Theory of Computation
Context-Free Languages

Dimitris Diochnos
School of Computer Science
University of Oklahoma

Outline

Table of Contents

Introduction

- With finite automata and regular expressions we can describe many languages, but some other simple languages such as $\{0^n 1^n | n \geq 0\}$ we cannot.
- *Context-free grammars* give us more power for such cases.
- **Applications**: Used for the specification and compilation of programming languages.
  - Parsers extract the meaning of a program
  - In some cases parsers can be created automatically given the grammar.

Context-free Languages

Languages of context-free grammars (CFGs) are called *context-free languages*.
(include regular languages and more)

Pushdown automata (PDAs) recognize context-free languages (CFLs).

Table of Contents

Context-Free Grammars

Grammar

A collection of **substitution rules**, also called **productions**.

Example of a grammar (call it G_1):

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

A is the start variable. B is a variable. 0, 1 and # are called terminals.

Derivations (How to generate strings):

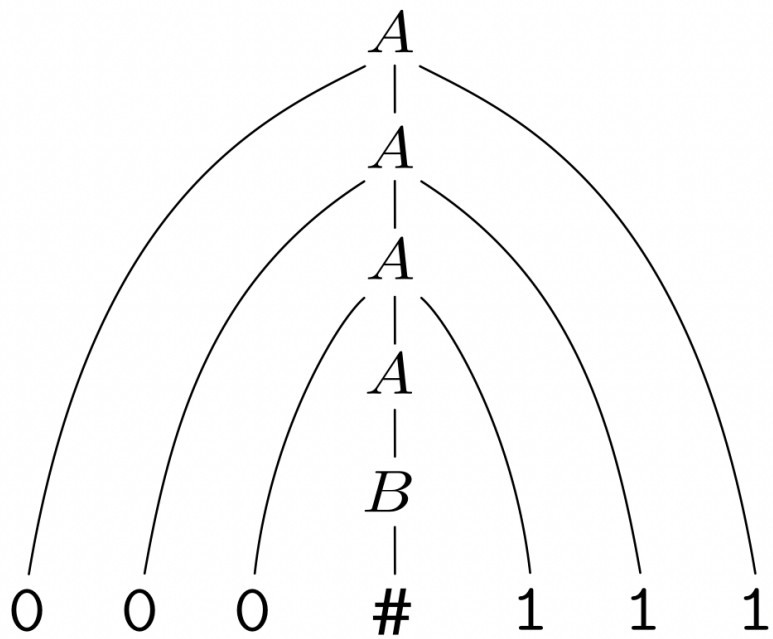
1. Write down the start variable.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the RHS of that rule.
3. Repeat step 2 until no variables remain.

Example

For example, grammar G1 generates the string 000#111:

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

Parse tree:



So: $L(G_1) = \{0^n\#1^n | n \geq 0\}$.

Convention: Abbreviate several rules with the same left-hand variable into a single line, using the symbol "|" as an "or".

Abbreviation

The set of rules

$$\begin{cases} A & \to & 0A1 \\ A & \to & B \end{cases}$$

is written down as

$$A \to 0A1|B.$$

Formal Definition of Context-Free Grammar

Definition 1 (Context-free grammar)

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

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

Formal Definition of Context-Free Grammar

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar,

- We say that uAv **yields** uwv, written $uAv \Rightarrow uwv$.
- We say that u **derives** v, written $u \overset{*}{\Rightarrow} v$,
  - if $u = v$, or
  - if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$.
- The **language of the grammar** is $\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$.

Example

Example 2

Consider grammar $G_3 = (\{S\}, \{a, b\}, R, S)$.
The set of rules, R, is

$$S \to aSb \mid SS \mid \lambda$$

This grammar generates strings such as *abab*, *aaabbb*, and *aababb*, i.e., strings that belong to the language of all *properly nested a's and b's* (in the sense of properly nested parentheses).

Designing Context-Free Grammars

- Trickier than designing finite automata
- If we can break a CFL into simpler pieces, do so! (Usually it is much simpler this way!)
  Then create grammars for the simpler parts and combine them (their rules) by adding a new rule

$$S \to S_1 | S_2 | \cdots | S_k$$

  where the variables S_i are the start variables for the individual grammars.

Example

Example 3

Design a grammar for the language $\{0^n1^n|n \geq 0\} \cup \{1^n0^n|n \geq 0\}$.

Grammar 1: $S_1 \rightarrow 0S_1 1 \mid \lambda$ (for language $\{0^n1^n|n \geq 0\}$)
Grammar 2: $S_2 \rightarrow 1S_2 0 \mid \lambda$ (for language $\{1^n0^n|n \geq 0\}$)
So the requested grammar is:

$$\begin{aligned} S &\rightarrow S_1|S_2 \\ S_1 &\rightarrow 0S_1 1|\lambda \\ S_2 &\rightarrow 1S_2 0|\lambda \end{aligned}$$

Conversion of a DFA to a CFG

Easy to construct a CFG when the language is **regular** (and we can construct a DFA for that language).

Conversion of a DFA into an equivalent CFG:

1. Make a variable R_i for each state q_i of the DFA.
2. Add the rule
$$R_i \rightarrow aR_j$$
to the CFG if $\delta(q_i, a) = q_j$ is a valid transition in the DFA.
3. Add the rule
$$R_i \rightarrow \lambda$$
if q_i is an accept state of the DFA.
4. Make R_0 the start variable of the grammar (corresponding to q_0 which is the start state of the machine).

Conversion of a DFA to a CFG

- Certain CFLs contain strings that themselves contain substrings that are somehow "linked".
- For example, the language $\{0^n 1^n | n \geq 0\}$. Then we can create a rule of the form

$$R \rightarrow uRv$$

  so that we can keep track of the u's and the v's simultaneously.
- Finally, we can have more complex languages, in which the strings may contain certain structures that appear recursively as part of other (or the same) structures.

Ambiguity

Definition 4 (Ambiguous grammar)

A string w is derived **ambiguously** in context-free grammar G if it has two or more different leftmost derivations.
Grammar G is **ambiguous** if it generates some string ambiguously.

A derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost remaining variable is the one replaced.

Remarks:

- Some ambiguous grammars can be replaced by unambiguous grammars that generate the same language.
- Some CFLs, however, can be described **only** by ambiguous grammars. Such languages are called *inherently ambiguous*.

Chomsky Normal Form

Definition 5 (Chomsky normal form)

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

$$\begin{aligned} A &\rightarrow B\,C \\ A &\rightarrow a \end{aligned}$$

where a is any terminal and A, B, C are any variables—except that B and C may not be the start variable.
In addition, we permit the rule

$$S \rightarrow \lambda$$

where S is the start variable.

Chomsky Normal Form

Theorem 6

Any CFL is generated by a CFG in Chomsky normal form.

Proof idea.

Rules that violate the conditions are replaced with equivalent ones that are satisfactory.

1. Add a new start variable.
2. Eliminate all λ-**rules** of the form $A \to \lambda$.
3. Eliminate all **unit rules** of the form $A \to B$.
4. Patch up the grammar to be sure that it still generates the same language.
5. Convert the remaining rules into the proper form.

□

Chomsky Normal Form

Proof.

1. Add a new start variable S_0 and the rule $S_0 \to S$, where S was the original start variable $\Rightarrow$
   Guarantee no S_0 on the RHS (right-hand side) of a rule.
2. Remove an λ-rule $A \to \lambda$ (where A is not a start variable).
   For each rule where A occurs on the RHS of a rule, add a new rule with that occurrence deleted.

Example

Deleting 1st rule from: $\begin{cases} A & \to & \lambda \\ R & \to & uAvAw \end{cases}$ we get: $\begin{cases} R & \to & uvAw \\ R & \to & uAvw \\ R & \to & uvw \end{cases}$

Remark: When eliminating $A \to \lambda$ for a rule $R \to A$, we substitute it with $R \to \lambda$ (unless we had previously removed $R \to \lambda$).

Chomsky Normal Form

Proof (cont).

3. Handle all unit rules.
   Remove a unit rule $A \to B$ and for each $B \to u$ rule (u: string of variables and terminals), add the rule $A \to u$ (unless this was a unit rule previously removed).
   Repeat until we eliminate all unit rules.
4. Convert all remaining rules into the proper form.
   For each rule $A \to u_1 u_2 \cdots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, write instead (i.e., replace it with):

$$\begin{cases} A & \to & u_1 A_1 \\ A_1 & \to & u_2 A_2 \\ & \vdots & \\ A_{k-2} & \to & u_{k-1} u_k \end{cases}$$

   The A_i's are all new variables. If $k = 2$ we replace any terminal u_i in the preceding rule(s) with the new variable U_i and add the rule $U_i \to u_i$. □

Example

Example 7 (Conversion of CFG to Chomsky normal form)

Given

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \lambda \end{aligned}$$

convert it to Chomsky normal form by using the conversion procedure just given.

Example (cont.)

Solution

Step 1: Make a new start variable

$$\begin{aligned} \mathbf{S_0} &\rightarrow \mathbf{S} \\ S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \lambda \end{aligned}$$

Example (cont.)

Solution (cont.)

Step 2: Eliminate λ-rules
(remove $B \to \lambda$)

$$\begin{array}{lcl} S_0 & \to & S \\ S & \to & ASA \mid aB \mid \mathbf{a} \quad (\text{introduced } \mathbf{a}) \\ A & \to & B \mid S \mid \lambda \quad (\text{introduced } \lambda) \\ B & \to & b \end{array}$$

(remove $A \to \lambda$)

$$\begin{array}{lcl} S_0 & \to & S \\ S & \to & ASA \mid aB \mid a \mid \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S} \quad (\text{introduced } \mathbf{SA}, \mathbf{AS}, \mathbf{S}) \\ A & \to & B \mid S \\ B & \to & b \end{array}$$

Example (cont.)

Solution (cont.)

Step 3: Remove unit rules
(remove $S \to S$)

$$\begin{array}{lcll} S_0 & \to & S & \\ S & \to & ASA \mid aB \mid a \mid SA \mid AS & (\text{removed } S) \\ A & \to & B \mid S & \\ B & \to & b & \end{array}$$

(remove $S_0 \to S$)

$$\begin{array}{lcll} S_0 & \to & ASA \mid aB \mid a \mid SA \mid AS & (\text{removed } \mathbf{S}) \\ S & \to & ASA \mid aB \mid a \mid \mathbf{SA} \mid \mathbf{AS} & \\ A & \to & B \mid S & \\ B & \to & b & \end{array}$$

Example (cont.)

Solution (cont.)

Step 3: Remove unit rules (cont.)
(remove $A \to B$)

$$\begin{aligned} S_0 &\to ASA \mid aB \mid a \mid SA \mid AS \\ S &\to ASA \mid aB \mid a \mid SA \mid AS \\ A &\to b \mid S \quad (\text{removed } \mathbf{B}) \\ B &\to b \end{aligned}$$

(remove $A \to S$)

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

Example (cont.)

Solution (cont.)

Step 4: Final Simplifications

$$
\begin{array}{rcl}
S_0 & \rightarrow & A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS \\
S & \rightarrow & A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS \\
A & \rightarrow & b \mid A\mathbf{A_1} \mid \mathbf{U}B \mid a \mid SA \mid AS \\
B & \rightarrow & b \\
A_1 & \rightarrow & SA \quad \text{(introduced)} \\
U & \rightarrow & a \quad \text{(introduced)}
\end{array}
$$

Remark: Technically we should have had instead of a single U, several U_is leading to a. All these are now combined and the grammar is simpler.

Regular Grammars

Definition 8

A grammar $G = (V, T, S, P)$ is said to be **right-linear** if all productions are of the form

$$\begin{aligned} A &\rightarrow xB \\ \text{or } A &\rightarrow x \end{aligned},$$

where $A, B \in V$ and $x \in T^*$. A grammar is said to be **left-linear** if all productions are of the form

$$\begin{aligned} A &\rightarrow Bx \\ \text{or } A &\rightarrow x \end{aligned},$$

Definition 9 (Regular Grammar)

A **regular grammar** is a grammar that is either right-linear or left-linear.

Linear Grammars

Definition 10 (Linear Grammar)

A **linear grammar** is a grammar in which at most one variable can occur on the right side of any production, without restriction on the position of the variable.

A regular grammar is always linear, but not all linear grammars are regular.

Example 11 (Linear but not regular)

$$G \begin{cases} S & \rightarrow & A \\ A & \rightarrow & aB \mid \lambda \\ B & \rightarrow & Ab \end{cases}$$

Remark 1

Every regular language has a regular grammar (we saw a construction with a right-linear grammar). The equivalence also holds for left-linear grammars.

Table of Contents

Pushdown Automata (PDAs)

PDAs: Similar to NFAs but also have a **stack** (which serves as memory)

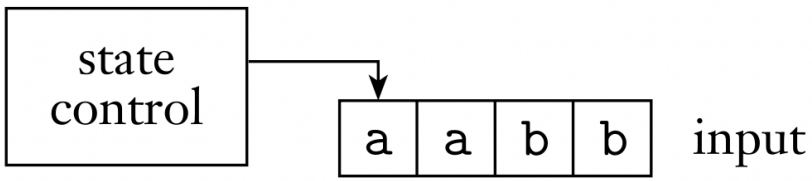
- Will be able to recognize some nonregular languages
- Equivalent in power to context-free Grammars

Proving that a language is context-free:

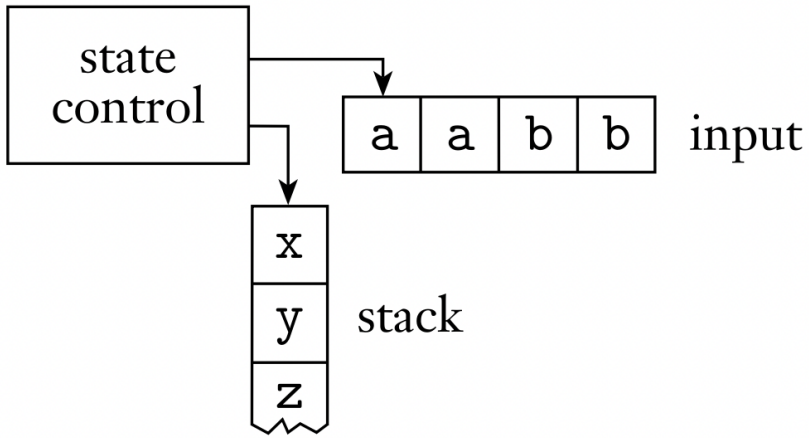
- Either give a context-free grammar generating the language, or
- give a push-down automaton (PDA) recognizing it

(use whichever is easier!)

Finite Automaton vs PDA Representation



finite automaton



PDA

Pushdown Automata

- Push/pop operations for interacting with the stack.
  (Of course, interaction only takes place with top of stack, as we can only read the symbol at the top of the stack.)
- Discuss a simple algorithm to recognize the language $\{0^n 1^n | n \geq 0\}$, by using a stack.
- Nondeterministic PDAs can recognize certain languages that no deterministic PDA can recognize!
- Only the Nondeterministic PDAs are equivalent in power to CFGs.
  (So we focus on Nondeterministic PDAs.)

Formal Definition of a PDA

- We have both an input alphabet Σ and a stack alphabet Γ.
- Transition function δ: $Q \times \Sigma_\lambda \times \Gamma_\lambda \to \mathcal{P}(Q \times \Gamma_\lambda)$.

Definition 12 (Pushdown automaton)

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. $\delta : Q \times \Sigma_\lambda \times \Gamma_\lambda \to \mathcal{P}(Q \times \Gamma_\lambda)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

How a PDA computes

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA.
M **accepts** input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\lambda$ and a sequence of states $r_0, r_1, \ldots, r_m \in Q$ and **strings** $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions.

1. $r_0 = q_0$ and $s_0 = \lambda$.
   (This condition signifies that M starts out properly, in the start state q_0 and with an empty stack.)
2. For $i = 0, \ldots, m-1$, 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 $a, b \in \Gamma_\lambda$ 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.)

Examples of Pushdown Automata

Example 13 (PDA construction)

Describe formally the PDA that recognizes the language $\{0^n 1^n | n \geq 0\}$.

Examples of Pushdown Automata

Solution

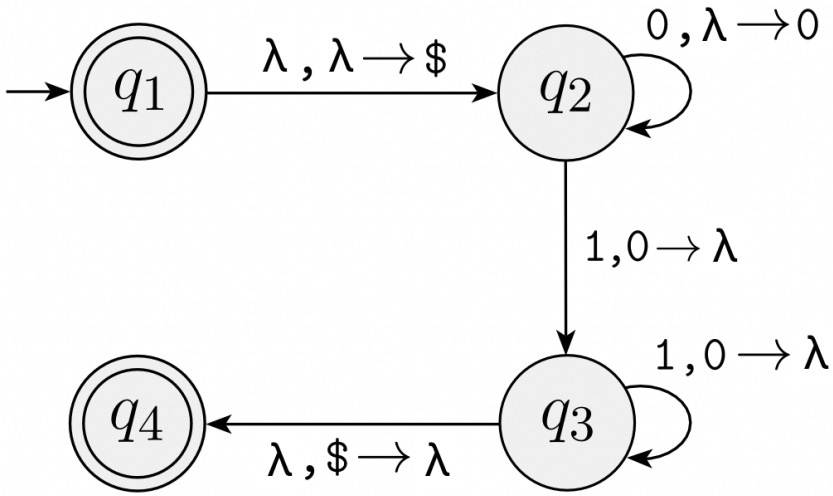
Define $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, F)$, where
$Q = \{q_1, q_2, q_3, q_4\}$,
$\Sigma = \{0, 1\}$,
$\Gamma = \{0, \$\}$,
$F = \{q_1, q_4\}$, and
δ is given by the following table, wherein blank entries signify $\emptyset$.

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

Example (cont.)

Solution (cont.)

State diagram for the PDA M_1 that recognizes $\{0^n 1^n | n \geq 0\}$.



Example (cont.)

Solution (cont.)

Labels:

- "$a, b \to c$" 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 a c. Any of a, b, and c may be λ.
- If $a = \lambda$, the machine may make this transition without reading any symbol from the input.
- If $b = \lambda$, the machine may make this transition without reading and popping any symbol from the stack.
- If $c = \lambda$, the machine does not write any symbol on the stack when going along this transition.

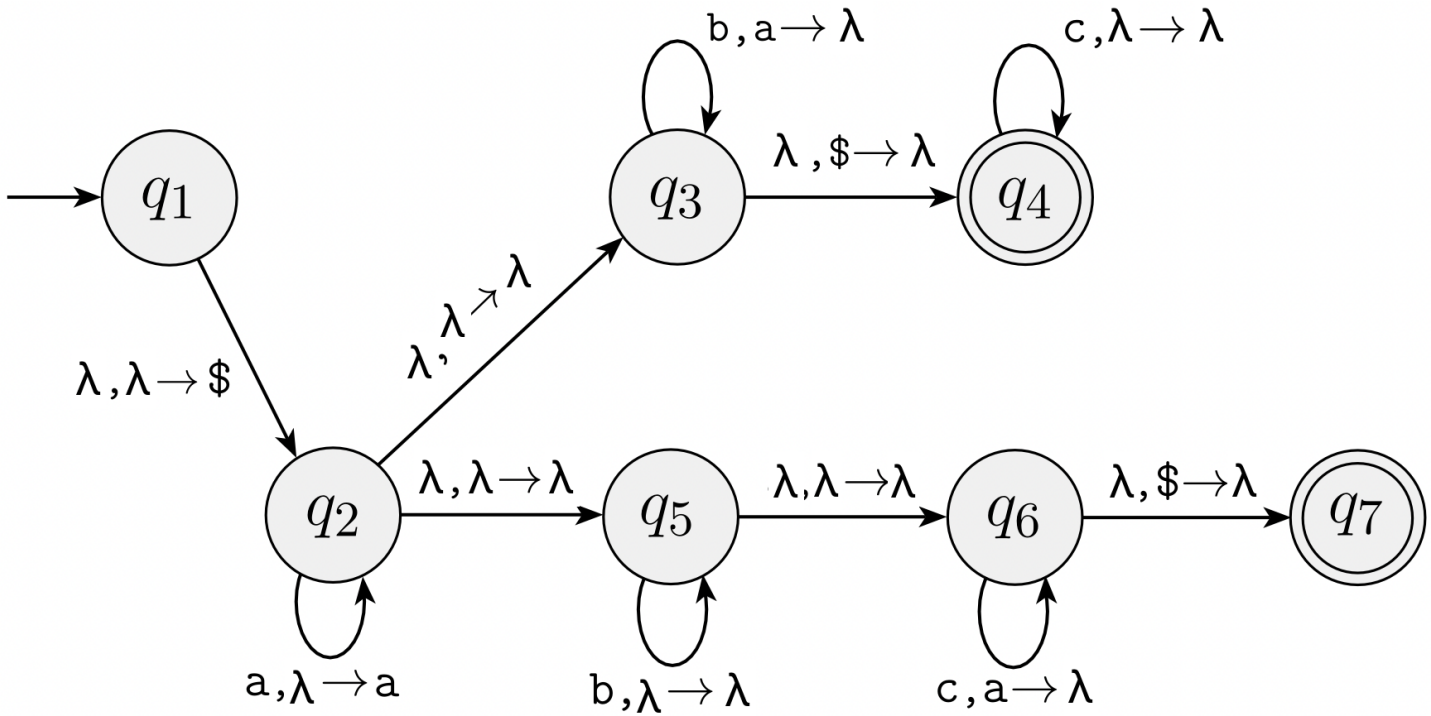
Example

Example 14 (PDA construction)

Design a PDA that recognizes the language $\{a^i b^j c^k | i, j, k \geq 0 \text{ and } (i = j \text{ or } i = k)\}$.

Example (cont.)

Solution



Example

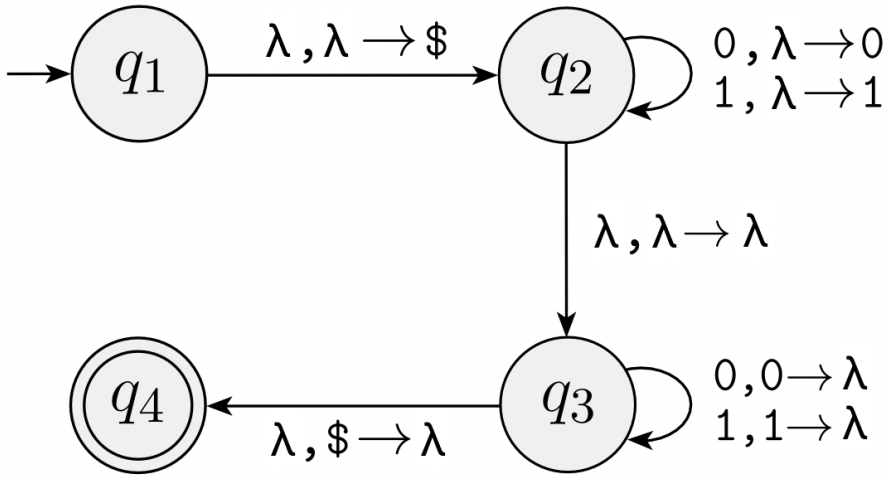
Example 15 (PDA construction)

Design a PDA recognizing the language $\{ww^R | w \in \{0,1\}^*\}$.
Recall that w^R is w written backwards.

Example (cont.)

Solution

Idea: Nondeterministically guess the midpoint.



Equivalence of PDAs with Context-Free Grammars

Theorem 16

A language is context free if and only if some PDA recognizes it.

We will prove both directions separately.

Lemma 17

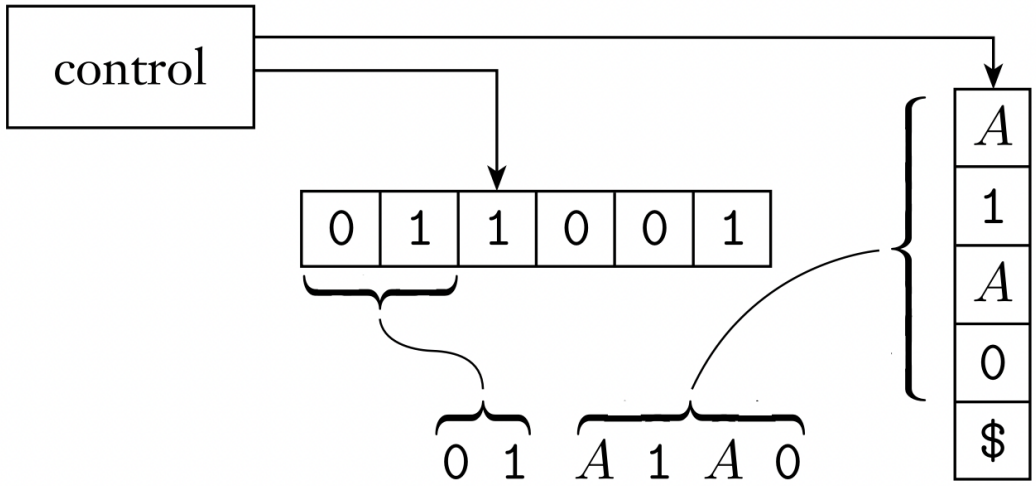
If a language is context free, then some PDA recognizes it.

Equivalence of PDAs with Context-Free Grammars

Proof idea.

The PDA P tries to 'guess' a derivation from the grammar G that gives the input w.

So that we can use the stack in this direction, we match terminals as they are generated or read from the top of the stack, and only store part of the intermediate string.



P representing the intermediate string $01A1A0$

Equivalence of PDAs with Context-Free Grammars

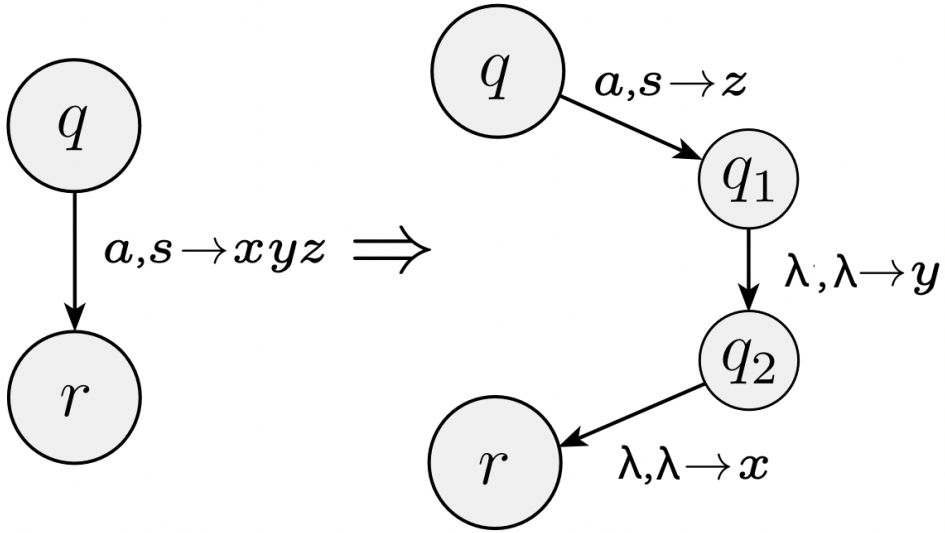
Informal description of P:

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a **variable** A, nondeterministically select one of the rules for A and substitute A with the RHS of that rule.
   b. 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.
   Otherwise, reject this branch of nondeterminism.
   c. If the top of stack is the symbol \$, enter the accept state.
   (Doing so accepts the input if it has all been read.)

Equivalence of PDAs with Context-Free Grammars

Proof.

We will allow the transition function to write an entire string on the stack. We can simulate such an action with intermediate states. Say the PDA goes from q to r reading symbol a at the input, popping s from the stack:



We will denote this as $(r, xyz) \in \delta(q, a, s)$.

Equivalence of PDAs with Context-Free Grammars

Proof (cont.)

- States of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, 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}.
- Specifying the transition function:
  Step (1):

$$\delta(q_{\text{start}}, \lambda, \lambda) = \{(q_{\text{loop}}, S\$)\}$$

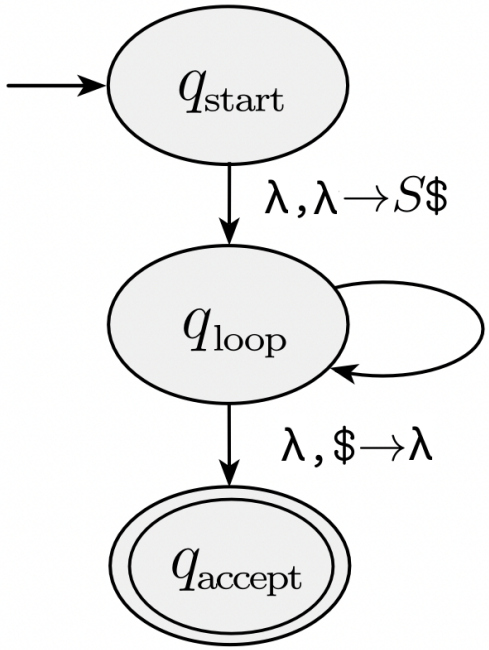
  Step (2):

$$(a) \quad \delta(q_{\text{loop}}, \lambda, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}.$$

$$(b) \quad \delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \lambda)\}.$$

$$(c) \quad \delta(q_{\text{loop}}, \lambda, \$) = \{(q_{\text{accept}}, \lambda)\}.$$

Equivalence of PDAs with Context-Free Grammars



$\lambda, A \to w$ for rule $A \to w$

$\mathtt{a}, \mathtt{a} \to \lambda$ for terminal $\mathtt{a}$

State diagram of P

Example

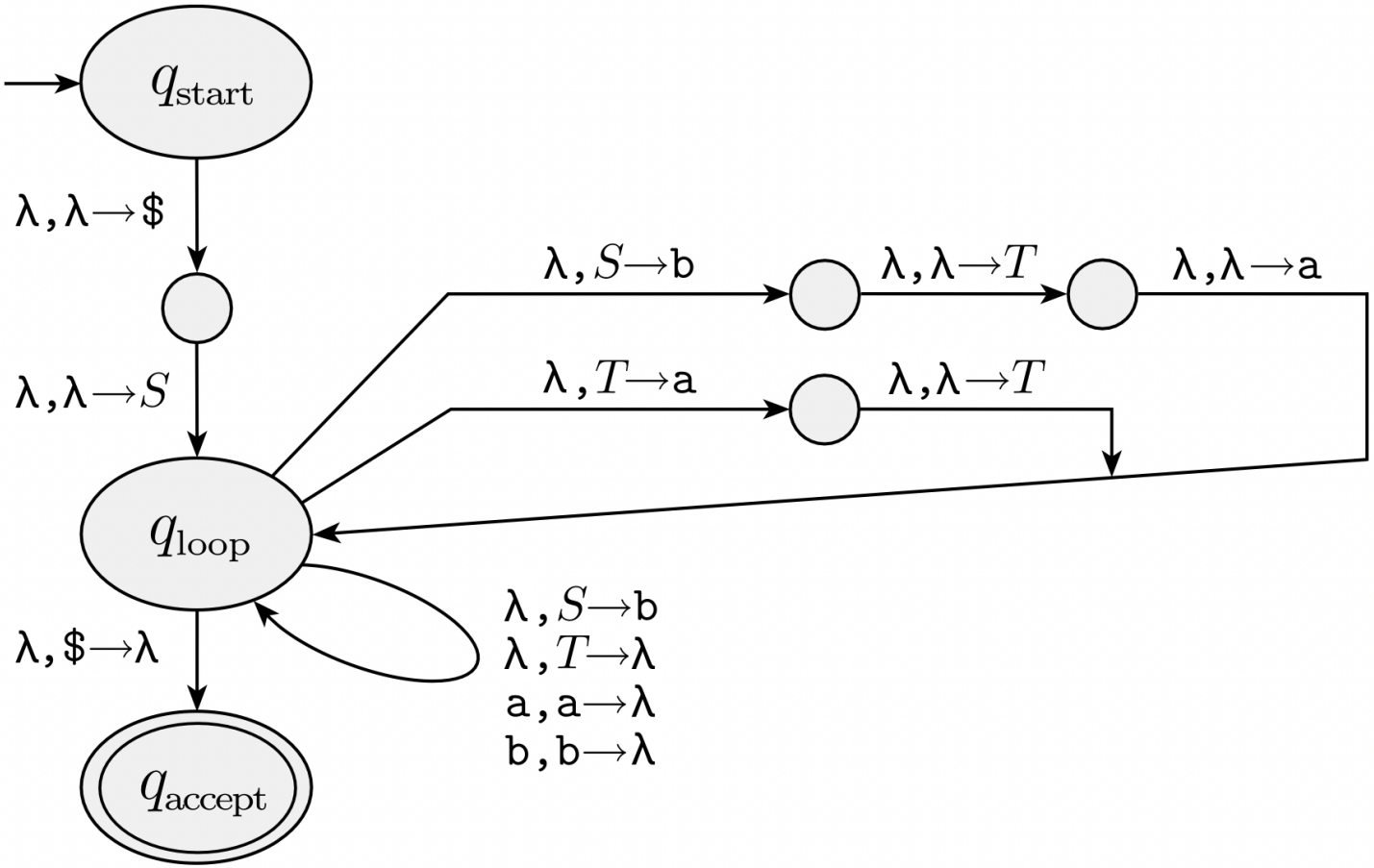
Example 18 (PDA construction)

Construct a PDA from the following context-free grammar G.

$$\begin{aligned} S &\rightarrow aTb \mid b \\ T &\rightarrow Ta \mid \lambda \end{aligned}$$

Example (cont.)

Solution



State diagram of the PDA

Equivalence of PDAs with Context-Free Grammars

Now we prove the reverse direction of the Theorem.

Lemma 19

If a PDA recognizes some language, then it is context free.

Proof idea.

(It is trickier because 'programming' a grammar is not as easy as 'programming' an automaton.)

- 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. (I.e., leave the stack intact during this transition.)

Equivalence of PDAs with Context-Free Grammars

Proof idea (cont.)

- Modify 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.

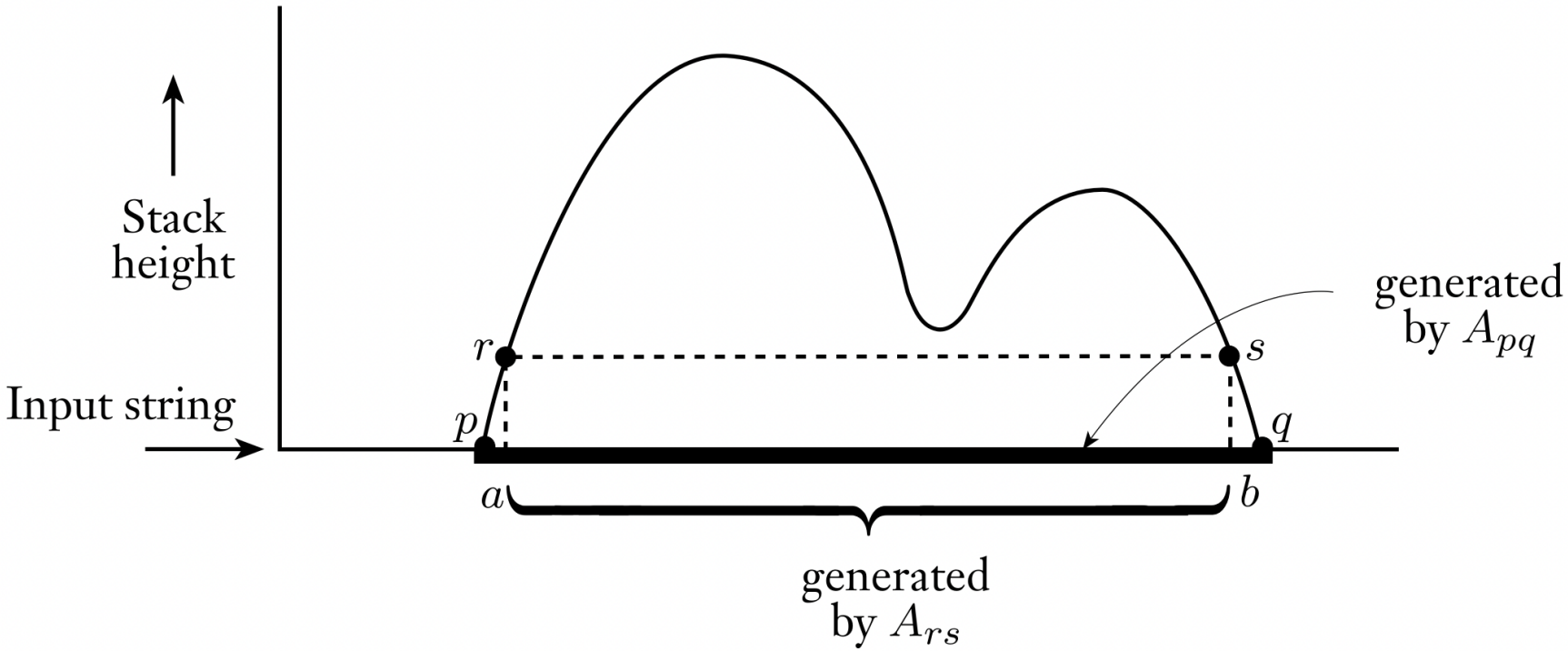
□

Remark:
Giving P features (1) and (2) is easy.
For (3) a simultaneous pop-push goes through an intermediate state.
Similarly, for (3) a transition that neither pops nor pushes can be simulated with an arbitrary push-pop (in 2 steps).

Equivalence of PDAs with Context-Free Grammars

PDA's P computation on string x:

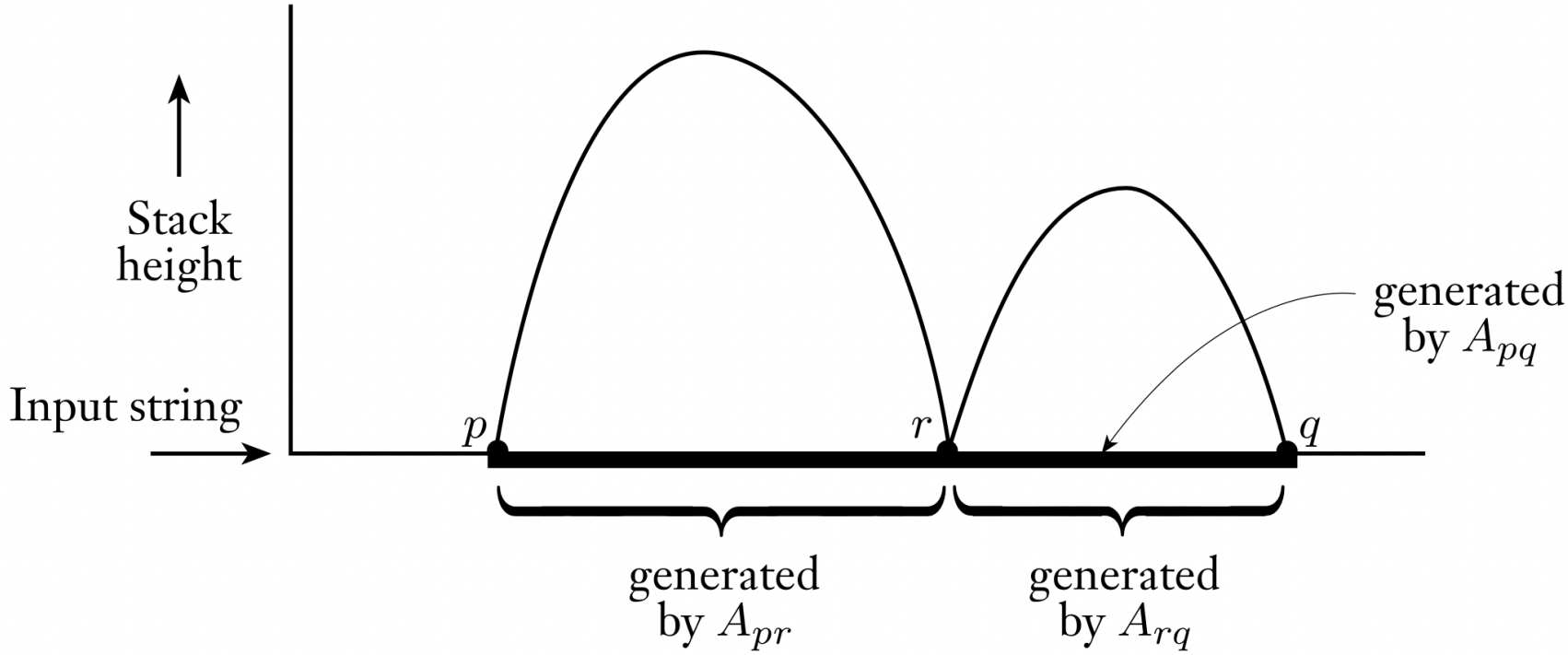
- Either the symbol popped at the end is the symbol that was pushed at the beginning:



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

Equivalence of PDAs with Context-Free Grammars

- or not:



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

Equivalence of PDAs with Context-Free Grammars

Proof.

Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G.
The variables of G are $\{A_{pq} | p, q \in Q\}$.
The start variable is $A_{q_0}, q_{\text{accept}}$.
The grammar G has the following rules:

- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\lambda$,
  if $\delta(p, a, \lambda)$ contains (r, t) and $\delta(s, b, t)$ contains (q, λ),
  put the rule $A_{pq} \to aA_{rs}b$ in G.
- For each $p, q, r \in Q$,
  put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.
- Finally, for each $p \in Q$,
  put the rule $A_{pp} \to \lambda$ in G.

Equivalence of PDAs with Context-Free Grammars

Claim

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

We prove this claim by induction on the number of steps in the derivation of x from A_{pq}.

Proof.

Basis: The derivation has 1 step.
Then the RHS of the rule contains no variables.
$\Rightarrow$ The only possibility is $A_{pp} \rightarrow \lambda$.
$\Rightarrow$ input λ takes P from p with empty stack to p with empty stack, so the basis is proved.

Equivalence of PDAs with Context-Free Grammars

Proof (cont.)

Induction Hypothesis: Assume the claim is true for derivations of length at most k, where $k \geq 1$ (and prove true for derivations of length $k+1$).

Induction Step: Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ in $(k+1)$ steps.
The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$.
We handle these two cases separately.

- The first step is $A_{pq} \Rightarrow aA_{rs}b$.

Consider the portion y of x generated by A_{rs}, so that $x = ayb$.
Because $A_{rs} \stackrel{*}{\Rightarrow} y$ in k steps, the induction hypothesis tells us that P can go from r with empty stack to s with empty stack.
Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G, $\delta(p, a, \lambda)$ contains (r, t) and $\delta(s, b, t)$ contains (q, λ), for some stack symbol t.
So P can go from p with empty stack to q with empty stack.

Equivalence of PDAs with Context-Free Grammars

Proof (cont.)

- The first step is $A_{pq} \Rightarrow A_{pr}A_{rq}$.

Now x is written as $x = yz$ where $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps each one of them.
So, by the induction hypothesis it follows that P can go from p with empty stack to r with empty stack, and from r with empty stack to s with empty stack.

Either way, we have proved the claim. □

Equivalence of PDAs with Context-Free Grammars

Claim

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

(Proof by induction on the number of steps in the computation.)

Equivalence of PDAs with Context-Free Grammars

Proof.

Basis: The computation has 0 steps.
If a computation has 0 steps, it starts and ends at the same state — say, p.
So we must show that $A_{pp} \stackrel{*}{\Rightarrow} x$.
In 0 steps, P only has time to read the empty string, so $x = \lambda$.
By construction, G has the rule $A_{pp} \to \lambda$, so the basis is proved.

Induction hypothesis: Assume that for computations of length at most k, where $k \geq 0$, it holds that if x can bring P from p with empty stack to q with empty stack, then A_{pq} generates x.

Induction step: Look at computations of length $k + 1$.
Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

Equivalence of PDAs with Context-Free Grammars

Proof (cont.)

• If the stack is empty only at the beginning and end of this computation, the symbol, say t, that was pushed at the first move, is the one popped at the last move.

Let a be the symbol read in the first move, b the symbol read in the last move, r the state after the first move, and s the state before the last move.

Then $(r, t) \in \delta(p, a, \lambda)$ and $(q, \lambda) \in \delta(s, b, t)$.

So the rule $A_{pq} \to aA_{rs}b$ is in G.

Let $x = ayb$. Computing y has $(k + 1) - 2 = k - 1$ steps.

Thus, by the induction hypothesis, $A_{rs} \stackrel{*}{\Rightarrow} y$. Hence $A_{pq} \stackrel{*}{\Rightarrow} x$.

• If the stack is empty in between, say in state r,

Then the portions of the computation from p to r and from r to q each contain at most k steps.

By the induction hypothesis, $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$. Because rule $A_{pq} \to A_{pr}A_{rq}$ is in G, $A_{pq} \stackrel{*}{\Rightarrow} x$, and the proof is complete. □

Equivalence of PDAs with Context-Free Grammars

Corollary 20

Every regular language is context-free.

Proof.

Every regular language is recognized by a finite automaton.
Every finite automaton is automatically a pushdown automaton that simply ignores its stack.
Pushdown automata recognize the class of context-free languages. □

Equivalence of PDAs with Context-Free Grammars

Therefore:

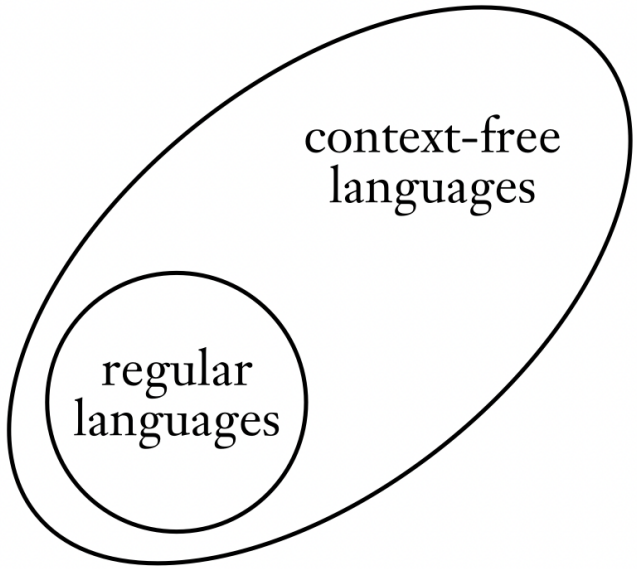


Table of Contents

Non-Context-Free Languages

A more involved pumping lemma follows.

Theorem 21 (Pumping lemma for context-free languages)

If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces

$$s = uvxyz$$

satisfying the conditions:

1. *for each $i \geq 0$, $uv^i xy^i z \in A$,*
2. *$|vy| > 0$, and*
3. *$|vxy| \leq p$.*

Condition (2) means that either v or y is $\neq \lambda$. Otherwise the theorem would be trivially true.

Non-Context-Free Languages

Proof idea.

Let $s \in A$, so that s is 'sufficiently long'. Since $s \in A$, s has a parse tree because it is derivable from G.
The idea is that the parse tree is tall because s is very long.
So the parse tree must contain some long path from the root to some terminal symbol (some leaf).
On this long path, some variable symbol R must be repeating (by pigeonhole principle).

Non-Context-Free Languages

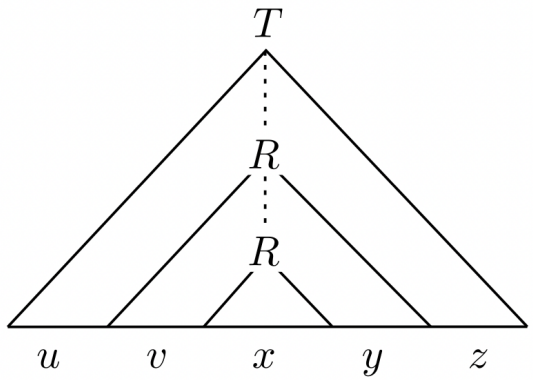
Proof idea (cont.)

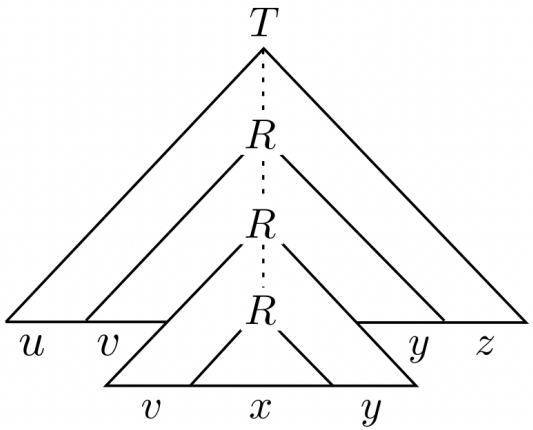
As the following figure shows, we can replace the subtree under the second occurrence of R with the subtree under the first occurrence of R and still get a legal parse tree.

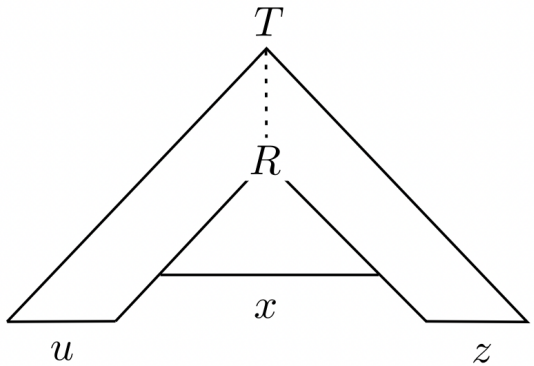
Therefore, we may cut s into five pieces $uvxyz$ as the figure indicates, and we may repeat the 2nd and 4th pieces and obtain a string still in the language. In other words, $uv^ixy^iz \in A$ for any $i \geq 0$.

Non-Context-Free Languages

Proof idea (cont.)







Non-Context-Free Languages

Proof.

Let G be a CFG for CFL A. Let b be the maximum number of symbols (variables or terminals) in the RHS of a rule (assume $b \geq 2$).
In any parse tree using G, a node has $\leq b$ children.
So, at most b leaves are 1 step from the start variable;
at most b^2 leaves are within 2 steps of the start variable; etc.
In other words, if a generated string is at least l long, then the parse tree must be at least $\lceil \log_b(l) \rceil$ tall.
Note: $b^{h-1} + 1 \leq |w| \leq b^h \Rightarrow b^{h-1} \leq |w| \leq b^h \Rightarrow h - 1 < \log_b(|w|) \leq h$

Non-Context-Free Languages

Proof (cont.)

- Set the pumping length p to be

$$p = b^{|V|+1}$$

where $|V|$ is the number of variables in G.

- If s has several parse trees, pick the parse tree that has the smallest number of nodes.

If s is a string such that $|s| \geq p$, then the parse tree for s be at least $|V| + 1$ high, so there is a path of length at least $|V| + 1$.

That path has at least $|V| + 2$ nodes (one at a terminal, the others at variables).

Hence that path has $\geq |V| + 1$ variables.

Due to pigeonhole principle, some variable R appears more than once. For convenience later, pick R among the lowest $|V| + 1$ variables on this path.

Non-Context-Free Languages

Proof (cont.)

• Divide s into $uvxyz$ according to Figure above.

The upper occurrence of R has a larger subtree and generates vxy, whereas the lower occurrence generates just x with a smaller subtree. Both of these subtrees are generated by the same variable (R), so we may substitute one for the other and still obtain a valid parse tree.

Replacing the smaller by the larger repeatedly gives parse trees for the strings uv^ixy^iz at each $i > 1$. (Condition 1)

• v and y can not be simultaneously λ. (We selected the parse tree that has the fewest number of nodes generating s.) (Condition 2)

• Condition 3 ($|vxy| \leq p$) is justified since we looked at the last $|V| + 1$ variables on the path. □

Examples

Example 22

Show that the language $B = \{a^n b^n c^n | n \geq 0\}$ is not context-free, using the pumping lemma.

Examples

Solution

Towards contradiction, assume that B is a context-free language.
Let p be the pumping length for B (that is guaranteed to exist by the pumping lemma). Select the string

$$s = a^p b^p c^p$$

to show that one of the conditions will be violated, no matter how we split s into $uvxyz$.
By Condition 2, either v or y is nonempty.

Examples

Solution (cont.)

Case I: Both v and y contain only one type of alphabet symbol.
Then, in the string

$$uv^2xy^2z$$

we are increasing the occurrences of at least one symbol, but certainly not all three simultaneously.
So, $uv^2xy^2z \notin B$.

Case II: One of v or y contains at least two types of symbols.
Then, in the string

$$uv^2xy^2z$$

some symbols are out of order.
So, again, $uv^2xy^2z \notin B$.

Examples

Example 23

Let $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$.
Use the pumping lemma to show that C is not a CFL.

Examples

Solution

Towards contradiction, assume that C is context-free.
Let p be the pumping length given by the pumping lemma.
Again use the string

$$s = a^p b^p c^p.$$

We split s again into $uvxyz$ and distinguish two cases.

Case I: Both v and y contain only one type of alphabet symbol. Then some symbol among a, b, c is missing from both. [Note that the reasoning used in example 22, case 1, does not apply immediately.]
We further subdivide this case into three subcases according to which symbol does not appear.

Examples

Solution (cont.)

- a's do not appear. Then we try pumping down to obtain the string

$$uv^0xy^0z = uxz.$$

That contains the same number of as as s does, but fewer b's or c's (or both). Therefore, $uxz \notin C$, and a contradiction occurs.

- b's do not appear. Then either a's or c's must appear in v or y because both can not be λ.
  - If a's appear, the string $uv2xy2z \notin C$ since it contains more a's than b's.
  - If c's appear, the string $uv0xy0z \notin C$ since it contains more b's than c's.

So, either way contradiction.

Examples

Solution (cont.)

• c's do not appear. Then $uv^2xy^2z \notin C$ since it contains more a's or more b's than c's, and a contradiction occurs.

Case II: When either v or y contains more than one type of symbol, then $uv^2xy^2z \notin C$ since it will not contain the symbols in the correct order, and a contradiction occurs.

Examples

Example 24

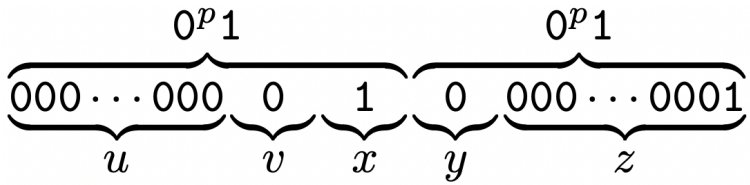
Let $D = \{ww | w \in \{0,1\}^*\}$.
Use the pumping lemma to show that D is not a CFL.

Examples

Solution

Towards contradiction, assume that D is a CFL. Let p be the pumping length given by the pumping lemma.
This time choosing string s is less obvious.

Candidate 1: 0^p10^p1.
This string can be pumped with the decomposition shown below, so it is not adequate for our purposes.



Examples

Solution (cont.)

Candidate 2: $0^p 1^p 0^p 1^p$.

Divide s into $uvxyz$, where $|vxy| \leq p$.

- If vxy is to the left of the midpoint of s, then

$$uv^2xy^2z$$

results in having at least one 1 to the right of the midpoint. But then $uv^2xy^2z \notin D$ (starts with 0 and after midpoint starts with 1).

- If vxy is to the right of the midpoint of s, then

$$uv^2xy^2z$$

moves a 0 into the last position of the first half and so again $uv^2xy^2z \notin D$

Examples

Solution (cont.)

- If vxy contains the midpoint of s, pumping down to

$$uxz$$

we have the form $0^p 1^i 0^j 1^p$, where i and j cannot both be p.
So again a contradiction, since $uxz \notin D$.