

# Equivalence of Pushdown Automata with Context-Free Grammar

Teodor Rus

`rus@cs.uiowa.edu`

The University of Iowa, Department of Computer Science

# Motivation

- CFG and PDA are equivalent in power:  
a CFG generates a context-free language, and,  
a PDA recognizes a context-free language.
- We show here how to convert a CFG into a PDA that recognizes the language specified by the CFG and vice versa.

**Application:** this equivalence allows to:

1. specify programming languages using CFG-s, and
2. implement compilers using equivalent PDA-s.

# Theorem 2.20

A language is context-free iff some pushdown automaton recognizes it.

**Note:** This means that:

1. if a language  $L$  is context-free then there is a PDA  $M_L$  that recognizes it;
2. if a language  $L$  is recognized by a PDA  $M_L$  then there is a CFG  $G_L$  that generates  $L$ .

# Lemma 2.21

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

## Proof idea:

1. Let  $A$  be a CFL. From the definition we know that  $A$  has a CFG  $G$ , that generates it;
2. We will show how to convert  $G$  into a PDA  $P$  that accepts strings  $w$  if  $G$  generates  $w$ ;
3.  $P$  will work by determining a derivation of  $w$ .

# What is a derivation?

**Recall:** For  $G = (V, \Sigma, R, S)$ ,  $w \in L(G)$ :

- A derivation of  $w$  is a sequence of substitutions

$$S \xRightarrow{S \rightarrow \alpha_1} w_1 \xRightarrow{A \rightarrow \alpha_2} \dots \xRightarrow{B \rightarrow \alpha_k} w, S \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, B \rightarrow \alpha_k \in R,$$

where  $S, A, \dots, B$  are not necessarily distinguished;

- Each step in the derivation yields an *intermediate string* of variables and terminals;
- Hence,  $P$  will determine whether some series of substitutions using rules in  $R$  can lead from start variable  $S$  to  $w$ .

# Difficulties expected

- How should we figure out which substitution to make? Nondeterminism allows  $P$  to guess:  
At each step of the derivation one of the rules for a particular variable is selected nondeterministically.
- How does  $P$  starts?  
 $P$  begins by writing the start variable on the stack and then continues working this string.

# How does $P$ terminate?

If while consuming  $w$ ,  $P$  arrives at a string of terminals that equals  $w$  then accept; otherwise reject!

# More questions

- The initial string (the start variable) is on the stack. How does  $P$  store the other intermediate strings?
- Using the stack doesn't quite work because the PDA needs to find the variables in the intermediate string and make substitutions.

**Note:** stack does not support this because only the top is accessible!

# The way around

- $P$  reconstructs the leftmost derivation of  $w$ ;
- $P$  keeps only part of the intermediate string on the stack starting with the first variable in the intermediate string;
- Any terminal symbol appearing before the first variable is matched with symbols in the input;

An example of graphic image of  $P$  is in Figure 1.

# An intermediate string

Assume that  $S \xRightarrow{*} 01A1A0$ .

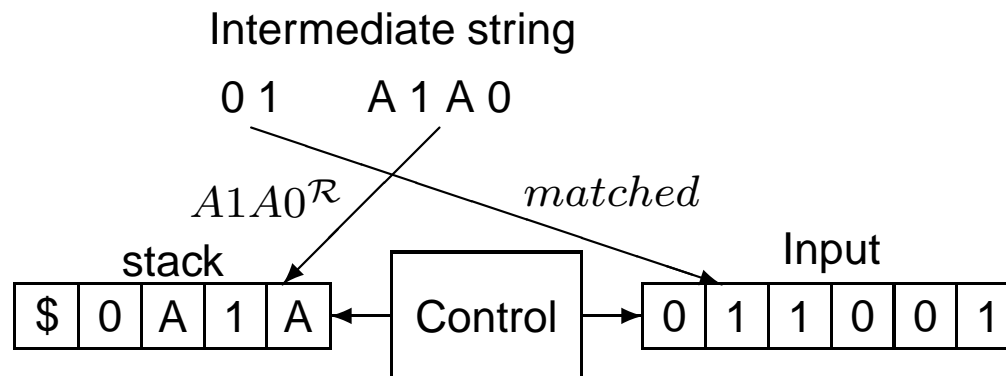


Figure 1:  $P$  representing intermediate string  $01A1A0$

# Informal description of $P$

- Place the marker symbol  $\$$  and the start variable on the stack;
  - **Repeat**
    1. If the top of the stack is a variable symbol  $A$ , nondeterministically select a rule  $r$  such that  $lhs(r) = A$  and replace  $A$  by the string  $rhs(r)$ ;
    2. If the top of the stack is a terminal symbol,  $a$ , read the next input symbol and compare it with  $a$ . If they match pop the stack; if they don't match reject on this branch of nondeterminism;
    3. If the top of the stack is the symbol  $\$$ , enter the accept state:  
**accept state:** *if all text has been read accept, otherwise reject.*
- until accept or reject.**

# Proof of lemma 2.21

Now we can give formal details of the construction of the PDA  $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$ .

- First we introduce an appropriate notation for transition function that provides a way to write an entire string  $rhs(r)$  on the stack in one step of the machine;
- **Simulation:** this action can be simulated by introducing additional states to write the string symbol by symbol.

# Formal construction

- Let  $q, r \in Q$ ,  $a \in \Sigma_\epsilon$  and  $s \in \Gamma_\epsilon$ .
- Assume that we want  $P$  to go from  $q$  to  $r$  when it reads  $a$  and pops  $s$
- In addition, we want  $P$  to push on the stack the string  $u = u_1 \dots u_k$  at the same time

# Implementation

This construction can be implemented by introducing the new states  $q_1, \dots, q_{k-1}$  and setting transition function as follows:

# Setting $\delta(q, a, s)$ :

$$\begin{aligned}(q_1, u_k) &\in \delta(q, a, s), \\ \delta(q_1, \epsilon, \epsilon) &= \{(q_2, u_{k-1})\}, \\ \delta(q_2, \epsilon, \epsilon) &= \{(q_3, u_{k-2})\}, \\ &\dots \quad \dots \\ \delta(q_{k-1}, \epsilon, \epsilon) &= \{(r, u_1)\}\end{aligned}$$

**Note:** transitions that push  $u_1 u_2 \dots u_k$  on the stack operate on the reverse of  $u$ . Why?

# Notation

$(r, u) \in \delta(q, a, s)$  means that when  $P$  is in state  $q$ ,  $a$  is the next input symbol, and  $s$  is the symbol on top of the stacks,  $P$ :

1. reads  $a$ ,
2. pop  $s$ ,
3. pushes  $u$  on the stack,

and then goes to state  $r$ , as seen in Figure 2.

**Hence:**  $(r, u) \in \delta(q, a, s)$  is equivalent with  $q \xrightarrow{a, s \rightarrow u} r$

# Graphic

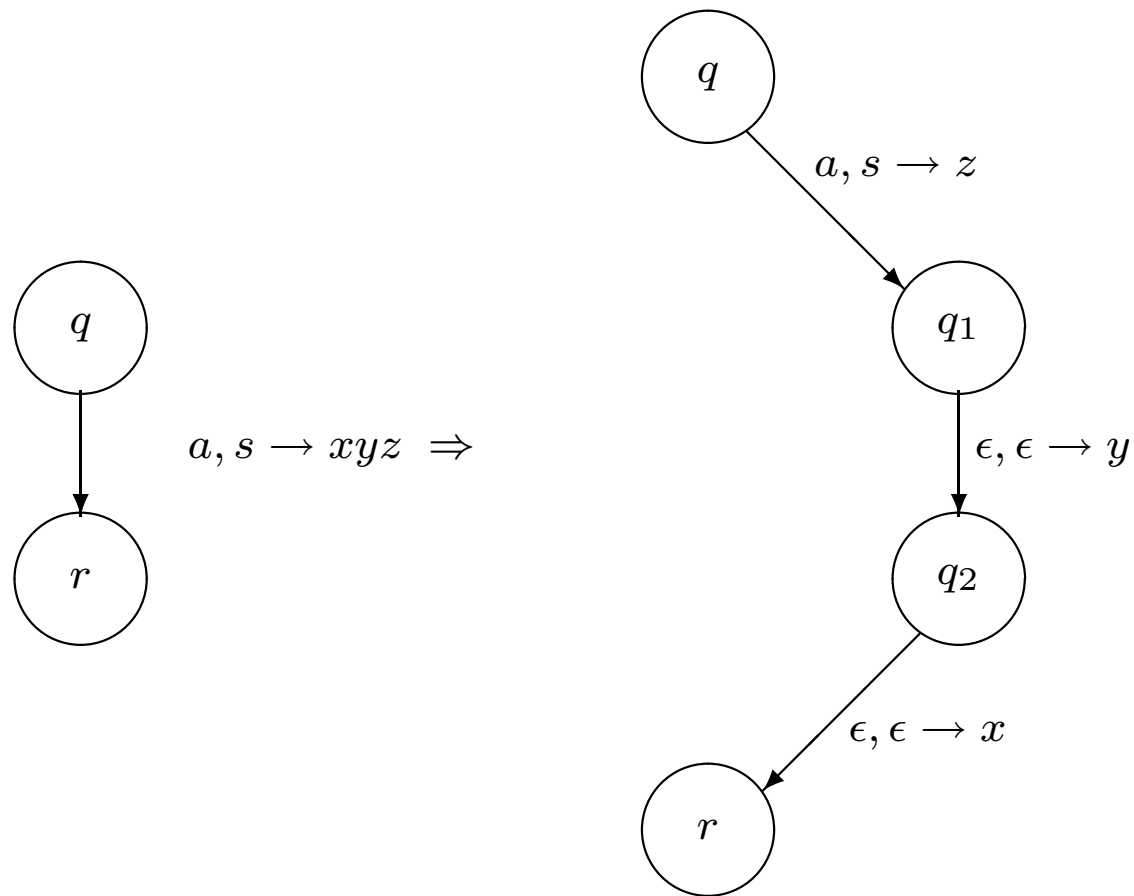


Figure 2: Implementing the shorthand  $(r, xyz) \in \delta(q, a, s)$

# Construction of $P$

- The states of  $P$  are

$Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$  where  $E$  is the set of states that we need to implement the shorthands.

- The transition function is defined as follows:

1. Initialize the stack to contain  $\$$  and  $S$ , i.e.,

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

2. Construct transitions for the main loop

# Main loop transitions

1. First we handle the case where the top of the stack is a variable, by setting:

$\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) \mid A \rightarrow w \in R\}$  where  $R$  is the set of rules of CFG generating the language

2. Then we handle the case where the top of the stack is a terminal, setting:

$$\delta(q_{loop}, a, a) = \{(q_{loop}, \epsilon)\}$$

3. Finally, if the top of stack is \$ we set:

$$\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$$

The state diagram of  $P$  is in Figure 3

# State diagram of $P$

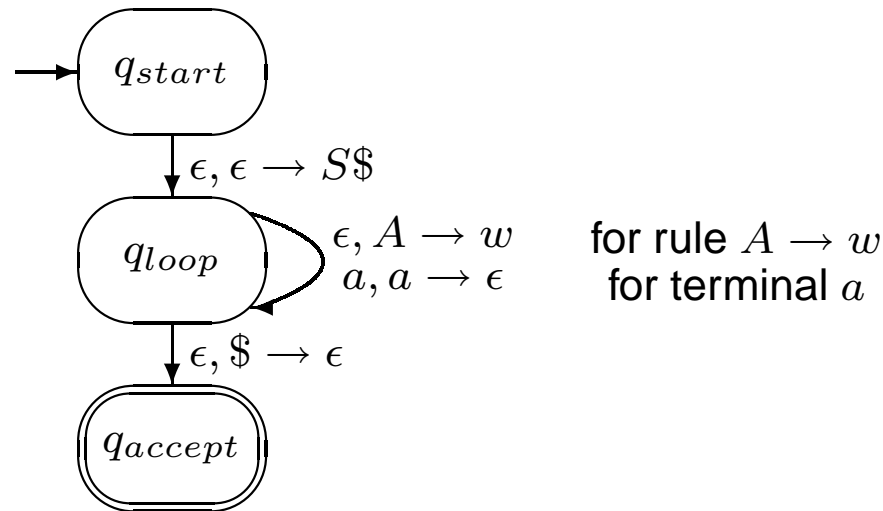


Figure 3: State transition diagram of  $P$

**Notation:**  $\epsilon, A \rightarrow w$ : means reads  $\epsilon$ , pops  $A$ , pushes  $w$ ;

$a, a \rightarrow \epsilon$ : means reads  $a$ , pops  $a$ , pushes  $\epsilon$ .

# Example

We use the procedure developed during the proof of Lemma 2.21 to construct the PDA  $P_G$  that recognizes the language generated by the CFG  $G$  with the rules:

$$S \rightarrow aTb|b$$

$$T \rightarrow Ta|\epsilon$$

A direct application of the construction in Figure 3 leads us to the PDA in Figure 4

# State diagram of $P_G$

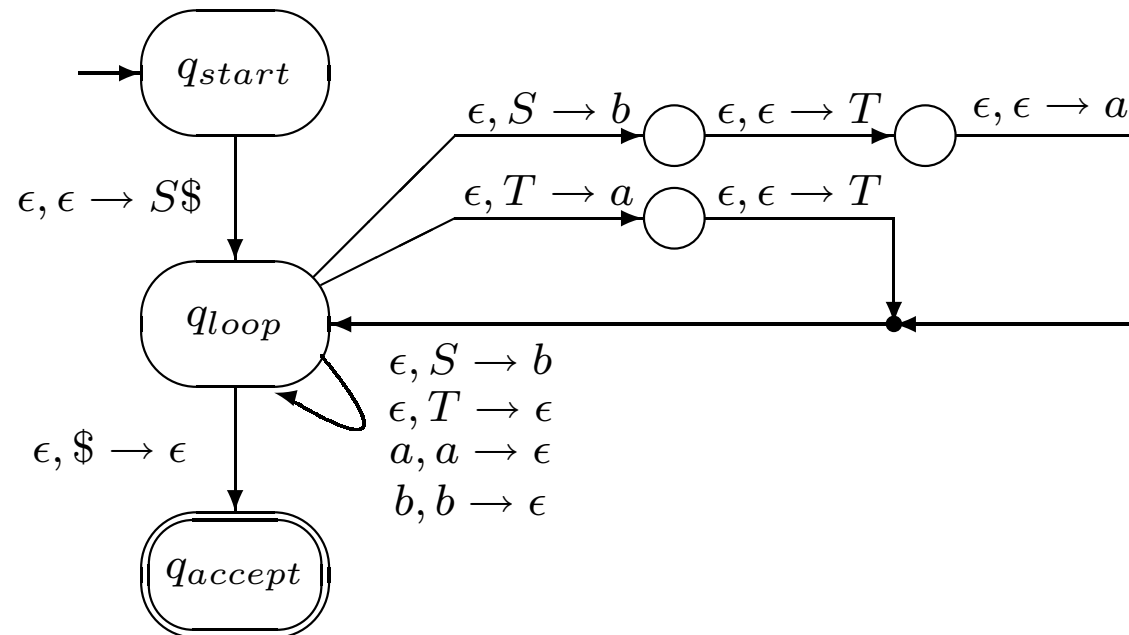


Figure 4: State transition diagram of  $P_G$

## Lemma 2.27

If a language  $L$  is recognized by a pushdown automaton then  $L$  is a context-free language.

# Proof idea

- Here we have a PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$  and want to construct a CFG  $G$  that generates all strings recognized by  $P$ .
- For this we design  $G$  to do somewhat more:  
For each pair of states  $p, q \in Q$ ,  $G$  will have a variable  $A_{pq}$  that generates all strings that can take  $P$   
**from  $p$  with an empty stack to  $q$  with an empty stack;**  
Assume that  $F = \{q_a\}$ . Then  $A_{q_0q_a}$  is the start symbol of the grammar.

**Note:** such strings can take  $P$  from  $p$  to  $q$  regardless of stack contents at  $p$ , leaving the stack at  $q$  the same as it was at  $p$ .

# Assumptions on $P$

1.  $P$  has a single accept state denoted  $q_a$ ;
2.  $P$  empties its stack before accepting;
3. At each transition  $P$  either pushes a symbol on the stack or pops of a symbol from the stack, but does not do both at the same time.

# Construction 1

Giving features (1) and (2) to  $P$  is easy.

1. To give feature (1) to  $P$  add a new state say  $q_a$  to  $Q$ , set  $\{q_a\}$  the set of final states, and add the new transitions:

$$\forall q \in F \text{ set } \delta(q, \epsilon, \epsilon) = \{(q_a, \epsilon)\}$$

2. To give feature (2) to  $P$  just add the transitions:

$$\forall b \in \Gamma \text{ set } \delta(q_a, \epsilon, b) = \{(q_a, \epsilon)\}.$$

# Construction 2

To give feature (3) to  $P$

- Replace each transition that simultaneously pops and pushes with a two transitions sequence that goes through a new state;
- In addition, replace each transition which neither pop nor push with a two transitions sequence that pushes and then pops an arbitrary stack symbol

See Figure 5

# Making it uniform

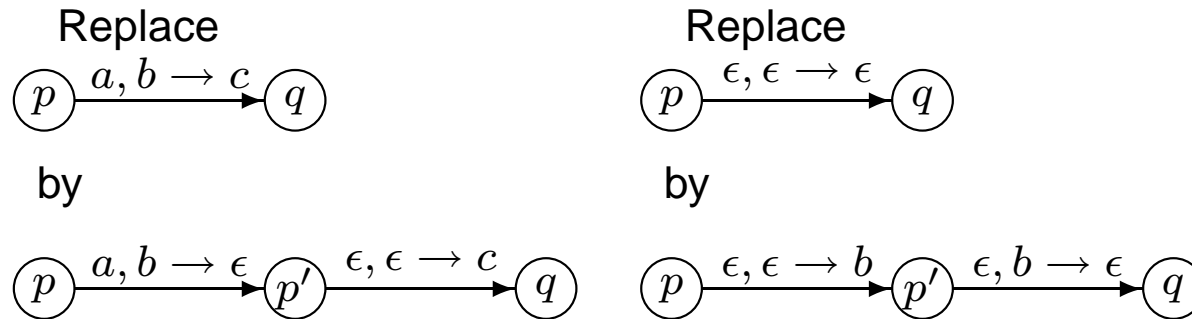


Figure 5: Giving feature 3 to  $P$

# Making it happens

How is  $P$  working on a string  $x$  while moving from  $p$  with an empty stack to  $q$  with an empty stack?

1.  $P$ 's first move on  $x$  must be a push, because each move is either a push or a pop, and the stack is empty.
2. Similarly, last move must be a pop because stack ends up empty.
3. Two possibilities occur during  $P$ 's computation on  $x$ :
  - (a) symbol popped at the end is the symbol pushed at the beginning;
  - (b) symbol popped at the end is not the symbol pushed at the beginning

# Note

- In case (a), the stack is empty only at the beginning and end of  $P$ 's computation;
- In case (b) initially pushed symbol must get popped before end and thus stack becomes empty at that point.

# Grammar simulation of $P$

- We simulate the case (a) of  $P$ 's computation by the rule  $A_{pq} \rightarrow aA_{rs}b$  where:
  1.  $a$  is the input symbol read at the first move,
  2.  $b$  is the symbol read at the last move,
  3.  $r$  is the state following  $p$ , and  $s$  is the state preceding  $q$ .
- We simulate the case (b) of  $P$ 's computation by the rule  $A_{pq} \rightarrow A_{pr}A_{rq}$  where:  
 $r$  is the state where the stack becomes empty.

# Fact

Since stack is empty at (p) and (q) and at every move  $P$  can do only one of **pus** or **pop** the computation represented by  $A_{pq} \rightarrow A_{pr}A_{rq}$  can be performed only with 3 or more states.

# The formal proof

Let  $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_a\})$ . Construct  $G = (\{A_{pq} | p, q \in Q\}, \Sigma, R, A_{q_0q_a})$  where  $R$  is constructed as follows:

1. For each  $p, q, r, s \in Q, t \in \Gamma, a, b \in \Sigma_\epsilon$ , if  $(r, t) \in \delta(p, a, \epsilon)$  (i.e.,  $p \xrightarrow{a, \epsilon \rightarrow t} r$ ) and  $(q, \epsilon) \in \delta(s, b, t)$  (i.e.,  $s \xrightarrow{b, t \rightarrow \epsilon} q$ ) then put  $A_{pq} \rightarrow aA_{rs}b$  in  $R$ ;
2. For each  $p, q, r \in Q$  put the rule  $A_{pq} \rightarrow A_{pr}A_{rq}$  in  $R$ ;
3. For each  $p \in Q$  put the rule  $A_{pp} \rightarrow \epsilon$  in  $R$ .

Figures 6 and 7 provide the intuition of this construction.

# $P$ 's computation corresponding to $A_{pq} \rightarrow A_{pr}A_{rq}$

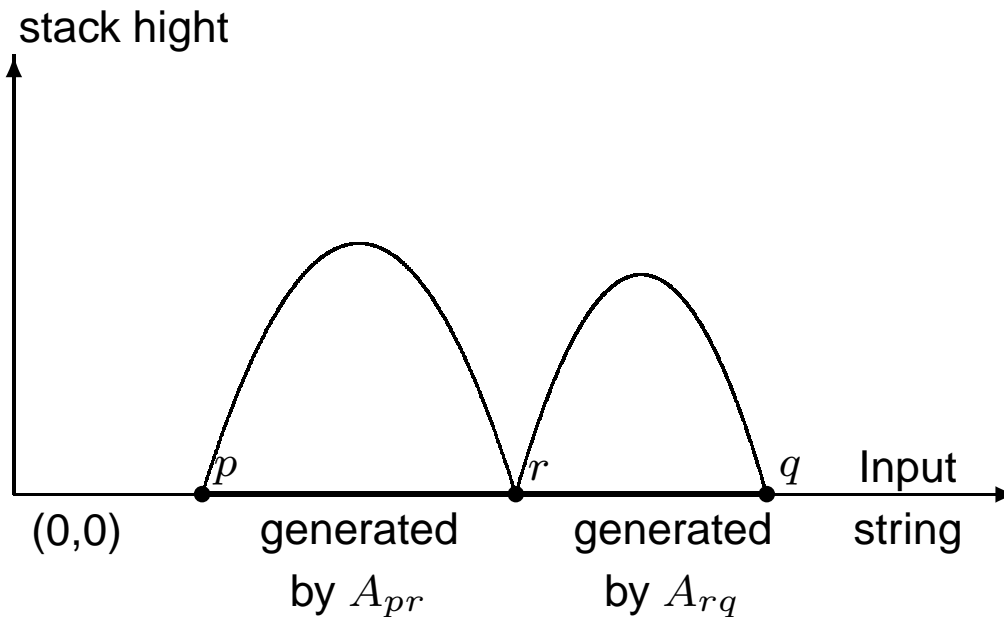


Figure 6:  $P$ 's computation for  $A_{pq} \rightarrow A_{pr}A_{rq}$

# $P$ 's computation corresponding to $A_{pq} \rightarrow aA_{rs}b$

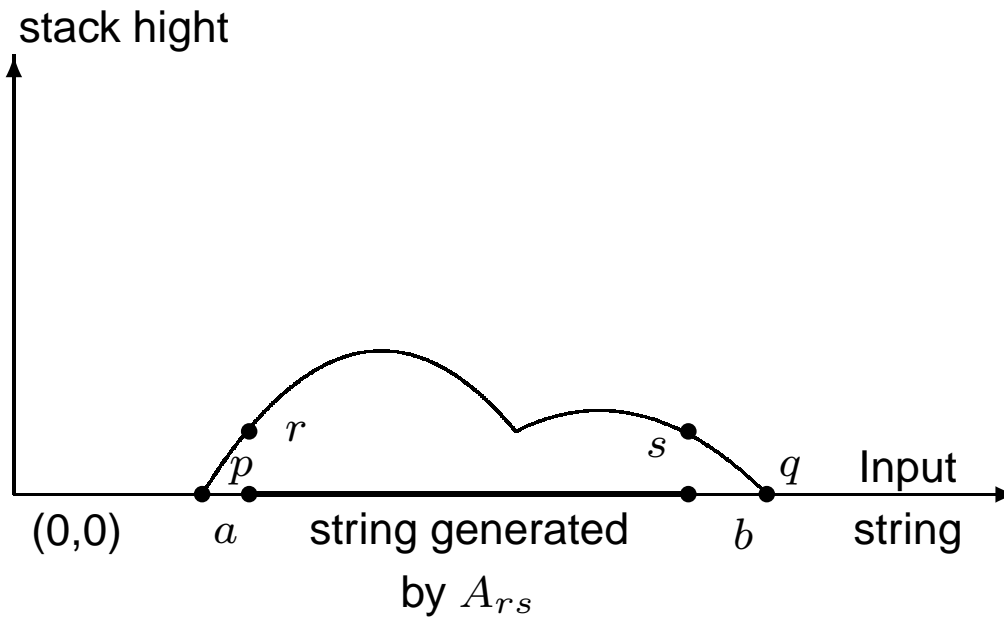


Figure 7:  $P$ 's computation for  $A_{pq} \rightarrow aA_{rs}b$

## Claim 2.30

If  $A_{pq}$  generates  $x$  then  $x$  brings  $P$  from  $p$  to  $q$  with empty stack.

**Proof:** by induction on the number of steps in the derivation of  $x$  from  $A_{pq}$ .

# Induction basis

## Derivation has 1 step

1. A derivation with a single step must use a rule whose *rhs* contains no variables;
2. The only rules in  $R$  whose *rhs* contain no variables are  $A_{pp} \rightarrow \epsilon$ ;
3. Clearly, the input  $\epsilon$  takes  $P$  from  $p$  with empty stack to  $p$  with empty stack.

# Induction step

Assume claim 2.30 true for derivations of length  $k$ ,  $k \geq 1$ , and prove it for  $k + 1$ .

1. Suppose  $A_{pq} \xRightarrow{*} x$  has  $k + 1$  steps. The first step of this derivation is either  $A_{pq} \Rightarrow aA_{rs}b$  or  $A_{pq} \Rightarrow A_{pr}A_{rq}$ ;
2. In the first case consider  $y$  portion of  $x$  generated by  $A_{rs}$ , i.e.,  $x = ayb$ ,  $A_{rs} \xRightarrow{*} y$ ;
3. Because  $A_{rs} \xRightarrow{*} y$  has  $k$  steps, induction hypothesis tells us that  $P$  can go from  $r$  to  $s$  with empty stack;
4. Because  $A_{pq} \rightarrow aA_{rs}b \in R$ , it follows that  $(r, t) \in \delta(p, a, \epsilon)$   
 $(p \xrightarrow{a, \epsilon \rightarrow t} r)$  and  $(q, \epsilon) \in \delta(s, b, t)$  ( $s \xrightarrow{b, t \rightarrow \epsilon} q$ ).

# Induction step continuation

5. Hence, if  $P$  starts at  $p$  with empty stack, after reading  $a$ , it can go to  $r$  and push  $t$  on the stack;
6. Then, reading  $y$  it can go to  $s$  and leaves  $t$  on the stack;
7. At  $q$ , because  $(q, \epsilon) \in \delta(s, b, t)$ , after reading  $b$ ,  $P$  can go to state  $q$  and pop  $t$  off the stack;
8. Hence,  $x$  brings  $P$  from  $p$  to  $q$  with an empty stack.

# Induction step, second case

1. Consider the portions  $y$  and  $z$  of  $x$  that are generated by  $A_{pr}$  and  $A_{rq}$ , respectively, i.e.,  $x = yz$ ,  $A_{pr} \xRightarrow{*} y$ ,  $A_{rq} \xRightarrow{*} z$ ;
2. Because  $A_{pr} \xRightarrow{*} y$  and  $A_{rq} \xRightarrow{*} z$  are derivations containing at most  $k$  steps,  $y$  and  $z$  respectively bring  $P$  from  $p$  to  $r$ , and from  $r$  to  $q$  respectively, with empty stack;
3. Hence,  $x$  can bring  $P$  from  $p$  to  $q$  with empty stack.

## Claim 2.31

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

**Proof:** by induction on the number of steps in the computation of  $P$  going from  $p$  to  $q$  with empty stack on the input  $x$ .

# Induction basis

## Computation has 0 steps.

1. With 0 steps, computation starts and ends with the same state  $p$ ;
2. So, we must show that  $A_{pp} \xRightarrow{*} x$  in 0 steps;
3. In zero steps  $P$  has only time to read the empty string, i.e.,  $x = \epsilon$ ;
4. By construction,  $R$  contains the rule  $A_{pp} \rightarrow \epsilon$ , hence  $A_{pp} \xRightarrow{*} x$ .

# Induction step

Assume claim 2.31 true for computations of length at most  $k$ ,  $k \geq 0$ , and show that it remains true for computations of length  $k + 1$ .

Two cases:

**Case a:**  $P$  has a computation of length  $k + 1$  wherein  $x$  brings  $P$  from  $p$  to  $q$  with empty stack.

**Case b:**  $P$  has a computation of length  $k + 1$  wherein the stack is empty at the begin and end, and stacks may become empty also during computation.

# Case (a)

stack is empty only at the beginning and end.

1. The symbol that is pushed at the first move must be the same as the symbol popped at the last move. Let it be  $t$ ;
2. Let  $a$  be the input read in the first move,  $b$  the input read at the last move,  $r$  be the state after the first move, and  $s$  be the state before the last move;
3. Then  $(r, t) \in \delta(p, a, \epsilon)$  and  $(q, \epsilon) \in \delta(s, b, t)$ , and  $A_{pq} \rightarrow aA_{rs}b \in R$ ;
4. Let  $y$  be the portion of  $x$  without  $a$  and  $b$ , i.e.,  $x = ayb$ . Using induction we know that  $y$  brings  $P$  from  $r$  to  $s$  without touching  $t$  because we can remove the first and the last step of computation, hence  $A_{rs} \xrightarrow{*} y$ ;
5. But then  $A_{pq} \Rightarrow aA_{rs}b \xrightarrow{*} ayb = x$ .

# Induction step, case (b)

stack may be empty between beginning and end.

Let  $r$  be the state where the stack becomes empty other than at the beginning or end of computation on the input  $x$

1. The portions of the computation from  $p$  to  $r$  and from  $r$  to  $q$ , both contain at most  $k$  steps;
2. Let  $y$  be the input read during the computation from  $p$  to  $r$ , and  $z$  be the input read during the computation from  $r$  to  $q$ ;
3. Using induction hypothesis we have  $A_{pr} \xRightarrow{*} y$  and  $A_{rq} \xRightarrow{*} z$ ;
4. Because  $A_{pq} \rightarrow A_{pr}A_{rq} \in R$  it results that  $A_{pq} \Rightarrow A_{pr}A_{rq} \xRightarrow{*} yz = x$ .

# Facts

1. We have proved that pushdown automata recognize the class of context free languages;
2. Every regular language is recognized by a finite automaton;
3. Every finite automaton is a pushdown automaton that ignores its stack.

**Conclusion:** every regular language is a context-free language.