Chapter 2 - Context-Free Languages (Lenguajes libres de contexto)

Q: What did we see in chapter 1?

→ Methods for describing languages
- Finite Automata
- Regular Expressions

} Both of which are equivalent (i.e. recognize the same languages)

Q: What will we see in chapter 2?

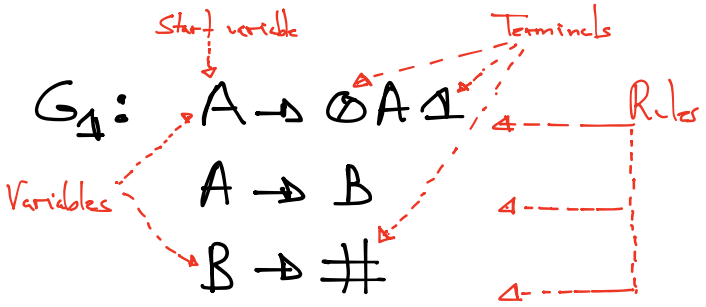
→ Context-free grammars, a more powerful method of describing languages:
- Can describe certain features that have a recursive structure
- Application: Specification and compilation of programming languages.

→ Context-free languages, collection of languages associated with context-free grammars

2.1 Context-Free Grammars

Example

G_1:



Grammar consists of:

1. **Substitution rules** Each rule appears as a line in the grammar, comprising:

   1.1 Symbol, called a **variable**, represented by capital letters

   1.2 String, consisting of other variables and other symbols called **terminals**. The terminals are analogous to the input alphabet and are represented by lowercase letters

2. **Start variable**, left-hand side of the topmost rule

Q: How does a grammar work?

A grammar is used to describe a language by generating each string in the following manner:

→

① Write down 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 right-hand side of that rule.

③ Repeat step 2 until no variables remain

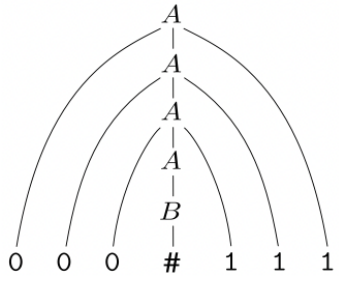
Grammar G_1 can generate the string:

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

- The sequence of substitutions is called a **derivation**

Árvore Sintática

The derivation can also be represented by a **parse tree**



Q: What is the language of grammar G_1?

$$L(G_1) = \{ 0^n \# 1^n \mid n \geq 0 \}$$

Any language that can be generated by some context-free grammar is called a **context-free language** (CFL)

Example

G_2 describes a fragment of the english language:

Substantivo

<SENTENCE> → <NOUN-PHRASE> | <VERB-PHRASE>

<NOUN-PHRASE> → <CMPX-NOUN> | <COMPLEX-NOUN> <PREP-PHRASE>

<VERB-PHRASE> → <CMPX-VERB> | <CMPLX-VERB> <PREP-PHRASE>

Preposição

<PREP-PHRASE> → <PREP> <COMPLEX-NOUN>

<COMPLEX-NOUN> → <ARTICLE> <NOUN>

<COMPLEX-VERB> → <VERB> | <VERB> <NOUN-PHRASE>

<ARTICLE> → a | the

<NOUN> → boy | girl | flower

<VERB> → touches | likes | sees

<PREP> → with

Let's see one possible derivation:

<SENTENCE> → <NOUN-PHRASE> <VERB-PHRASE>
→ <CMPLX-NOUN> <VERB-PHRASE>
→ <ARTICLE><NOUN><VERB-PHRASE>
→ a <NOUN><VERB-PHRASE>
→ a boy <VERB-PHRASE>
→ a boy <CMPLX-VERB>
→ a boy <VERB>
→ a boy sees

Examples of other strings in G_2:

s_1: a boy sees

s_2: the boy sees a flower

s_3: a girl with a flower likes the boy

Q: What is the formal definition of a context-free grammar?

Definition 2.2

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
4. $S \in V$ is the **start variable**

Important Notations (1/2):

- Let u, v and w be string of variables and terminals
- Let $A \to w$ be a rule
- 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

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

Important Notations (2/2):

- The language of the grammar is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$

Q:	What is the formal definition of CFG G_1?

$G_1 = (V, \Sigma, R, S)$ where:

- $V = \{A, B\}$
- $\Sigma = \{0, 1, \#\}$
- $S = A$
- R = collection of rules described in page 2

Q:	What is the formal definition of CFG G_2?

$G_2 = (V, \Sigma, R, S)$ where:

- $V = \{$<SENTENCE>, <NOUN-PHRASE>, <VERB-PHRASE>, <PREP-PHRASE>, <COMPLX-NOUN>, <COMPLX-VERB>, <ARTICLE>, <NOUN>, <VERB>, <PREP>$\}$
- $\Sigma = \{a, b, c, \ldots, z, \underbrace{" "}_{\text{White space character}}\}$
- $S =$ <SENTENCE>
- R = collection of rules described in page 4

Example 2.3

Variables, Terminals, Rules, Start variable

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

$$S \to aSb \mid SS \mid \varepsilon$$

Q: What is the language generated by G_3, i.e. $L(G_3)$?

Let's see possible derivations:

① $S \to \varepsilon$

② $S \to aSb$
$\to a\varepsilon b$
$\to ab$

③ $S \to aSb$
$\to aaSbb$
$\to aa\varepsilon bb$
$\to aabb$

④ $S \to aSb$
$\to aaSbb$
$\to aaaSbbb$
$\to aaa\varepsilon bbb$
$\to aaabbb$

⑤ $S \to SS$
$S \to aSb\ aSb$
$S \to a\varepsilon b\ a\varepsilon b$
$S \to ab\ ab$

⑥ $S \to SS$
$S \to aSb\ SS$
$S \to a\varepsilon b\ aSb\ SS$
$S \to ab\ a\varepsilon b\ aSb\varepsilon$
$S \to ab\ ab\ a\varepsilon b$
$S \to ab\ ab\ ab$

⑦ $S \to aSb$
$\to aSSb$
$\to aaSbb\ Sb$
$\to aa\varepsilon bb\ aSb\ b$
$\to aa\varepsilon bb\ a\varepsilon b\ b$
$\to aabb\ ab\ b$

∴ G_3 generates strings such as : abab, aaabbb, aababb

Q:
- Substitute the terminal a by the left parenthesis "("
- Substitute the terminal b by the right parenthesis ")"
- What is $L(G_3)$?

↳ Viewed this way: $L(G_3)$ is the language of all strings of properly nested parentheses.

Example 2.4

Consider $G_4 = (V, \Sigma, R, \langle EXPR \rangle)$ where:

(Variables, Terminals, Rules, Start Variable)

- $V = \{\langle EXPR \rangle, \langle TERM \rangle, \langle FACTOR \rangle\}$
- $\Sigma = \{a, +, \times, (,)\}$ (Left Parenthesis, Right Parenthesis)
- Rules R are:

$\langle EXPR \rangle \rightarrow \langle EXPR \rangle + \langle TERM \rangle \mid \langle TERM \rangle$

$\langle TERM \rangle \rightarrow \langle TERM \rangle \times \langle FACTOR \rangle \mid \langle FACTOR \rangle$

$\langle FACTOR \rangle \rightarrow (\langle EXPR \rangle) \mid a$

Q: What is the language generated by G_4, i.e. $L(G_4)$?

Let's see possible derivations:

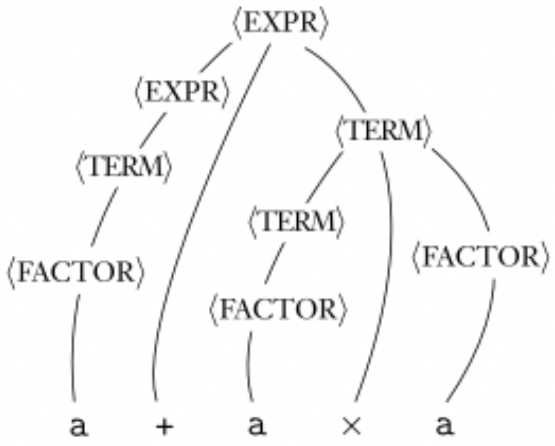
① $\langle EXPR \rangle \rightarrow \langle EXPR \rangle + \langle TERM \rangle$

$\rightarrow \langle TERM \rangle + \langle TERM \rangle \times \langle FACTOR \rangle$

$\rightarrow \langle FACTOR \rangle + \langle FACTOR \rangle \times \langle FACTOR \rangle$

$\rightarrow a + a \times a$

The corresponding parse tree for this derivation:



Notes:

The tree "groups" the × operator and its operands (the second two a's) as one operator of the + operator. This tree represents the standard precedence of multiplication before addition.

② ⟨EXPR⟩ → ⟨TERM⟩

→ ⟨TERM⟩ × ⟨FACTOR⟩

→ ⟨FACTOR⟩ × ⟨FACTOR⟩

→ (⟨EXPR⟩) × a

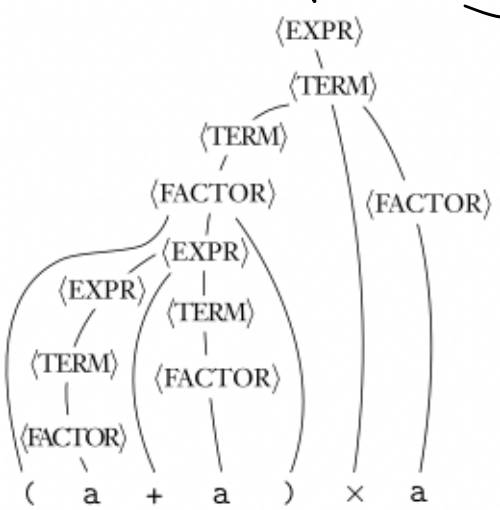
→ (⟨EXPR⟩ + ⟨TERM⟩) × a

→ (⟨TERM⟩ + ⟨FACTOR⟩) × a

→ (⟨FACTOR⟩ + a) × a

→ (a + a) × a

The corresponding parse tree for this derivation:



Notes:

The tree "grouping" is reversed: the use of parentheses overrides the standard precedence of multiplication i.e., instead of a + a × a the grouping results in (a + a) × a

∴ G_4 describes a fragment of a programming language concerned with arithmetic expressions.

Important Observations:

- Compilers translate code written in a programming language ("human easy") to one more suitable for execution ("machine easy")

- A compiler extract the meaning of a code in a process called parsing. The parse tree is one representation of this meaning

Designing Context-Free Grammars

Q: So, how do we design CFGs?

→ Mixture: Creativity and trial and error

Let's see some possible techniques:

Technique 1: Many CFLs are the union of simpler CFLs

→

1.1 Construct CFGs for the simpler CFLs. Solving simpler problems is often easier than solving one complicated problem

1.2 Merge the individual grammars to get a grammar for the original language. This can be done by combining their rules and the adding the new rule:

Start variable for CFG_1 | Start variable for CFG_2 | Start variable for CFG_k

$$S \to S_1 \mid S_2 \mid \dots \mid S_k$$

Example

Construct a grammar for the language:

$$\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

[Resolution:]

① Construct grammar for the language: $\{0^n 1^n \mid n \geq 0\}$

$$S_1 \rightarrow 0 S_1 1 \mid \varepsilon$$

② Construct grammar for the language: $\{1^n 0^n \mid n \geq 0\}$

$$S_2 \rightarrow 1 S_2 0 \mid \varepsilon$$

③ Construct global grammar:

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow 0 S_1 1 \mid \varepsilon$$
$$S_2 \rightarrow 1 S_2 0 \mid \varepsilon$$

Technique 2: Constructing a CFG is easy if you can construct a DFA for the language. Any DFA can be converted into an equivalent CFG as follows:

2.1 Create a variable R_i for each state q_i of the DFA

2.2 Add the rule $R_i \to aR_j$ if $\delta(q_i, a) = q_j$ is a transition in the DFA.

2.3 Add the rule $R_i \to \varepsilon$ if q_i is a final state of the DFA.

2.4 Make R_0 the start variable

Technique 3: Certain CFLs contain strings with two substrings that are "linked". This occurs in the language $\{0^n 1^n \mid n \geq 0\}$ since the machine needs to remember the number of 0s in order to verify that it equals the number of 1s. A CFG can be constructed by using a rule of the form $R \to 0R1$ which generates strings with an equal number of 0s and 1s.

Technique 4: In more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. In example 2.4 (page 10) any time a symbol a appears, an entire parenthesized expression might appear recursively instead (see the third rule). To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

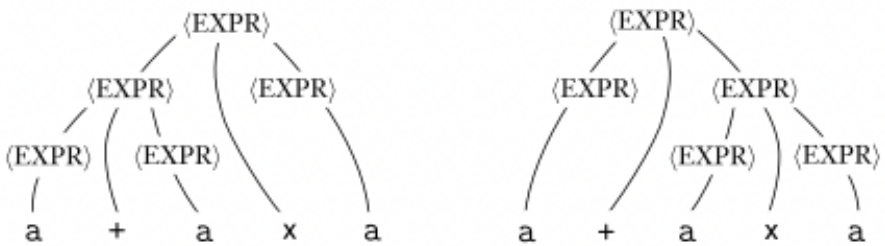
Grammar Ambiguity

Sometimes a grammar can generate the same string in several different ways. This means ≠ parse trees will exist and, consequently, ≠ meanings will exist. Ambiguity is undecidable for programming languages.

Example Consider grammar G_5:

$$\langle EXPR \rangle \rightarrow \langle EXPR \rangle + \langle EXPR \rangle \mid \langle EXPR \rangle \times \langle EXPR \rangle \mid (\langle EXPR \rangle) \mid a$$

G_5 generates the string $a + a \times a$ ambiguously:



∴1 G_5 does not capture the usual precedence relations. Depending on which parse tree is evaluated it may calculate wrongly (ambiguous)

∴2 This contrast with G_4 that generates a unique parse tree (unambiguous)

Q: So, what is the formal definition of ambiguity?

A grammar is ambiguous if a string has two ≠ parse trees, not two ≠ derivations.

Two derivations may differ merely in the order in which they replace variables yet not in their overall structure

To concentrate on structure, we define a type of derivation that replaces variables in a fixed-order. A derivation of a string w in a grammar G is a leftmost derivation if at every step the left most remaining variable is the one replaced.

Example of a leftmost derivation

Grammar G_2 of page 5:

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩
⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩
⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩
⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩
⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩
⟨ARTICLE⟩ → a | the
⟨NOUN⟩ → boy | girl | flower
⟨VERB⟩ → touches | likes | sees
⟨PREP⟩ → with

Left-most derivation:

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⇒ ⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩
⇒ ⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a ⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a boy ⟨VERB-PHRASE⟩
⇒ a boy ⟨CMPLX-VERB⟩
⇒ a boy ⟨VERB⟩
⇒ a boy sees

Definition 2.7

A string w is derived **ambiguously** in CFG G if it has two or more $\neq$ left-most derivations. Grammar G is **ambiguous** if it generates some string ambiguously

Important:

① **Sometimes** when we have an ambiguous CFG we can find an unambiguous CFG that generates the same language

② **However**, some CFL can only be generated by ambiguous CFG Such languages are called: **inherently ambiguous**

Chomsky Normal Form

Notes: Noam Chomsky fuck yeah!!! Still alive and one of the great minds of our time.

When working with CFGs it is often convenient to have them in simplied form: **Chomsky Normal Form**

→

Definition 2.8

A CFG 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 with the exception that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Theorem 2.9

Any CFL is generated by a CFG in Chomsky normal form

Proof Idea:

(1) Any grammar G can be converted to Chomsky normal form

(2) Conversion: Rules that violate Chomsky conditions are replaced with equivalent ones that are satisfactory:

(2.1) Add a new start variable;

(2.2) Eliminate all ε-rules of the form $A \to \varepsilon$;

(2.3) Eliminate all unit rules of the form $A \to B$.

Q:	What is a unit rule?

↳ A rule that merely goes to a single rule (without any strings of variables). I.e: rules of the form $A \to B$

(3) Patch up the grammar to make sure it generates the same language.

(4) Convert remaining rules into the proper form

Proof:

① Add a new start variable S_0
Add a new rule $S_0 \to S$ (Original Start Variable)

This guarantees that the start variable does not occur in the right-hand side of a rule.

② Remove any ε-rule $A \to \varepsilon$, where A is not the start variable.

For each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted

Examples:

2.1 $R \to A$, then $R \to \varepsilon$ is added

2.2 $R \to uAv$, then $R \to uv$ is added

Notes: u and v are strings of variables and literals

2.3 $R \to uAvAw$, then the following rules are added:

$R \to uvAw$

$R \to uAvw$

$R \to uvw$

Notes:

We either substitute the first A or

we substitute the second A or

we substitute both A's

(proof continues on the next page)

③ Remove all unit rules (i.e. rules of the form $A \to B$).

Example

Unit Rule

$A \to B$
$B \to u$

} Is converted to $A \to u$

Notes:
u and v are strings of variables and literals

④ Convert all remaining rules into the proper form. 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:

$A \to u_1 A_1$
$A_1 \to u_2 A_2$
$A_2 \to u_3 A_3$
$\vdots$
$A_{k-2} \to u_{k-1} u_k$

Notes:
This is closer to Chomsky's normal form but is still a little bit $\neq$ ...

→ Any terminal u_i in the preceding rules are replaced with a new variable U_i and add the rule $U_i \to u_i$.

Notes:
This final step guarantees that every rule is in accordance with Definition 2.8

Example 2.10 Convert G_6 to Chomsky normal form:

G_6: $S \to ASA \mid aB$
$A \to B \mid S$
$B \to b \mid \varepsilon$

[resolution:]

① Add a new start variable S_0. Add a new rule $S_0 \to S$.

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

② Remove ε-Rules

(2.1) Removing $B \to \varepsilon$ means adding new rules with that occurrence deleted whenever B appears on the right-hand side:

(2.1.1) What we want to remove

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

(2.1.2) With new rules:

$S_0 \to S$
$S \to ASA \mid aB \mid a\varepsilon$
$A \to B \mid S \mid \varepsilon$
$B \to b$

(2.1.3) Final grammar:

$S_0 \to S$
$S \to ASA \mid aB \mid a$
$A \to B \mid S \mid \varepsilon$
$B \to b$

(2.2) Removing $A \to \varepsilon$ means adding new rules with that occurrence deleted whenever A appears on the right-hand side:

(2.1.1) What we want to remove

$S_0 \to S$
$S \to ASA \mid aB \mid a$
$A \to B \mid S \mid \varepsilon$
$B \to b$

(2.1.2) With new rules:

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid \varepsilon SA \mid AS \varepsilon \mid \varepsilon S \varepsilon$
$A \to B \mid S$
$B \to b$

(2.1.3) Final grammar:

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to b$

(3) Remove unit rules

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to b$

Unit Rules

(3.1) Remove unit rules $S \to S$:

(3.1.1) Unit rule to be removed

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid SA \mid AS \mid S$
$A \to B \mid S$
$B \to b$

(3.1.2) After removing unit rule

$S_0 \to S$
$S \to ASA \mid aB \mid a \mid SA \mid AS$
$A \to B \mid S$
$B \to b$

3.2 Remove unit rules $S_0 \to S$

3.2.1 Unit rule to be removed

$S_0 \to \cancel{S}$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to B \mid S$

$B \to b$

3.2.2 After removing unit rule

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to B \mid S$

$B \to b$

3.3 Remove unit rules $A \to B$:

3.3.1 Unit rule to be removed

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to \cancel{B} \mid S$

$B \to b$

3.3.2 After removing unit rule

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to b \mid S$

$B \to b$

3.4 Remove unit rules $A \to S$:

3.4.1 Unit rule to be removed

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to b \mid \cancel{S}$

$B \to b$

3.4.2 After removing unit rule

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to b \mid ASA \mid aB \mid a \mid SA \mid AS$

$B \to b$

(4) Convert remaing rules into the proper form by adding variables and rules

(4.1) Latest version of the grammar

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS$

$S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to b \mid ASA \mid aB \mid a \mid SA \mid AS$

$B \to b$

(4.2) Lets replace terminal by variable

$S_0 \to ASA \mid UB \mid a \mid SA \mid AS$

$S \to ASA \mid UB \mid a \mid SA \mid AS$

$A \to b \mid ASA \mid UB \mid a \mid SA \mid AS$

$B \to b$

$U \to a$

(4.3) Convert to two variable maximum

$S_0 \to AA_1 \mid UB \mid a \mid SA \mid AS$

$S \to AA_1 \mid UB \mid a \mid SA \mid AS$

$A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS$

$B \to b$

$U \to a$

$A_1 \to SA$

(proof continues on next page)

2.2 Pushdown Automata (PDA)

- This section introduces a new type of computational model **pushdown automata** ("autômato com pilha"):

- **PDA** are similar to NFA but have an extra component called a **stack**:

  - Provides additional memory beyond the finite amount available in the control.

  - Allows PDA to recognize to recognize **some** nonregular languages

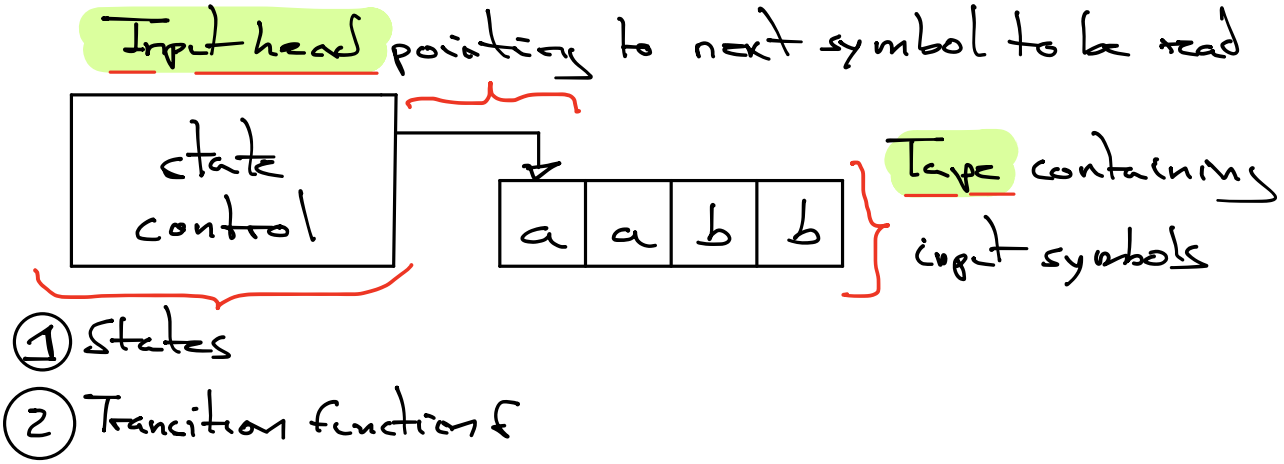
- PDA are **equivalent** in power to context-free grammars. This is useful because it gives us two options for proving that a language is context-free:

  1. Describe a CFG for the CFL or
  2. Describe a PDA for the CFL

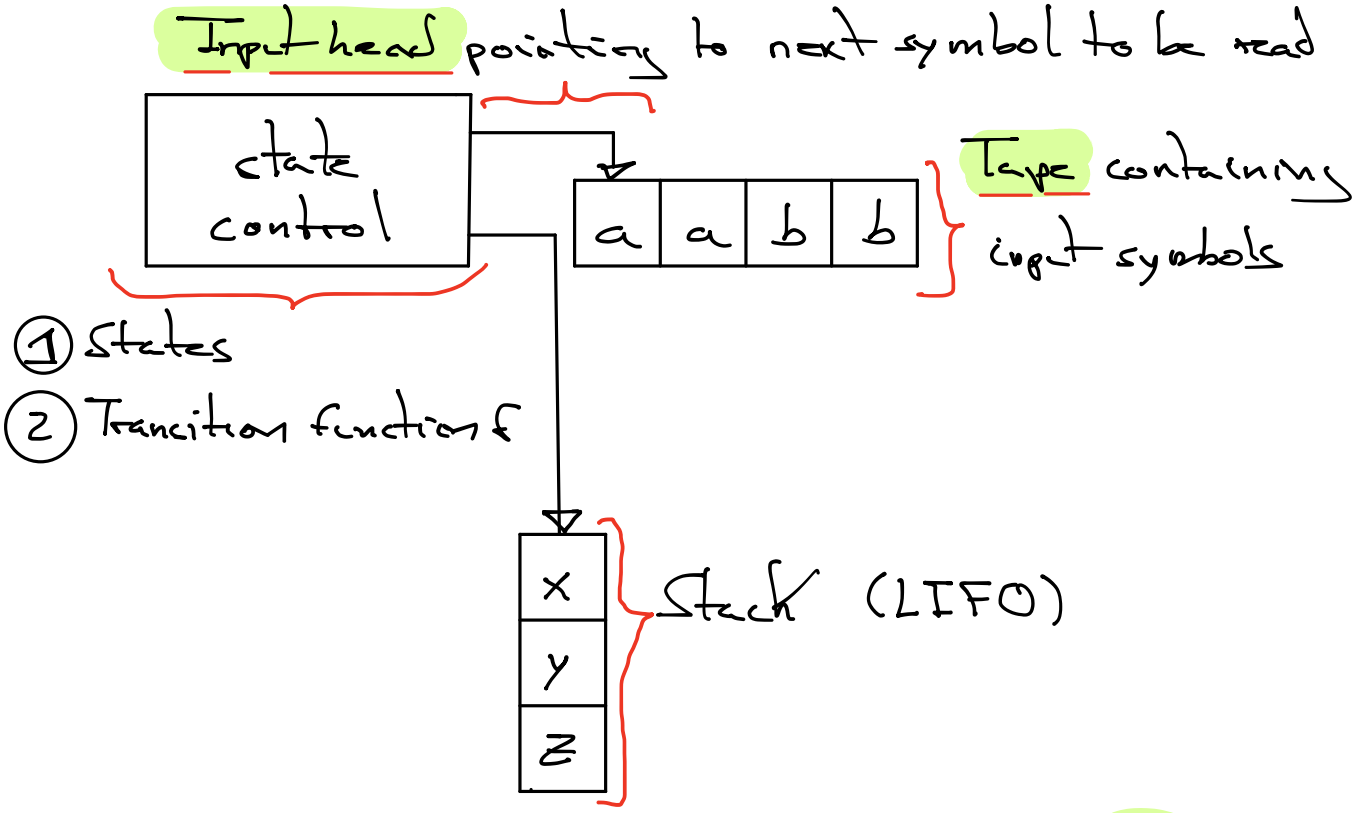
Notes:
Certain languages are more easily described by CFG whereas others are more easily described by PA

Attention: Do not confuse with a PA!!!

- Schematic representation of a finite automaton:



- Schematic representation of a pushdown automata:



- PDA can write symbols on the stack (push) and read them back later (pop)
- Writing a symbol "pushes down" all other symbols on the stack thus the name PDA.

Stack is valuable because it can hold an unlimited amount of information:

① Recall: A FA is unable to recognize $\{0^n 1^n \mid n \geq 0\}$ because it cannot store very large numbers in its finite memory.

② A PDA is able to recognize $\{0^n 1^n \mid n \geq 0\}$ because it can use the stack to store the number of 0s it has seen. Procedure:

2.1 Read symbols from the input

2.2 As each 0 is read, push it onto the stack

2.3 As each 1 is read, pop a 0 from the stack

2.3 If the last 1 results in an empty stack: accept

2.4 Otherwise: reject

Important:

- PDA may be deterministic or nondeterministic
- Deterministic PDA and Non-Deterministic PDA are not equivalent in power
- Non-Deterministic PDA recognize certain languages that no deterministic PDA can recognize (we will see this in Section 2.4)
- Recall: NFA and DFA are equivalent

Formal definition of a PDA

- Similar to a FA, except for the stack which contains symbols drawn from some alphabet.

- Machine may use ≠ alphabets for its input and its stack:

Σ := input alphabet and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

(tau sweet symbol) Γ := stack alphabet and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$

- Domain of the transition function δ is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$:

Q := current state
Σ_ε := next input symbol
Γ_ε := 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.

(proof idea continues on next page)

- Range of the transition function δ is $Q \times \Gamma_\varepsilon$:

① Machine may enter some new state (Q) and possibly write a symbol on the top of the stack

② Because non-determinism is allowed the machine may have several legal next moves. The transition function incorporates nondeterminism by returning a set of members of $Q \times \Gamma_\varepsilon$, ie. a member of $P(Q \times \Gamma_\varepsilon)$

↳ Power set := the collection of all subsets in $Q \times \Gamma_\varepsilon$

∴ Transition function δ is of the form $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$

Definition 2.13

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. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function
5. $q_0 \in Q$ is the initial state
6. $F \subseteq Q$ is the set of final states

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows:

(1) If accepts input w if w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\varepsilon$

A sequence of zero or more symbols of Γ (including ε)

(2) 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.

Notes:
The string s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

(2.1) initial state $r_0 = q_0$ and empty stack $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.

(2.2) For state indexes $i = 0, \dots, m-1$ 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 $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$.

New state; symbol to push; current state; next input symbol; top symbol of stack

(t i.e. $\in$ is a string over Γ)

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.

Notation:

$a, b \to c$ signifies that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with symbol c.

If $a == \varepsilon$: Machine may make the transition without reading any symbol from the input.

If $b == \varepsilon$: Machine may make the transition without popping any symbol from the stack.

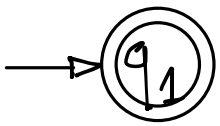
If $c == \varepsilon$: Machine may make the transition without pushing any symbol from the stack.

Example 2.14

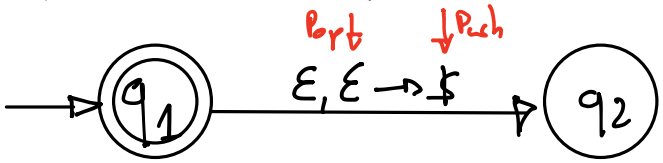
- Build the PDA automata M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$

[resolution:]

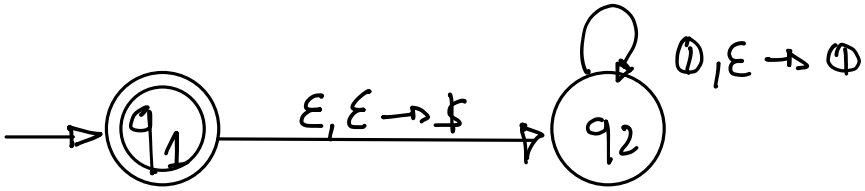
① Recall that $\{0^n 1^n \mid n \geq 0\}$, this means that ε string results in an accept state:



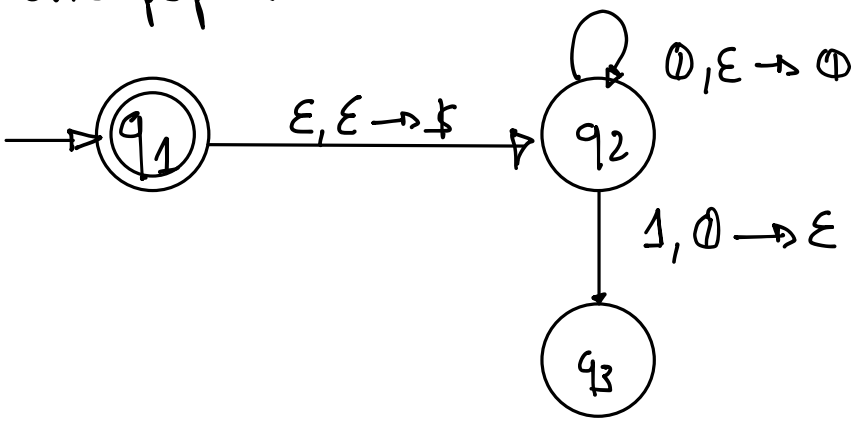
② Let's start by adding a special symbol ($) to our stack so to mark the start of the stack. This way, whenever we read $ from the stack we know that we have reached its end:



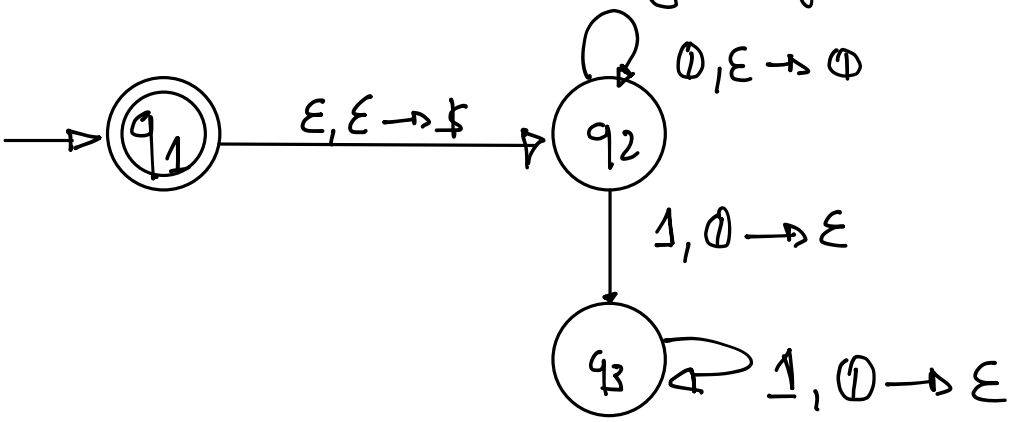
③ Whenever we see 0's we need to add them to the stack:



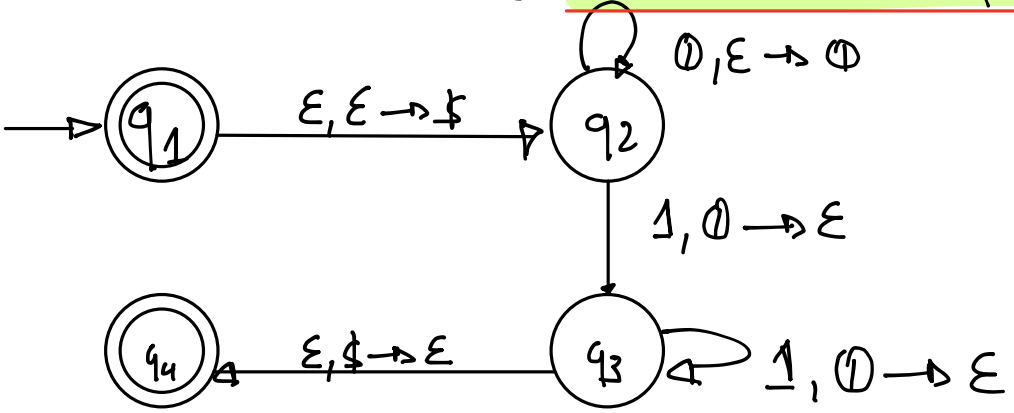
④ When we see the first one we need to transition to a new state and pop a 0.



⑤ Whenever we see 1's we just pop the corresponding 0's



⑥ If the input finishes (ε) and we also have the empty stack. This can be done by always testing with each symbol read if the machine finishes (in a non-deterministic way):



$\therefore M_1 = (Q, \Sigma, \Gamma, \delta, q_1, F)$ where:

① $Q = \{q_1, q_2, q_3, q_4\}$

② $\Sigma = \{0, 1\}$

③ $\Gamma = \{0, \$\}$

④ $F = \{q_1, q_4\}$

⑤ δ is given by the following table:

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									

New state

Push onto stack

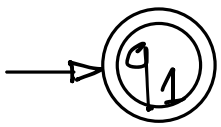
Example 2.46

- Build the PDA automata M_2 that recognizes the language:

$$\{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i = j \text{ or } i = k \}$$

[resolution:]

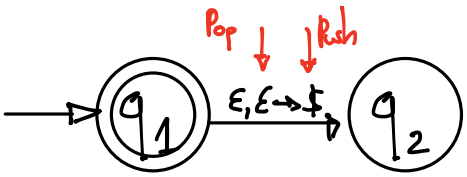
① The ε string results in an accept state:



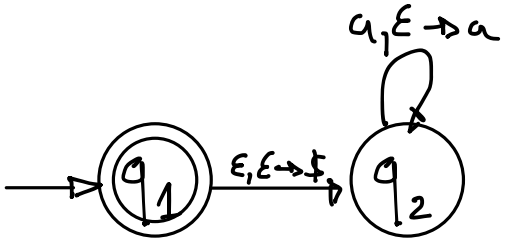
Notes:

We will change this later, but for now let q_1 be a final state

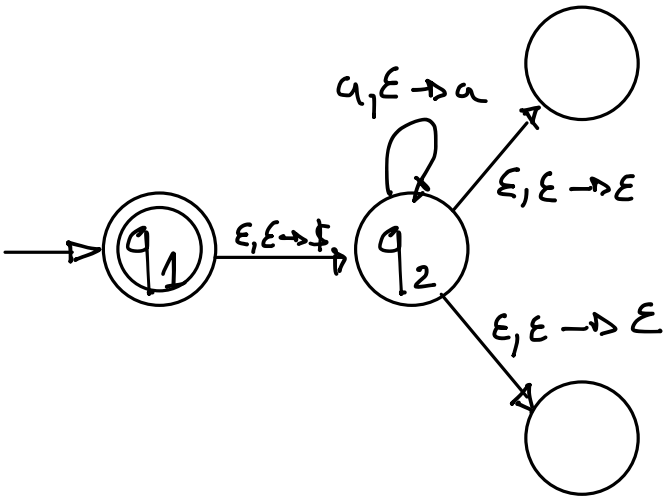
② Lets start with adding \$ to the stack:



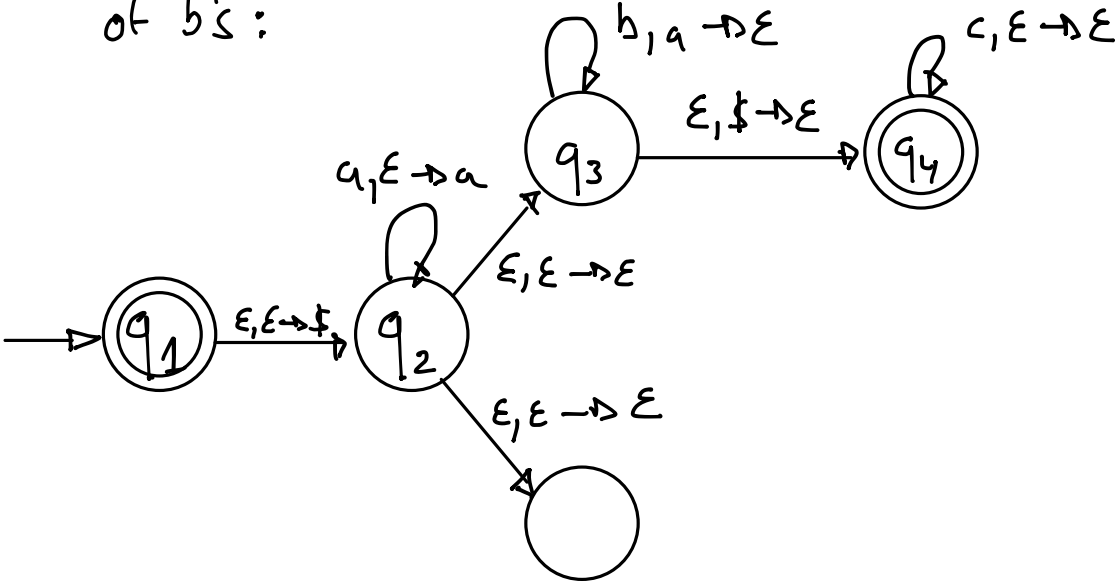
③ Now we need to count the number of a's, we can do this by pushing onto the stack:



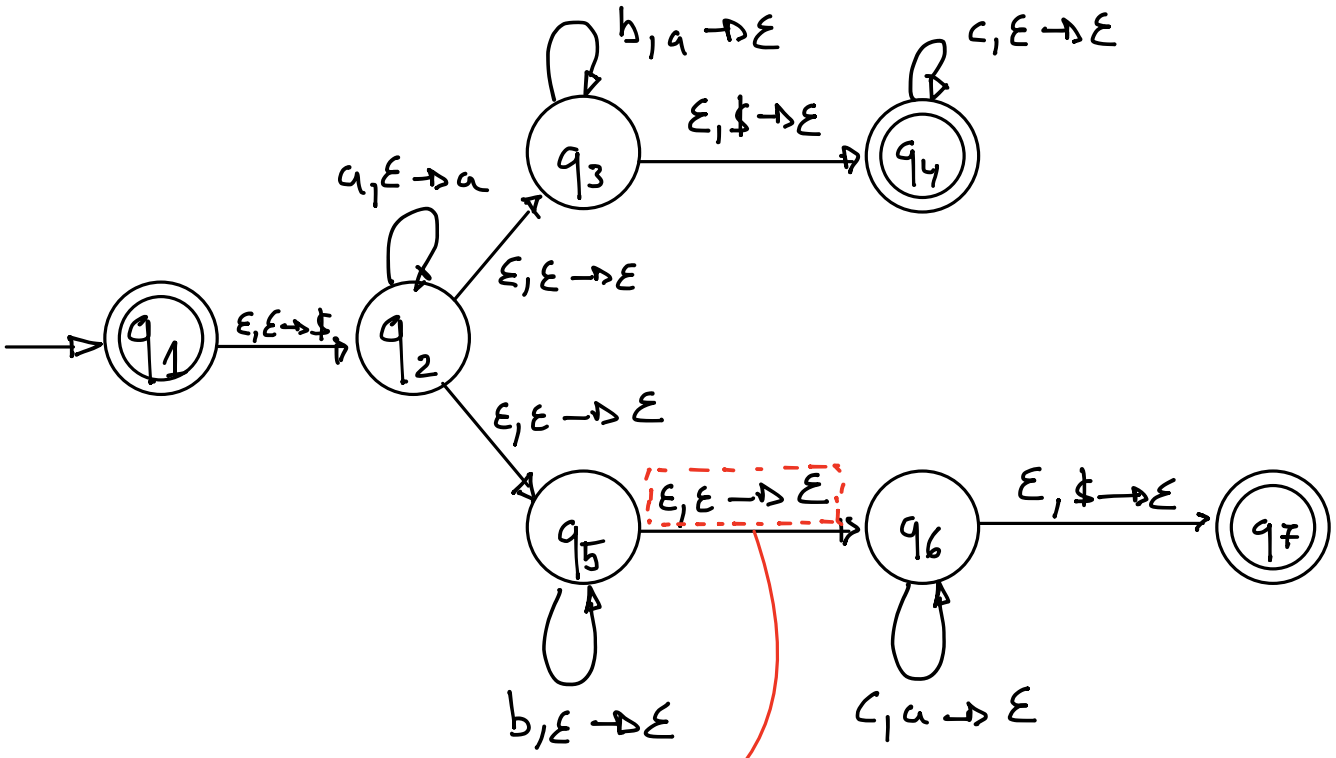
④ Now we need to check if the number of a's matches with the number of b's or the number of c's. This or condition can be checked with non-determinism.



⑤ Let's focus first on verifying if the number of a's matches the number of b's:



⑥ Now we can focus on verifying if the number of a's matches the number of c's

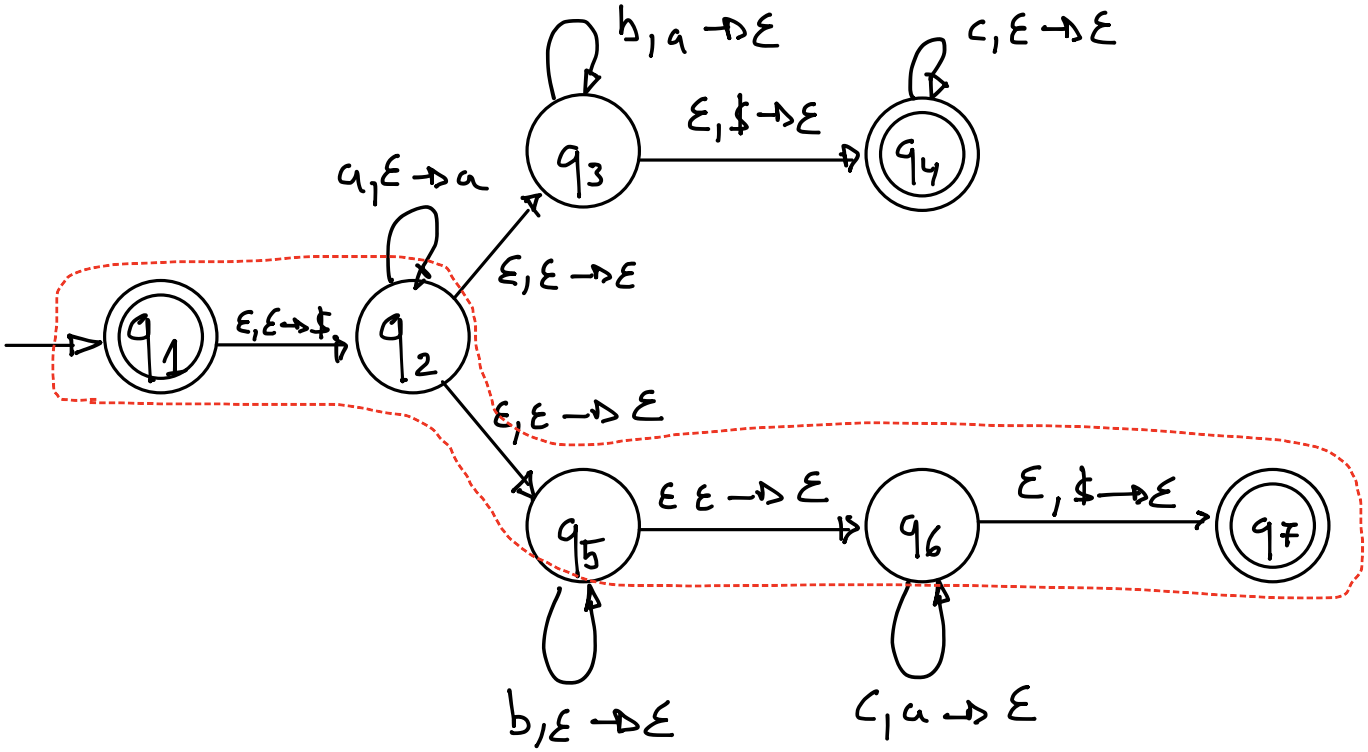


Notes:

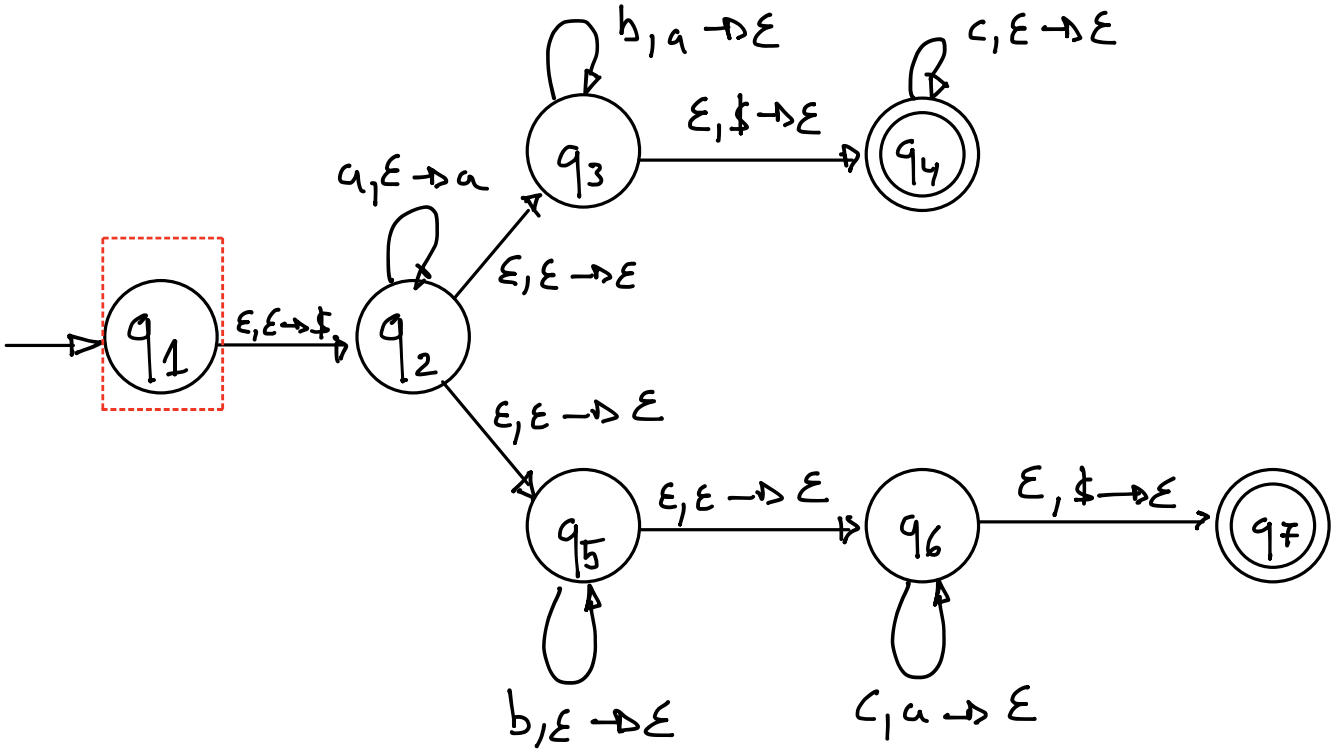
Why this transition?

The number of b's might zero, e.g. "aacc". Therefore, if we read a series of a's we need to test if this is followed by a sequence of c's

(7) Notice that with this latest version it is possible to start in the initial state (q_1) and reach one of the final states (q_7) by following the ε-transitions:



This means that the initial state no longer needs to be also a final state:



Example 2.18

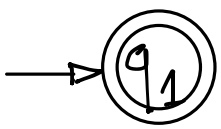
- Build the PDA automata M_2 that recognizes the language:

$$\{ ww^R \mid w \in \{0,1\}^* \}$$

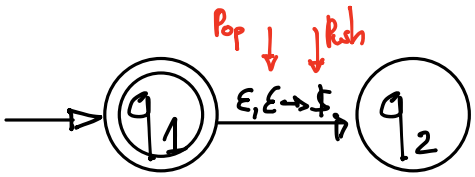
Notes:
w^R - means w written backwards

[resolution:]

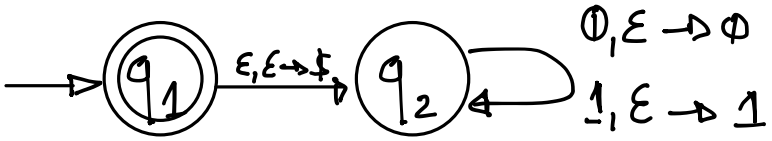
① The ε string results in an accept state:



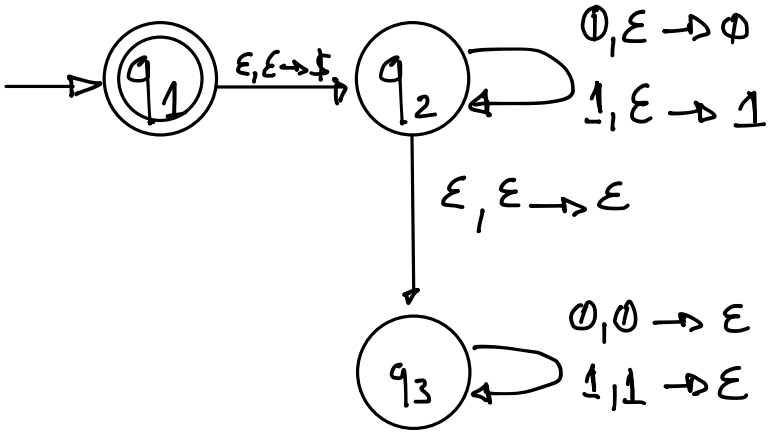
② Let's start with adding $ to the stack:



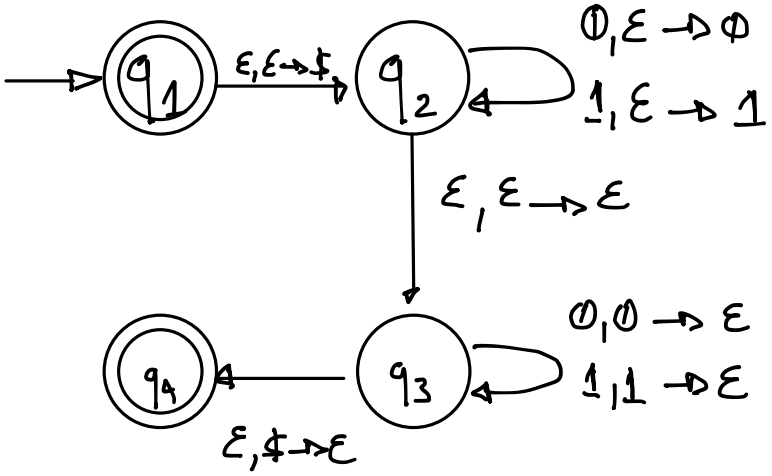
③ Let's push into the stack the symbols read:



④ We do not know when $\underline{w}$ has been fully read. Therefore, we need to always test with each step in a non-deterministic manner:



⑤ We also need to check if any branch of the computation ends the input (ε) and also results in an empty stack:



Equivalence with Context-Free Grammars

CFGs and PDAs are equivalent in power:

- Both are capable of describing the class of CFL
- Remember what is a CFL?

  CFL := Any Language that can be described with a CFG

- So, let's see why this equivalence exists :)

(For practical reasons I decided to start on the next page)

Theorem 2.20

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

"If and only if" theorems always have two directions to prove. First, we will do the easier forward direction:

Lemma 2.21

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

But later on we will also do the reverse direction:

Lemma 2.22

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

Let's start with Lemma 2.21 👍

Proof Idea:

① Let A be a CFL. From the definition we know that A has a CFG G generating it.

② We show how to convert G into an equivalent PDA P

③ P will accept input w if G generates that input, by determining whether there is a derivation for w

3.1 Recall: A derivation is simply the sequence of substitutions made as a grammar generates a string

3.2 Each step of the derivation yields an intermediate string of variable and terminals.

3.3 P is designed to determine whether some series of substitutions using the rules of G can lead from the start variable to w.

(proof idea continues on next page)

(4) One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make:

(4.1) PDA's nondeterminism allows it to guess the sequence of correct substitutions by testing all possible computational branches.

(4.2) At each step of the derivation, one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

(5) PDA P begins by:

(5.1) Writing the start variable on its stack.

(5.2) Then goes through a series of intermediate strings, making one substitution after another.

(5.3) Eventually: P may arrive at a string that contains only terminal symbols. P accepts if this string is identical to the input string w

(proof idea continues on next page)

(6) But how can PDA P store the intermediate strings?

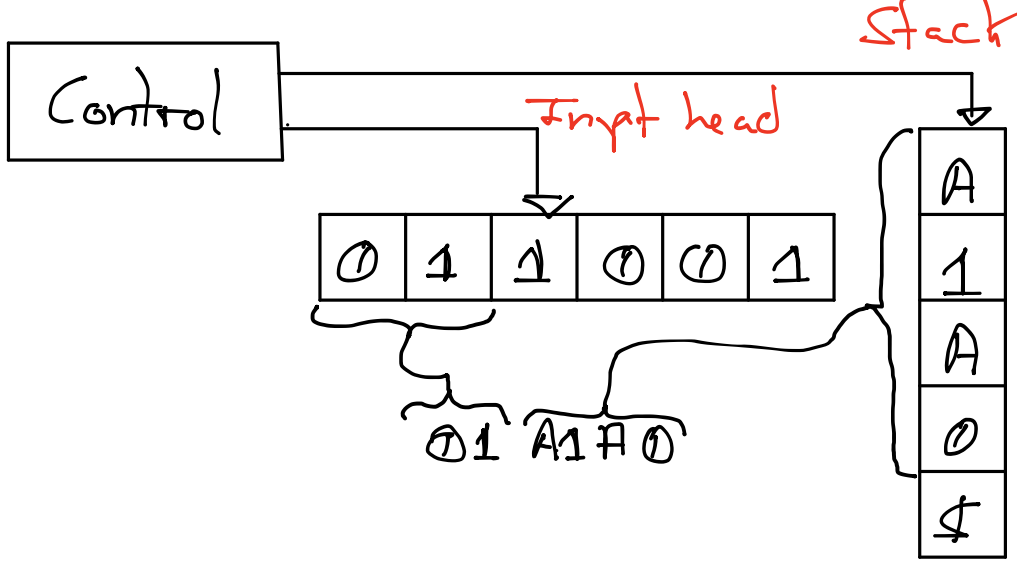
(6.1) Using the stack does not work because the PDA needs to find the variables in the intermediate string and make substitutions...

(6.2) PDA can access only the top symbol on the stack and that may be a terminal symbol instead of a variable.

(6.3) Solution: keep only part of the intermediate string on the stack. Namely, the symbols starting with the first variable in the intermediate string.

(6.4) Any terminal symbols appearing before the 1st variable are matched immediately with symbols in the input string.

PDA P: Representing intermediate string 01A1A0



(proof idea continues on next page)

(7) Informal description of P:

(7.1) Place marker symbol $ and start variable on stack.

(7.2) Repeat the following steps forever:

Notes:

Why do we need to loop forever?

Because the grammar has a recursive structure.

(7.2.a) If top of stack is variable A:

Nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

(7.2.b) If top of stack is terminal a:

Read next symbol from the input and compare it to a. If they match, repeat. Otherwise, reject this branch of nondeterminism.

(7.2.c) If top of stack is $:

Enter accept state, doing so accepts the input if it has all been read.

Proof:

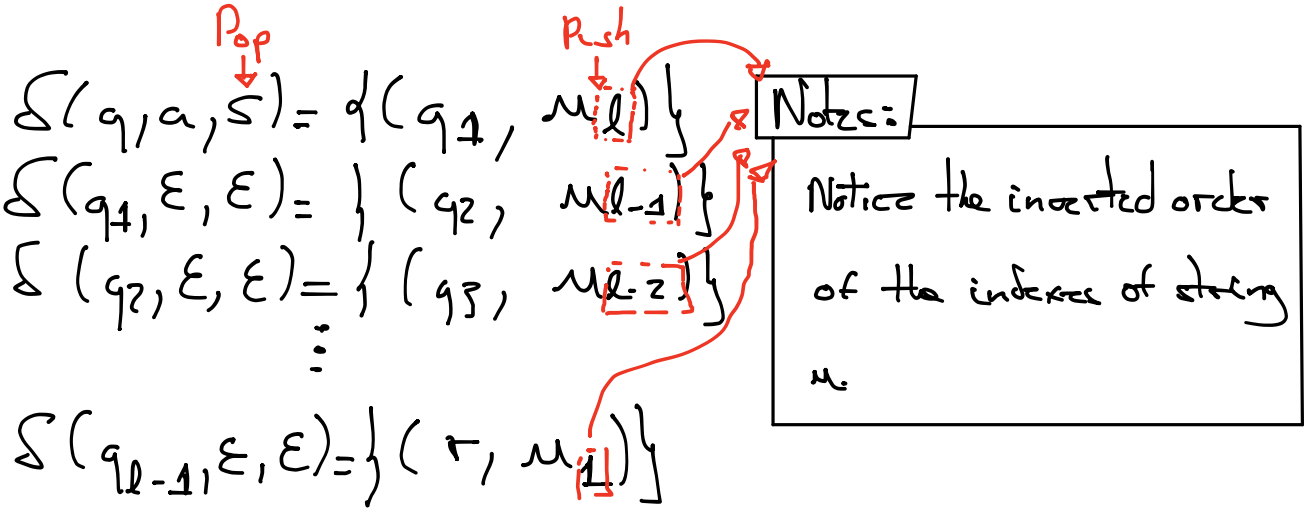
(1) We now give the formal details for PDA $P = (Q, \Sigma, \Gamma, q_{start}, F)$

(2) Let:

- $q, r \in Q$
- $a \in \Sigma_\varepsilon$
- $s \in \Gamma_\varepsilon$

(3) Say that we want the PDA to go from state q to r when it reads a and pops s. Furthermore, we want to push the entire string $u = u_1 \dots u_l$ onto the stack at the same time.

This can be done by introducing new states $E = \{q_1, \dots, q_{l-1}\}$ and setting the transition function as follows:

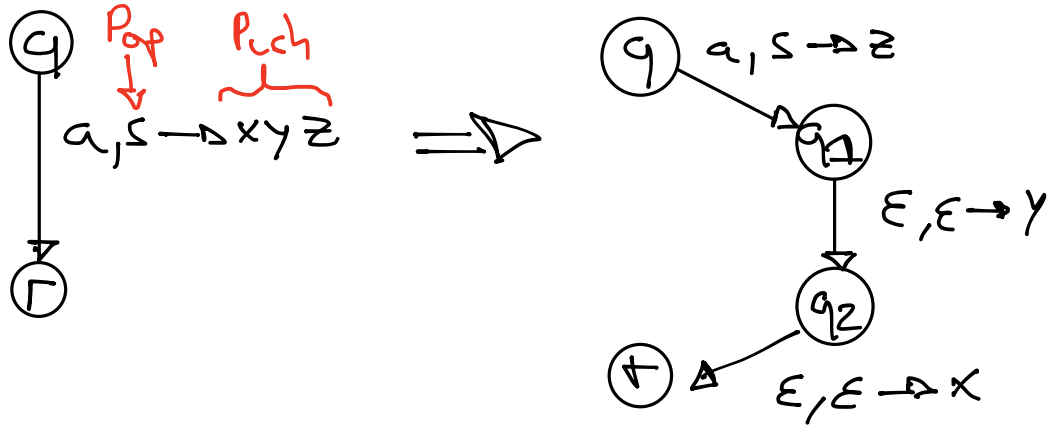


This is known as shorthand notation and is represented as

$(r, u) \in \delta(q, a, s)$

(proof continues on next page)

Shorthand notation example:



④ The states of P are $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$ where E is the set of states for implementing the shorthand notation. The start state is q_{start} and the only accept state is q_{accept}.

⑤ δ is defined as follows: (see informal definition, page 49)

5.1 Place marker symbol \$ and start variable on stack:

Input symbol, Pop, Push

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

(proof continues on next page)

(5.2) Repeat the following steps forever (q_{loop}):

(5.2.a) If top of stack is variable A:

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

(5.2.b) If top of stack is terminal a:

Read next symbol from the input and compare it to a. If they match, repeat. Otherwise, reject this branch of nondeterminism.

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

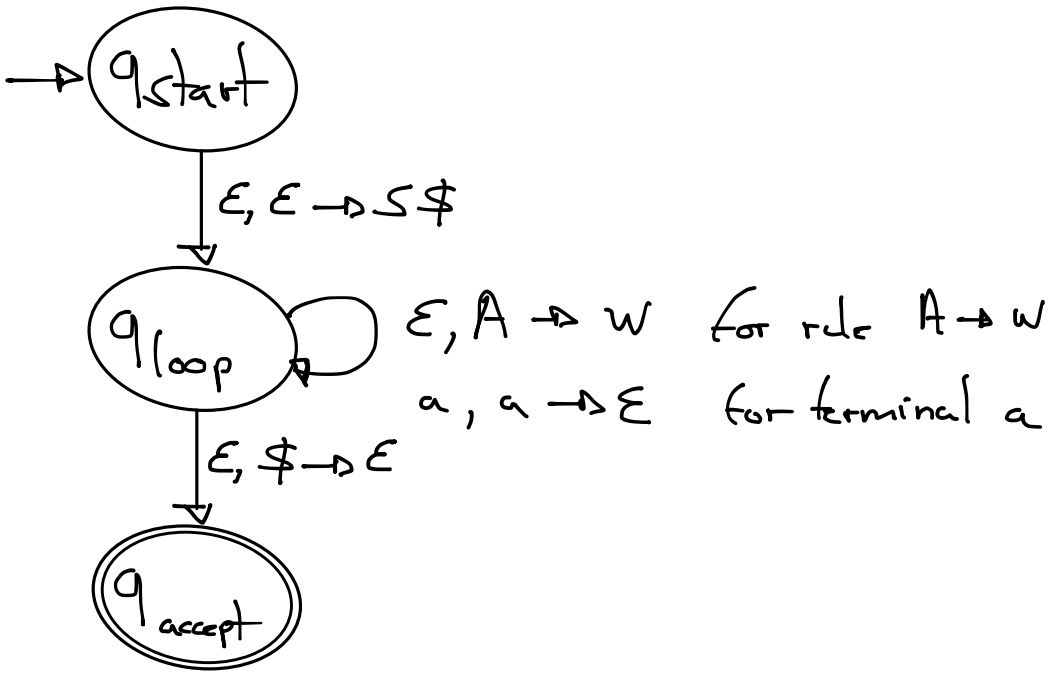
(7.2.c) If top of stack is $\$$:

Enter accept state, doing so accepts the input if it has all been read.

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

(proof continues on next page)

∴ State diagram of P:



 (end of proof)

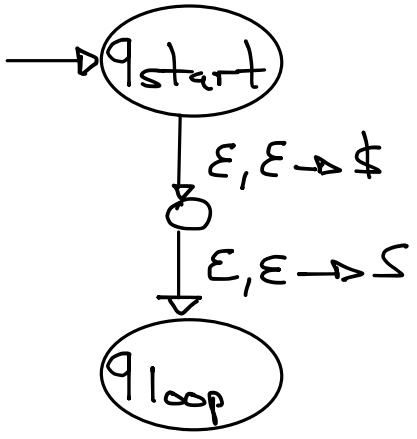
Example 2.25

Use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G:

$$S \to aTb \mid b$$
$$T \to Ta \mid \varepsilon$$

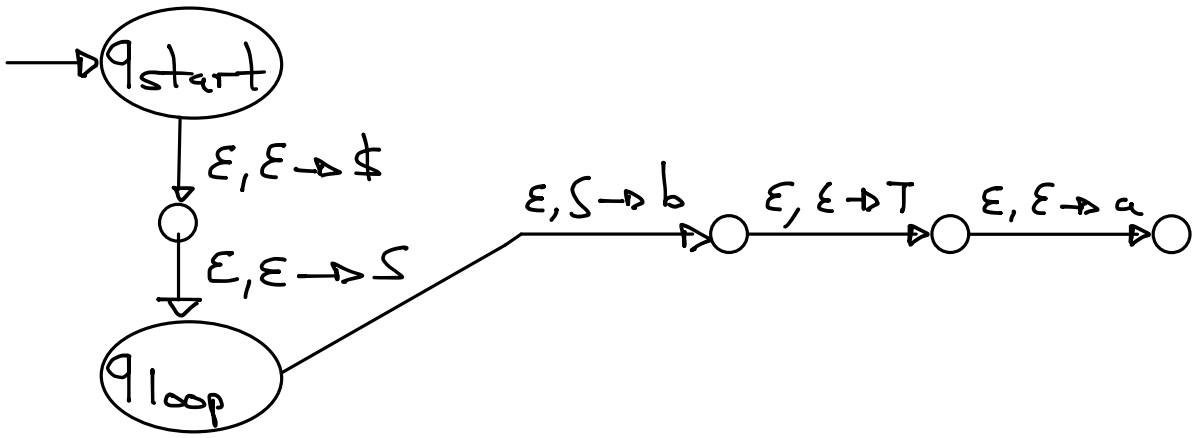
[resolution:] $P_1 = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$

(1) Lets start by pushing $S\$$ onto the stack:

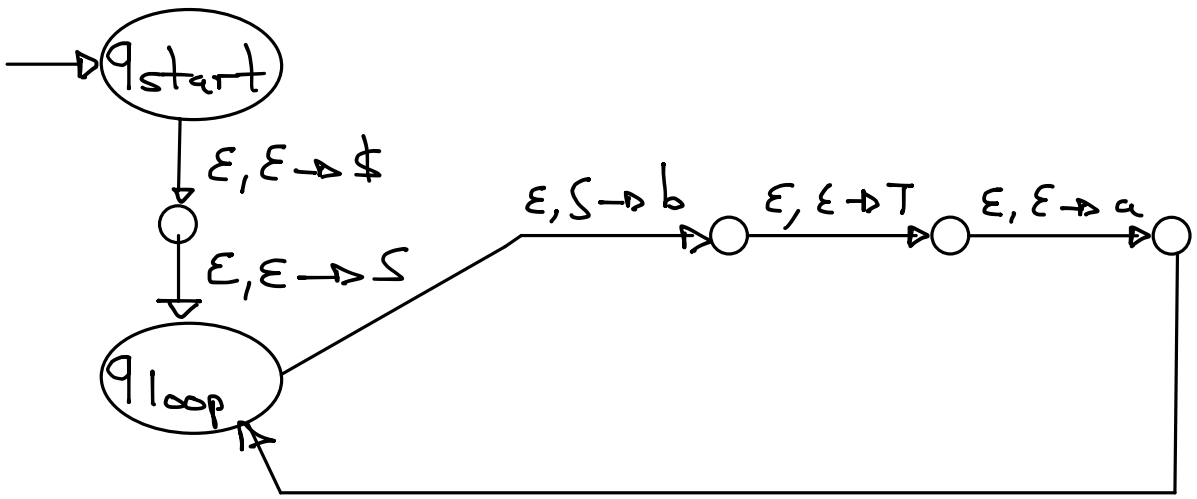


(2) Now lets care of the rule $S \to aTb$:

This means that when an S is popped from the stack we need to write the string "aTb" onto the stack.

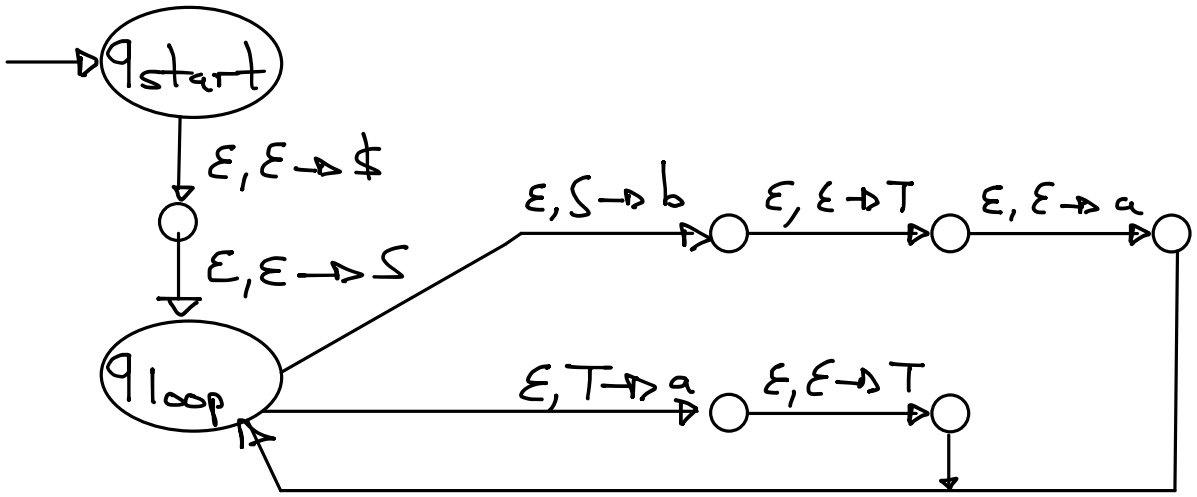


(3) Because of the recursive nature of the grammar we need to make sure that these strings representing the rule can be substituted in a loop:

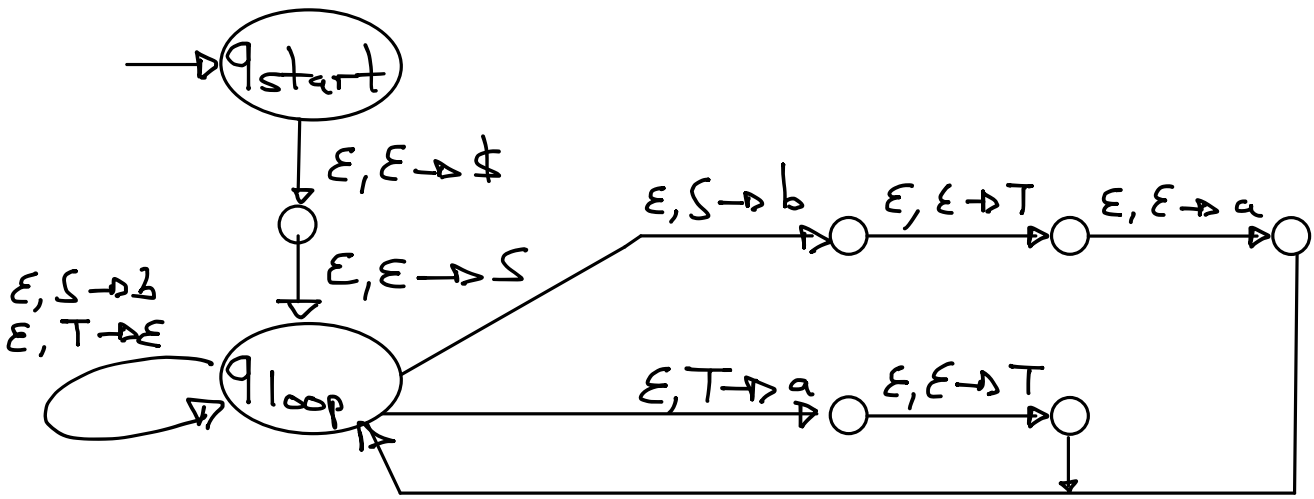


④ Now lets care of the rule $T \to Ta$:

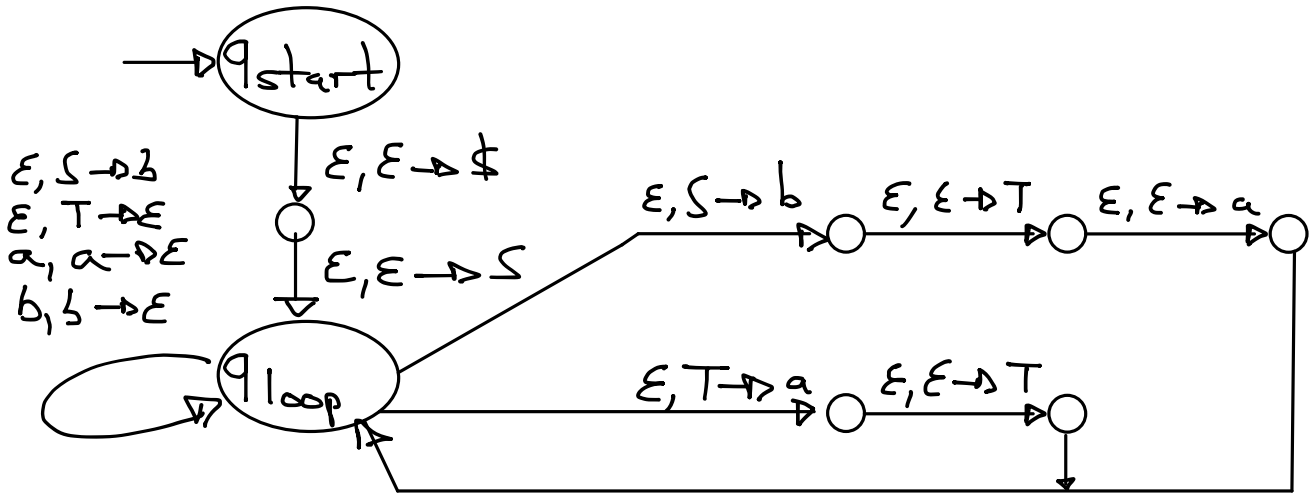
This means that when an T is popped from the stack we need to write the string "Ta" onto the stack. (lets also do the loop part)



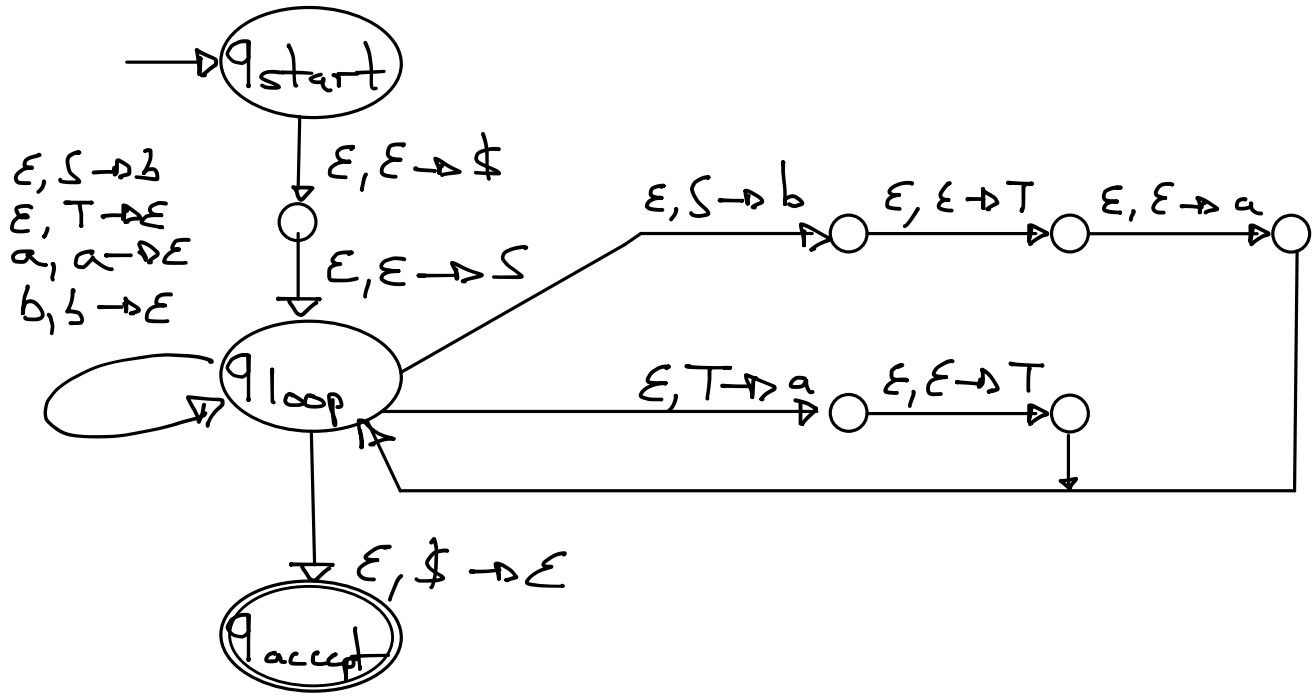
⑤ Lets take care of the rules $S \to b$ and $T \to \varepsilon$:



⑥ Now we need to take care of what happens when a terminal symbol is read:



⑦ Finally we add the accept state:



(end of the example)

Now we need to prove the inverse direction of Theorem 2.20 (page 47):

- Forward direction: Convert a CFG to a PDA
- Reverse direction: Convert a PDA to a CFG

Lemma 2.22

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

Proof Idea:

(1) Convert a PDA P to a CFG G that generates all the strings that P accepts.

(2) We design a grammar that does somewhat more:

(2.1) For each pair of states p and q in P the grammar will have a variable A_{pq}.

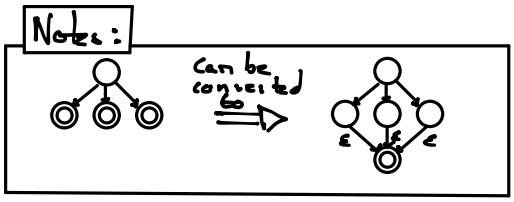
(2.2) A_{pq} generates all strings that can take P from p with an empty stack to q with an empty stack.

(2.3) Such string 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.

(proof idea continues on next page)

(3) First, we simplify our task by modifying P slightly to give it the following three features:

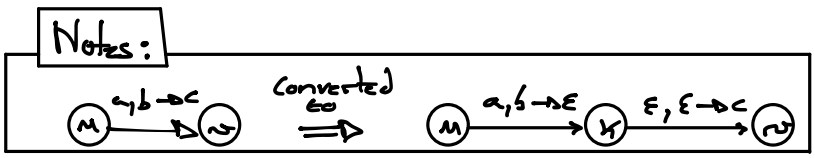
(3.1) It has a single accept state



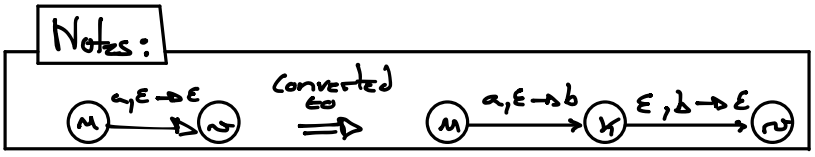
(3.2) It empties its stack before accepting

(3.3) Each transition either pushes a symbol or pops one off the stack, but it does not do both at the same time:

(3.3.a) This requires replacing each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state.



(3.3.b) In addition, each transition that neither pops nor pushes is replaced with a two transition sequence that pushes then pops an arbitrary stack symbol.



(proof idea continues on next page)

(4) Recall: A_{pq} generates all strings that can take P from p with an empty stack to q with an empty stack.

(4.1) For any string x, P's first move must be a push:

Because every move is either a push or a pop and P can't pop an empty stack.

(4.2) Similarly, the last move on x must be a pop:

For the stack to end up empty, two possibilities occur:

(4.2.1) The symbol popped at the end is the symbol that was pushed at the beginning.

Accordingly, the stack can only be empty at the beginning and end of P's computation. We simulate this with the rule:

$$A_{pq} \rightarrow a\, A_{rs}\, b$$

where:

a := input read at the first move
b := input read at the last move
r := state following state p
s := state preceding state q

(proof idea continues on next page)

(4.2.2) The symbol popped at the end is not the symbol that was pushed at the beginning.

Accordingly, the initially pushed symbol must be popped at some point before the end of string x and thus the stack becomes empty at this point. This can be simulated with the rule:

$$A_{pq} \rightarrow A_{pr} A_{rq}$$

where:

$r :=$ state when stack becomes empty.

(end of proof idea)

Proof:

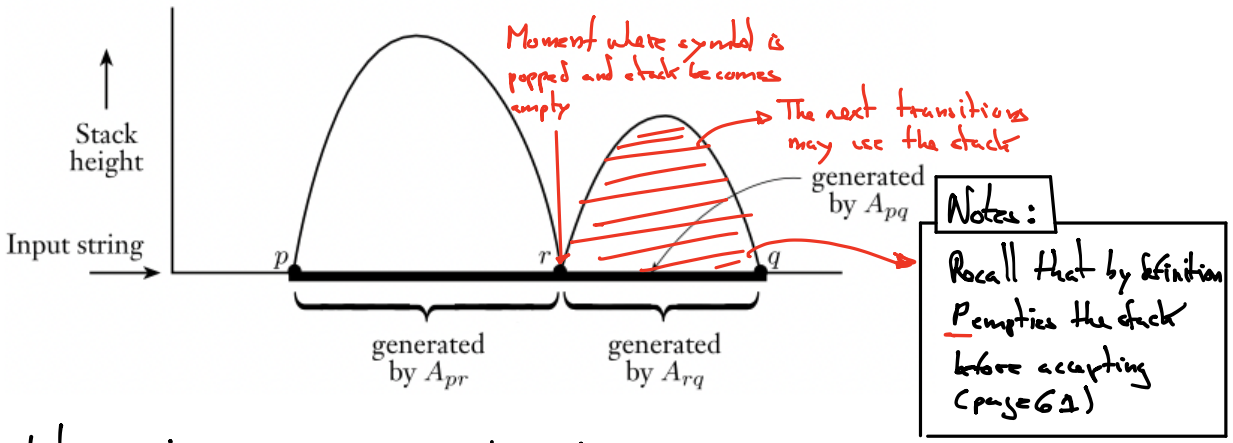
① Given PDA $P = (Q, \Sigma, T, \delta, q_0, \{q_{accept}\})$ construct G.

② Variables of G are $\{A_{pq} \mid p, q \in Q\}$ (All possible pairs of states)

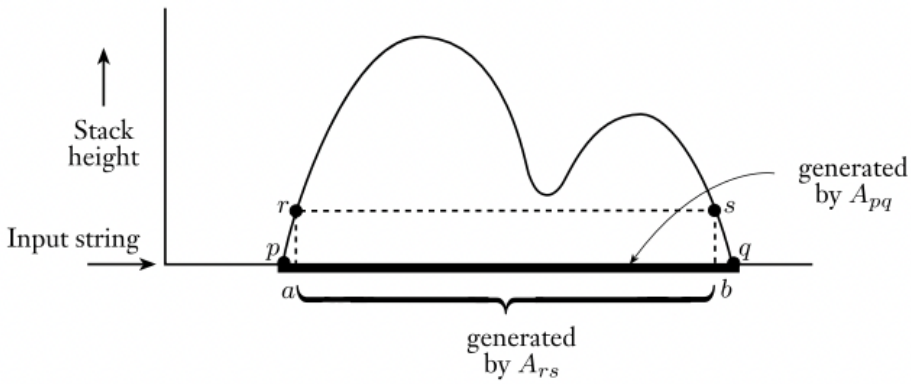
③ Start variable is $A_{q_0, q_{accept}}$, i.e. we want to go from the initial state to the final state.

④ Let's gain some insights from the following figures:

④.① If the initially pushed symbol is popped early:



④.② If the last symbol popped is the first symbol pushed:



(5) G's rules can then be described in three parts:

(4.1) For each $p, q, r, s \in Q$, $u \in \Gamma$ and $a, b \in \Sigma_\varepsilon$, if:

- $\delta(p, a, \varepsilon) = (r, u)$
- $\delta(s, b, u) = (q, \varepsilon)$

} Then put rule $A_{pq} \to a A_{rs} b$ in G

State, Input Symbol, Pop, State, Push

The stack starts empty

The stack ends empty

(4.2) For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr} A_{rq}$ in G

(4.3) Finally, for each $p \in Q$, put the rule $A_{pp} \to \varepsilon$ in G

Let's see why this construction works by demonstrating that A_{pq} generates x iff x can bring P from p with empty stack to q with empty stack. Because of the iff we need to consider two claims:

Claim 2.30

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

Claim 2.31

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

Let's start by proving claim 2.30 first ⇓

Claim 2.30

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

Proof:

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

Notes:
A grammar always needs to apply at least one rule.

(2) **Basis:** When the derivation of the rules has 1 step:

(2.1) A derivation with a single step must use a rule whose right-hand side contains no variables.

(2.2) The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \varepsilon$.

(2.3) Clearly, input ε takes P from p with empty stack to p with empty stack, so the basis is proved.

(3) **Induction:**

(3.1) Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k+1$.

derives

(3.?) Suppose $A_{pq} \overset{*}{\Rightarrow} x$ with $k+1$ steps. Then we have two possibilities:
$A_{pq} \Rightarrow a A_{rs} b$ or $A_{pq} \Rightarrow A_{pr} A_{rq}$

(3.3.a) Lets focus on $A_{pq} \Rightarrow a A_{rs} b$:

(3.3.a.1) Consider the portion y of x that A_{rs} generates so $x = ayb$

(3.3.a.2) Because $A_{rs} \overset{*}{\Rightarrow} y$ with k steps the induction hypothesis tells us that P can go from r on empty stack to s on empty stack.

(3.3.a.3) Because $A_{pq} \to a A_{rs} b$ is a rule of G then $\delta(p, a, \varepsilon) = (r, u)$ *1 and $\delta(s, b, u) = (q, \varepsilon)$ *2 for some stack symbol u

(3.3.a.4) If P starts at p with an empty stack, after reading a it can go to state r and push u onto the stack. See *1

(3.3.a.5) Then reading y can bring it to s and leave u on the stack.

(3.3.a.6) Then after reading b it can go to state q and pop u off the stack. See *2

$\therefore_\Delta$ X can bring P from p with empty stack to q with empty stack.

3.3.5 Now, let's consider $A_{pq} \Rightarrow A_{pr} A_{rq}$:

3.3.5.1 Consider the portions y and z of x that A_{pr} and A_{rq} generate so $x = yz$.

3.3.5.2 Because $A_{pr} \overset{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps, the induction hypothesis tells us that y can bring P from p to r, and z can bring P from r to q, with empty stacks at the beginning and end.

$\therefore_2$ x can bring P from p with empty stack to q with empty stack.

∎

Now let's prove Claim 2.3 ✌

Claim 2.31

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

Proof:

(1) We prove this claim by induction on the number of steps in the computation of P that goes from p to q with empty stack on input x.

Notes:
The computation represented by an automaton can have 0 steps so there should be the basis case.

(2) **Basis:** The computation has 0 steps.

(2.1) If a computation has 0 steps, it starts and ends at the same state, say p. So we must show that $A_{pp} \overset{*}{\Rightarrow} x$.

(2.2) In 0 steps, P cannot read any characters, so $x = \varepsilon$.

(2.3) By definition, G has the rule $A_{pp} \to \varepsilon$, so the basis is proved.

(3) **Induction:**

(3.1) Assume true for computations of length at most k, where $k \geq 0$ and prove true for computations of length $k+1$

(3.2) Suppose that P has a computation wherein x brings p to q with empty stacks in $k+1$ steps. Then we have two possibilities: $A_{pq} \Rightarrow a A_{rs} b$ or $A_{pq} \Rightarrow A_{pr} A_{rq}$

(3.2.a) Let's focus on $A_{pq} \Rightarrow a A_{rs} b$:

(3.2.a.1) The symbol that is pushed at the first move must be the same as the symbol that is popped at the last move. Call this symbol u

(3.2.a.2) Let: a be the input read on 1st move
b be the input read on last move
r be the state after the 1st move
s be the state before the last move

(3.2.a.3) Then: $\delta(p, a, \varepsilon) = (r, u)$
$\delta(s, b, u) = (q, \varepsilon)$
Therefore, rule $A_{pq} \to a A_{rs} b \in G$

(3.2.a.4) Let y be the portion of x without a and b, so $x = ayb$

(3.2.a.5) Input y can bring P from r to s without touching the symbol u that is on the stack

3.2.a.6 Therefore, P can go from r with an empty stack to s with an empty stack on input y.

3.2.a.7 We have performed until now two computational steps:

1st step: $\delta(p, c, \varepsilon) = (r, u)$

2nd step: $\delta(s, b, u) = (q, \varepsilon)$

This means that the first and last steps of the $k+1$ steps in the original computation on x so the computation on y has $(k+1) - 2 = k-1$ steps.

3.2.a.8 Thus, the induction hypothesis tells us that $A_{rs} \overset{*}{\Rightarrow} y$.

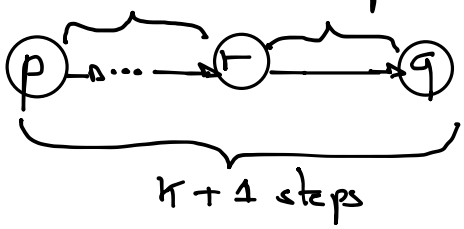
3.2.a.9 This means that: $A_{pq} \to a A_{rs} b \to a y b \to x$ } i.e. $A_{pq} \overset{*}{\Rightarrow} x$

(3.2.5) Let's focus on $A_{pq} \Rightarrow A_{pr} A_{rq}$:

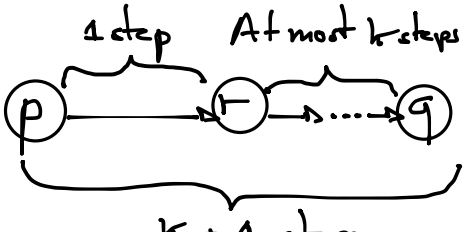
(3.2.5.1) Let r be a state where the stack becomes empty other than at the beginning or end of the computation on x.

(3.2.5.2) Then the portions of the computation from p to r and from r to q each contain at most k steps:

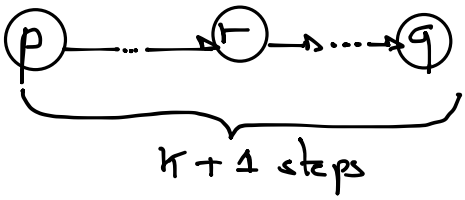
Hypothesis 1: At most k steps, 1 step



Hypothesis 2: 1 step, At most k steps



Hypothesis 3: Something in the "middle" of Hypothesis 1 and Hypothesis 2



(3.2.5.3) Say that y is the input read during the 1st portion and z is input read during the 2nd portion, i.e. $x = yz$.

(3.2.5.4) The induction hypothesis tells us that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$.

(3.2.5.5) Because rule $A_{pq} \to A_{pr}A_{rq}$ is in G:

$A_{pq} \to A_{pr}A_{rq}$
$\to y^z$
$\to x$

$\Big\}$ $A_{pq} \overset{*}{\Rightarrow} x$ ▨

Q: Ok, let's recapitulate, what did we just prove?

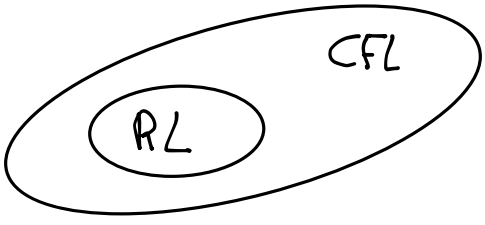
↳ ∴ PDAs recognize the class of CFLs

It now becomes possible to establish a relationship between regular languages and context-free languages:

① Every RL is recognized by a finite automaton

② Every finite automaton is a PDA that simply ignores its stack

$\Big\}$ ∴ Every RL is also CFL



2.3 Non-Context-Free Languages

Recall: In Section 1.4 we introduced the **pumping lemma** for showing that certain languages are not regular.

This section: Presents a similar pumping lemma for context-free languages. I.e. certain languages are not context-free.

Q: What is a context-free language?

→ Language that is generated by a CFG.

Every CFL has a special value called the **pumping length** such that all longer strings in the language can be "pumped":

- This time the meaning of pumped is a bit more complex.

- It means that the string can be divided into five parts.

- The 2nd and 4th parts may be repeated together any number of times and the resulting string still remains in the language.

(perfect description of the PL for CFL)

The pumping lemma for context-free languages

Theorem 2.34

Pumping lemma for context-free languages: If A is a CFL 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:

$$s = u\,v\,x\,y\,z$$

Satisfying the conditions:

1. for each $i \geq 0$, $u\,v^i\,x\,y^i\,z \in A$
2. $|vy| > 0$, and
3. $|vxy| \leq p$

Let's see why this is true !!

Proof Idea:

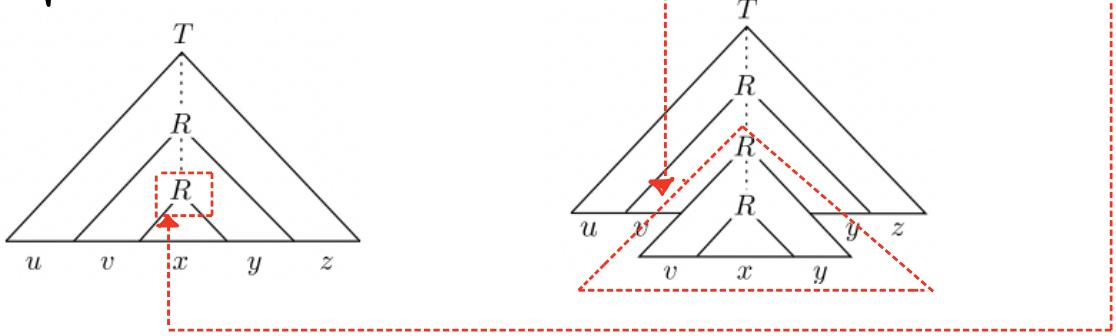
(1) Let A be a CFL and let G be a CFG that generates it. Show that any sufficiently long string $s \in A$ can be pumped and remain in A.

(3) Let s be a very long string in A:

(3.1) Because $s \in A$, it is derivable from G and so has a parse tree (A Árvore Sintática)

(3.2) Parse tree for s must be very tall because s is very long. That is, the parse tree must contain some long path from the start variable at the root of the tree to one of the terminal symbols at a leaf.

(3.3) On this long path, some variable symbol R must repeat. As the following figure shows, this repetition allows us to replace the subtree under the 2nd occurrence of R with the subtree under the 1st occurrence of R and still get a legal parse tree:



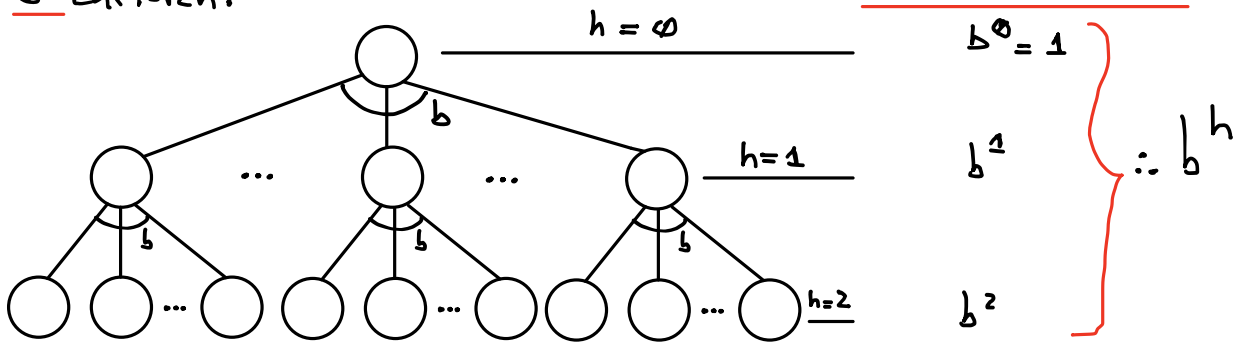
Notice that this procedure turned $uvxyz$ into uv^2xy^2z

(3.4) $\therefore uv^ixy^iz \in A \quad \forall i \geq 0$

Proof:

① Let G be a CFG for CFL A. Let b be the maximum number of symbols in the right-hand side of a rule (assume at least 2).

② In any parse tree using this grammar, we know that a node can have no more than b children:



③ ∴1 The length of the string generated is at most b^h

∴2 If a generated string has length at least b^h+1 each of its parse trees must be at least $h+1$ high.

④ Let $|V|$ be the number of variables in G. We set p (pumping length) to be $b^{|V|+1}$.

⑤ If $s \in A$ and $|s| \geq p$, its parse tree must be at least $|V|+1$ high, since:

$$|s| \geq p \iff |s| \geq b^{|V|+1} \Rightarrow b^h \geq b^{|V|+1} \iff h \geq |V|+1$$

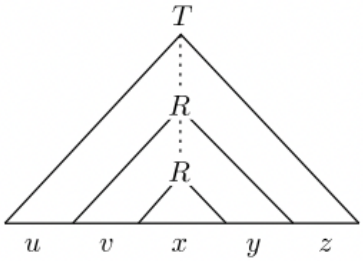
I.e. for the parse tree to generate a string of length at least $b^{|V|+1}$ the height of the tree needs to be at least $|V|+1$.

⑥ To see how to pump any such string let T be one of its parse trees. If s has several parse trees, choose T to be a parse tree that has the smallest number of nodes. We know that T must be at least $|V|+1$ high, this means that:

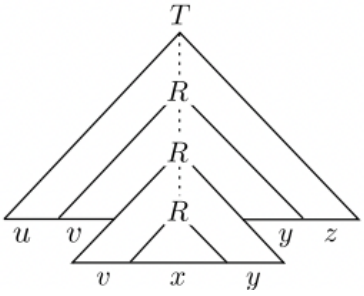
- Longest path from root to a leaf has length at least $|V|+1$
- Longest path from root to a leaf has at least $|V|+2$ nodes.
- Longest path from root to a leaf consists of nodes representing variables and terminals. The first nodes are all variables and the last node is a terminal. If this path has $|V|+2$ nodes, then this means that $|V|+1$ variables exist.

⑦ Recall that $|V|$ is the number of variables in G, if the longest path has $|V|+1$ variables, then this means that some variable R appears more than once on that path. For convenience, we select R to be a variable that repeats among the lowest $|V|+1$ variables on this path.

(8) We divide s into $u\ v\ x\ y\ z$ according to the figure:

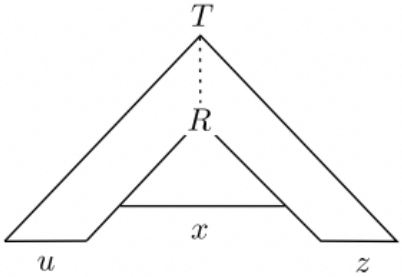


- Each occurrence of R has a subtree, generating a part of string s.
- Upper occurrence of R generates $v\ x\ y$
- Lower occurrence of R generates x.
- Both subtrees are generated by the same variable, so we may substitute one for the other and still obtain a valid parse tree:



$\therefore_1$ Replacing the smaller subtree by the larger repeatedly gives parse trees for the strings: uv^ixy^iz $\forall i > 1$.

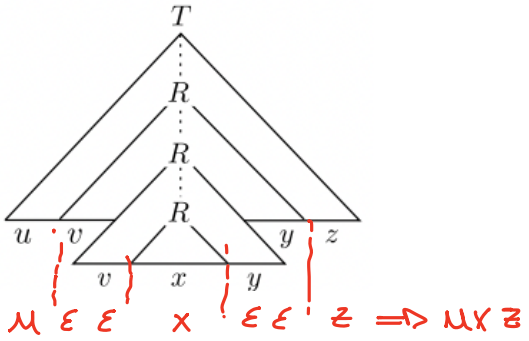
$\therefore_2$ Replacing the larger by the smaller generates string uxz:



$\therefore_3$ If $\therefore_1$ and $\therefore_2$ are combined we get condition 1 of the PL:

$$uv^ixy^iz \quad \forall i \geq 0$$

(9) To get condition 2 we must be sure that v and y are not both ε. If they were, the parse tree obtained by substituting the smaller subtree for larger one:



This tree would have fewer nodes than T and would still generate s which is a contradiction.

(10) To get condition 3 we need to be sure that vxy has length at most p. In the parse tree for s the upper occurrence of R generates vxy. We chose R so that both occurrences fall within the $|V|+1$ variables on the path, and we chose the longest path in the parse tree, so the subtree where R generates vxy is at most $|V|+1$ high. A tree of this height can generate a string of length at most $b^{|V|+1} = p$ (which we defined to be the pumping length).

Example 2.36

Use PL to show that the language $B = \{a^n b^n c^n \mid n \geq 0\}$ is not a CFL.

Execution :)

① Assume B is CFL and obtain a contradiction

② Let p be the pumping length for B that is guaranteed to exist by the PL.

③ Select the string $s = a^p b^p c^p$ (i.e. $s \in B$ and $|s| \geq p$)

④ No matter how s is divided into $u\,v\,x\,y\,z$ one of the conditions of the PL is violated

⑤ By condition 2: $|vy| > 0$, i.e. either v or y is nonempty. Let's consider whether v and y contain only one type of alphabet symbol or more:

⑤.1 Case 1: Both v and y contain only one type of alphabet symbol, i.e. v does not contain both a's and b's or both b's and c's (same for y).

Example:

$p = 2$: $s = a^2 b^2 c^2 = \overset{u}{a}\overset{v}{a}\overset{x}{b}\overset{y}{b}\overset{z}{c}c$. According to PL $u\,v^i x\,y^i z \in B$. However, this is not true since we will obtain strings where the number of a's and c's will not match the number of b's. (contradiction)

(5.2) **Case 2:** Both v and y contain more than one type of symbol

Example:

$p=3$: $s = a^3b^3c^3 = aaa\,bbb\,ccc$ (u = aaa, v = bbb, x = ccc, y = ccc, z = ccc) According to PL $uv^ixy^iz \in B$.

However, this is not true since we will obtain strings where the the order of the symbols is not correct. (contradiction)

Example 2.37

Use PL to show that the language $C = \{a^i b^j c^k \mid 0 \le i \le j \le k\}$ is not a CFL.

Execution :)

① Assume C is CFL and obtain a contradiction

② Let p be the pumping length for C that is guaranteed to exist by the PL.

③ Select the string $s = a^p b^p c^p$ (i.e. $s \in C$ and $|s| \ge p$)

④ No matter how s is divided into $uvxyz$ one of the conditions of the PL is violated

⑤ By condition 2: $|vy| > 0$, i.e. either v or y is nonempty. Let's consider whether v and y contain only one type of alphabet symbol or more:

⑤.1 Case 1: Both v and y contain only one type of alphabet symbol, i.e. v does not contain both a's and b's or both b's and c's (same for y).

Example:

$p = 2$: $s = a^2 b^2 c^2 = \overset{u}{|}\overset{v}{|}\overset{x}{|}\overset{y}{|}\overset{z}{|}\,aabbcc$. According to PL $uv^i xy^i z \in C$. However, this is not true since we will obtain strings where the number of a's and c's will be greater than the number of b's. (contradiction)

(5.2) **Case 2:** Both v and y contain more than one type of symbol

Example:

p=3: $s = a^3b^3c^3 = aaabbbccc$ According to PL $uv^ixy^iz \in C$.

However, this is not true since we will obtain strings where the the order of the symbols is not correct. (contradiction)

Example 2.38

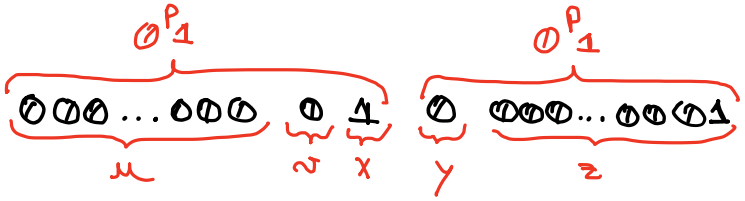
Use PL to show that the language $D = \{ww \mid w \in \{0,1\}^*\}$ is not a CFL.

Resolution :)

① Assume D is CFL and obtain a contradiction

② Let p be the pumping length for D that is guaranteed to exist by the PL.

③ Select the string $s = 0^p 1 0^p 1$ (i.e. $s \in D$ and $|s| \geq p$). This string is not a good candidate since it can be pumped as follows:



④ Let's try another candidate: $s = 0^p 1^p 0^p 1^p$ (i.e. $s \in D$ and $|s| \geq p$)

⑤ No matter how s is divided into $uvxyz$ one of the conditions of the PL is violated

⑥ Example

$p=2$: $0^2 1^2 0^2 1^2 = \underbrace{00}_{u}\underbrace{11}_{v}\underbrace{0}_{x}\underbrace{0}_{y}\underbrace{11}_{z}$ According to PL $uv^i xy^i z \in B$.

Left side Right side

However, this is not true since the number of 1's in the left side will not match the number of 1's in the right side. Similarly, the number of 0's in the right side will not match the number of 0's in the left side.

2.4 Deterministic Context-Free Languages