Introduction to Languages and Computation

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Context-Free Languages and Pushdown Automata

- We introduced finite automata and regular expressions.
- We now present context-free grammars, a more powerful method of describing languages.
- They were first used in the study of human languages.
- Another important application occurs in the specification and compilation of programming languages. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. There are tools that can automatically generate the parser from the grammar.
- The collection of languages associated with context-free grammars are called the context-free languages. They include all the regular languages and many additional languages.
- Pushdown automata is a class of machines recognizing the context-free languages.

Subsection 1

Context-Free Grammars

Introducing Grammars

- The following grammar G_1 is an example of a context-free grammar:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

- A grammar consists of a collection of **substitution rules**, also called **productions**. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow:
  - The symbol is called a **variable**. The variable symbols often are represented by capital letters.
  - The string consists of variables and other symbols called **terminals**. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols.
- One variable is designated as the **start variable**. It usually occurs on the left-hand side of the topmost rule.
- Example: Grammar G_1 contains three rules. G_1's variables are A and B; A is the start variable. Its terminals are 0, 1, #.

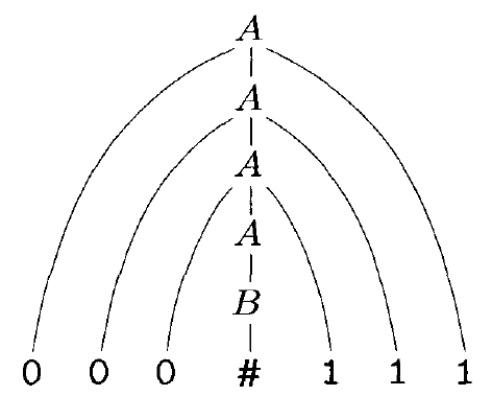
Derivations in a Grammar

- Grammars describe a language by generating each string of that language in the following manner:
  1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
  2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
  3. Repeat step 2 until no variables remain.
- Example: G_1 generates the string $000\#111$.
- The sequence of substitutions to obtain a string is called a **derivation**.
- Example: A derivation of $000\#111$ in G_1 is

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

Parse Trees and Languages

- The same information as in a derivation can be given with a **parse tree**:



- All strings generated in this way constitute the **language of the grammar**. We write $L(G_1)$ for the language of grammar G_1.
- In the example, $L(G_1) = \{0^n\#1^n : n \geq 0\}$.
- Any language that can be generated by some context-free grammar is called a **context-free language** (**CFL**).
- Sometimes, for convenience, we abbreviate several rules with the same left-hand variable. E.g., we write $A \to 0A1 \mid B$ in place of both $A \to 0A1$ and $A \to B$.

A Context-Free Grammar G_2

- The following is a context-free grammar G_2 which describes a fragment of the English language:

$$
\begin{array}{rcl}
\langle \text{SENTENCE} \rangle & \rightarrow & \langle \text{NOUN} - \text{PHRASE} \rangle \langle \text{VERB} - \text{PHRASE} \rangle \\
\langle \text{NOUN} - \text{PHRASE} \rangle & \rightarrow & \langle \text{CMPLX} - \text{NOUN} \rangle \mid \\
 & & \langle \text{CMPLX} - \text{NOUN} \rangle \langle \text{PREP} - \text{PHRASE} \rangle \\
\langle \text{VERB} - \text{PHRASE} \rangle & \rightarrow & \langle \text{CMPLX} - \text{VERB} \rangle \mid \\
 & & \langle \text{CMPLX} - \text{VERB} \rangle \langle \text{PREP} - \text{PHRASE} \rangle \\
\langle \text{PREP} - \text{PHRASE} \rangle & \rightarrow & \langle \text{PREP} \rangle \langle \text{CMPLX} - \text{NOUN} \rangle \\
\langle \text{CMPLX} - \text{NOUN} \rangle & \rightarrow & \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX} - \text{VERB} \rangle & \rightarrow & \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN} - \text{PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle & \rightarrow & \text{a} \mid \text{the} \\
\langle \text{NOUN} \rangle & \rightarrow & \text{boy} \mid \text{girl} \mid \text{flower} \\
\langle \text{VERB} \rangle & \rightarrow & \text{touches} \mid \text{likes} \mid \text{sees} \\
\langle \text{PREP} \rangle & \rightarrow & \text{with}
\end{array}
$$

Grammar G_2 has 10 variables, 27 terminals and 18 rules. Strings in $L(G_2)$ include

a boy sees

the boy sees a flower

a girl with a flower likes the boy

A Derivation in Grammar G_2

- The following is a derivation of the first string on this list:

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN} - \text{PHRASE} \rangle \langle \text{VERB} - \text{PHRASE} \rangle \\
&\Rightarrow \langle \text{CMPLX} - \text{NOUN} \rangle \langle \text{VERB} - \text{PHRASE} \rangle \\
&\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB} - \text{PHRASE} \rangle \\
&\Rightarrow \text{a} \langle \text{NOUN} \rangle \langle \text{VERB} - \text{PHRASE} \rangle \\
&\Rightarrow \text{a boy} \langle \text{VERB} - \text{PHRASE} \rangle \\
&\Rightarrow \text{a boy} \langle \text{CMPLX} - \text{VERB} \rangle \\
&\Rightarrow \text{a boy} \langle \text{VERB} \rangle \\
&\Rightarrow \text{a boy sees.}
\end{aligned}
$$

Context-Free Grammars

Definition (Context-Free Grammar)

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

1. V is a finite set of **variables**,
2. Σ is a finite set, disjoint from V, of **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**.

- 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 $uAv \Rightarrow uwv$.
- 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 $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.
- The **language of the grammar** is $\{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$.

Examples

- In grammar G_1,
  - $V = \{A, B\}$,
  - $\Sigma = \{0, 1, \#\}$,
  - $S = A$,
  - R is the collection of the three rules $A \to 0A1$, $A \to B$, $B \to \#$.
- In grammar G_2,
  - $V = \{\langle \text{SENTENCE} \rangle, \langle \text{NOUN} - \text{PHRASE} \rangle, \langle \text{VERB} - \text{PHRASE} \rangle, \langle \text{PREP} - \text{PHRASE} \rangle, \langle \text{CMPLX} - \text{NOUN} \rangle, \langle \text{CMPLX} - \text{VERB} \rangle, \langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{PREP} \rangle, \langle \text{VERB} \rangle\}$,
  - $\Sigma = \{a, b, c, \ldots, z, \text{“ ”}\}$. The symbol “ ” is the blank symbol, placed invisibly after each word (a, boy, etc.), so the words not run together.
- Often we specify a grammar by writing down only its rules:
  - Variables are the symbols that appear on the left-hand side of the rules;
  - Terminals are the remaining symbols.
  - The start variable is the variable on the left-hand side of the first rule.

Grammars G_3 and G_4

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

$$S \to aSb|SS|\varepsilon.$$

This grammar generates strings such as *abab*, *aaabbb*, and *aababb*. We can understand more easily the language of this grammar by thinking of a as a left parenthesis and b as a right parenthesis. Viewed in this way, $L(G_3)$ consists of all strings of properly nested parentheses.

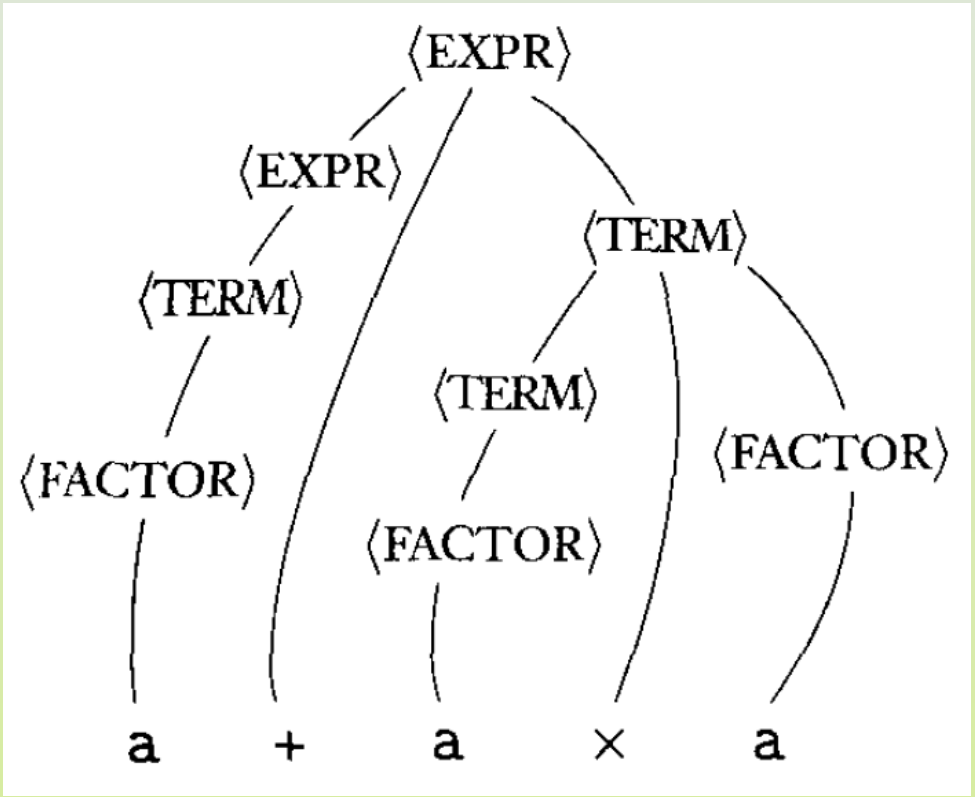
- Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.
  - $V = \{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$,
  - $\Sigma = \{a, +, \times, (,)\}$.
  - The rules are

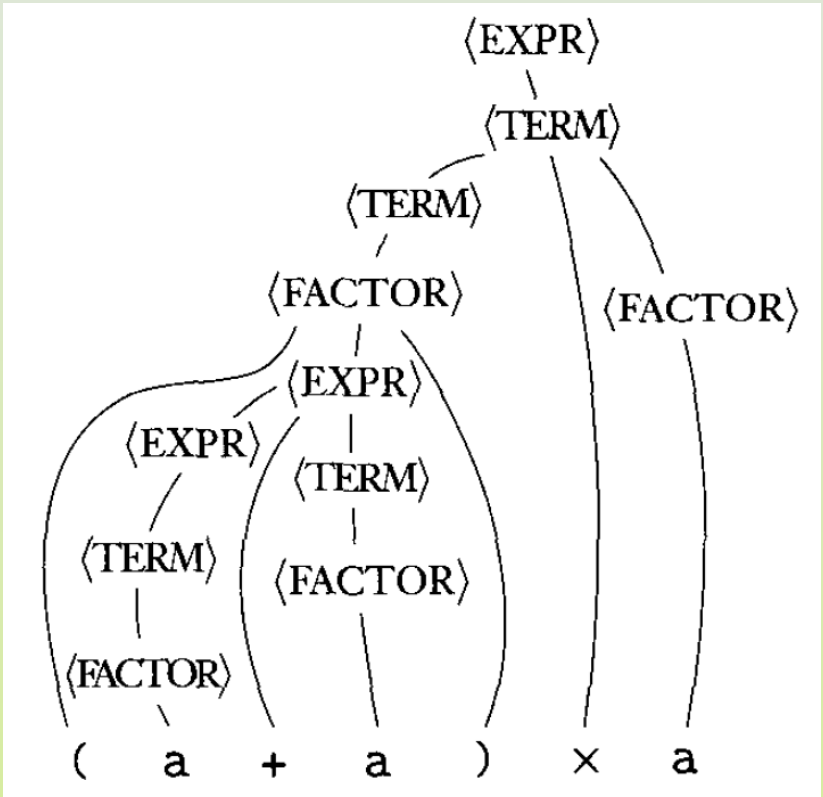
$$\begin{aligned} \langle \text{EXPR} \rangle &\to \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\to \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\to (\langle \text{EXPR} \rangle) \mid \text{a} \end{aligned}$$

The two strings $a + a \times a$ and $(a + a) \times a$ can be generated with grammar G_4.

Two Parse Trees in G_4

- The left tree is a parse tree for $a + a \times a$ in G_4:





- The right tree is a parse tree for $(a + a) \times a$ in G_4.

Designing a CFG: Breaking into Pieces

- Many CFLs are the union of simpler CFLs. To construct a CFG for a CFL that can be broken into simpler pieces, we construct individual grammars for each piece. These grammars can be merged into a single grammar by combining their rules and then adding the new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars.
- Example: To get a grammar for the language $\{0^n1^n : n \geq 0\} \cup \{1^n0^n : n \geq 0\}$, construct:

$$\begin{aligned} S_1 &\to 0S_11 \mid \varepsilon, \quad \text{for the language } \{0^n1^n : n \geq 0\} \\ S_2 &\to 1S_20 \mid \varepsilon, \quad \text{for the language } \{1^n0^n : n \geq 0\}. \end{aligned}$$

Then add the rule $S \to S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to 0S_11 \mid \varepsilon \\ S_2 &\to 1S_20 \mid \varepsilon \end{aligned}$$

Designing a CFG: Regular Languages

- Constructing a CFG for a language that happens to be regular is easy.
  - First construct a DFA for that language.
  - Then, convert the DFA into an equivalent CFG as follows:
    - Make a variable R_i for each state q_i of the DFA.
    - Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA.
    - Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA.
    - Make R_0 the start variable of the grammar, where q_0 is the start state of the machine.

    The resulting CFG generates the same language that the DFA recognizes.

Designing a CFG: Linked Strings

- Certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring.
- This situation occurs in the language $\{0^n1^n : n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s.
- You can construct a CFG to handle this situation by using a rule of the form

$$R \to uRv,$$

which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Designing a CFG: Recursive Patterns

- Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other structures.
- That situation occurs in the grammar that generates arithmetic expressions in the preceding example.
- Any time the symbol a appears, an entire parenthesized expression might appear recursively instead.
- 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.

  The grammar

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \times \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid \text{a} \end{aligned}$$

  is of this type.

The Issue of Ambiguity

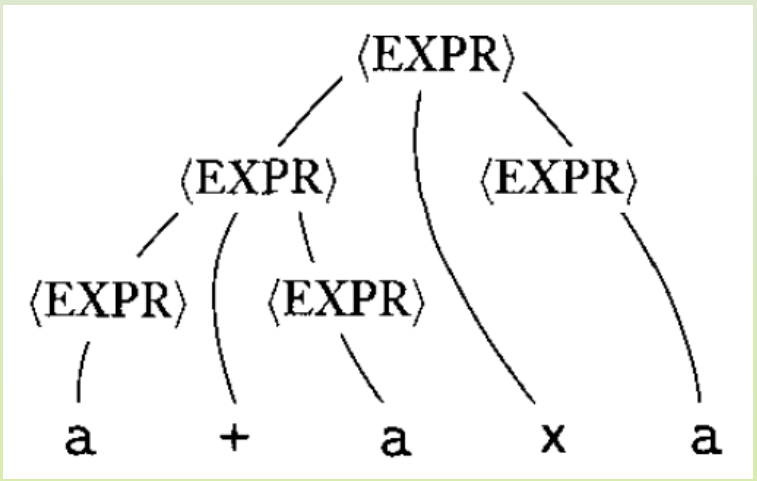
- Sometimes a grammar can generate the same string in several different ways.
- Such a string will have several different parse trees and thus several different meanings.
- This result may be undesirable for certain applications, such as programming languages, where a given program should have a unique interpretation.
- If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar.
- If a grammar generates some string ambiguously we say that the grammar is ambiguous.

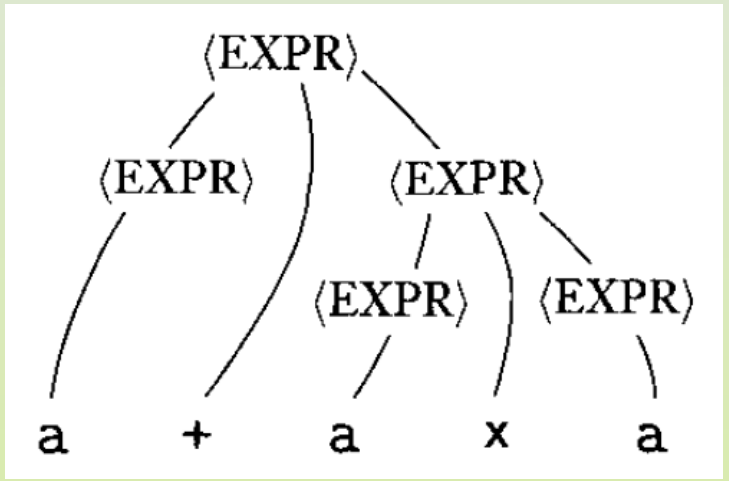
Example of an Ambiguous Grammar

- Consider grammar G_5:

  $$\langle \text{EXPR} \rangle \to \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid a$$

  This grammar generates the string $a + a \times a$ ambiguously:





  This grammar does not capture the usual precedence relations and so may group the $+$ before the $\times$ or vice versa.
- In contrast grammar G_4 generates exactly the same language, but every generated string has a unique parse tree.
- Hence G_4 is unambiguous, whereas G_5 is ambiguous.

Ambiguity

- A grammar generating a string ambiguously means that the string has two different parse trees, not two different derivations.
- Two derivations may differ only in the order in which they replace variables, but not in their overall structure.
- 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.

Definition (Ambiguous Derivation and 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.

- Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.
- Some context-free languages can be generated only by ambiguous grammars. They are called **inherently ambiguous**.
- Example: $\{a^i b^j c^k : i = j \text{ or } j = k\}$ is inherently ambiguous.

Chomsky Normal Form

Definition (Chomsky Normal Form)

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

$$A \to BC, \qquad A \to a,$$

where a is any terminal and A, B and C are any variables - except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Theorem (Chomsky Normal Form)

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

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

Proof of the Chomsky Normal Form Theorem

- We add a new start variable S_0 and the rule $S_0 \to S$, where S was the original start variable.
- We take care of all ε rules:
  - We remove an ε-rule $A \to \varepsilon$, where A is not the start variable.
  - Then for *each occurrence* of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted.
  - If we have the rule $R \to A$, we add $R \to \varepsilon$ unless we had previously removed the rule $R \to \varepsilon$.
  - We repeat until we eliminate all ε rules not involving the start variable.
- We now handle all unit rules:
  - We remove a unit rule $A \to B$.
  - Then, whenever a rule $B \to u$ appears, we add the rule $A \to u$ unless this was a unit rule previously removed.
  - We repeat these steps until we eliminate all unit rules.
- We convert all remaining rules into the proper form.
  - We replace each rule $A \to u_1 u_2 \cdots u_k$, where $k \geq 3$, with the rules $A \to u_1 A_1$, $A_1 \to u_2 A_2$, $A_2 \to u_3 A_3$, $\ldots$, $A_{k-2} \to u_{k-1} u_k$.
  - If $k = 2$, replace a terminal u_i with new variable U_i and add $U_i \to u_i$.

Example of Conversion to Chomsky Normal Form

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

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

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

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

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

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

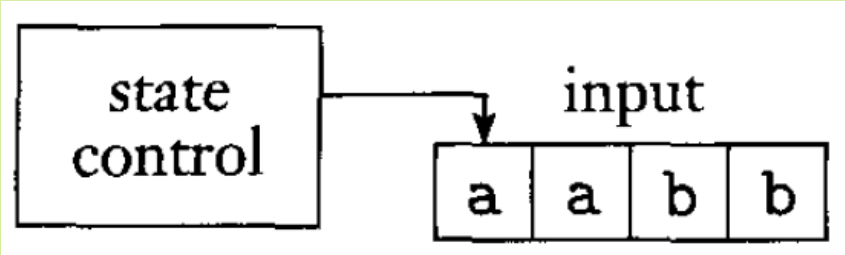
$$S_0 \to AA_1 \mid UB \mid a \mid SA \mid AS$$
$$S \to AA_1 \mid UB \mid a \mid SA \mid AS$$
$$A \to b \mid AA_1 \mid UB \mid a \mid SA \mid AS$$
$$A_1 \to SA$$
$$U \to a$$
$$B \to b$$

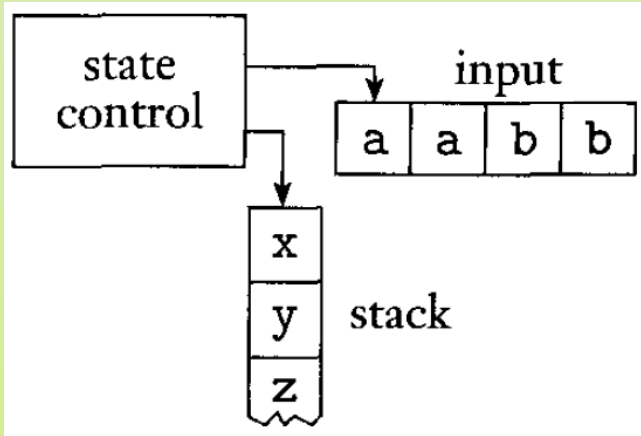
Subsection 2

Pushdown Automata

Pushdown Automata: Adding a Stack to Finite Automata

- We introduce a new type of computational model called pushdown automata.
- They are like nondeterministic finite automata but have an extra component called a stack.
- The stack provides additional memory beyond that available in the control, thus allowing recognition of some nonregular languages.
- Pushdown automata are equivalent to context-free grammars. Thus, to prove that a language is context free we can give
  - either a context-free grammar generating it
  - or a pushdown automaton recognizing it.





Operation of the Stack

- A pushdown automaton (PDA) can write symbols on the stack and read them back later.
  - Writing a symbol "pushes down" all the other symbols on the stack.
  - At any time, the symbol on the top of the stack can be read and removed and the remaining symbols then move back up.
- Writing a symbol on the stack is often referred to as **pushing** the symbol.
- Removing a symbol is referred to as **popping** it.
- Note that all access to the stack, for both reading and writing, may be done only at the top, i.e., a stack is a "last in, first out" storage device.
- If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

An Informal Example

- A stack can hold an unlimited amount of information.
- A finite automaton is unable to recognize the language $\{0^n 1^n : n \geq 0\}$ because it cannot store very large numbers in its finite memory.
- A PDA is able to recognize this language because it can use its stack to store the number of 0s it has seen. Thus, the unlimited nature of a stack allows the PDA to store numbers of unbounded size:

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are seen, pop a 0 off the stack for each 1 read.

- If reading the input is finished exactly when the stack becomes empty of 0s, accept the input.
- If the stack becomes empty while 1s remain or if the 1s are finished while the stack still contains 0s or if any 0s appear in the input following 1s, reject the input.

Determinism and Nondeterminism in PDAs

- As mentioned earlier, pushdown automata may be nondeterministic.
- Deterministic and nondeterministic pushdown automata are not equivalent in power. Nondeterministic pushdown automata recognize certain languages which no deterministic pushdown automata can recognize, though we will not prove this fact.
- Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different.
- Nondeterministic pushdown automata are the ones that are equivalent in power to context-free grammars.

Components of a Pushdown Automaton

- In a pushdown automaton different alphabets may be used for the input and the stack; the input alphabet is Σ and the stack alphabet Γ.
- A transition function describes the behavior:
  - If $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$, the domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Thus, the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. 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.
  - The automaton may enter some new state and possibly write a symbol on the top of the stack. The function δ can indicate this action by returning a member of Q together with a member of Γ_ε, that is, a member of $Q \times \Gamma_\varepsilon$. Nondeterminism is reflected in the transition function returning a set of members of $Q \times \Gamma_\varepsilon$, i.e., a member of $\mathcal{P}(Q \times \Gamma_\varepsilon)$.

  In conclusion, $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$.

Pushdown Automata (PDAs)

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

Operation of Pushdown Automata (PDAs)

- A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows: It **accepts** input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and there exist sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ that satisfy the following three conditions:
  1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state 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 $a, b \in \Gamma_\varepsilon$ 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.

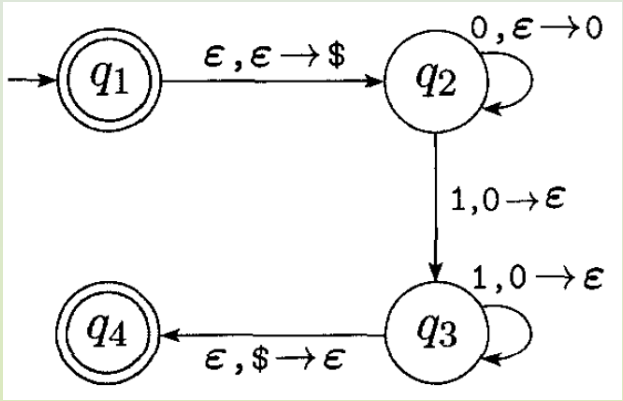
An Example of a PDA

- The following is the formal description of the PDA (informally described previously) that recognizes the language $\{0^n 1^n : n \geq 0\}$.
- Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where
  1. $Q = \{q_1, q_2, q_3, q_4\}$;
  2. $\Sigma \in \{0, 1\}$;
  3. $\Gamma = \{0, \$\}$;
  4. $F = \{q_1, q_4\}$, and
  5. δ is given by the following table wherein blank entries signify $\emptyset$:

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

Diagram of a PDA Transition Function

- We can also use a state diagram to describe the PDA:



We write "$a, b \to c$" to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c. Any of a, b and c may be ε.

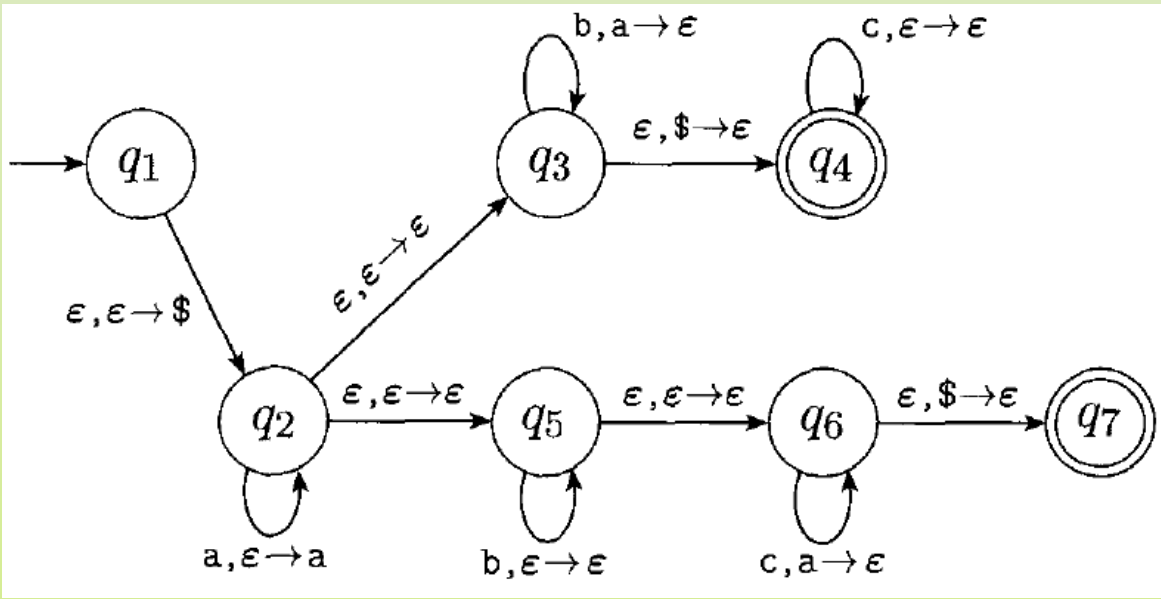
- If a is ε, the machine may make this transition without reading any symbol from the input.
- If b is ε, the machine may make this transition without reading and popping any symbol from the stack.
- If c is ε, the machine does not write any symbol on the stack when going along this transition.

Testing for Empty Stack and for End of Input

- The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol $ on the stack. Then, if it ever sees the $ again, it knows that the stack effectively is empty.
- Similarly, PDAs cannot test explicitly for having reached the end of the input string. The PDA just presented is able to achieve that effect because the accept state takes effect only when the machine is at the end of the input. We assume that PDAs can test for the end of the input, which can be implemented in this manner.

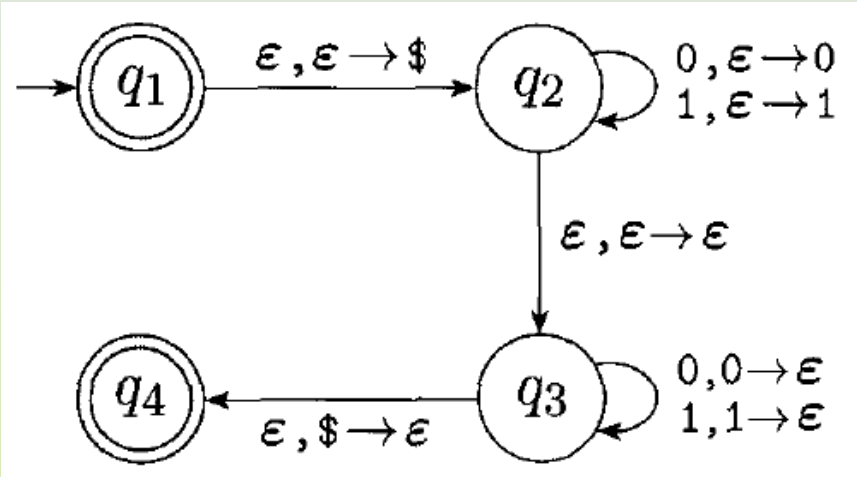
An Example of a PDA

- A pushdown automaton that recognizes the language $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$ works as follows:
  - First reads and pushes the a's.
  - When the a's are done the machine has all of them on the stack so that it can match them with either the b's or the c's.
  - Since the machine does not know in advance whether to match the a's with the b's or the c's, it uses nondeterminism to guess whether to match the a's with the b's or with the c's:



Another PDA

- A PDA M_3 recognizing the language $\{ww^{\mathcal{R}} : w \in \{0,1\}^*\}$ is shown:



Recall that $w^{\mathcal{R}}$ means w written backwards.

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

Equivalence of CFGs and PDAs

- Context-free grammars and pushdown automata are equivalent in power.
- Both are capable of describing the class of context-free languages.
- To establish this equivalence, we show how to convert any context-free grammar into a pushdown automaton that recognizes the same language and vice versa.

Theorem (Equivalence of CFGs and PDAs)

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

- Since this is an "if and only if" theorem, both directions have to be proved.

From CFGs to PDAs: Outline

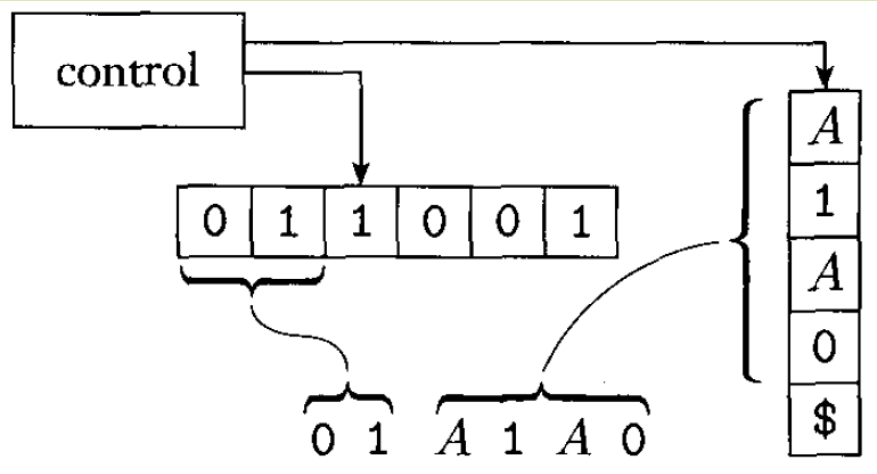
Lemma

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

- Let A be a CFL. By definition, there exists a CFG G generating A. We show how to convert G into an equivalent PDA P. P accepts its input w, if G generates that input. It determines whether some series of substitutions using the rules of G can lead from the start variable to w. It uses nondeterminism to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.
  - The PDA begins by writing the start variable on its stack.
  - It goes through a series of intermediate strings, making one substitution after another.
  - Eventually it may arrive at a string that contains only terminal symbols, meaning that it has used the grammar to derive a string.
  - It accepts if this string is identical to the string it has received as input.

From CFGs to PDAs: Fine-tuning

- The PDA must store intermediate strings.
- Simply using the stack does not work because the PDA needs to find the variables to make substitutions and the PDA can only access the top symbol on the stack which may be a terminal symbol instead of a variable.
- So, the PDA keeps only part of the intermediate string on the stack:
  - the symbols starting with the first variable in the intermediate string.
  - Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string.



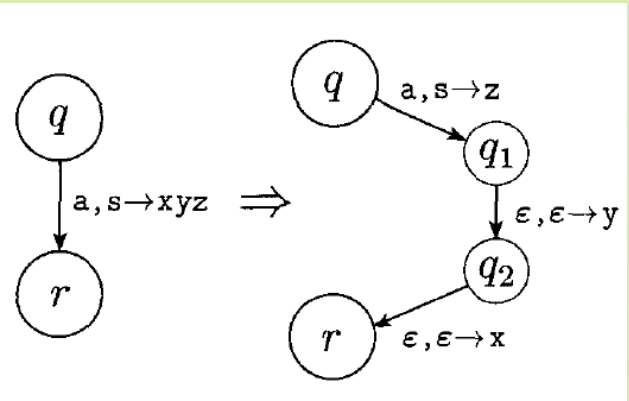
From CFGs to PDAs: Informal Description

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 symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the 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.
      - If they do not match, reject on this branch of the 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.

From CFGs to PDAs: Shorthand Notation

- Let $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$.
- We use a shorthand notation, which provides a way to write an entire string on the stack in one step of the machine.
- This may be simulated by introducing additional states to write the string one symbol at a time:
  - Let q and r be states, a be in Σ_ε and s be in Γ_ε. Say that we want the PDA to go from q to r when it reads a and pops s. Furthermore we want it to push the entire string $u = u_1 \cdots u_\ell$ on the stack. Introduce new states $q_1, \ldots, q_{\ell-1}$ and set:

$$\begin{aligned}
&\delta(q, a, s) \text{ to contain } (q_1, u_\ell),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{\ell-1})\}\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{\ell-2})\},\\
&\vdots\\
&\delta(q_{\ell-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

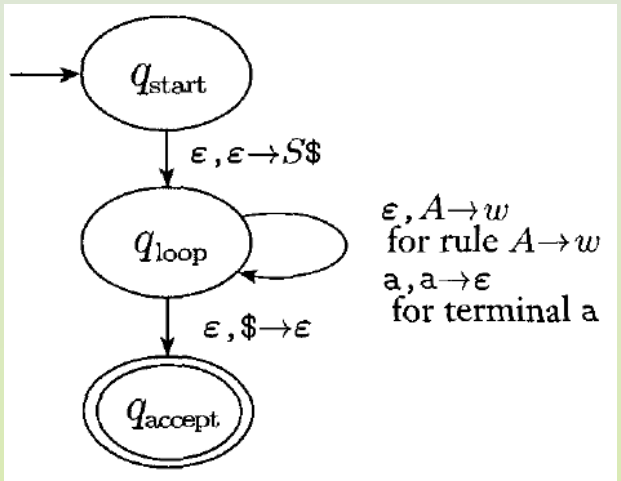


- The notation $(r, u) \in \delta(q, a, s)$ signifies this move.

From CFGs to PDAs: Formal Proof

- The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states implementing the shorthand described previously.

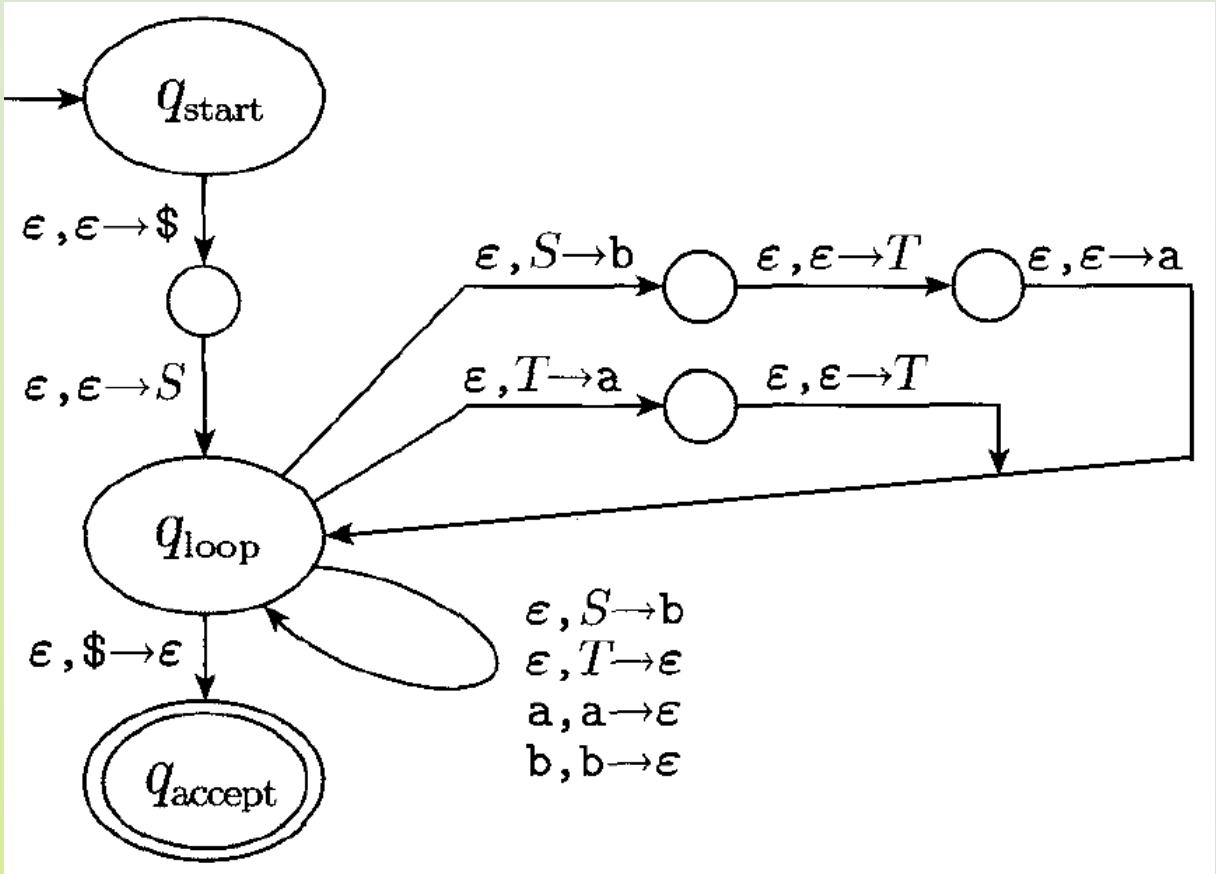
- The start state is q_{start}.
- The only accept state is q_{accept}.
- For the transition function:



  - Initialize the stack to contain \$ and S: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$.
  - We add transitions for the main loop of Step 2:
    - First, we handle Case (a) wherein the top of the stack contains a variable: $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w)\}$, where $A \to w$ is a rule in R.
    - Second, we handle Case (b) wherein the top of the stack contains a terminal: $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.
    - Finally, we handle Case (c) wherein the empty stack marker \$ is on the top of the stack: $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

An Example Illustrating the Proof

- We construct a PDA P from the CFG G: $S \to aTb \mid b$, $T \to Ta \mid \varepsilon$.



PDAs to CFGs: Outline

Lemma

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

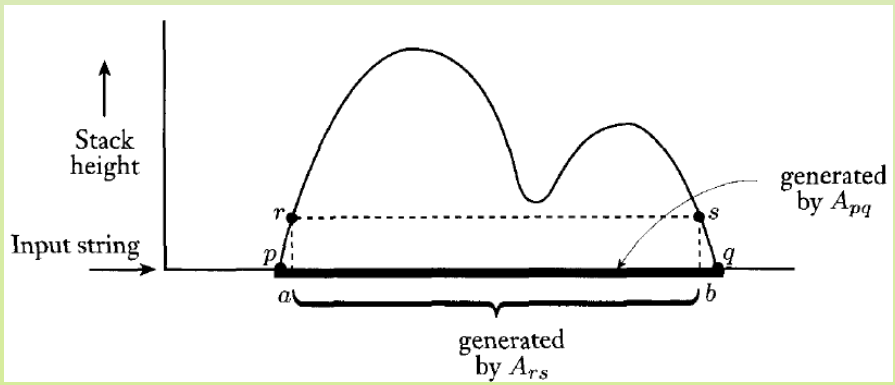
- We have a PDA P, and want to construct a CFG G that generates all the strings that P accepts.
- We design a grammar that does somewhat more:
  - For each pair of states p and q in P the grammar has a variable A_{pq}.
  - This variable generates all the strings that can take P from p with an empty stack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

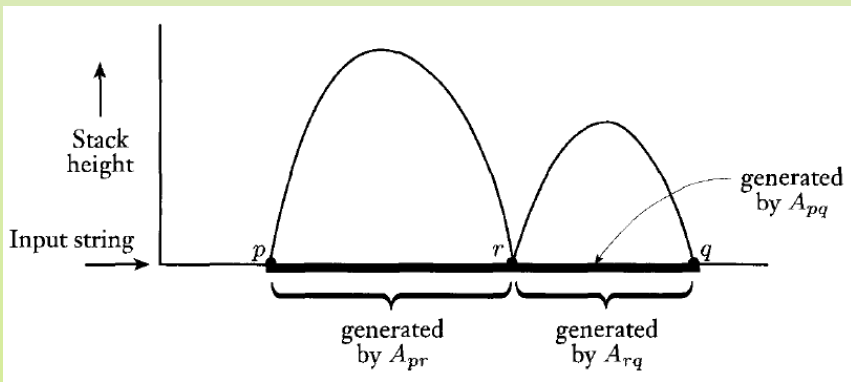
PDAs to CFGs: Informal Description

- To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we look at P's operation on these strings:
  - For any such x, P's first move on x must be a push, because every move is either a push or a pop and P cannot pop an empty stack.
  - The last move on x must be a pop, because the stack ends up empty.
  - Either the symbol popped at the end is the symbol that was pushed at the beginning, or not:
    - If so, the stack is empty only at the beginning and end of P's computation on x. This possibility is simulated by the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q.
    - If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. This possibility is simulated by the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

PDAs to CFGs: Formal Proof

- Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G:
  - The variables of G are $\{A_{pq} : p, q \in Q\}$.
  - The start variable is $A_{q_0 q_{\text{accept}}}$.
  - Now we describe G's rules:
    - For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), 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 $A_{pp} \to \varepsilon$ in G.





Proof of the Role of A_{pq} Part I

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

  Proof is by induction on the number of steps in the derivation of x from A_{pq}.
  - Basis: The derivation has 1 step. A derivation with a single step must use a rule whose right-hand side contains no variables. The only rules in G where no variables occur on the right-hand side are $A_{pp} \to \varepsilon$. Input ε takes P from p with empty stack to p with empty stack.
  - Induction step: Assume true for derivations of length at most k, where $k \geq 1$, and prove true for derivations of length $k+1$. Suppose that $A_{pq} \stackrel{*}{\Rightarrow} x$ with $k+1$ steps. The first step is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$.

Proof of the Role of A_{pq} Part I (Cont'd)

- The first step is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$.
  - In the first case, consider the portion y of x that A_{rs} generates, so $x = ayb$. Because $A_{rs} \stackrel{*}{\Rightarrow} y$ with k steps, the induction hypothesis tells us that P can go from r on empty stack to s on empty stack. Because $A_{pq} \rightarrow aA_{rs}b$ is a rule of G, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack symbol t. Hence, if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack. Then after reading b it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.
  - In the second case, consider the portions y and z of x that A_{pr} and A_{rq} respectively generate, so $x = yz$. Because $A_{pr} \stackrel{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \stackrel{*}{\Rightarrow} z$ 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. Hence x can bring it from p with empty stack to q with empty stack.

Proof of the Role of A_{pq} Part II

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

  Proof is by induction on the number of steps in the computation of P that goes from p to q with empty stacks on input x.
  - 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 $x = \varepsilon$. By construction, G has the rule $A_{pp} \to \varepsilon$, so the basis is proved.
  - Induction step: Assume true for computations of length at most k, where $k \geq 0$, and prove true for computations of length $k + 1$. Suppose that P has a computation wherein x brings p to q with empty stacks in $k + 1$ steps. Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.

Proof of the Role of A_{pq} Part II (Cont'd)

- Either the stack is empty only at the beginning and end of this computation, or it becomes empty elsewhere, too.
  - In the first case, the symbol t that is pushed at the first move must be the same as the symbol that is popped at the last move. Let a, b be the inputs read in the first, last move, respectively, r, s be the states after the first and before the last move. Then $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε). So the rule $A_{pq} \to aA_{rs}b$ is in G. Let y be the portion of x without a and b, so $x = ayb$. Input y can bring P from r to s without touching t on the stack, so P can go from r with empty stack to s with empty stack on input y. The computation on y has $k - 1$ steps. By the hypothesis, $A_{rs} \stackrel{*}{\Rightarrow} y$. Hence, $A_{pq} \stackrel{*}{\Rightarrow} x$.
  - In the second case, let r be a state where the stack becomes empty other than at the beginning or end of the computation on x. The computations from p to r and from r to q each contain at most k steps. Say that y is the input read during the first and z during the second portion. The induction hypothesis tells us that $A_{pr} \stackrel{*}{\Rightarrow} y$ and $A_{rq} \stackrel{*}{\Rightarrow} z$. Because rule $A_{pq} \to A_{pr}A_{rq}$ is in G, $A_{pq} \stackrel{*}{\Rightarrow} x$.

Regular Languages are Context-Free

- We have just proved that pushdown automata recognize the class of context-free languages.
- Recall that every regular language is recognized by a finite automaton and every finite automaton is automatically a pushdown automaton that simply ignores its stack.
- Therefore, every regular language is also a context-free language.

Corollary

Every regular language is context-free.

Subsection 3

Non-Context-Free Languages

Pumping Lemma for Context-Free Languages

- The pumping lemma states that all strings longer than a **pumping length** in a context-free language can be "pumped" (in a way different from that in regular languages).

Pumping Lemma for Context-Free Languages

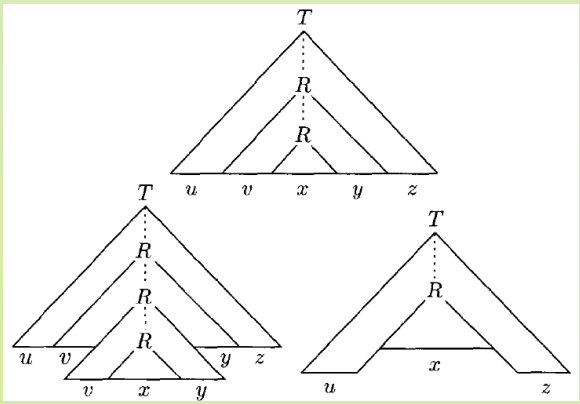
If A is a context-free language, then there is a number p (the **pumping length**) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$, satisfying the conditions

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

- When s is being divided into $uvxyz$, Condition 2 says that either v or y is not the empty string.
- Condition 3 states that the pieces v, x and y together have length at most p. This technical condition is useful in proving languages to be non-context-free.

Idea Behind the Proof

- Let A be a CFL and let G be a CFG that generates it. We must show that any sufficiently long string s in A can be pumped remaining in A. Since s is in A, it is derivable from G. So it has a parse tree. The parse tree for s must be very tall because s is assumed to be "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. On this long path some variable symbol R must repeat because of the pigeonhole principle.



This repetition allows us to 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 in the figure, and we may repeat the second and fourth pieces and obtain a string still in the language.

Proof of the Pumping Lemma: Pumping Length

- Let G be a CFG for CFL A. Let $b \geq 2$ be the maximum number of symbols in the right-hand side of a rule. In any parse tree using this grammar, a node can have no more than b children, i.e., at most b leaves are 1 step from the start variable; at most b^2 leaves are within 2 steps of the start variable; and at most b^h leaves are within h steps of the start variable. So, if the height of the parse tree is at most h, the length of the string generated is at most b^h.

  Conversely, if a generated string is at least $b^h + 1$ long, each of its parse trees must be at least $h + 1$ high.

  Say $|V|$ is the number of variables in G. We set p, the **pumping length**, to be $b^{|V|+1} > b^{|V|} + 1$. If s is a string in A and its length is p or more, its parse tree must be at least $|V| + 1$ high.

Proof of the Pumping Lemma: Splitting a Parse Tree

- To see how to pump any such string s, let τ be one of its parse trees. If s has several parse trees, choose τ to be a parse tree that has the smallest number of nodes. We know that τ must be at least $|V|+1$ high, so it must contain a path from the root to a leaf 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 at least $|V|+1$ variables. With G having only $|V|$ variables, some variable R appears more than once on that path. For convenience later, we select R to be a variable that repeats among the lowest $|V|+1$ variables on this path.

  We divide s into $uvxyz$ according to the previous figure. Each occurrence of R has a subtree under it, generating part of the string s. 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, so we may substitute one for the other and still obtain a valid parse tree.

Proof of the Pumping Lemma: Proving the Properties

- Replacing the smaller by the larger subtree repeatedly gives parse trees for the strings uv^ixy^iz at each $i \geq 1$. Replacing the larger by the smaller generates the string uxz. That establishes Condition 1 of the lemma.
- To get Condition 2 we must be sure that not both v and y are ε. If they were, the parse tree obtained by substituting the smaller subtree for the larger would have fewer nodes than τ does and would still generate s. This result is not possible because we had already chosen τ to be a parse tree for s with the smallest number of nodes.
- In order to get Condition 3 we must show 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 bottom $|V| + 1$ variables, and we chose the longest path, 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 $b^{|V|+1} = p$.

Example I

- The language $B = \{a^n b^n c^n : n \geq 0\}$ is not context free.

  Assume that B is a CFL. Let p be the pumping length for B. Select the string $s = a^p b^p c^p$. Clearly s is a member of B and of length at least p. The pumping lemma states that s can be pumped. However, we show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated. Condition 2 stipulates that either v or y is nonempty. The substrings v and y either both contain only one type of alphabet symbol or not.

  1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. In this case the string uv^2xy^2z cannot contain equal numbers of a's, b's, and c's. Therefore it cannot be a member of B. That violates Condition 1 of the lemma. So, we obtain a contradiction.
  2. When either v or y contain more than one type of symbol uv^2xy^2z may contain equal numbers of the three alphabet symbols but not in the correct order. Hence, it cannot be a member of B.

Example II

- The language $C = \{a^i b^j c^k : 0 \leq i \leq j \leq k\}$ is not a CFL. Assume that C is a CFL. Let p be the pumping length. We use the string $s = a^p b^p c^p$, but now we must pump both up and down. Let $s = uvxyz$ and again consider the previous cases:
  1. When both v and y contain only one type of symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. In this case, one of a, b, or c does not appear in v or y.
     a. If a's do not appear, then we pump down to obtain $uv^0xy^0z = uxz$. It contains the same number of a's as s does, but it contains fewer b's or fewer c's. Therefore it is not a member of C.
     b. If b's do not appear, then either a's or c's must appear in v or y because not both are empty. If a's appear, the string uv^2xy^2z contains more a's than b's, so it is not in C. If c's appear, the string uv^0xy^0z contains more b's than c's, so it is not in C.
     c. If c's do not appear, then the string uv^2xy^2z contains more a's or more b's than c's, so it is not in C, and a contradiction occurs.
  2. When v or y contain more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order.

Example III

- The language $D = \{ww : w \in \{0,1\}^*\}$ is not a CFL.
  Assume that D is a CFL. Let p be the pumping length. The string 0^p10^p1 appears to be a good candidate, but it can be pumped:

$$\overbrace{\underbrace{000\cdots000}_{u}\underbrace{0}_{v}\underbrace{1}_{x}}^{0^p1}\overbrace{\underbrace{0}_{y}\underbrace{000\cdots0001}_{z}}^{0^p1}.$$

  We consider $s = 0^p1^p0^p1^p$. Condition 3 says we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$. First, we show that the substring vxy must straddle the midpoint of s.
  - If it occurs only in the first half of s, pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww.
  - If vxy occurs in the second half of s, pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww.

  If vxy straddles the midpoint of s, pumping s down to uxz gives $0^p1^i0^j1^p$, where i and j cannot both be p.