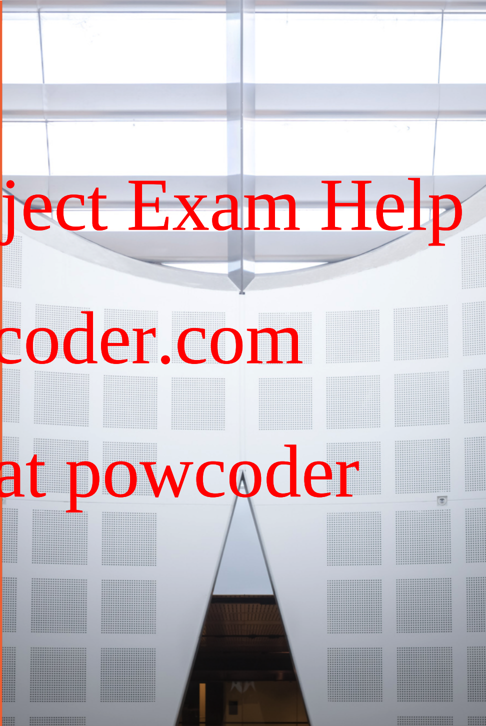
COMP2022|2922
Models of Computation

Lesson 9a: Chomsky Normal Form

Presented by

Sasha Rubin
School of Computer Science

THE UNIVERSITY OF SYDNEY



- A contex-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:

Input: a CFG G and string w. Output: 1 if the string w is generated by G, and 0 otherwise.

- 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 and declare "the string w cannot be derived from G"?
   - This problem can be fixed, but the resulting algorithm takes exponential time in the worst case, i.e., is very slow.
2. Use a table-filling algorithm (aka tabulation, aka dynamic programming).
   - Systematically compute, for every substring v of w, which non-terminals of G derive v (if any).
   - Then check if the start state S is in the set computed for the whole string w.
   - The resulting algorithm takes polynomial time in the worst case, i.e., is acceptably fast.
   - This is the approach we will take today! It is called the CYK algorithm

Input: a CFG G and string w.
Output: 1 if the string w is generated by G, and 0 otherwise.

Plan

1. Convert G into a grammar G' such that $L(G) = L(G')$ and G' is in a special normal form called Chomsky Normal Form.
2. Apply the table-filling algorithm to G' and w, and read off the answer.

Skeptic

Q. Why do we do the normal form?

A. In order to make the table small and easier to implement.

E.g., the parse-trees of a grammar in CNF are binary trees!

Chomsky Normal Form

Definition

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

- $A \to BC$ (A, B, C are any variables, except that neither B nor C is the start symbol)
- $A \to a$ (A is any variable and a is a terminal)
- $S \to \varepsilon$ (where S is the start symbol).

In the next slides, we will give a 5-step algorithm that transforms every CFG into an equivalent one in Chomsky Normal Form:

1. START: Eliminate the start symbol 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 symbol 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 symbol S_0 and the rule $S_0 \to S$

Chomsky Normal Form: algorithm

1. Eliminate the start symbol 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 symbol 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 $A \to X_1 X_2 \ldots X_n$ (where $n > 2$), delete it and create new variables $A_1, A_2, \ldots, A_{n-2}$ and rules:

$$
\begin{aligned}
A &\to X_1 A_1 \\
A_1 &\to X_2 A_2 \\
&\vdots \\
A_{n-3} &\to X_{n-2} A_{n-2} \\
A_{n-2} &\to X_{n-1} X_n
\end{aligned}
$$

Chomsky Normal Form: algorithm

1. Eliminate the start symbol 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 $S_0 \to \varepsilon$)

- Remove the rule.
- For each rule $A \to \alpha$ containing U, add every possible rule $A \to \alpha'$ where α' is α with one or more U's removed; only add the rule $A \to \epsilon$ if this rule has not already been removed.

Chomsky Normal Form: algorithm

1. Eliminate the start symbol 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 $A \to B$:

- Remove the rule $A \to B$
- For each rule of the form $B \to \alpha$ add the new rule $A \to \alpha$ (unless it was previously removed)

Chomsky Normal Form: example

$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

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

$$\mathbf{S_0 \to S}$$
$$S \to ASA \mid aB$$
$$A \to B \mid S$$
$$B \to b \mid \varepsilon$$

Chomsky Normal Form: example

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid aB \\
A &\to B \mid S \\
B &\to b \mid \varepsilon
\end{aligned}
$$

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

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid \mathbf{N_a} B \\
A &\to B \mid S \\
B &\to b \mid \varepsilon \\
\mathbf{N_a} &\to \mathbf{a}
\end{aligned}
$$

Chomsky Normal Form: example

$$
\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid N_a B \\
A &\to B \mid S \\
B &\to b \mid \varepsilon \\
N_a &\to a
\end{aligned}
$$

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

$$
\begin{aligned}
S_0 &\to S \\
S &\to \mathbf{AS_1} \mid N_a B \\
\mathbf{S_1} &\to \mathbf{SA} \\
A &\to B \mid S \\
B &\to b \mid \varepsilon \\
N_a &\to a
\end{aligned}
$$

Chomsky Normal Form: example

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \\
S_1 &\to SA \\
A &\to B \mid S \\
B &\to b \mid \varepsilon \\
N_a &\to a
\end{aligned}
$$

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

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \mid \mathbf{N_a} \\
S_1 &\to SA \\
A &\to B \mid S \mid \varepsilon \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

Chomsky Normal Form: example

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \mid N_a \\
S_1 &\to SA \\
A &\to B \mid S \mid \varepsilon \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

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

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \mid N_a \mid \mathbf{S_1} \\
S_1 &\to SA \mid \mathbf{S} \\
A &\to B \mid S \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

Chomsky Normal Form: example

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \mid N_a \mid S_1 \\
S_1 &\to SA \mid S \\
A &\to B \mid S \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

Step 5 (UNIT): Eliminate unit rules $S \to N_a$, $A \to B$, $S \to S_1$

$$
\begin{aligned}
S_0 &\to S \\
S &\to AS_1 \mid N_aB \mid \mathbf{a} \mid \mathbf{SA} \qquad \text{(and useless } S \to S\text{)} \\
S_1 &\to SA \mid S \\
A &\to \mathbf{b} \mid S \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

Chomsky Normal Form: example

$$S_0 \to S$$
$$S \to AS_1 \mid N_aB \mid a \mid SA$$
$$S_1 \to SA \mid S$$
$$A \to b \mid S$$
$$B \to b$$
$$N_a \to a$$

Step 5 (UNIT): Eliminate unit rules $S_0 \to S$, $S_1 \to S$, $A \to S$

$$S_0 \to \mathbf{AS_1} \mid \mathbf{N_aB} \mid \mathbf{a} \mid \mathbf{SA}$$
$$S \to AS_1 \mid N_aB \mid a \mid SA$$
$$S_1 \to \mathbf{AS_1} \mid \mathbf{N_aB} \mid \mathbf{a} \mid SA$$
$$A \to b \mid \mathbf{AS_1} \mid \mathbf{N_aB} \mid \mathbf{a} \mid \mathbf{SA}$$
$$B \to b$$
$$N_a \to a$$

Chomsky Normal Form: example

All done!

$$
\begin{aligned}
S_0 &\to AS_1 \mid N_aB \mid a \mid SA \\
S &\to AS_1 \mid N_aB \mid a \mid SA \\
S_1 &\to AS_1 \mid N_aB \mid a \mid SA \\
A &\to b \mid AS_1 \mid N_aB \mid a \mid SA \\
B &\to b \\
N_a &\to a
\end{aligned}
$$

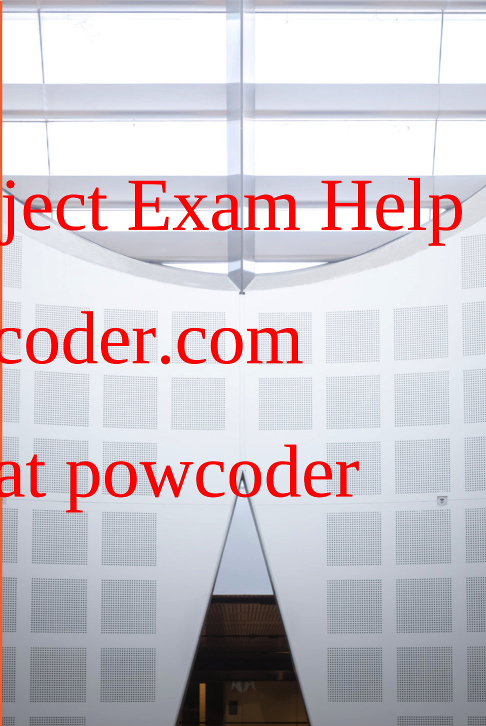
COMP2022|2922 Models of Computation

Lesson 9b: Membership problem for CFGs in CNF

Presented by

Sasha Rubin
School of Computer Science

THE UNIVERSITY OF SYDNEY



Membership problem for CFG in CNF

Input: a CFG G that is in CNF, and string w.
Output: 1 if the string w is generated by G, and 0 otherwise

Table-filling algorithm (aka Dynamic Programming)

- This approach accumulates 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)

- Main steps: define the subproblems, find the recursion, make sure you solve each subproblem once.
- You will see this again in `COMP3027:Algorithm Design`
- The algorithm we will see is known as the CYK algorithm (Cocke–Younger–Kasami).

What are the subproblems?

Idea

– A parse tree for a string w is built from a root, a left subtree, and a right subtree (remember that G is in CNF)

– The left (right) subtree is parse tree of a prefix (suffix) of w.

– So, the immediate subproblems for w are computing which variables generate prefixes/suffixes of w.

– So, the subproblems for w are computing which variables generate which substrings of w.

What are the entries in the table?

Given G in CNF, and a non-empty string $w = w_1 w_2 \cdots w_n$:

- Write $table(i,j)$ for the set of variables A that generate the substring $w_i w_{i+1} \cdots w_j$.
- The algorithm will compute $table(i,j)$ for all $1 \leq i < j \leq n$.
- Once the table is computed, just check if $S \in table(1,n)$. If yes, then the grammar generates w, and if no, then it doesn't.

Computing the table recursively

Compute $table(i,j)$ using the following recursive procedure:

1. If $i = j$ then $table(i,j)$ is the set of variables A such that $A \to w_i$ is a rule of the grammar.
2. If $i < j$ then $table(i,j)$ is the set of variables A for which there is a rule $A \to BC$ and an integer k with $i \leq k < j$ such that $B \in table(i,k)$ and $C \in table(k+1,j)$.

Q: Why is the recursion correct?

Computing the table recursively

Compute $table(i, j)$ using the following recursive procedure:

1. If $i = j$ then $table(i, j)$ is the set of variables A such that $A \to w_i$ is a rule of the grammar.
2. If $i < j$ then $table(i, j)$ is the set of variables A for which there is a rule $A \to BC$ and an integer k with $i \leq k < j$ such that $B \in table(i, k)$ and $C \in table(k + 1, j)$.

Q: Why does this recursion stop?

- At each step, we call the procedure on "smaller" problems.
- In what sense is $table(i, k)$ and $table(k + 1, j)$ smaller than $table(i, j)$? The size of the intervals $[i, k]$ and $[k + 1, j]$ is smaller than the size of the interval $[i, j]$.

We want to avoid computing table entries more than once.

- So, before making a recursive call just check if the value has already been computed. If yes, use that value and don't recurse. If not, recurse.

Can we write this with a bunch of loops?

Yes, but it is harder to read. See Sipser (edition 3) Theorem 7.16.

$D =$ "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)

Can we fill the table by hand?

Yes, but this is best done by a computer!

(1,7)						
(1,6)	(2,7)					
(1,5)	(2,6)	(3,7)				
(1,4)	(2,5)	(3,6)	(4,7)			
(1,3)	(2,4)	(3,5)	(4,6)	(5,7)		
(1,2)	(2,3)	(3,4)	(4,5)	(5,6)	(6,7)	
(1,1)	(2,2)	(3,3)	(4,4)	(5,5)	(6,6)	(7,7)
w_1	w_2	w_3	w_4	w_5	w_6	w_8

Q: Where do I put the entry $table(i, j)$?

- horizontal co-ordinate = starting position of substring = i
- vertical co-ordinate = length of substring = $j - i + 1$

Q: What entries are needed to compute $table(i, j)$?

- You have to look at the pairs $table(i, k), table(k + 1, j)$ for $k = i, \ldots, j - 1$; it's the right-angled triangle below $table(i, j)$.

Q: In what order are the entries computed?

- Row by row, bottom to top, left to right

Example

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	VP, V	Det	N	P	Det	N
she	eats	a	fish	with	a	fork
1	2	3	4	5	6	7

$$S \to NP\ VP$$
$$VP \to VP\ PP \mid V\ NP$$
$$PP \to P\ NP$$
$$NP \to Det\ N$$

Q: In what order are the entries computed?

- To compute an entry, you need the entries in the "right-angled triangle below it".
- Row by row, bottom to top, left to right

Example

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	VP, V	Det	N	P	Det	N
she	eats	a	fish	with	a	fork
1	2	3	4	5	6	7

$S \rightarrow NP\ VP$

$VP \rightarrow VP\ PP \mid V\ NP$

$PP \rightarrow P\ NP$

$NP \rightarrow Det\ N$

$VP \in table(2, 4)$ because

- the string from position 2 to 4 is "eats a fish",
- which can be split into "eats" from position 2 to 2,
- and "a fish" from position 3 to 4, and
- and $VP \rightarrow V\ NP$ is a rule, $V \in table(2, 2)$, and $NP \in table(3, 4)$.

Example

S						
	VP					
S						
	VP			PP		
S		NP			NP	
NP	VP, V	Det	N	P	Det	N
she	eats	a	fish	with	a	fork
1	2	3	4	5	6	7

$$S \rightarrow NP\ VP$$
$$VP \rightarrow VP\ PP \mid V\ NP$$
$$PP \rightarrow P\ NP$$
$$NP \rightarrow Det\ N$$

$VP \in table(2, 7)$ because

- the string from position 2 to 7 is "eats a fish with a fork",
- which can be split into "eats a fish" from position 2 to 4,
- and "with a fork" from position 5 to 7, and
- and $VP \rightarrow VP\ PP$ is a rule, $VP \in table(2, 4)$, and $PP \in table(5, 7)$.

How efficient is this algorithm?

Time complexity

- There are $O(n^2)$ entries in the table,
- and each entry requires $O(n|G|)$ work to compute, since one must check each rule and check at most n splits,
- So the total time is $O(n^3|G|)$.
- Here $|G|$, the size of G, is the number of bits required to write the grammar down, which is polynomial in the number of rules, variables and terminals.

Asides

- For fixed G and varying w, the time is $O(n^3)$.
- If the input is large (e.g., a compiling a very large program), then this complexity is too high. So, in this case, one uses restricted grammars for which there are faster algorithms, see `COMP3109:Programming Languages and Paradigms`

What if I want to compute a derivation?

You can adjust the algorithm to store more information in order to produce a derivation (or parse tree).

- Store in $table(i, j)$ a rule $A \to BC$ and splitting point k ($i \leq k < j$) that can be used to derive $A \Rightarrow^* w_i w_{i+1} \ldots w_j$.
- You can then deduce a rightmost derivation using a stack.

- Start by pushing the element $(S, 1, n)$ onto the stack, and then repeat the following:
  - if (A, i, i) is the top element of the stack then apply the rule $A \to w_i$ and pop the stack.

  - if (A, i, j) is the top-element of the stack, then apply the rule $A \to BC$, pop the stack, and push the element (B, i, k) followed by $(C, k+1, j)$ onto the stack.

Good to know

- There is a machine-theoretic characterisation of context-free languages (pushdown automaton = NFA + stack).
  - Come to COMP2022 to learn more! or see Sipser Chapter 2.2
- Not every language is context-free. E.g., $\{ww : w \in \{0, 1\}^*\}$ is not context-free.
  - The proof of this uses a pumping argument, see Sipser Chapter 2.3.

Where are we going?

Next week we start learning about an even more powerful model of computation that can even recognise non-context-free languages — the Turing machine (= automaton + unbounded memory).

This is the most powerful model of computation that we know of, and is a model of a general purpose computer.