X &\to TB \mid B \\
A &\to a \\
B &\to b
\end{aligned}
$$

Step 5 (UNIT): Eliminate unit rules (first $S \to T$, then $X \to B$)

$$
\begin{aligned}
S &\to AX \mid \epsilon \\
T &\to AX \\
X &\to TB \mid b \\
A &\to a \\
B &\to b
\end{aligned}
$$

Chomsky Normal Form: example

All done!

$$
\begin{aligned}
S &\to AX \mid \epsilon \\
T &\to AX \\
X &\to TB \mid b \\
A &\to a \\
B &\to b
\end{aligned}
$$

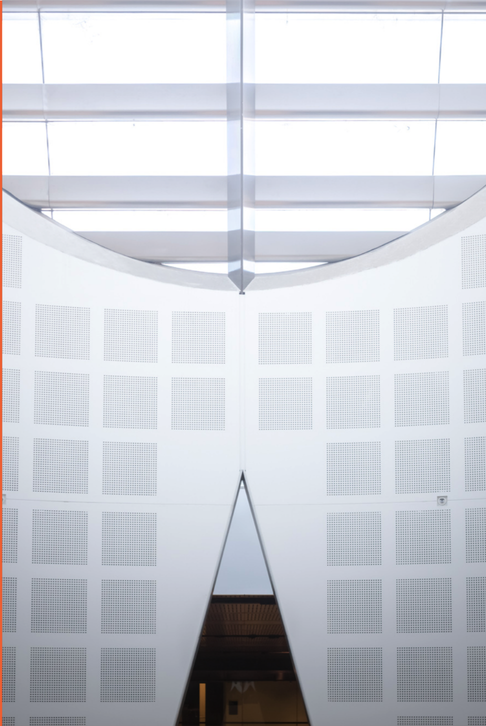
COMP2022|2922 Models of Computation

CYK Algorithm for Parsing CFGs in CNF

Sasha Rubin

October 27, 2022

THE UNIVERSITY OF SYDNEY



Membership problem for CFG in CNF

Problem

Given a CFG G in CNF and string w decide if G derives w (i.e., if $S \Rightarrow^* w$)

Dynamic Programming

- Accumulate information about smaller subproblems to solve the larger problem (similar to divide and conquer)
- The table records the solution to the subproblems, so we only need to solve each subproblem once (aka memoisation)
- Steps in dynamic programming:
  1. Define the subproblems.
  2. Find the recurrence relating the subproblems.
  3. Make sure each subproblem is solved once.
- The algorithm we will see is known as the CYK algorithm (Cocke–Younger–Kasami).

Steps in dynamic programming:

1. Define the subproblems.
2. Find the recurrence relating the subproblems.
3. Make sure each subproblem is solved once.

Step 1: Define the subproblems

If $S \to AB$, then in order to know if there is a derivation

$$S \Rightarrow AB \Rightarrow^* w$$

we need to know if w can be split into uv such that

$$A \Rightarrow^* u$$

and

$$B \Rightarrow^* v$$

- But now we have the same problem again, but on subwords u, v of w and other nonterminals A, B.
- So, the general problem we need to solve is this: for every infix z of w, and every non-terminal X, if $X \Rightarrow^* z$.
- Introduce a 2D array $Sub(x, y)$ is the set of non-terminals that derive the infix of w of length y starting in position x.

Step 1: Define the subproblems

Introduce a 2D array $Sub(x, y)$ is the set of non-terminals that derive the infix of w of length y starting in position x.

- in math: $Sub(x, y) = \{A \in V : A \Rightarrow^* w_x w_{x+1} \cdots w_{x+y-1}\}$

Example

$$
\begin{aligned}
S &\to AB \mid AX \mid \epsilon \\
T &\to AB \mid AX \\
X &\to TB \\
A &\to a \\
B &\to b
\end{aligned}
$$

4	S,T			
3		X		
2		S,T		
1	A	A	B	B
	1	2	3	4

$w = aabb$

Step 2: Find the recurrence

4				
3				
2				
1		A?		
	1	2	3	4

$x = 2, y = 1$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.

Step 2: Find the recurrence

4				
3	A?			
2				
1				
	1	2	3	4

$x = 1, y = 3$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4				
3	A?			
2		C		
1	B			
	1	2	3	4

$$x = 1, y = 3, l = 1$$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4				
3	A?			
2	B			
1			C	
	1	2	3	4

$$x = 1, y = 3, l = 2$$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4	A?			
3				
2				
1				
	1	2	3	4

$x = 1, y = 4$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4	A?			
3		C		
2				
1	B			
	1	2	3	4

$$x = 1, y = 4, l = 1$$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4	A?			
3				
2	B		C	
1				
	1	2	3	4

$$x = 1, y = 4, l = 2$$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \leq l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 2: Find the recurrence

4	A?			
3	B			
2				
1				C
	1	2	3	4

$$x = 1, y = 4, l = 3$$

1. If $y = 1$ then $Sub(x, y)$ is the set of variables A such that $A \to w_x$ is a rule of the grammar.
2. If $y > 1$ then $Sub(x, y)$ is the set of variables A for which there is a rule $A \to BC$ and an integer l with $1 \le l < y$ such that $B \in Sub(x, l)$ and $C \in Sub(x + l, y - l)$.

Step 3: each subproblem solved once

We want to avoid computing table entries more than once.

– If your algorithm is recursive, just check if the value has already been computed. If yes, use that value and don't recurse. If not, recurse.

– If your algorithm is iterative, just build in order: row by row, bottom to top, left to right.

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

4				
3				
2				
1				
	1	2	3	4

$$w = aabb$$

$$
\begin{aligned}
S &\to AB \mid AX \mid \epsilon \\
T &\to AB \mid AX \\
X &\to TB \\
A &\to a \\
B &\to b
\end{aligned}
$$

4				
3				
2				
1	A	A	B	B
	1	2	3	4

$$w = aabb$$

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

4				
3				
2		S,T		
1	A	A	B	B
	1	2	3	4

$$w = aabb$$

$$S \rightarrow AB \mid AX \mid \epsilon$$

$$T \rightarrow AB \mid AX$$

$$X \rightarrow TB$$

$$A \rightarrow a$$

$$B \rightarrow b$$

4				
3		X		
2		S,T		
1	A	A	B	B
	1	2	3	4

$$w = aabb$$

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

4	S,T			
3		X		
2		S,T		
1	A	A	B	B
	1	2	3	4

$$w = aabb$$

Can we write this iteratively?

$D =$ "On input $w = w_1 \cdots w_n$:

1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. ⟦ $w = \varepsilon$ case ⟧
2. For $i = 1$ to n: ⟦ examine each substring of length 1 ⟧
3.   For each variable A:
4.   Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5.   If so, place A in $table(i, i)$.
6. For $l = 2$ to n: ⟦ l is the length of the substring ⟧
7.   For $i = 1$ to $n - l + 1$: ⟦ i is the start position of the substring ⟧
8.   Let $j = i + l - 1$. ⟦ j is the end position of the substring ⟧
9.   For $k = i$ to $j - 1$: ⟦ k is the split position ⟧
10.   For each rule $A \to BC$:
11.   If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

- Pseudocode from "Introduction to the theory of computation" by Michael Sipser, 3rd edition, Theorem 7.16.
- NB. Sipser uses $table(i, j)$ to mean the variables A that derive the substring starting at position i and ending at position j.

How efficient is this algorithm?

$|w|$ = length of w, $|G|$ = size of G (num. bits required to store G).

Time complexity

- $O(|w|^2)$ entries in the table,
- and each entry requires $O(|w||G|)$ work to compute, since one must check each rule and check $< n$ splits.
- So the total time is $O(|w|^3|G|)$.

Asides

- For fixed G and varying w, the time is $O(|w|^3)$.
- If the input is large (e.g., a compiling a very large program), then $O(|w|^3)$ is too high. So, one often resorts to using restricted grammars for which there are linear-time algorithms.
- Btw, there are subcubic algorithms for parsing CFGs based on the fact that Matrix Multiplication can be done in subcubic time (!)

What if I want to compute a derivation?

- Store more information!
- Idea: for every $A \in Sub(x, y)$ store a rule $A \to BC$ and a split l that witnessed why A got added to $Sub(x, y)$.
- You can then compute a rightmost derivation using a stack containing elements of the form (A, x, y) which represents the rightmost variable A and the substring of w that needs to be produced from A.

1. Push $(S, 1, n)$ onto the stack, and repeat the following:
2. Look at the top element of the stack (A, x, y) and get $A \to BC$ and l from $Sub(x, y)$.
3. if $y = 1$ then apply the rule $A \to w_x$ and pop the stack.
4. if $y > 1$ then apply the rule $A \to BC$, pop the stack, and push the element (B, x, l) followed by $(C, x + l, y - l)$ onto the stack.

Summary

We have studied some fundamental models of computation:

1. Regular expressions, finite automata
2. Context-free grammars
3. Turing machines

There is a machine-theoretic characterisation of context-free languages . . .

- Pushdown automaton = nondeterministic automaton + stack
- See Sipser Chapter 2.2