CS112: Theory of Computation (LFA)

Lecture 9: Pushdown Automata

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

April 29, 2025

Table of contents

Section 1

Previously on CS112

Intuition

- A grammar consists of a collection of **substitution rules**, also called **productions**
- Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow
- The symbol is called a **variable**. The string consists of variables and other symbols called **terminals**
- The variable symbols often are represented by capital letters
- The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols
- One variable is designated as the start variable
- It usually occurs on the left-hand side of the topmost rule

Example

For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and $\#$

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Intuition

You use a grammar to describe a language by generating each string of that language in the following manner:

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule
3. Repeat step 2 until no variables remain

Example

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a **derivation**. A derivation of string 000#111 in grammar G_1 is:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Example

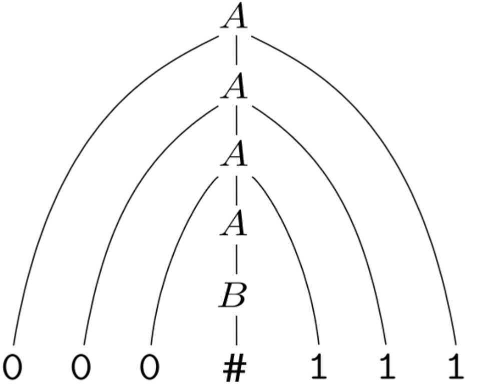

Figure: Parse tree for 000#111 in grammar G_1

Formal definition

Definition

A **context-free grammar** is a 4-tuple (V, Σ, R, S) where:

- V is a finite set called the **variables**
- Σ is a finite set, disjoint from V called the **terminals**
- R is a finite set of rules, with each rule being a variable and a string of variables and terminals
- $S \in V$ is the start variable

Section 2

Chomsky normal form

Noam Chomsky

Noam Chomsky is a groundbreaking thinker whose work reshaped linguistics, computer science, and philosophy. A longtime professor at MIT, he also became a leading voice for political activism and critical thinking about media, power, and society.

- According to The New York Times, Noam Chomsky is **arguably the most important intellectual alive**
- `https://www.goodreads.com/author/list/2476.Noam_Chomsky`
- If in doubt, start with: **How the World Works** and **Understanding Power**

Chomsky normal form

- Working with context-free grammars, it is often convenient to have them in simplified form
- The simplest and most useful forms is called **the Chomsky normal form**
- Chomsky normal form is useful in giving algorithms for working with context-free grammars

Formal definition

Definition

A context-free grammar is in Chomsky normal form if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables — except that B and C may not be the start variable. In addition, we allow the rule $S \rightarrow \epsilon$, where S is the start variable

Converting to Chomsky normal form

Theorem

Any context-free language is generated by a context-free grammar in Chomsky normal form

- Proof idea is to convert any grammar G into Chomsky normal form
- The conversion has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory

Proof idea

- First, we add a new start variable
- Then, we eliminate all ϵ-rules of the form $A \to \epsilon$
- We also eliminate all unit rules of the form $A \to B$
- In both cases we patch up the grammar to be sure that it still generates the same language
- Finally, we convert the remaining rules into the proper form

Formal proof

Proof.

First, we add a new start variable S_0 and the rule $S_0 \rightarrow S$, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule

Second, we take care of all ϵ-rules. We remove an ϵ-rule $A \rightarrow \epsilon$, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \rightarrow uAv$ is a rule in which u and v are strings of variables and terminals, we add rule $R \rightarrow uv$. We do so for each occurrence of an A, so the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$, $R \rightarrow uAvw$ and $R \rightarrow uvw$. If we have the rule $R \rightarrow A$, we add $R \rightarrow \epsilon$ unless we had previously removed the rule $R \rightarrow \epsilon$. We repeat these steps until we eliminate all ϵ-rules not involving the start variable.

Formal proof

Proof.

Third, we handle all unit rules. We remove a unit rule $A \to B$. Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed. As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.
Finally, we convert all remaining rules into the proper form. We replace each rule $A \to u_1 u_2 \dots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \to u_1 A_1, A_1 \to u_2 A_2, A_2 \to u_3 A_3, \dots, A_{k-2} \to u_{k-1} u_k$ The A_i's are new variables. We replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \to u_i$ $\square$

Section 3

Examples

Example

- Let G_6 be the following CFG and convert it to Chomsky normal form by using the conversion procedure just given.
- The series of grammars presented illustrates the steps in the conversion.

$$
\begin{aligned}
S &\to ASA|aB \\
A &\to B|S \\
B &\to b|\epsilon
\end{aligned}
$$

Example

1. The result of applying the first step to make a new start variable appears on the right

$$\begin{aligned} S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\epsilon \end{aligned} \qquad\qquad \begin{aligned} S_0 &\to S \\ S &\to ASA|aB \\ A &\to B|S \\ B &\to b|\epsilon \end{aligned}$$

2. Remove ϵ-rules $B \to \epsilon$, shown on the left and $A \to \epsilon$, shown on the right

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|aB|a \\ A &\to B|S|\epsilon \\ B &\to b \end{aligned} \qquad\qquad \begin{aligned} S_0 &\to S \\ S &\to ASA|aB|a|SA|AS|S \\ A &\to B|S \\ B &\to b \end{aligned}$$

Example

3. Remove unit rules $S \to S$, shown on the left and $S_0 \to S$, shown on the right

$$\begin{aligned} S_0 &\to S \\ S &\to ASA|aB|a|SA|AS \\ A &\to B|S \\ B &\to b \end{aligned}$$

$$\begin{aligned} S_0 &\to ASA|aB|a|SA|AS \\ S &\to ASA|aB|a|SA|AS \\ A &\to B|S \\ B &\to b \end{aligned}$$

4. Remove unit rules $A \to B$ and $A \to S$

$$\begin{aligned} S_0 &\to ASA|aB|a|SA|AS \\ S &\to ASA|aB|a|SA|AS \\ A &\to S|b \\ B &\to b \end{aligned}$$

$$\begin{aligned} S_0 &\to ASA|aB|a|SA|AS \\ S &\to ASA|aB|a|SA|AS \\ A &\to b|ASA|aB|a|SA|AS \\ B &\to b \end{aligned}$$

Example

5. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. We simplified the resulting grammar by using a single variable U and rule $U \to a$

$$
\begin{aligned}
S_0 &\to AA_1 | UB | a | SA | AS \\
S &\to AA_1 | UB | a | SA | AS \\
A &\to b | AA_1 | UB | a | SA | AS \\
A_1 &\to SA \\
U &\to a \\
B &\to b
\end{aligned}
$$

Section 4

Pushdown Automata

Context setup

Corresponding to Sipser 2.2

Context setup

- Next we introduce a new type of computational model called **pushdown automata**
- These automata are like nondeterministic finite automata but have an extra component called a **stack**
- The stack provides additional memory beyond the finite amount available in the control
- The stack allows pushdown automata to recognize some nonregular languages
- **Pushdown automata are equivalent in power to context-free grammars**
- This equivalence is useful because it gives us two options for proving that a language is context free
- We can give either a context-free grammar generating it or a pushdown automaton recognizing it
- Certain languages are more easily described in terms of generators, whereas others are more easily described by recognizers

Section 5

Intuition

Schematics

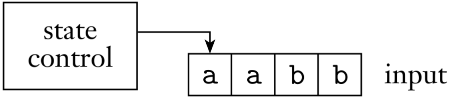

Figure: Schematic of a finite automaton

- The left figure is a schematic representation of a finite automaton
- The control represents the states and transition function
- The tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read

Schematics

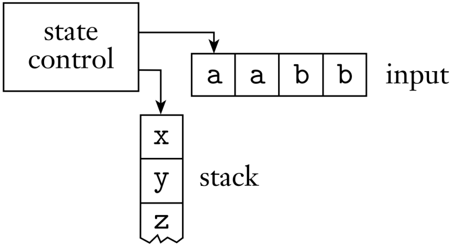

Figure: Schematic of a pushdown automaton

- A **pushdown automaton** (PDA) can write symbols on the stack and read them back later
- Writing a symbol "pushes down" all the other symbols on the stack.
- At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up
- Writing a symbol on the stack is often referred to as pushing the symbol, and removing a symbol is referred to as popping it

Schematics

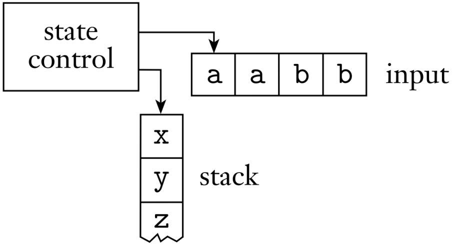

Figure: Schematic of a pushdown automaton

- Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage device
- If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed
- A stack is valuable because it can hold an unlimited amount of information

Example

Let's take the language $\{0^n 1^n | n \geq 0\}$. While a DFA is unable to recognize, a PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen. Thus the unlimited nature of a stack allows the PDA to store numbers of unbounded size, This is how:

- Read symbols from the input. As each 0 is read, push it onto the stack
- As soon as 1s are seen, pop a 0 off the stack for each 1 read
- If reading the input is finished exactly when the stack becomes empty of 0s, accept the input
- If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input

Nondeterminism

- PDA may be nondeterministic. Deterministic and nondeterministic PDA are *not* equivalent in power
- Nondeterministic PDA recognize certain languages that no deterministic PDA can recognize
- Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the PDA situation is different
- We will focus on nondeterministic PDA because these automata are equivalent in power to context-free grammars

Section 6

Formal definition

Transition function

- Formal definition of a PDA is similar to that of a finite automaton, except for the stack
- The stack is a device containing symbols drawn from some alphabet
- The machine may use different alphabets for its input and its stack, so now we specify both an input alphabet Σ and a stack alphabet Γ
- Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$
- Core aspect of any formal definition of an automaton is the transition function, which describes its behavior
- The domain of the transition function is $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$
- Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a PDA
- Either symbol may be ϵ, causing the machine to move without reading a symbol from the input or without reading a symbol from the stack

Transition function

- We need to consider what to allow the automaton to do when it is in a particular situation
- It may enter some new state and possibly write a symbol on the top of the stack
- The function δ can indicate this action by returning a member of Q together with a member of Γ_ϵ, that is, a member of $Q \times \Gamma_\epsilon$
- Because we allow nondeterminism in this model, a situation may have several legal next moves
- The transition function incorporates nondeterminism in the usual way, by returning a set of members of $Q \times \Gamma_\epsilon$, that is a member of $\mathcal{P}(Q \times \Gamma_\epsilon)$
- Gluing everything together we get: $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$

Formal definition

Definition

A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are all finite sets, and

1. Q is the set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function
5. $q_0 \in Q$ is the start state
6. $F \in Q$ is the set of accept states

PDA computation

A PDA $M = (Q, \Sigma, \Gamma, \delta, \delta_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \dots w_m$ where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, \dots, r_m \in Q$ and strings $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions:

1. $r_0 = q_0$ and $s_0 = \epsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack
2. For $i = 0, \dots, m-1$ we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol
3. $r_m \in F$. This condition states that an accept state occurs at the input end

The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation

Section 7

Examples

Example

This is the formal description of the PDA M_1 that recognizes the language $\{0^n 1^n | n \geq 0\}$.
$M_1 = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- $F = \{q_1, q_4\}$
- δ is described in the below table, wherein blank entries signify $\emptyset$

Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Example

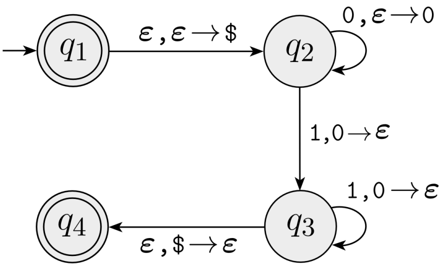

Figure: State diagram for the PDA M_1

- We can also use a state diagram to describe a PDA
- Such diagrams are similar to the state diagrams used to describe finite automata, modified to show how the PDA uses its stack when going from state to state
- We write $a, b \to c$ to signify that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with c

Example

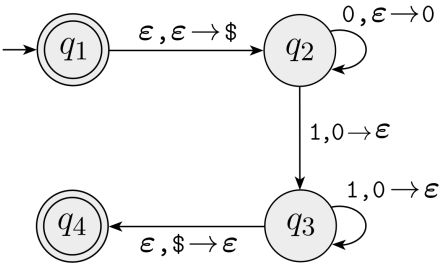

Figure: State diagram for the PDA M_1

- Any or a, b, c may be ϵ
- If a is ϵ, the machine may make this transition without reading any symbol from the input
- If b is ϵ, the machine may make this transition without reading and popping any symbol from the stack
- If c is ϵ, the machine does not write any symbol on the stack when going along this transition

Assumptions

- The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. M_1 is able to get the same effect by initially placing a special symbol \$ on the stack. Then if it ever sees the \$ again, it knows that the stack effectively is empty. So, when we refer to testing for an empty stack in an informal description of a PDA, we implement this procedure.
- Also, PDAs cannot test explicitly for having reached the end of the input string. M_1 is able to achieve that effect because the accept state takes effect only when the machine is at the end of the input. Thus from now on, we assume that PDAs can test for the end of the input, and we know that we can implement it in the same manner

Example

Next, let's study the PDA M_2 that recognizes the language $\{a^i b^j c^k | i, j, k \geq 0 \textit{ and } i = j \textit{ or } i = k\}$.

- Informally, the PDA for this language works by first reading and pushing the a's
- When the a's are done, the machine has all of them on the stack so that it can match, them with either the b's or the c's
- This maneuver is a bit tricky because the machine doesn't know in advance whether to match the a's with the b's or the c's
- **Nondeterminism** to the rescue!
- Using its nondeterminism, the PDA can guess whether to match the a's with the b's or with the c's
- Think of the machine as having two branches of its nondeterminism, one for each possible guess. If either of them matches, that branch accepts and the entire machine accepts

Example

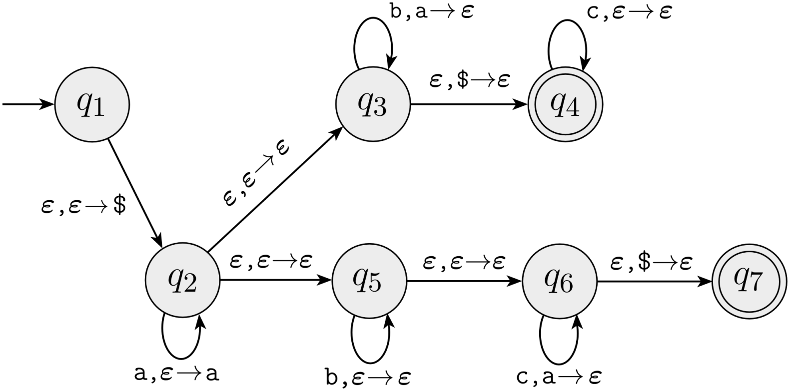

Figure: State diagram for PDA M_2

Example

Next, let's study the PDA M_3 that recognizes the language $\{ww^r | w \in \{0,1\}^*\}$ where w^r is w in written backwards.

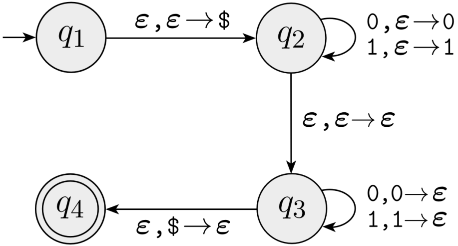

Figure: State diagram for the PDA M_3

- Start by pushing the symbols that are read onto the stack. At each point, nondeterministically guess that the middle of the string has been reached and then change into popping off the stack for each symbol read, checking to see that they are the same
- If they were always the same symbol and the stack empties at the same time as the input is finished, accept; otherwise reject