Chapter # 4
Context Free Grammar (CFG)

Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

1

Introduction

- A grammar is a set of rules by which the valid sentences in a language are constructed.
- Every language either that is natural language or artificial language have a certain grammar that helps in constructing the valid sentences in that language.
- In natural languages, invalid sentences violating the grammatical rules can be constructed and still they are understandable.
- But, this in not true in case of computer programming languages.

2

Example

- Some of the rules of English grammar are these:
  1. A *sentence* can be a *subject* followed by a *predicate*.
  2. A *subject* can be a *noun-phrase*.
  3. A *noun-phrase* can be an *adjective* followed by a *noun-phrase*.
  4. A *noun-phrase* can be an *article* followed by a *noun-phrase*.
  5. A *noun-phrase* can be a *noun*.
  6. A *predicate* can be a *verb* followed by a *noun-phrase*.
  7. A *noun* can be : *apple, bear, cat, dog*.
  8. A *verb* can be : *eats, follows, gets, hugs*.
  9. A *adjective* can be : *itchy, jumpy*.
  10. An *article* can be : *a, an, the*.
  11. A *predicate* can be a *verb.*

3

Example

- Now if have to form the sentence:

  The itchy bear hugs the jumpy dog.
- The sequence of application of the grammar rules to generated the above sentence is as follows:

1.	Sentence	→subject predicate	Rule 1
2.		→noun-phrase predicate	Rule 2
3.		→noun-phrase verb noun-phrase	Rule 6
4.		→article noun-phrase verb noun-phrase	Rule 4
5.		→article adjective noun-phrase verb noun-phrase	Rule 3
6.		→article adjective noun verb noun-phrase	Rule 5
7.		→article adjective noun verb article noun-phrase	Rule 4
8.		→ article adjective noun verb article adjective noun-phrase	Rule 3
9.		→ article adjective noun verb article adjective noun	Rule 5
10.		→ The adjective noun verb article adjective noun	Rule 10
11.		→ The itchy noun verb article adjective noun	Rule 9

4

Example

12. → The itchy bear verb article adjective noun — Rule 7
13. → The itchy bear hugs article adjective noun — Rule 8
14. → The itchy bear hugs the adjective noun — Rule 10
15. → The itchy bear hugs the jumpy noun — Rule 9
16. → The itchy bear hugs the jumpy dog — Rule 7

- The arrows indicates that a substitution was made according to the rules of grammar stated above.
- What we did:
  - We started with the initial symbol sentence.
  - We then applied the rules for producing the given sentence.
  - We then replaced the grammar words with the vocabulary words and hence get the required sentence.
- The words that cannot be replaced by anything are called terminal .
- The words that can be replaced with by other words are called non-terminals.

The sequence of application of the rules that produces the finished string of terminals from the starting symbol is called a derivation.

5

Syntax and Semantics

- Syntax is concerned with the grammatical structure of the sentences of the language.
  - Grammatical structure means the set of rules that are needed to construct valid sentences in the language.
- Semantics is concerned with the meaning of the sentence generated as a result of application of the grammatical rules.
- Sometimes, a sentence will be syntactically correct but semantically it will be incorrect.
- For example in the previous grammar we have:
  - Sentence → noun predicate.
  - Predicate → verb.

6

Syntax and Semantics

- By using this grammar, we can construct sentences like:
  - Birds sings.
  - Wednesday sings.
  - Coal mines sings.
- The first sentence is both synthetically and semantically correct.
- But, the last two are synthetically correct but semantically incorrect.
- For a sentence to be valid, it should be both synthetically and semantically correct.

7

Example

- Lets write grammar for valid arithmetic expressions.
  - Start → AE
  - AE → (AE + AE)
  - AE → (AE – AE)
  - AE → (AE * AE)
  - AE → (AE / AE)
  - AE → (AE ** AE)
  - AE → (AE)
  - AE → – (AE)
  - AE → Any-Number
  - Any-Number → First-Digit
  - First-Digit → First-Digit Other-Digit
  - First-Digit → 0 1 2 3 4 5 6 7 8 9
  - Other-Digit → 0 1 2 3 4 5 6 7 8 9

- Now using this grammar derive the expression:

1. (4-5)
2. ((5+4)*4)
3. ((9+5)*(8+2))

8

Grammar

- A grammar can be represented by four tuple:

  G = (Σ, N, S, P)

  - Σ is a finite non-empty set called alphabets.
    - The elements of Σ is called terminals and usually represented by lower-case letters .i.e. a, b, c etc.
  - N is a finite non-empty set of symbols such that N ∩ Σ =Ǿ.
    - The elements of N is called non-terminals are represented by uppercase letters .i.e. A, B, C etc.
  - S is a distinguished element of N called start symbol such that S € N.
  - P is the set of production rules or substitutions rules.
    - A production rule is of the form:

      Non-Terminal → finite set of Terminals and Non-Terminals.

9

Types of Grammar

- There are four types of grammars and is commonly called Chomsky hierarchy.
  - Type – 0 grammar.
  - Type – 1 grammar.
  - Type – 2 grammar.
  - Type – 3 grammar.

10

Type -0 grammar

- Type – 0 grammar is also called phrase structure grammar or un-restricted grammar.
- It is of four tuple (Σ, N, S, P).
- A production rule in an un-restricted grammar is of the form:

$$U \rightarrow V$$

  - Where U and V are strings of terminals, non-terminals or both of them.
  - Only restriction is that $U \neq \varepsilon$.
  - These production allows for a completely replacement of one string 'U' by another 'V'.
- The language generated by type-0 grammar is called type-0 language.

11

Example

- Let

  $\Sigma = \{a, c, d\}$

  $N = \{A, B, C\}$

  $S = \{A\}$

  and P is:

A	→	aBCc
aB	→	cad
Bc	→	aBa
BCc	→	Bcc
Cc	→	ε

Now to derive cad:

1. S → aBCc
2. aBCc → cadCc
3. cadCc → cad

Now to derive acadac

1. S → aBCc
2. aBCc →aBcc
3. aBcc → aaBac
4. aaBac → acadac

12

Type-1 grammar

- It is called context-sensitive grammar.
- It is also of four tuple (Σ, N, S, P).
- A production rule of the following form:

$$ά\, A\, β \rightarrow ά\, σ\, β$$

  - Where A is a non-terminal and $σ \neq ε$ is any non-empty string of terminals or non-terminals or both.
  - ά and β may either terminals, non-terminals or both.
  - The idea is that we may replace the non-terminal A by σ but only if A is surrounded by in the context of ά and β.

13

Type – 2 grammar

- It is also called context-free grammar (CFG).
- It is also of four tuple (Σ, N, S, P).
- A production rule of the following form:

$$A \rightarrow σ$$

  - Where A is a single non-terminal symbol and σ is any string of terminals or non-terminals or both.
- This is the most suitable grammar for computer languages and almost all computer programming languages are CFG.

14

Type – 3 grammar

- It is also called regular grammar or linear grammar.
- It is also of four tuple (Σ, N, S, P).
- In this type of grammar, we replace a single non-terminal with either a single terminal, a single terminal with a single non-terminal or ε.
- There are two types of this grammar.
  - Right linear or Right regular.
  - Left linear or Left regular.

15

Right Linear

- A grammar G is said to be of the right linear if every one of its productions has one of the following form.

$$A \rightarrow \varepsilon$$
$$A \rightarrow aB$$
$$A \rightarrow a$$

- Here A, B are non-terminals and ‘a’ is terminal.

16

Left Linear

- A grammar G is said to be of the left linear if every one of its productions has one of the following form.

$$A \rightarrow \varepsilon$$
$$A \rightarrow Ba$$
$$A \rightarrow a$$

- Here A, B are non-terminals and 'a' is terminal.

17

Context-free Language

- A language generated by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions.
- A language generated by a CFG is called context-free language.
  - It can also be said as the language defined by CFG or the language derived from the CFG or the language produced by the CFG.
- The language defined by a CFG can also be describe by a regular expression.
  - This can also be said as the language defined by a RE can also be defined by a CFG.

18

Example

- Let the only terminal be a.
- Let the production be:

  Prod 1 S $\rightarrow$ aS

  Prod 2 S $\rightarrow$ ε
- If we apply prod 1 six times and then apply prod 2, we generate the following:

$$
\begin{aligned}
S &\rightarrow aS \\
&\rightarrow aaS \\
&\rightarrow aaaS \\
&\rightarrow aaaaS \\
&\rightarrow aaaaaS \\
&\rightarrow aaaaaaS \\
&\rightarrow aaaaaa
\end{aligned}
$$

- This is a derivation of a^6 in this CFG.

19

Example

- The string a^n comes from n times application of prod 1 followed by one application of prod 2.
- If we apply prod 2 with out prod 1, we find that the null string is itself in the language of this CFG.
- Since the only terminal is a it is clear that no words outside of a* can possibly be generated.
- The language generated by this CFG is exactly a*.

20

Example

- Let the terminal be a and b.
- Let the non-terminals be S, X and Y.
- Let the production rules be:

S	→	X
S	→	Y
X	→	ε
Y	→	aY
Y	→	bY
Y	→	a
Y	→	b

- The language generated by this CFG is (a|b)*.

21

Example

- Let the terminal be a and b.
- Let the non-terminals be S and X.
- Let the production rules be:

S	→	XaaX
X	→	aX
X	→	bX
X	→	ε

- The language generated by this CFG is (a|b)*aa(a|b)*.
  - The language of all words with at least a double a in them somewhere.

22

RE and CFG

- Every language that can be described by a RE can also be described by a CFG.
- It is rather difficult to write grammar directly.
- FA can be converted into corresponding grammar.
- To convert FA to grammar, the following rules should be used.
  - For each state "i" of the FA create a non-terminal symbol "A".
  - If a state "i" has a transition to state "j" on a symbol "a" .i.e. $\delta(i, a) = j$, then introduce a production rule of the following form.

    $A_i \rightarrow aA_j$

23

RE and CFG

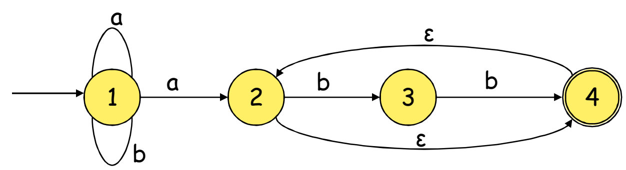
  - If state "i" goes to state "j" on input "ε", then introduce a production rule of the form.

    $A_i \rightarrow A_j$
  - If state "i" is an accepting state, then introduce a production rule of the form.

    $A_i \rightarrow \varepsilon$
  - If state "i" is the start state, then A_i is the start symbol (non-terminal) of the grammar.

24

Example



- Regular expression for the FA is (a|b)*a(bb)*.
- Its corresponding grammar will be:

  $\Sigma = \{a, b\}$

  $N = \{A_1, A_2, A_3, A_4\}$

  $S = \{A_1\}$

  The production rules will be:

25

Example

$A_1 \rightarrow aA_1$

$A_1 \rightarrow bA_1$

$A_1 \rightarrow aA_2$

$A_2 \rightarrow bA_3$

$A_2 \rightarrow A_4$

$A_3 \rightarrow bA_4$

$A_4 \rightarrow A_2$

$A_4 \rightarrow \varepsilon$

26

Example

- Consturct FA and grammar for the following RE.

(a|b)* bbb (a|b)*

Sentential and Sentence form of a Grammar

- A sentential form of a grammar G is any string X_i of symbols, such that X_i is set of terminals plus non-terminals or only non-terminals. Such that

  X_i Є Σ U N

  - If S $\overset{*}{\Rightarrow}$ ά, where ά contains of non-terminal, then we can say that ά is a sentential form of grammar.
- A sentential form of a grammar G that cannot be further derived or expended .i.e. a sentential form of a grammar G that consists of terminal symbols only is called sentence form of grammar.
  - If *w* Є L(G) and S $\overset{*}{\Rightarrow}$ *w*, where *w* is denotes the string contain only terminals symbols then w is called a sentence.

Types of Derivation

- Replacing of a non-terminal in the current state with its corresponding production rule in the grammar in order to obtained the required string is called derivation.
- Two types of derivation.
  - Left most derivation.
  - Right most derivation.

29

Types of Derivation

- Left-most Derivation.
  - The derivation in which only the left-most non-terminal in any sentential form is expanded at each step is called left most derivation.
- Right-most Derivation.
  - The derivation in which only the right-most non-terminal in any sentential form is expanded at each step is called left most derivation.

30

Example

- Consider the following grammar.

$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow a
\end{aligned}
$$

- No derive (a + a) by using both left-most and right-most derivations.

31

Using Right-most Derivation

$$
\begin{aligned}
E &\rightarrow (E) \\
&\rightarrow (E + T) \\
&\rightarrow (E + F) \\
&\rightarrow (E + a) \\
&\rightarrow (T + a) \\
&\rightarrow (F + a) \\
&\rightarrow (a + a)
\end{aligned}
$$

32

Using Left-most Derivation

$$
\begin{aligned}
E &\rightarrow (\ E\ ) \\
&\rightarrow (\ E + T\ ) \\
&\rightarrow (\ T + T\ ) \\
&\rightarrow (\ F + T\ ) \\
&\rightarrow (\ a + T\ ) \\
&\rightarrow (\ a + T\ ) \\
&\rightarrow (\ a + a\ )
\end{aligned}
$$

33

Example

- Derive the string (a + a * a) by using the grammar stated in the previous slides by using both left-most and right-most derivation.

34

Backus-Naur Form

- BNF stands for Backus-Naur Form or Backus Normal Form.
- A meta language is a language that is used to describe another language.
- BNF is a meta language (formal notations) for programming languages syntax.
- A production is a rule relating to a pair of strings, say ά and β, specifying how one may be transformed into the other. This may be denoted

  ά → β.
- For simple theoretical grammars, upper case letters are used for non-terminals and lower case letters are used for terminals.
- For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions is to use the notations invented by Backus commonly called BNF.
- These notations were first introduced by Backus for describing ALGOL 58.
- These notations were later modified slightly by Peter Naur for the description of ALGOL 60.

35

Backus-Naur Form

- In BNF:
  - A non-terminal and terminal are usually given some descriptive names.
  - Non-terminals symbols are written in angle brackets to distinguish it from a terminal symbol.
  - If there are multiple definitions for the same non-terminal symbol, then they can be written as single rule, separated from each by using (|) vertical bar which means logical OR.

36

Example

```
G = {N , T , S , P}
N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }
T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps ,
      mystifies , tall , thin , thirsty }
S = <sentence>
P = {  <sentence>        →  the <qualified noun> <verb>      (1)
                          | <pronoun> <verb>                 (2)
       <qualified noun>  →  <adjective> <noun>               (3)
       <noun>            →  man | girl | boy | lecturer      (4, 5, 6, 7)
       <pronoun>         →  he | she                         (8, 9)
       <verb>            →  talks | listens | mystifies      (10, 11, 12)
       <adjective>       →  tall | thin | sleepy             (13, 14, 15)
    }
```

- Derive the "The sleepy boy listens" by using the above grammar.

37

Parse Trees

- A grammar naturally describe the hierarchical syntactic structure of the sentences of the language they define.
- The hierarchical structure is called Parse Tree, Syntax Tree, Derivation Tree or Production Tree..
- To draw a parse tree for a sentence generated by a grammar:
  - We start with the start symbol "S".
  - Every time we used to replace a non-terminal by a string, we draw downward lines from the non-terminal to each character (terminal and non-terminal) in the production rule.
  - These replacements are continued until we label the downward lines with terminal symbols only (leaf nodes).
  - Every internal node of a parse tree is labeled with a non-terminal symbol and leaf nodes are labeled with terminal symbols.

38

Example

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

- Now to derive and also draw parse tree for the following expression by using the above grammar.

  A = B * (A + C)

- Also draw parse tree for the sentence derived in slide # 39.

Problems of a CFG

- Three types of problems are mainly faced in a CFG.
  - Ambiguity.
  - Left Recursion.
  - Common Prefixes.
- Three problems must be removed from a CFG, otherwise the grammar will not work accurately.

Ambiguity

- An ambiguous grammar is one that:
  - Produces more than one parse trees for the same sentence.
  - Produces more than one leftmost derivations or rightmost derivations for the same sentence.
- A grammar becomes ambiguous when a single non-terminal appears twice or more times on the L.H.S of the production rules in the grammar.
- If more than one parse trees can be produced for a sentence; then the compiler would not be able to generate the code uniquely.

Example

- Consider the following grammar.

  <assign> → <id> = <expr>
  <id> → A | B | C
  <expr> → <expr> + <expr>
  | <expr> * <expr>
  | (<expr>)
  | <id>

- Now show that this grammar is ambigous for the sentence A = B + C * A.

Elimination of Ambiguity

- Consider the following grammar.

<if_stmt> → **if** <logic_expr> **then** <stmt>
| **if** <logic_expr> **then** <stmt> **else** <stmt>
<stmt> → <if_stmt>

- Now to show that this grammar is ambiguous for the following sentence.

if <logic_expr> **then if** <logic_exper> **then** <stmt> **else** <stmt>

- Now to eliminate ambiguity from the above grammar, we have to rewrite the above grammar.

43

Elimination of Ambiguity

- To rewrite unambiguous grammar, we have to following the following rule.
  - The rule for if statement in most languages is that an else clause, when present, is matched with the nearest previous unmatched then.
  - Therefore, between a then and its matching else, there cannot be an if statement without an else.
  - So for this situation, statements must be distinguished between those that are matched and those that are unmatched.
  - Where unmatched statements are else – less ifs.

44

Elimination of Ambiguity

- So the unambiguous grammar will be:

<stmt> → <matched> | <unmatched>
<matched> → if <logic_expr> then <matched> else <matched>
| any non-if statement
<unmatched> → if <logic_expr> then <stmt>
| if <logic_expr> then <matched> else <unmatched>

- Now try this out on the previous sentence.

45

Left Recursion

- A grammar is said to be left recursive, if it has a a non-terminal ‘A’ such that there is a derivation

  $A \stackrel{*}{\rightarrow} A\acute{\alpha}$

  for some string x.
- When a grammar rule has its L.H.S also appearing at the beginning of its R.H.S, the rule/grammar is said to be left recursive.
- For example

  $S \rightarrow Sa$

  Now, replacing S by Sa we get Saa, then Saaa and then Saaaa and so on.
- Top-down parsing method cannot handle left-recursive grammar, so it has to be eliminated.

46

Left – Recursive Removal

- Consider the following grammar.

  $A \rightarrow A\acute{\alpha} \mid \beta$

  This is a left recursive grammar and it can be removed by replacing it with the following productions.

  $A \rightarrow \beta A^{/}$

  $A^{/} \rightarrow \acute{\alpha} A^{/} \mid \varepsilon$

47

Example

- Consider the grammar.

  $S \rightarrow Ab \mid b$

  $A \rightarrow Ac \mid Sd \mid \varepsilon$

- This grammar can be expanded as:

  $S \rightarrow Ab \mid b$

  $A \rightarrow Ac \mid Abd \mid bd \mid \varepsilon$

- After removing left – recursion we get.

  $S \rightarrow Ab \mid b$

  $A \rightarrow bdA^{/} \mid \varepsilon A^{/}$

  $A^{/} \rightarrow cA^{/} \mid bdA^{/} \mid \varepsilon$

48

Example

- Consider the following left – recursive grammar.

$$E \rightarrow E + T \mid T$$

- In this case

$$+T = ά$$
$$T = \beta$$

- By eliminating left – recursion, it can be written as:

$$E \rightarrow TE^{/}$$
$$E^{/} \rightarrow +TE^{/} \mid \varepsilon$$

49

Common Prefixes

- When a grammar has two or more productions with the same non-terminal of the L.H.S and the same prefixes on their R.H.S but different suffixes.
- Consider the following grammar:

$$A \rightarrow ά\beta_1$$
$$A \rightarrow ά\beta_2$$

  - So, it is not clear that which of the two alternative productions is to be used to expand non – terminal ‘A’.
- The removal of common prefixes is called left factoring,

50

Common Prefixes Removal

- Consider the following grammar:

$$A \rightarrow ά\beta_1$$
$$A \rightarrow ά\beta_2$$

- After left factoring, it will be as:

$$A \rightarrow άA^{/}$$
$$A^{/} \rightarrow \beta_1 \mid \beta_2$$

51

Example

- Consider the following grammar:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

- After left factoring, we get:

$$S \rightarrow iEtSS^{/} \mid a$$
$$S^{/} \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

52

Operator Precedence

- Grammar is used to create parse tree for a sentence.
- Therefore, parse tree is used to determine the meaning of the sentence.
- In case of mathematical expression:
  - An operator in an arithmetic expression that is generated lower in the parse tree has precedence over an operator produced higher up in the tree.
- If we consider the grammar shown in slide # 44 and create parse trees for the given expressions.
  - We will find that in one tree the multiplication operator is generated lower in the tree, indicating that it has precedence over the addition operator in the expression.
  - The second parse tree indicates just the opposite.

53

Operator Precedence

- Therefore an ambiguous grammar can also create the problem of operator precedence.
- To maintain the operator precedence, we have to restructure the grammar.
- We have to write the grammar in such away so that can easily reflect the operator precedence.
- In case of the grammar shown on slide # 44.
  - A grammar can be written to separate the addition and multiplication operators so they are consistently in a higher-to-lower ordering, respectively, in the parse tree.
  - This ordering can be maintained regardless of the order in which the operators appear in an expression.
  - The correct ordering is specified by using separate rules for the operands of the operator that have different precedence.
  - This requires additional non-terminals and some new rules.

54

Operator Precedence

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <term> + <expr>
| <term> - <expr>
| <term>
<term> → < factor > * <term >
| < factor > / <term>
| <factor>
<factor> → (<expr>)
| <id>

55

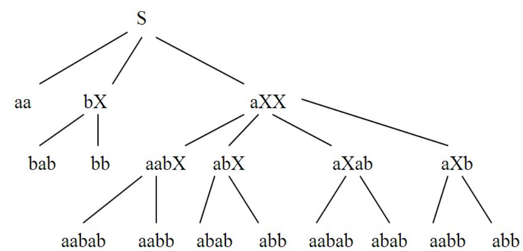
The Total – Language Tree

- A total – language tree shows the generation of all the words in the language of CFG simultaneously in one big (possibly infinite) tree.
- For a given CFG, total – language tree is:
  - Start with the start symbol S as its root and whose nodes are working strings of terminals and non-terminals.
  - The descendent of each node are all the possible results of applying every production of the non-terminals in the node, one at a time.
  - A string of all terminals is the terminal node in the tree.
  - The resultant tree is called the total language tree of the CFG.

56

The Total – Language Tree

- Example : For the CFG
  - S → aa | bX | aXX
  - X → ab | b
- The total language tree is:



- This total language has only seven different words.
- Four of its words { abb, aabb, abab, aabab } have two different possible derivations.

57

Class Work

- Find a CFG for each of the languages defined by the following regular expressions.
  1. aa*bb*
  2. (a|b)* a (a|b)* a (a|b)*
  3. b* a (a|b)* a b*
- Consider the CFG

  S → aX
  X → aX | bX | ε

  What is the language this CFG generates.
- Consider the CFG

  S → XaXaX
  X → aX | bX | ε

  What is the language this CFG generates.
- Consider the CFG

  S → aS | bb

  Prove that this grammar generates the language defined by the RE a*bb.

58

- End of Chapter # 4