Chapter Three: Syntax Analysis

Outlines:

- ✓ The Role of the Parser
- ✓ Grammars
- ✓ Chomsky Hierarchy
- ✓ Parse Tree
- ✓ Reduction of CFGs
- ✓ Issues of CFG for the Programming Languages
- ✓ Pushdown Machines

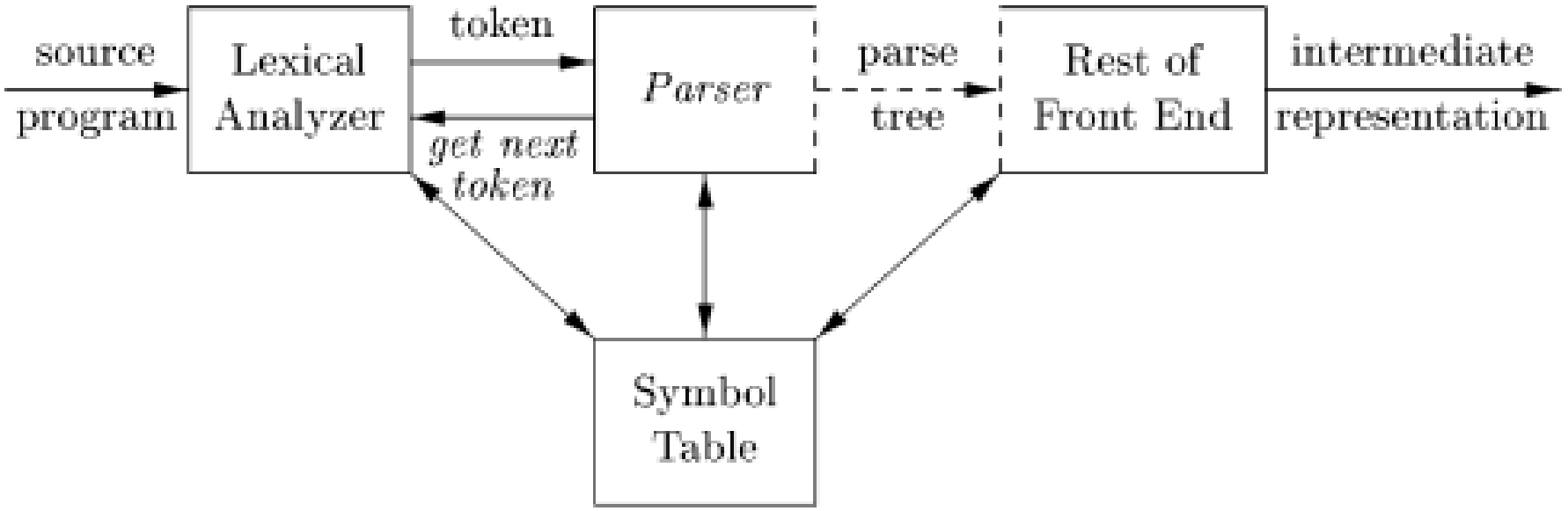
Objectives

- This chapter presents the **basic concepts of parsing**.
- **Formal methods** to *specify* and *construct* the syntax analysis.
  - Introduce the concept of a formal grammar as a means of:
    - ✓ ***Specifying*** the programming language
    - ✓ ***Implementing*** the syntax analysis phase.

The Role of the Parser

- **Syntax** analyzer (**Parser**) or **Hierarchical** analyzer.
- A **Parser**
  - ✓ Implements the source language **grammar**.
  - ✓ Obtains a stream of **tokens** from the scanner.
  - ✓ Verifies that the stream can be generated by the grammar.
  - ✓ Report any syntax errors in an intelligible fashion.
  - ✓ Recover from commonly occurring errors.
  - ✓ Constructs a parse tree and passes it to the rest of the compiler.
- The grammar that a parser implements is called a **Context Free Grammar** or **CFG**.

The Role of the Parser *Cont.*



- Three general types of parsers are ***universal***, ***top-down***, and ***bottom-up***.
  - ✓ **Universal** parsing methods can parse any grammar but too inefficient.
  - ✓ The **top-down** or **bottom-up** methods are commonly used.

The Role of the Parser *Cont.*

- The role of the parser is twofold:
  - ✓ To check syntax (= string recognizer) and report syntax errors accurately.
- To invoke semantic actions
  - ✓ For static semantics checking, e.g. type checking of expressions, functions, etc.
- The difference between Syntax and Semantic is:
  - ✓ **Syntax** is the way in which we construct sentences by following principles and rules.
  - ✓ **Semantics** is the interpretations of and meanings derived from the sentence transmission and understanding of the message or in other words are the logical sentences making sense or not.

The Role of the Parser *Cont.*

- If there are no syntax errors, the **output** of parser is a stream of **atoms** or **syntax trees**.
  - ✓ An **atom** is ***a primitive operation which can be implemented using only a few machine language instructions***.
  - ✓ Each atom includes operands which are ultimately converted to memory addresses on the target machine.
- A syntax tree is a data structure in which the interior nodes represent operations, and the leaves represent operands.
- The **syntax directed translation** process is not only checking for proper syntax but producing output as well.

Grammars

- Every programming language has *precise rules* that prescribe the *syntactic* structure of well-formed programs.
- Grammars offer *significant benefits* for language designers and compiler writers.
  - ✓ A grammar gives a precise syntactic specification of a programming language.
  - ✓ Automatically, we can construct an efficient parser that determines the syntactic structure of a source program.
  - ✓ The designed grammar is useful for translating source programs into correct object code and for detecting errors.

Grammars *Cont.*

- ✓ A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks.
  - By follow the grammatical structure of the language.

- There are some concepts of formal language theory which the student must understand.
  - ✓ A **language** is a set of strings from a specific alphabet.
  - ✓ Ways of formally specifying a language are:
    - Regular expressions
    - Finite Automata (**FAs**) or Finite State Machines (**FSMs**)
    - Grammars

Grammars *Cont.*

- A **grammar** is a list of ***rules*** which can be used to ***produce*** or ***generate*** all the strings of a language, and which does ***not generate*** any strings which are not in the language.
- Mathematically, a **grammar** is defined as a 4-tuple;

$$\mathbf{G: (\Sigma, N, P, S)}$$

  - ✓ **Σ** is a finite set of characters, called the ***input alphabet***, the ***input symbols***, or ***terminal symbols***.
  - ✓ **N** is a finite set of symbols, distinct from the terminal symbols, called ***non-terminal symbols*** (***grammar variables***).

Grammars *Cont.*

- **P** is a finite list of ***rewriting rules***, also called ***productions***, which define how strings in the language may be generated. Each of these rewriting rules is of the form $\alpha \rightarrow \beta$, where α and β are arbitrary strings of terminals and non-terminals, and α *is not null*.
- **S** is a particular non-terminal called the ***goal*** *symbol*, which represents exactly all the strings in the language. part of N
- The set of ***terminals and non-terminals*** is called the ***vocabulary*** of the grammar.

Grammars *Cont.*

- The following symbols will have particular meanings:

A, B, C, . . .	**a single non-terminal.**
a, b, c, . . .	**a single terminal.**
. . ., X, Y, Z	**a single terminal or non-terminal.**
. . ., x, y, z	**a string of terminals.**
α, β, γ, . . .	**a string of terminals and non-terminals.**

Thus, a generic production can be written as A → α

- ✓ **Terminals**: Lower case letters, operator symbols, punctuation symbols, digits, boldface strings are all terminals.
- ✓ **Non-Terminals**: Upper case letters, lower case italic names are usually non terminals.

Grammars *Cont.*

- ✓ The starting non-terminal is always **S** unless otherwise specified.
- ✓ For reference purposes, each of the grammars shown will be numbered (**G1, G2, G3, ...**).

- ❑ If **G** is a grammar, the language specified by **G** is **L(G)**.
- ❑ For example, let production for any grammar are:

$$S \rightarrow a\ S\ a$$

$$S \rightarrow b\ S\ b$$

$$S \rightarrow c$$

$$G: (\Sigma, N, P, S)$$

$$\Sigma = \{a, b, c\} \quad N = \{S\}$$

$$P = \{S \rightarrow a\ S,\ S \rightarrow b\ S\ b,\ S \rightarrow c\}\ \ S = S$$

Grammars *Cont.*

- The most common way of specifying productions is called **Backus-Naur Form** (**BNF**)
  - ✓ non-terminals are enclosed in angle brackets ‹ ›,
  - ✓ The arrow is replaced by a **::=** as: ‹ **S** › **::= a** ‹ **S** › **b** which is the BNF version of the grammar rule: **S → a S b**
- So, in BNF, productions have the form

  left side **→** ***definition***

  where ***left side*** **∈ (Σ U N) *** and ***definition*** **∈ (Σ U N) ***
- For ***left side*** contains one non-terminal, ***left side*** **∩ N = ϕ.**

Grammars *Cont.*

- For multiple definitions of one non-terminal, production rules are abbreviated by listing the definitions as a set of one or more alternatives, separated by a vertical bar symbol "|".
- For example, **‹ S › ::= a ‹ S › b | ε** which is the **BNF** version of two grammar rules: **S → a S b, S → ε**.
- Let production for any grammar are:

  S → a S a

  S → b S b

  S → c

  can represented as: **S → a S a | b S b | c**

Chomsky Hierarchy

- **Noam Chomsky** defined *four levels* of grammars according to complexity and the corresponding four classes of ***automata*** or abstract machine types have been identified.

Chomsky Language Class	Grammar	Recognizer
3	Regular	Finite State Automaton
2	Context-Free	Push-Down Automaton
1	Context-Sensitive	Linear-Bounded Automaton
0	Unrestricted	Turing Machine

Chomsky Hierarchy *Cont.*

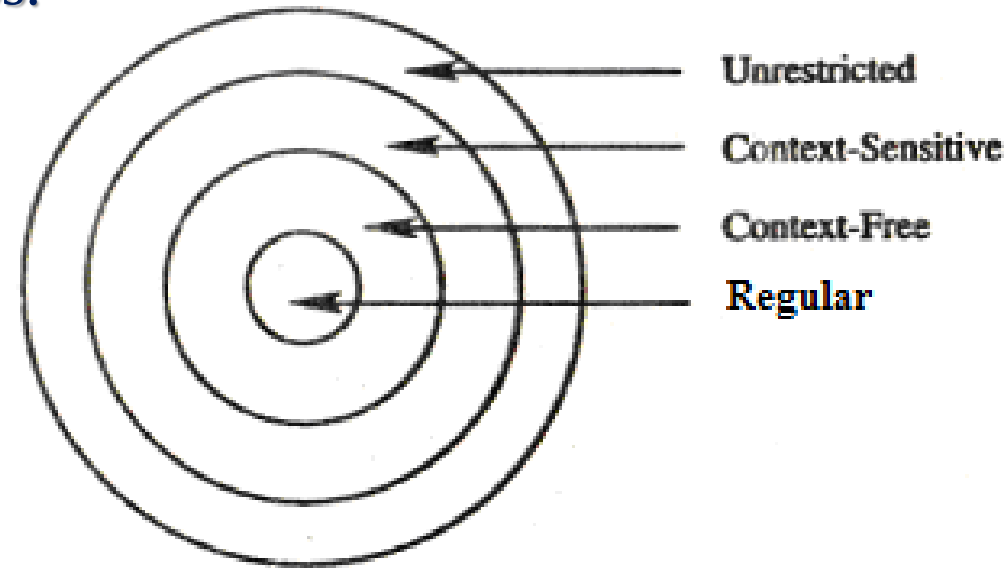
- **0. Unrestricted** grammar is one in which there are *no restrictions* on the *rewriting rules*. Each production rule on the form $\alpha \rightarrow \beta$ where α and β are arbitrary string of grammar symbols with $\alpha \neq \varepsilon$.
- **1. Context-Sensitive** grammar is one in which each rule must be of the form: $\alpha\ A\ \gamma \rightarrow \alpha\ \beta\ \gamma$ where α, β, and γ are any string of terminals and non-terminals (including ε), and **A** represents a single non-terminal. β is at least as long as α that is clearly $|\alpha| \leq |\beta|$

Chomsky Hierarchy *Cont.*

- **2. Context-Free Grammar (CFG)** is one in which each rule must be of the form: $A \rightarrow \alpha$ where **A** represents a single non-terminal and α is any string of terminals and non-terminals ($A \in N$ and $\alpha \in (\Sigma \cup N)^*$).
- **3. Regular Grammar** - If all production rules of a CFG are of the form: $A \rightarrow wB$ or $A \rightarrow w$ where **A** and **B** are non-terminals and $w \in \Sigma^*$, then we say that is a ***right linear*** grammar. If all production rules of a CFG are of the form: $A \rightarrow Bw$ or $A \rightarrow w$, we call it a ***left linear*** grammar.

Chomsky Hierarchy *Cont.*

- Every context-sensitive grammar is in the unrestricted class.
- Every **CFG** is in the context-sensitive and unrestricted classes.
- Every regular grammar is in the context-free, context-sensitive, and unrestricted classes.



Chomsky Hierarchy *Cont.*

- A grammar **G** is said to be
  - ✓ **Regular** if it is *right linear* where each production is of the form

$$A \rightarrow wB \quad \text{or} \quad A \rightarrow w$$

    or *left linear* where each production is of the form

$$A \rightarrow Bw \quad \text{or} \quad A \rightarrow w$$

  - ✓ **Context-free** if each production is of the form $A \rightarrow \alpha$

    where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$
  - ✓ **Context sensitive** if each production is of the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

    where $A \in N$, $\alpha, \gamma, \beta \in (N \cup \Sigma)^*$, $|\gamma| > 0$
  - ✓ **Unrestricted**

Chomsky Hierarchy *Cont.*

- **L(regular) ⊂ L(context free) ⊂ L(context sensitive) ⊂ L(unrestricted)**
  - ✓ *Where L(T) = {L(G) | G is of type T} That is: the set of all languages generated by grammars G of type T.*
- **`Examples:`**
  - ✓ *Every finite language is regular! (construct a FSA for strings in L(G))*
  - ✓ $L_1 = \{ a^n b^n \mid n \geq 1 \}$ *is context free*
  - ✓ $L_2 = \{ a^n b^n c^n \mid n \geq 1 \}$ *is context sensitive*
  - ✓ ***G1 is an** example of a context-sensitive grammar.*

G1:

1. S → aSBC
2. S → ε
3. aB → ab
4. bB → bb
5. C → c
6. CB → CX
7. CX → BX
8. BX → BC

Derivation

- *A **derivation** is a sequence of rewriting rules, applied to the starting non-terminal, ending with a string of terminals.*
  - ✓ *It demonstrates that a particular string is a member of the language.*
  - ✓ *Assuming **S** is the starting non-terminal, derivations are written as:*
  - ✓ **α**, **β**, *and* **γ** *are strings of terminals and/or non-terminals, and x is a string of terminals.* $S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \gamma \Rightarrow \ldots \Rightarrow x$
- *The one-step derivation is defined by* $\alpha A \beta \Rightarrow \alpha \gamma \beta$

  where $A \rightarrow \gamma$ *is a production in the grammar*
- *The language generated by G is defined by*

$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

Derivation *Cont.*

- A ***sentential form*** is the goal or start symbol, or any string that can be derived from it, that is any string w such that $S \overset{*}{\Rightarrow} w$

  where $w \in (\Sigma \cup N)^*$

- A *recursive* grammar permits derivations of the form

  $A \overset{*}{\Rightarrow} w_1 A w_2$ (where $A \in N$ and w_1 and $w_2 \in (\Sigma \cup N)^*$)

- It is a *left recursive* if $A \overset{*}{\Rightarrow} A w$ and *right recursive* if $A \overset{*}{\Rightarrow} w A$

- A *self-embedding* grammar permits derivations of the form

  $A \overset{*}{\Rightarrow} w_1 A w_2$ (where $A \in N$ and w_1 and $w_2 \in (\Sigma \cup N)^*$)

but where w_1 or w_2 contains at least one terminal that is $(w_1 \cap \Sigma) \cup (w_2 \cap \Sigma) \neq \phi$.

Derivation *Cont.*

- *In addition, we define*
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- *Grammar* **G = ({+,*, (,), -, id}, {E}, P, E)** *with*

Productions **P =**

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow - E$$
$$E \rightarrow id$$

Derivation *Cont.*

- Example derivations:

$$E \Rightarrow -E \Rightarrow -\text{id}$$

$$E \Rightarrow_{rm} E+E \Rightarrow_{rm} E+\text{id} \Rightarrow_{rm} \text{id}+\text{id}$$

$$E \Rightarrow^{*} E$$

$$E \Rightarrow^{*} \text{id}+\text{id}$$

$$E \Rightarrow^{+} \text{id}*\text{id}+\text{id}$$

Derivation *Cont.*

- **Example 1: G2** consists of:
  - ✓ Four rules, the terminal symbols {0, 1}, and
  - ✓ The starting non-terminal **S**.
  - ✓ An example of a derivation using G1 is:

G2:

1. $S \rightarrow 0S0$
2. $S \rightarrow 1S1$
3. $S \rightarrow 0$
4. $S \rightarrow 1$

$$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001S100 \Rightarrow 0010100$$

  - ✓ Thus, **0010100** is in **L(G1)**.
  - ✓ **G2** specifies language of ***palindromes*** of odd length over the alphabet **{0,1}**.
  - ✓ A ***palindrome*** is a string which reads the same from left to right as it does from right to left.

L(G1) = {0, 1, 000, 010, 101, 111, 00000, . . . }

Derivation *Cont.*

- **Example 2: G3** consists of:

G3:

1. S ⟶ ASB
2. S ⟶ ε
3. A ⟶ a
4. B ⟶ b

  - ✓ Four rules, the terminal symbols {a, b}, and
  - ✓ ε is not a terminal symbol.
  - ✓ The starting non-terminal **S**.
  - ✓ An example of a derivation using **G3** is:

    S ⇒ **ASB ⇒ aSB ⇒ aASBB ⇒ aaSBB ⇒ aaBB ⇒ aabB ⇒ aabb**

  - ✓ Thus, ***aabb*** is in **L(G2)**.
  - ✓ **G3** specifies the set of all strings of ***a's*** and ***b's*** which contain the same number of ***a's*** as ***b's*** and in which all the ***a's*** precede all the ***b's***.

L(G2) = {ε, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, . . . }

= {$a^n b^n$} such that n greater than or equal to zero.

  - ✓ **G3** language is the set of all strings of ***a's*** and ***b's*** which consist of zero or more ***a's*** followed by exactly the same number of ***b's***.

Derivation *Cont.*

- Two grammars, **G1** and **G2**, are said to be ***equivalent*** if **L(G1) = L(G2)** – i.e., they specify the same language.
- In the following grammar, there can be several different derivations for a particular string e.g.

1. S ⟶ aSA
2. S ⟶ BA
3. A ⟶ ab
4. B ⟶ bA

S ⇒ aSA ⇒ aBAA ⇒ abAAA ⇒ ababAA ⇒ abababA ⇒ abababab

S ⇒ aSA ⇒ aSab ⇒ aBAab ⇒ aBabab ⇒ abAabab ⇒ abababab

S ⇒ BA ⇒ bAA ⇒ babA ⇒ babab

Derivation *Cont.*

- Let a grammar with productions $E \rightarrow E + E \mid E * E \mid x \mid y \mid z$

A leftmost derivation of x + y ∗ z is:

$E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E * E \Rightarrow x + y * E \Rightarrow x + y * z$

A rightmost derivation of x + y ∗ z is:

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * z \Rightarrow E + y * z \Rightarrow x + y * z$

Another leftmost derivation of x + y ∗ z is:

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow x + E * E \Rightarrow x + y * E \Rightarrow x + y * z$

The following is neither left- nor rightmost:

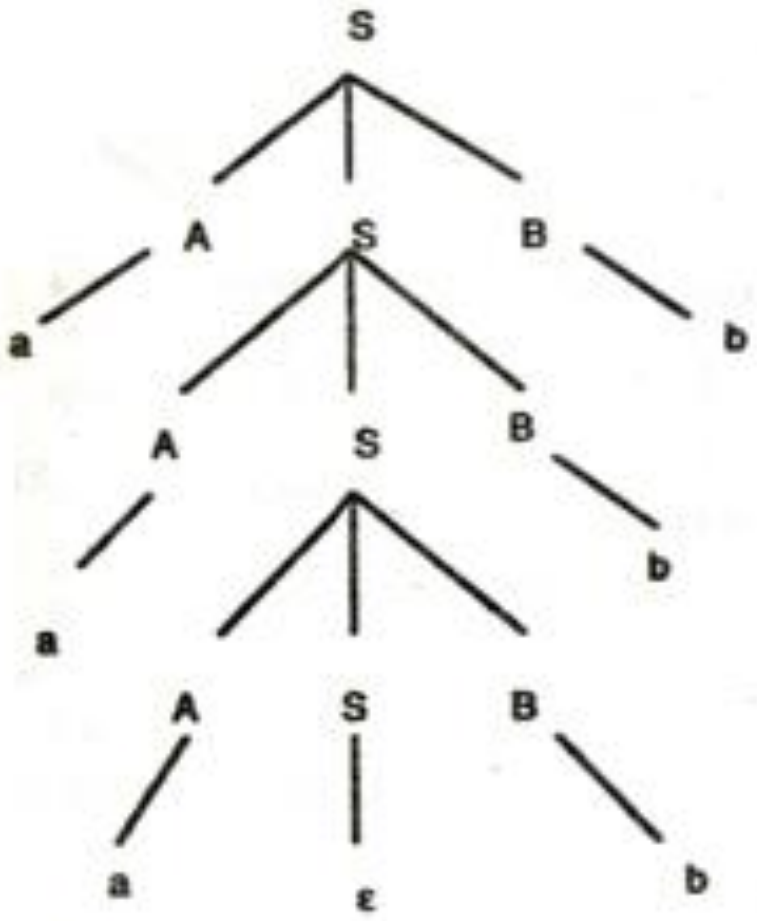
$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + y * E \Rightarrow x + y * E \Rightarrow x + y * z$

Parse Tree

- *A **derivation/parse tree** is a pictorial representation of a derivation.*
  - ✓ *The **root** of the tree is labeled by the **start** symbol*
  - ✓ *Each **leaf** of the tree is labeled by a **terminal** (=**token**) or **ε** in the derived string.*
  - ✓ *Each **interior node** is labeled by a **non-terminal** in a **sentential form.***
  - ✓ *If $A \rightarrow X_1\ X_2\ \ldots\ X_n$ is a production, then node **A** has immediate children $X_1, X_2, \ldots, X_n$ where X_i is a (non)terminal or **ε***
  - ✓ *A parse tree for the string **aaabbb** using **G3** is:*

Parse Tree *Cont.*

- *The **yield** of a parse tree is the concatenation of its leaves from left to right.*
- *The **yield** is always a string that is derived from the root node.*
- *The **yield** is a terminal string labelled from* S ∪ {ε}.



Reduction of CFGs

- *There are several ways to restrict the format of CFG without reducing the language generation power of CFG.*
- *Let L be a non-empty context free language generated by a CFG with elimination of:*
  1. *Useless symbols, those terminals or non-terminals that do not appears in any derivation of a terminal string from the start symbol.*
  2. *Unit productions, those of the form $X \rightarrow Y$ for some non-terminals X and Y.*
  3. *ε-productions, those of the form $X \rightarrow \varepsilon$ for some non-terminal X.*

Elimination of Useless Production/Symbols from CFG

- *Any symbol is useful only when it is deriving any terminal and if a symbol is deriving a terminal but not reachable from start state.*
- *All terminals will be useful symbols.*
- *A symbol that is useful will be both generating and reachable.*
- *For example: Find the reduced grammar that is equivalent to G4*

G4:	1. S → S B | a C 2. A → b S C a 3. B → a S B | b B C 4. C → a B C | a d

Elimination of Useless Production/Symbols from CFG *Cont.*

Solution:

1. *Since C → a d, C is a generating symbol and*
2. *Since S → a C, S is also a generating symbol.*
3. *According to the production A → b S C a, A is also a generating symbol.*
4. *Right side of B → a S B and B → b B C contains B, and B is not terminating, so B is not a generating symbol.*
5. *So, we can eliminate those productions and grammar becomes:*

G_4:	**1. S → a C** **2. A → b S C a** **3. C → a d**

Elimination of Useless Production/Symbols from CFG *Cont.*

Solution:

*Since **S** is the start symbol and right side of **S** does not contain **A**, hence **A** is not **reachable** as:*

*So, by eliminating **A** we get*

G4:	**1. S → a C** **2. C → a d**

which is reduced grammar equivalent to the given grammar, containing no useless symbol.

Eliminating Unit Productions

- *A unit production in a CFG is a production of the form:*

  Non -terminal → One non-terminal (A → B)

Eliminate unit productions Algorithm:

While (there exist a unit production ***A → B****)*

{

Select *a unit production, such that there exist a production* ***B → a****, where* ***a*** *is a terminal.*

for *(every non-unit production,* ***B → a****)*

add *production* ***A → a*** *to the grammar.*

Eliminate A → B *from the grammar*

}

Eliminating Unit Productions *Cont.*

- ***Example:*** *Remove the unit productions from* ***G5****:*
- ***Solution:*** *There are three-unit productions in the grammar (I)*

G5:	1. S → A B 2. A → a 3. B → C | b 4. C → D 5. D → E 6. E → a

I

	3. B → C 4. C → D 5. D → E

II

G5:	1. S → A B 2. A → a 3. B → a | b 4. C → a 5. D → a 6. E → a

III

G5:	1. S → A B 2. A → a 3. B → a | b

Eliminating Unit Productions *Cont.*

- *For* $D \rightarrow E$ *there is* $E \rightarrow a$ *so, add* $D \rightarrow a$ *to* $G5$ *and delete* $D \rightarrow E$ *from* $G5$.
- *Also,* $C \rightarrow D$ *so, add* $C \rightarrow a$ *to* $G5$ *and delete* $C \rightarrow D$ *from* $G5$.
- *Similarly,* $B \rightarrow C$ *by adding* $B \rightarrow a$ *and removing* $B \rightarrow C$ *So, the final grammar free of unit productions as in (II).*
- *See* C, D *and* E *are unreachable symbols. So, to get a completely reduced grammar, remove them from G5. The final G2 is in (III).*

Eliminate ε-Productions

- *Null productions are of the form* $A \rightarrow \varepsilon$.
- *All* **ε***-productions cannot be removed from a grammar if the language contains* **ε** *as a word, but if it does not remove all.*
- *In each CFG, a non-terminal* **N** *is nullable if there is a production* $N \rightarrow \varepsilon$ *or there is a derivation that starts at* **N** *and leads to* **ε** *:*

$$N \Rightarrow \ldots \Rightarrow \varepsilon$$

- *If* $A \rightarrow \varepsilon$ *is an eliminated production, look for all productions whose right side contains* **A**, *and replace each occurrence of* **A** *in each of these productions to obtain the non* **ε***-productions.*
- *Add the resultant non* **ε***-productions to the grammar to keep the language the same.*

Eliminate ε-Productions *Cont.*

- ***Example:*** *Remove the null productions from* ***G6****:*
- ***Solution:***

There are two null productions in the grammar $A \rightarrow \varepsilon$ *and* $B \rightarrow \varepsilon$.
To eliminate $A \rightarrow \varepsilon$ ***must*** *change the productions containing* A *in the right side. Those productions are* $S \rightarrow A\,B\,A\,C$ *and* $A \rightarrow a\,A$
So, replace each occurrence of A *by* ε. *Four new productions are in* ***(I)***

G6:	1. S → A B A C 2. A → a A | ε 3. B → b B | ε 4. C → c

	1. S → B A C | A B C | B C 2. A → a

I

Eliminate ε-Productions *Cont.*

- *Add these productions to the grammar and eliminate* $A \rightarrow \varepsilon$ *(II).*

G6:	1. $S \rightarrow ABAC \mid BAC \mid ABC \mid BC$ 2. $A \rightarrow aA \mid a$ 3. $B \rightarrow bB \mid \varepsilon$ 4. $C \rightarrow c$

II

	1. $S \rightarrow AAC \mid AC \mid C$ 2. $B \rightarrow b$

III

- *To eliminate* $B \rightarrow \varepsilon$*, change the productions containing B on the right side. Doing that generate new productions in (III).*

Eliminate ε-Productions *Cont.*

- *Add these productions to the grammar and remove the production* $B \rightarrow \varepsilon$ *from the grammar.*
- *The new grammar after removal of* **ε***-productions is:*

G6:	1. S → A B A C | A B C | B A C | B C | A A C | A C | C 2. A → a A | a 3. B → b B | b 4. C → c

Issues of CFGs

- *Writing Issues of a CFG include:*
  - ✓ *Ambiguity, Left Recursion, and Left Factoring*
  - ✓ *For example, consider a CFG of arithmetic expressions:*

G7:

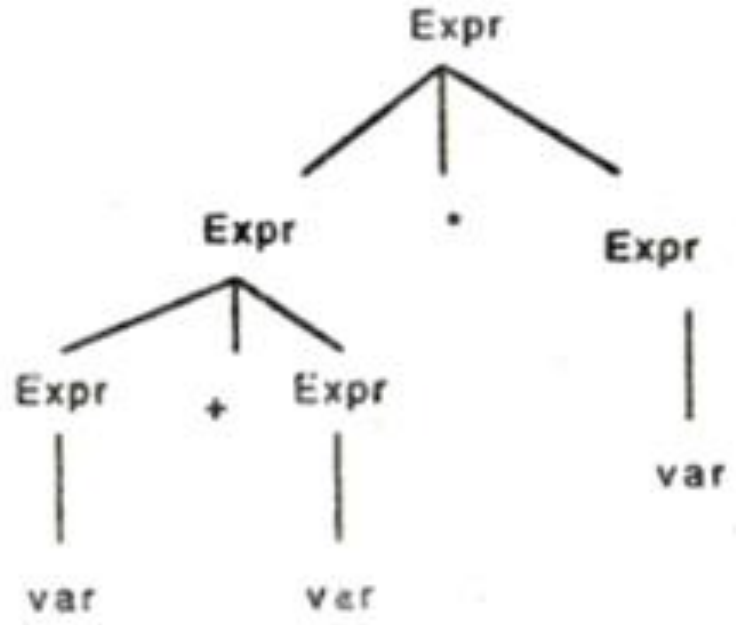
1. **Expr → Expr + Expr**
2. **Expr → Expr * Expr**
3. **Expr → (Expr)**
4. **Expr → var**
5. **Expr → const**

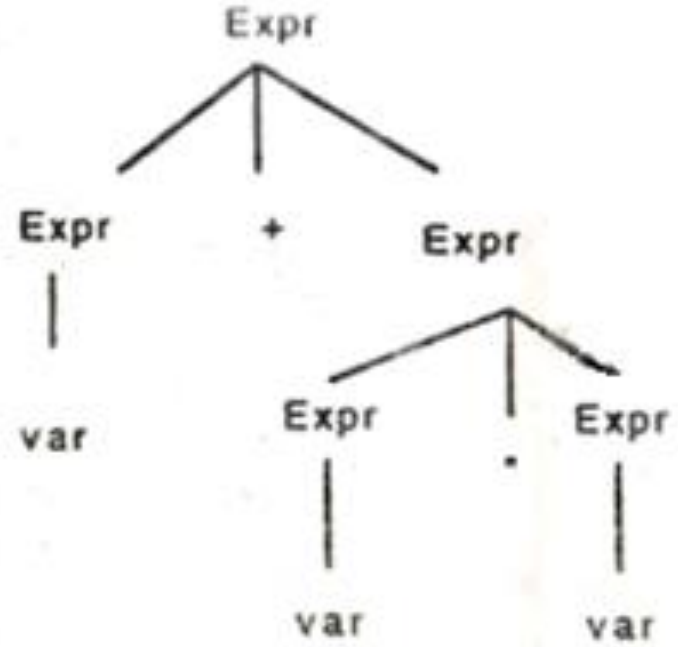
- ✓ *This is an ambiguous grammar and should be avoided.*

Ambiguity

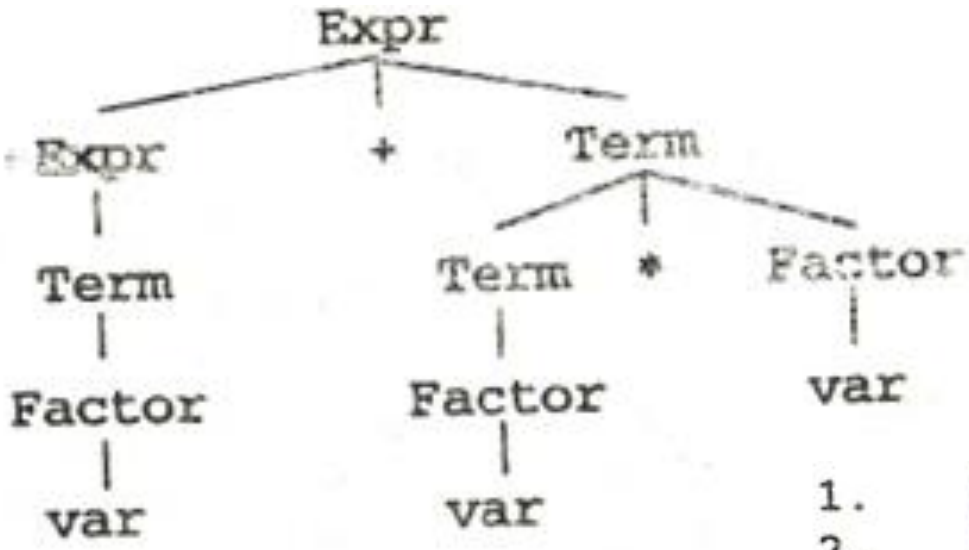
- *In natural languages, ambiguous phrases are those which may have more than one interpretation.*
- *A CFG is ambiguous* ***if there is more than one derivation tree for a particular string****.*
- *There are two different derivation trees for the string* ***var + var * var*** *using grammar* ***G7***
- *One way to resolve an ambiguity is to rewrite the grammar of the language to be unambiguous e.g.* ***G8*** *of* ***G7****.*
- *There is a derivation tree for* ***var + var * var*** *using grammar G8.*

Ambiguity *Cont.*





Ambiguity *Cont.*



1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var
7. Factor → const

Ambiguity *Cont.*

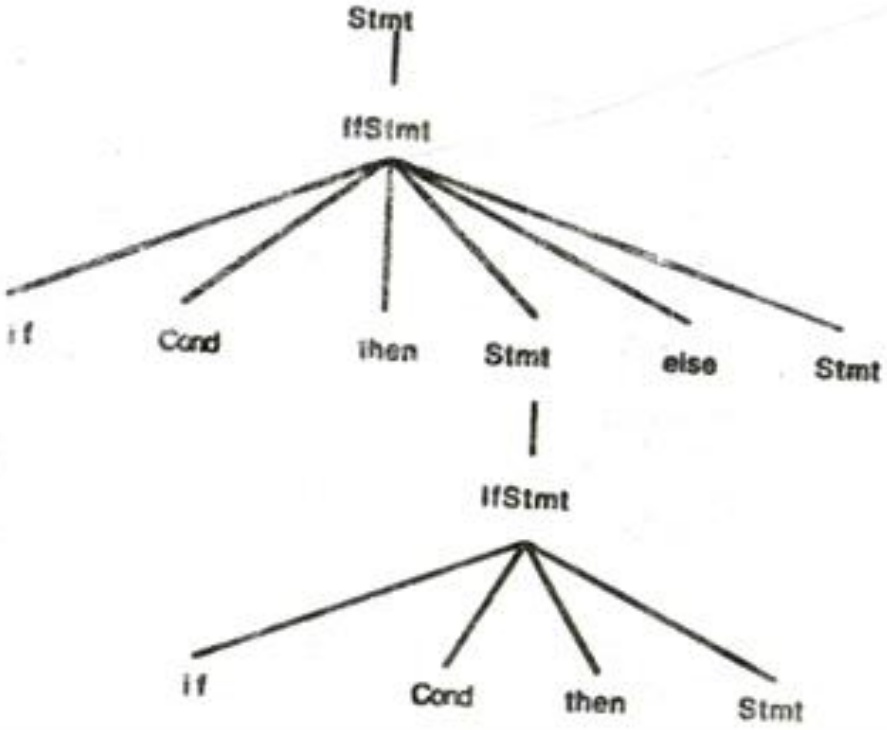
- *Another example of ambiguity in programming languages is the conditional statement as defined by grammar* ***G9****:*

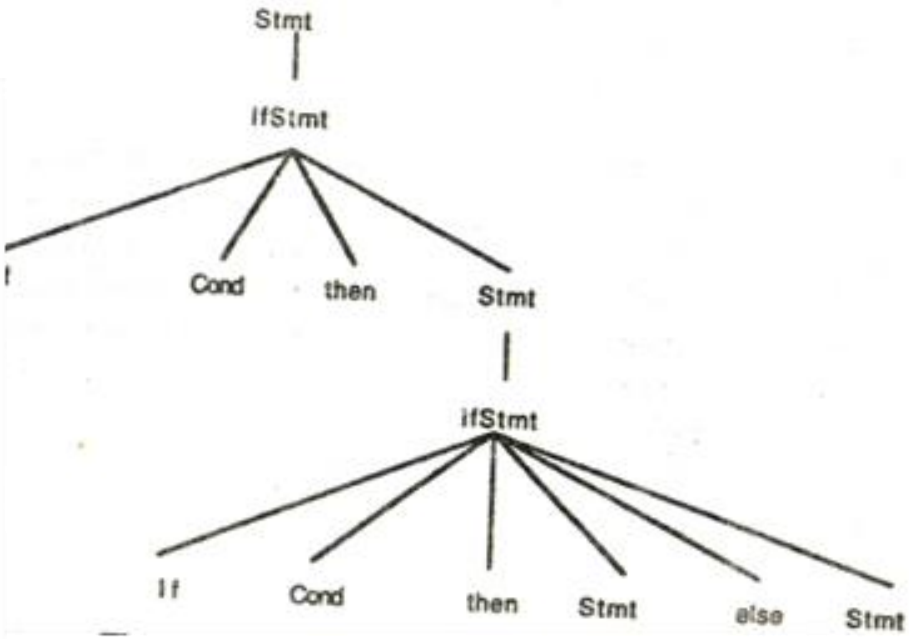
G9:

1. **Stmt → IfStmt**
2. **IfStmt → if Cond then Stmt**
3. **IfStmt → if Cond then Stmt else Stmt**

- *Two different derivation trees for: if Cond then if Cond then Stmt else Stmt are:*

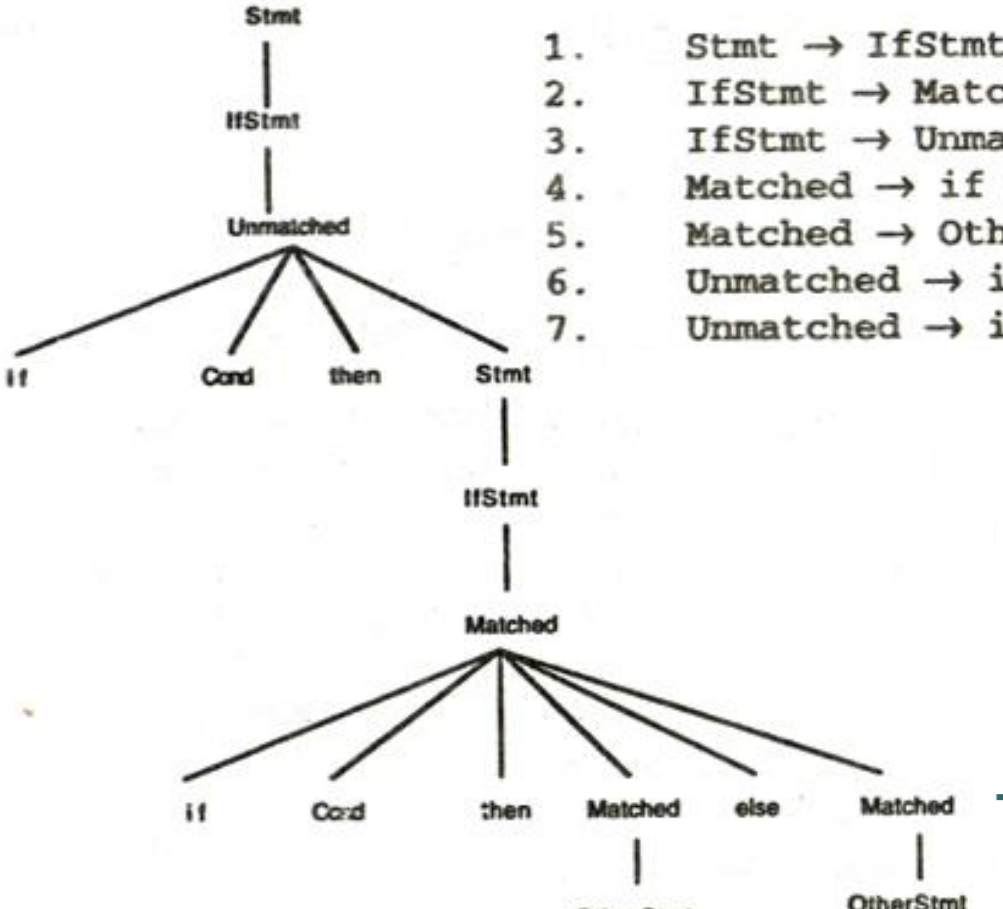
Ambiguity *Cont.*





Ambiguity *Cont.*

- *A derivation tree for* ***if Cond then if Cond then OtherStmt else OtherStmt*** *using grammar* ***G10*** *is:*



1. Stmt → IfStmt
2. IfStmt → Matched
3. IfStmt → Unmatched
4. Matched → if Cond then Matched else Matched
5. Matched → OtherStmt
6. Unmatched → if Cond then Stmt
7. Unmatched → if Cond then Matched else Unmatched

Left Recursion

- *A grammar is said to be immediately left recursive if there is* α *production of the form* $A \rightarrow A\,\alpha$.
- *Consider a grammar* **G** *with productions* $A \rightarrow A\,\alpha \mid \beta$
- *A new grammar* **G'** *defines the same language as* **G** *but without any left recursion, by replacing the above productions with*

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha\, A' \mid \varepsilon$$

- *In general, consider a grammar* **G** *with productions*

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \cdots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

where none of $\beta_1, \ldots, \beta_m$ *begin with an A.*

Left Recursion *Cont.*

- *A new grammar G'_7 without left recursion by replacing the above productions with*

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$$

- *The left recursion is removed as G_8:*

G7:

1. Expr → Expr + Expr
2. Expr → Expr * Expr
3. Expr → (Expr)
4. Expr → var
5. Expr → const

G_8:

1. Expr → Term Expr'
2. Expr' → + Term Expr' | ε
3. Term → Factor Term'
4. Term' → * Factor Term' | ε
5. Factor → (Expr) | var | const

G'_7:

1. Expr → Expr + Term | Term
2. Term → Term * Factor | Factor
3. Expr → (Expr) | var | const

Left-Factoring

- *Many simple parsing algorithms cannot cope with grammars with productions such as*

$$\langle \textit{if stmt} \rangle \rightarrow \textbf{\textit{if}} \langle \textit{exp} \rangle \textbf{\textit{then}} \langle \textit{stmt} \rangle \textbf{\textit{else}} \langle \textit{stmt} \rangle$$
$$\mid \textbf{\textit{if}} \langle \textit{exp} \rangle \textbf{\textit{then}} \langle \textit{stmt} \rangle$$

where two or more productions for a given non-terminal share a ***common prefix****.*

- *Factor-out the common prefix, the equivalent grammar rules:*

$$\langle \textit{if stmt} \rangle \rightarrow \textbf{\textit{if}} \langle \textit{exp} \rangle \textbf{\textit{then}} \langle \textit{stmt} \rangle \langle \textit{rest if} \rangle$$
$$\langle \textit{rest if} \rangle \rightarrow \textbf{\textit{else}} \langle \textit{stmt} \rangle \mid \varepsilon$$

Left-Factoring *Cont.*

- *In general, if situation of the productions are as follows:*

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

- *Then, it is difficult to decide whether to expand* **A** *to* $\alpha\beta_1$ *or to* $\alpha\beta_2$
- *We will rewrite the production:*

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$

- *After seeing the input derived from* α*, we expand*

  A′ to β_1 or to β_2.

Pushdown Machines

- *A **pushdown machine** is an abstract or theoretic machine.*
- *Pushdown machines can be used for **parsing**.*
- *A pushdown machine **consists of:***
  1. *A finite set of **states**, one is designated as the **starting state**.*
  2. *A finite set of input symbols, the **input alphabet**.*
  3. *An **infinite stack** and a finite set of **stack symbols**.*
     - *Pushed on or removed from it.*
     - *The stack symbol need not be distinct from the input symbols.*
     - *The stack must be initialized to contain at least one stack symbol before the first input symbol is read.*

Pushdown Machines *Cont.*

4. *A **state transition function** f (current state, input symbol, top stack symbol); its result is the new state of the machine.*
5. *On **each state transition,** the machine may **advance** to the next input symbol or **retain** the input pointer (i.e., not advance to the next input symbol).*
6. *On **each state transition,** the machine may perform one of the stack operations, **push(X)** or **pop**, where **X** is one of the **stack symbols**.*
7. *A **state transition** may include an **exit** from the machine labeled either **Accept** or **Reject**. This determines whether or not the input string is in the specified language.*

- *The pushdown machine is:* ***Infinite Stack + FSM***

G3:

1. S → ASB
2. S → ε
3. A → a
4. B → b

Pushdown Machines *Cont.*

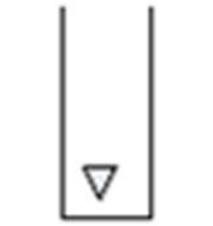
- A pushdown machine to accept the language of grammar **G3**
  - ✓ **rows** labeled by **stack symbols** and **columns** labeled by **input symbols.**
  - ✓ ↵ **end-marker**, indicating the end of the input string, and ▽ the **empty stack symbol**.
  - ✓ The states of the machine are **S1** (will always the starting state) and **S2**;
  - ✓ A **separate transition table** for each state.
  - ✓ Each cell shows a stack operation (**push(X)** or **pop**), an input pointer function (**advance** or **retain**), and the next state. "**Accept**" and "**Reject**" are exits from the machine.

G3:

1. S → ASB
2. S → ε
3. A → a
4. B → b

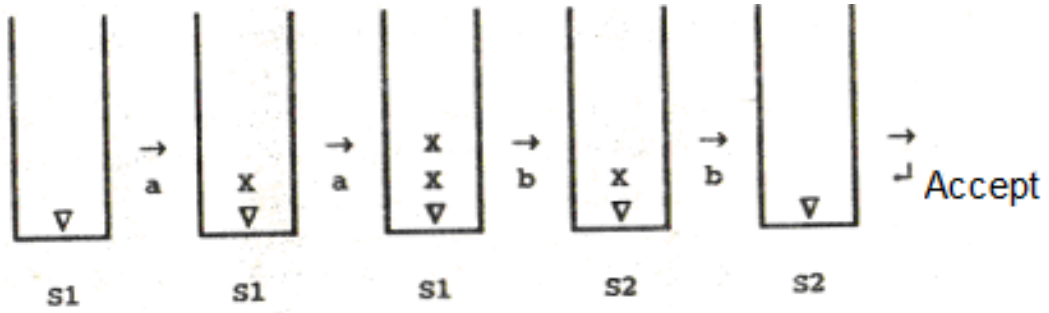
Pushdown Machines *Cont.*

S1	a	b	↵
X	Push (X) Advance S1	Pop Advance S2	Reject
▽	Push (X) Advance S1	Reject	Accept



S2	a	b	↵
X	Reject	Pop Advance S2	Reject
▽	Reject	Reject	Accept

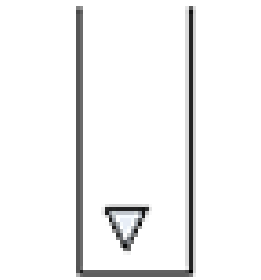
- Sequence of stacks as pushdown machine accepts the input string ***aabb*** is:



Pushdown Machines *Cont.*

- A pushdown machine to accept any string of well-balanced parentheses is:

S1	(	)	↵
X	Push (X) Advance S1	Pop Advance S2	Reject
▽	Push (X) Advance S1	Reject	Accept



Initial Stack

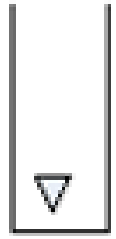
Pushdown Machines *Cont.*

- Pushdown translator for infix to postfix expressions.

S1	a	+	*	(	)	↵
E	Reject	Push (+)	Push (*)	Reject	Pop retain S3	Pop retain
Ep	Reject	Pop out (+)	Push (*)	Reject	Pop retain S2	Pop retain S2
L	Push (E) out (a)	Reject	Reject	Push (L)	Reject	Reject
Lp	Push (E) out (a)	Reject	Reject	Push (L)	Reject	Reject
+	Push (Ep) out (a)	Reject	Reject	Push (Lp)	Reject	Reject
*	Pop out (a*)	Reject	Reject	Push (L)	Reject	Reject
▽	Push (E) out (a)	Reject	Reject	Push (L)	Reject	Accept

S2	)	↵
+	Pop out (+) retain S3	Pop out (+) retain S1

S3	)
L	Rep (E) S1
Lp	Rep (Ep) S1
E	Pop retain
▽	Reject



Initial Stack

?