CS112: Theory of Computation (LFA)

Lecture12: Pushdown Automata

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

May 17, 2022

Table of contents

Section 1

Previously on CS112

CFG 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

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

CNF Formal definition

Definition

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

$$A \to BC$$
$$A \to a$$

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 $S \to \epsilon$, where S is the start variable

PDA Schematics

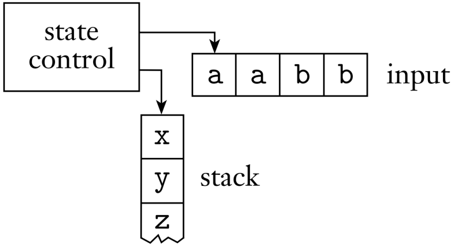


Figure: Schematic of a pushdown automaton

- 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

PDA 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

Section 2

Context setup

Context setup

Corresponding to Sipser 2.2

Context setup

- **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 3

Equivalence with Context-free grammars

Equivalence with Context-free grammars

- Next, we study that context-free grammars and pushdown automata are equivalent in power
- Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa
- We defined a context-free language to be any language that can be described with a context-free grammar and our objective is the following theorem

Equivalence with Context-free grammars

Theorem

A language is context free if and only if some pushdown automaton recognizes it

As usual for "if and only if" theorems, we have two directions to prove. We formulate both directions as lemmas. First, we do the easier forward direction

Lemma

If a language is context free, then some pushdown automaton recognizes it

Proof idea

- Let A be a CFL. From the definition we know that A has a CFG, G generating it. We show how to convert G into an equivalent PDA, which we call P
- The PDA P that we now describe will work by accepting its input w, if G generates that input, by determining whether there is a derivation for w
- Recall that a derivation is simply the sequence of substitutions made as a grammar generates a string
- Each step of the derivation yields an intermediate string of variables and terminals
- We design P to determine whether some series of substitutions using the rules of G can lead from the start variable to w
- One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make
- The PDA's nondeterminism allows it to guess the sequence of correct substitutions
- At each step of the derivation, one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable

Proof idea

- The PDA P begins by writing the start variable on its stack. It goes through a series of intermediate strings, making one substitution after another
- Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string
- Then P accepts if this string is identical to the string it has received as input
- Implementing this strategy on a PDA requires one additional idea. We need to see how the PDA stores the intermediate strings as it goes from one to another
- Simply using the stack for storing each intermediate string is tempting. However, that doesn't quite work because the PDA needs to find the variables in the intermediate string and make substitutions
- The PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable
- The way around this problem is to keep only part of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string
- Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string

Proof idea

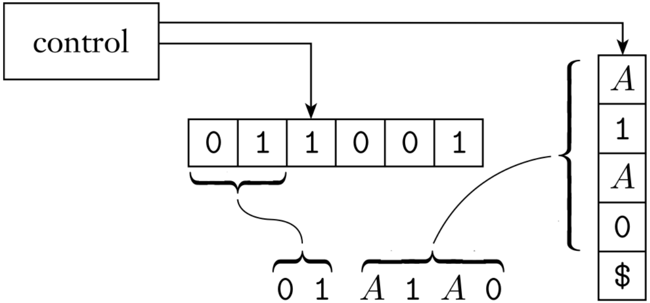


Figure: P representing the intermediate string $01A1A0$

Proof idea

The following is an informal description of P

1. Place the marker symbol \$ and the start variable on the stack
2. Repeat the following steps forever
    2.1 If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule
    2.2 If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism
    2.3 If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read

Lemma 1

Proof.

We now give the formal details of PDA $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$. To make the construction clearer, we use shorthand notation for the transition function. This notation provides a way to write an entire string on the stack in one step of the machine. We can simulate this action by introducing additional states to write the string one symbol at a time, as implemented in the following formal construction.

Let q and r be states of the PDA and let a be in Σ_ϵ and s be in Γ_ϵ. Say that we want the PDA to go from q to r when it reads a and pops s. Furthermore, we want it to push the entire string $u = u_1 \dots u_l$ on the stack at the same time.

Lemma 1

Proof.

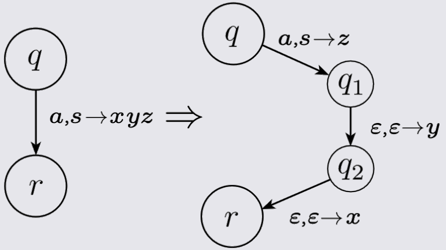
We can implement this action by introducing new states $q_1, \ldots, q_{l-1}$ and setting the transition function as follows:

$$\begin{aligned}
\delta(q, a, s) &\text{ to contain } (q_1, u_l) \\
\delta(q_1, \epsilon, \epsilon) &= \{(q_2, u_{l-1})\} \\
\delta(q_2, \epsilon, \epsilon) &= \{(q_2, u_{l-2})\} \\
&\vdots \\
\delta(q_{l-1}, \epsilon, \epsilon) &= \{(r, u_1)\}
\end{aligned}$$

Lemma 1

Proof.

We use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r



q
$a,s \to xyz$
r
$\Rightarrow$
q
$a,s \to z$
q_1
$\varepsilon,\varepsilon \to y$
q_2
$\varepsilon,\varepsilon \to x$
r

Figure: Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

Lemma 1

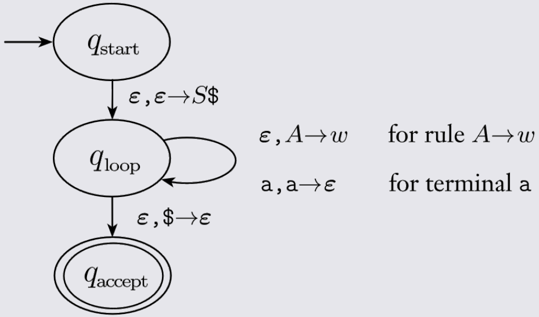
Proof.

The states of P are $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}. The transition function is defined as follows. We begin by initializing the stack to contain the symbols \$ and S, implementing *step 1* in the informal description: $\delta(q_{start}, \epsilon, \epsilon) = \{(q_{loop}, S\$)\}$. Then we put in transitions for the main loop of *step 2*. Next we handle all three cases:

1. Wherein the top of the stack contains a variable. Let $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) | where\ A \rightarrow w \text{ is a rule in } R\}$.
2. Wherein the top of the stack contains a terminal. Let $\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$.
3. Wherein the empty stack marker \$ is on the top of the stack. Let $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$.

Lemma 1

Proof.



q_{start}
$\varepsilon, \varepsilon \to S\$$
q_{loop}
$\varepsilon, A \to w$ for rule $A \to w$
$\mathtt{a}, \mathtt{a} \to \varepsilon$ for terminal $\mathtt{a}$
$\varepsilon, \$ \to \varepsilon$
q_{accept}

Figure: State diagram of P

$\square$

Example

Using the procedure from the above lema let us construct a PDA P_1 from the CFG G:

$$S \rightarrow aTb|b$$
$$T \rightarrow Ta|\epsilon$$

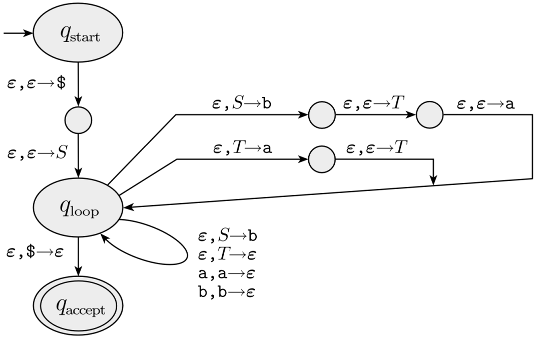

q_{start}
$\varepsilon, \varepsilon \rightarrow \$$
$\varepsilon, \varepsilon \rightarrow S$
$\varepsilon, S \rightarrow$ b
$\varepsilon, \varepsilon \rightarrow T$
$\varepsilon, \varepsilon \rightarrow$ a
$\varepsilon, T \rightarrow$ a
$\varepsilon, \varepsilon \rightarrow T$
q_{loop}
$\varepsilon, S \rightarrow$ b
$\varepsilon, T \rightarrow \varepsilon$
a, a $\rightarrow \varepsilon$
b, b $\rightarrow \varepsilon$
$\varepsilon, \$ \rightarrow \varepsilon$
q_{accept}

Figure: State diagram of P_1

Lemma 2

Now we prove the reverse direction of the theorem. For the forward direction, we gave a procedure for converting a CFG into a PDA. The main idea was to design the automaton so that it simulates the grammar. Now we want to give a procedure for going the other way: converting a PDA into a CFG. *We design the grammar to simulate the automaton*. This task is challenging because "programming" an automaton is easier than "programming" a grammar

Lemma

If a pushdown automaton recognizes some language, then it is context free

Proof idea

- We have a PDA P and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state
- We design a grammar that does somewhat more. For each pair of states p and q in P, the grammar will have a variable A_{pq} . This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p
- We simplify our task by modifying P slightly to give it the following three features:
    1. It has a single accept state, q_{accept}
    2. It empties its stack before accepting
    3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time

Proof idea

- Giving P features 1 and 2 is easy.
- To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol
- To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings
- For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop because the stack ends up empty

Proof idea

- So, two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not
- If that's the case then the stack could be empty only at the beginning and end of P 's computation on x
- If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point
- We simulate the former possibility with the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q
- We simulate the latter possibility with the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack becomes empty

Lemma 2

Proof.

Assume that $P = \{Q, \Sigma, \Gamma, \delta, q_0, \{q_accept\}\}$ and construct G. The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is $A_{q_0, q_{accept}}$. We describe the rules of G in three parts:

1. For each $p, q, r, s \in Q$, $u \in \Gamma$ and $a, b \in \Sigma$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ) we set the rule $A_{pq} \to aA_{rs}b$ in G
2. For each $p, q, r \in Q$ we set the rule $A_{pq} \to A_{pr}A_{rq}$ in G
3. For each $p \in Q$, we set the rule $A_{pp} \to \epsilon$ in G

Insights

You may gain some insight for this construction from the following figure:

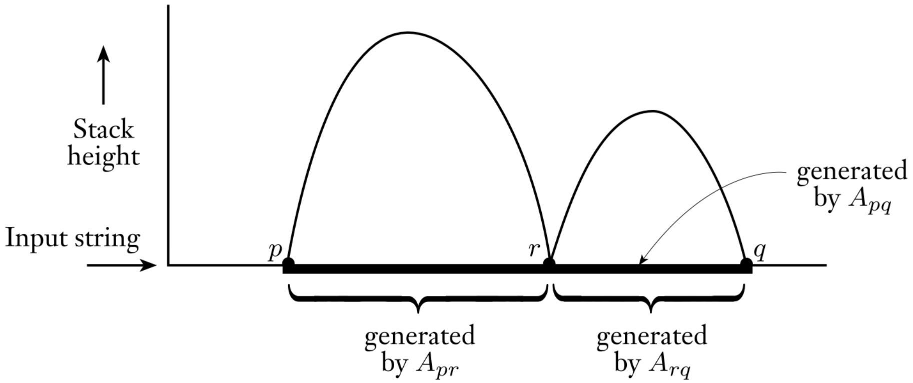


Figure: PDA computation corresponding to the rule $A_{pq} \to A_{pr}A_{rq}$

Insights

And from the following figure:

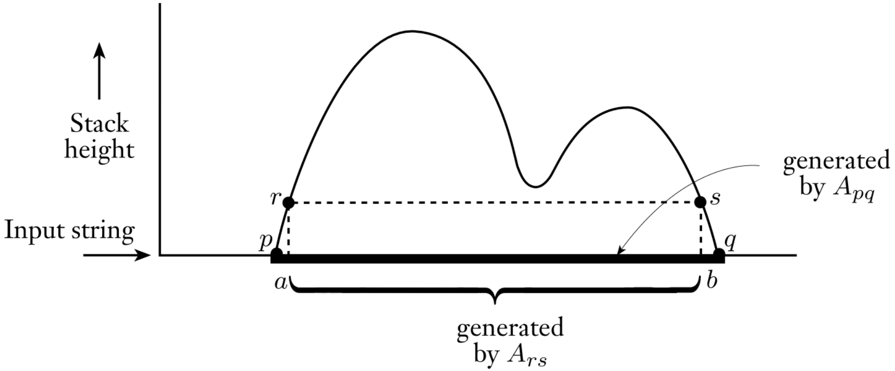


Figure: PDA computation corresponding to the rule $A_{pq} \rightarrow aA_{rs}b$

Lemma 2

Next, ~~we~~ you prove that this construction works by demonstrating that A_{pq} generates x if and only if (iff) x can bring P from p with empty stack to q with empty stack. We consider each direction of the iff as a separate claim

Claim

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack

Proof.

You prove this claim by induction on the number of steps in the derivation of x from A_pq
Basis: The derivation has 1 step. A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \epsilon$. Clearly, input ϵ takes P from p with empty stack to p with empty stack so the basis is proved.

Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k + 1$. $\square$

Lemma 2

Proof.

Using the above claims we have a proof of our (second) lemma □

- We have just proved that pushdown automata recognize the class of contextfree languages. This proof allows us to establish a relationship between the regular languages and the context-free languages
- Because every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack, we now know that every regular language is also a context-free language

Relationships

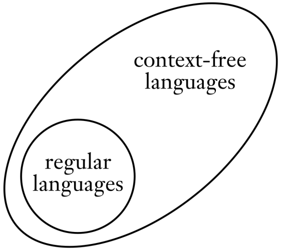


Figure: Relationship of the regular and context-free languages

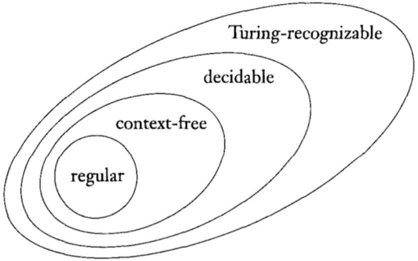


Figure: A broader view of future relationships

Section 4

Undecidability of The post correspondence problem

Section 5

Strong perfect graph theorem