Chapter # 5
Parsing Mechanisms

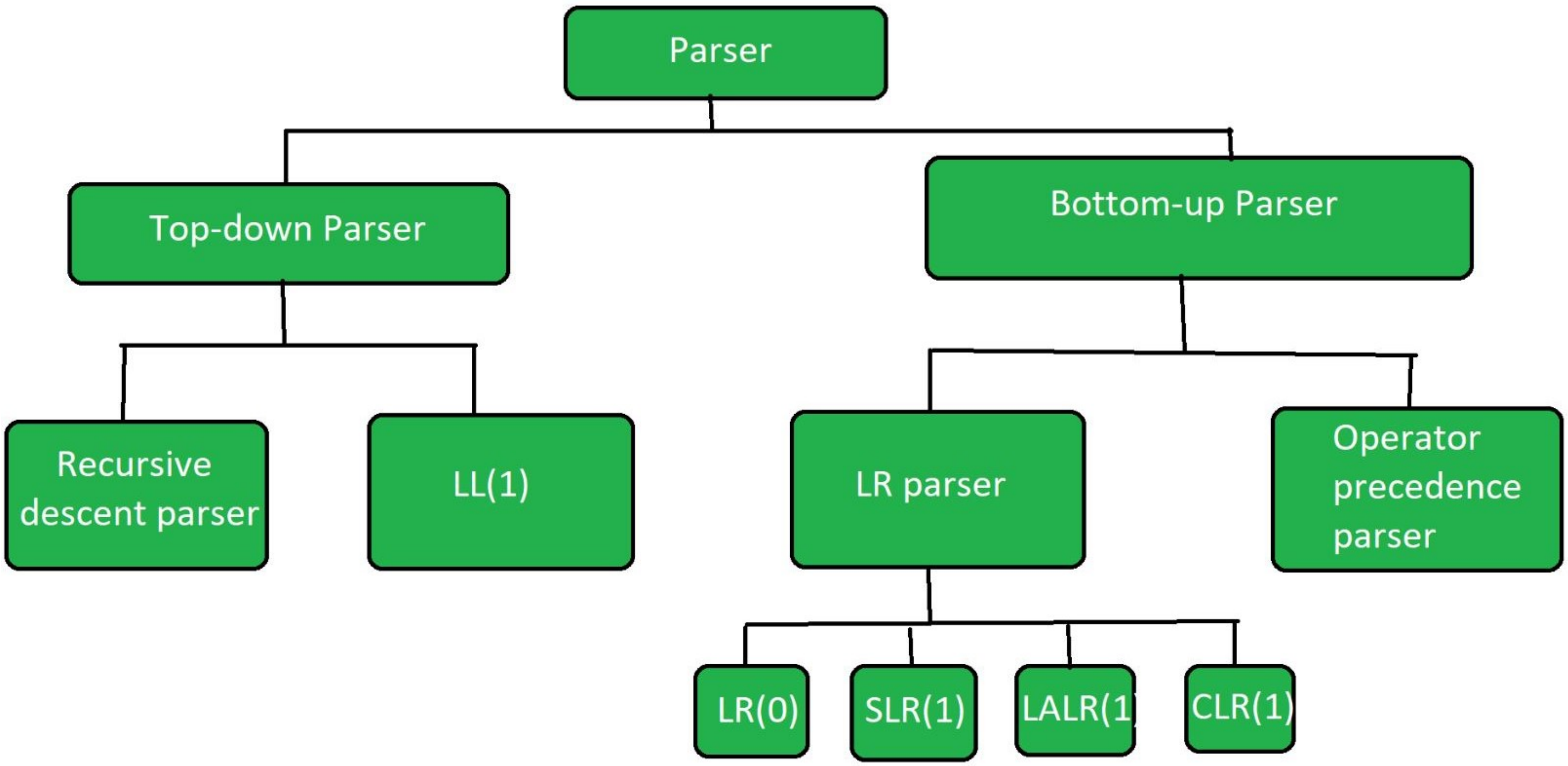
Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

Parser and Parsing

- **Parser** is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree.
  - Parser is also known as Syntax Analyzer.

- Parsing is a process that construct a syntactic structure (i.e., parse tree) from the stream of tokens
  - Parsing is the process of determining if a string of tokens can be generated by a grammar

Types of Parser



Types of Parser

- There are two types of parsers:
  - Top Down Parser (LL Parser).
    - Recursive Descent Parser.
    - Predictive Parser.
    - Non-Recursive Predictive Parser

    } LL(1)

  - Bottom-Up Parser (LR Parser).
    - Shift Reduce Parser
    - Simple LR Parser.
    - Canonical LR Parser.

    } LR(1)
- LL(1) means "L for left-to-right scanning of the input and L is for left most derivation and only one non-terminal expanded at each step.
- LR(1) means "L for left-to-right scanning of the input and R is for right most derivation and only one non-terminal expanded at each step.

Top-Down Parser

- Top-Down parsing can be viewed as an attempt to find a left-most derivation for an input string.
- We can say that to construct a parse tree for the input starting form the root and creating the nodes of parse tree in preorder.
- It works as under:
  - Expand the start symbol of a grammar into the string (on RHS of the start symbol).
  - At each expansion step, the non terminal symbol in the LHS of a particular production is replaced by the RHS of that production.
  - If the substitution is chosen correctly at each step, a left most derivation is traced out.

Example.

- Consider the following grammar:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow - (E)$$

$$E \rightarrow id$$

Now derive the string - (id + id).

Types of Top-Down Parsing.

- There are three types of Top-Down Parsers:
  - Recursive Descent Parser.
  - Predictive Parser.
  - Non-Recursive Predictive Parser.

Recursive Descent Parser.

- In this type of Top-Down Parsing, a non-terminal of the current derivation step is expanded using the production rule in the given grammar.
- If the expansion does not gives the desired result, the parser drops the current production and applies another production corresponding to the same non-terminal symbol.
- This process is repeated until the required result is obtained.
- The process of dropping the previous production and applying a new production is called BACKTRACKING.

Recursive Descent Parser.

- BackTracking occurs in Recursive Descent Parsers
  - Grammars that include multiple production for a single non-terminal and not left factored

- Disadvantage:
  - The main disadvantage of this technique is that it is slow because of backtracking.
  - When a grammar with left recursive production is given, then the parser might get into infinite loop. Hence, left recursion must be eliminated.

Example 1

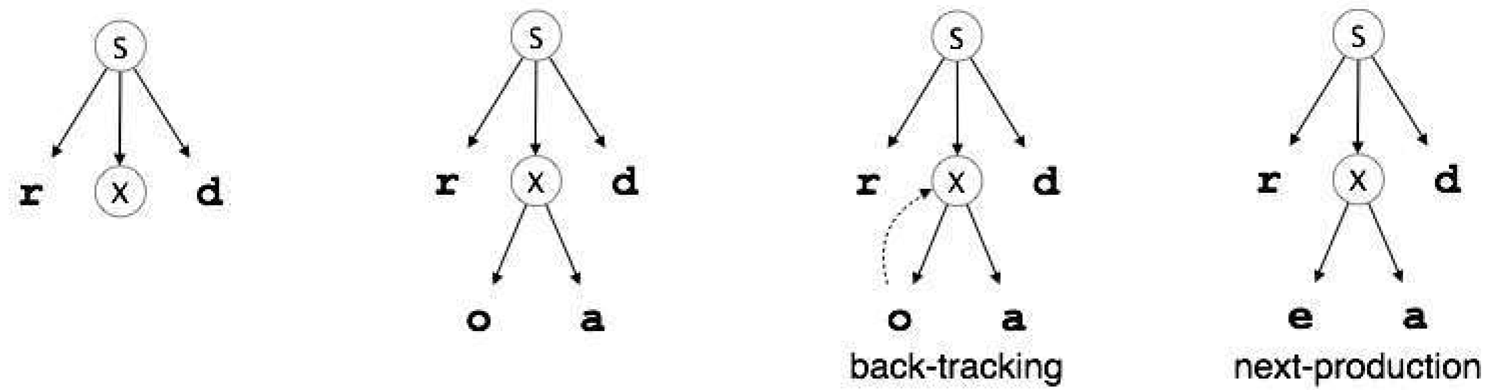
- Consider the grammar

  $S \rightarrow rXd \mid rZd$

  $X \rightarrow oa \mid ea$

  $Z \rightarrow ai$

- For an input string: read



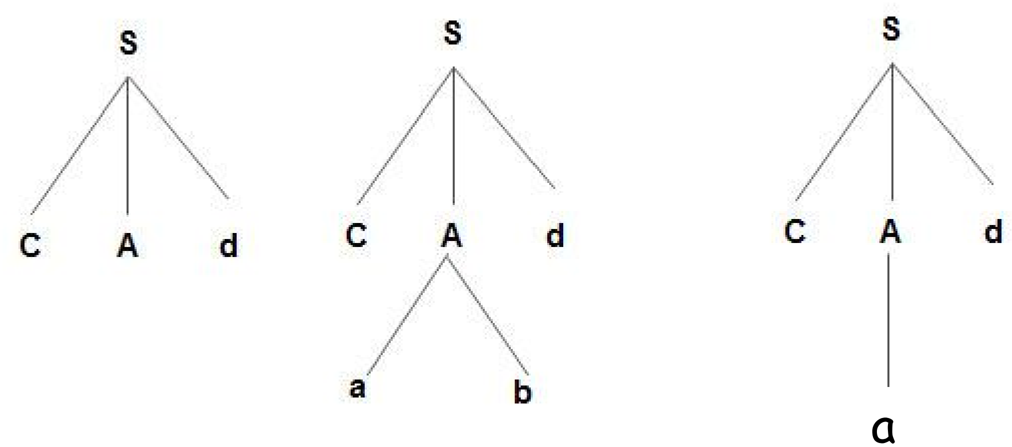
Example 2

- Consider the grammar:

  S → cAd

  A → ab | a

  Now derive the string cad.



Predictive Parsing.

- It is a special case of Recursive Descent Parser.
- In this parsing method the backtracking is removed.
  - In many cases, by eliminating left recursion and left factoring (common prefixes) form a grammar, we can obtain a grammar that can be parsed by a Recursive Descent Parser that needs no backtracking.
- This type of parsing technique works by attempting to predict the appropriate production to expand the non-terminal at the current derivaiton step, in case more than one productions corresponds to the same non-terminal.

Predictive Parsing.

- To construct a predictive parser, we must know:
  - Given the current input symbol α and the non-terminal to be expanded, which one of the alternatives of production A → α1 | α2 | α3 | ---- | αn is the unique alternative that derives a string beginning with α.
  - That is, the proper alternative must be detectable by looking at only the first symbol it derives.
- For example , if we have the productions:

stmt → if *expr* than *stmt* else *stmt*

| while *expr* than stmt

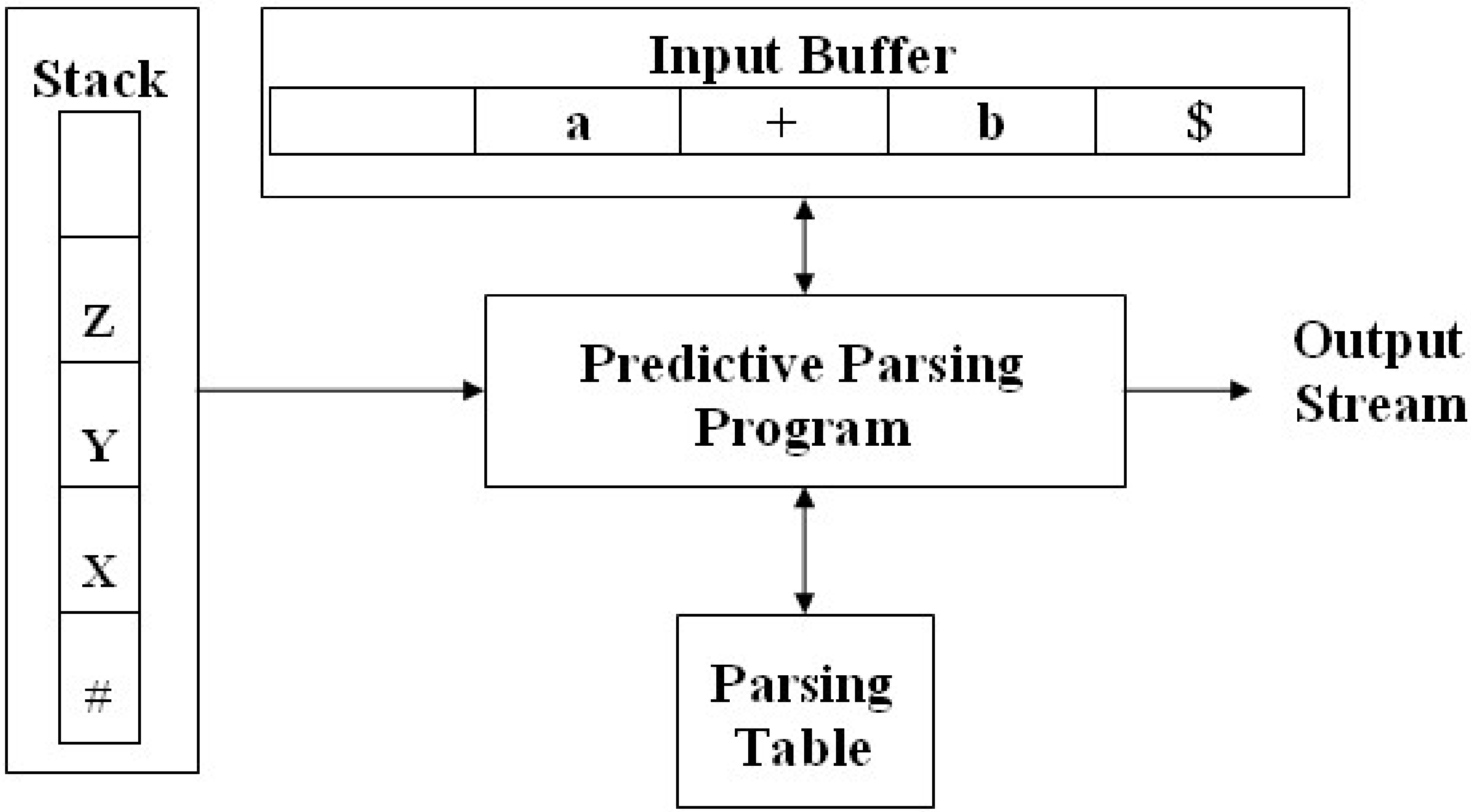
| begin *stmt_list* end

Then the keywords *if, while, begin* tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Non-Recursive Predictive Parser.

- The key problem in the predictive parsing is that of determining the production to be applied for a non-terminal.
- The Non-Recursive Predictive Parser is the implementation of Predictive Parser and solves the problem by implementing an implicit stack and parsing table.
- The Non-Recursive Predictive Parser looks up the production to be applied in a parsing table.
- The parsing table can be constructed directly from certain grammar.

Model of a Non-Recursive Predictive Parser:



Model of a Non-Recursive Predictive Parser.

- Input Buffer:
  - The input buffer contains the string to be parsed followed by $, a symbol used to indicate the end of the input string.
- Stack:
  - The stack contains a sequence of grammar symbols (terminal and non-terminal) with # or $ indicating the bottom of the stack.
- Parse Table:
  - A two dimensional array M[A,a], where A is a non-terminal and a is a terminal or the symbol $

Functions of Non-RPP

- Non-Recursive Predictive Parsing process may include the following functions.
- Considering X, the symbol on top of the stack and a the current input symbol.
  - If X = a = $, the parser halts and announces successful completion of parsing.
  - **POP:**
    - If X = a not equal to $, the parser pops X off the stack and advances the input pointer to the next input symbol.
  - **Apply:**
    - If X is a non-terminal, then X will be popped from the stack.
    - The parser consult M[X,a] of the parsing table M.

Functions of Non-RPP

- This entry will be either an X-production of the grammar or an error entry.
- If, for example, M[X,a] = { X → UVW}, the parser replaces X on top of the stack by WVU (with U on top).

– **Rejects:**
  - If M[A,a] = error, the parser calls an error recovery routine.

– **Accepts:**
  - If the current input is \$ .i.e. a = \$ and top of the stack is \$ .i.e. X = \$, then parser will declare the validity of the input string and give output as the structure of the parser.

FIRST and FOLLOW Sets

- The construction of a non-recursive predictive parser is aided by two functions associated with a grammar G
- These functions, FIRST and FOLLOW, allow us to fill in the entries of a parsing table for G, whenever possible
- We need to find FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position

Why FIRST Set

- If the compiler would have come to know in advance
  - what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees
  - It can wisely take decision on which production rule to apply

```
S -> cAd
A -> bc|a

And the input
string is "cad".
```

If it knew that after reading character 'c' in the input string and applying S->cAd, next character in the input string is 'a'

It would have ignored the production rule A->bc (because 'b' is the first character of the string produced by this production rule, not 'a')

Directly used the production rule A->a (because 'a' is the first character of the string produced by this production rule, and is same as the current character of the input string which is also 'a').

Why FIRST Set

- Hence it is validated
  - If the compiler/parser knows about first character of the string that can be obtained by applying a production rule
  - I can wisely apply the correct production rule to get the correct syntax tree for the given input string

Why FOLLOW Set

- The parser faces one more problem
- Let us consider below grammar to understand this problem

```
A -> aBb
B -> c | ε
And suppose the input string is “ab” to parse.
```

- As the first character in the input is a, the parser applies the rule A->aBb

```
  A
/ | \
a B b
```

- Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can’t get any string derivable from B that contains b as first character

Why FOLLOW Set

- But the Grammar does contain a production rule B -> ε
  - if that is applied then B will vanish, and the parser gets the input "ab"
  - But the parser can apply it only when it knows that the character that follows B is same as the current character in the input
- In RHS of A -> aBb
  - b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b
  - Hence the parser applies this rule. And it is able to get the string "ab" from the given grammar

Rules to Compute FIRST Set

- If X is a non-terminal symbol then
  - FIRST(X) is the set of terminals that begin the strings derivable from X
- If X is a non-terminal and have production rule X-> Ɛ, then add Ɛ to FIRST(X)
- If X->Y1 Y2 Y3….Yn is a production,
  - FIRST(X) = FIRST(Y1)
  - If FIRST(Y1) contains Ɛ then FIRST(X) = { FIRST(Y1) – Ɛ } U { FIRST(Y2) }
  - If FIRST (Yi) contains Ɛ for all i = 1 to n, then add Ɛ to FIRST(X)
- If x is a terminal, then FIRST(x) = { ‘x’ }

Example 1

Production Rules of Grammar

E -> TE'

E' -> +T E'|ϵ

T -> F T'

T' -> *F T' | ϵ

F -> (E) | id

FIRST sets

FIRST(E) = FIRST(T) = { (, id }

FIRST(E') = { +, ϵ }

FIRST(T) = FIRST(F) = { (, id }

FIRST(T') = { *, ϵ }

FIRST(F) = { (, id }

Example 2

```
Production Rules of Grammar
S -> ACB | Cbb | Ba
A -> da | BC
B -> g | Є
C -> h | Є
```

FIRST sets

```
FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C)
         = { d, g, h,    b, a}
FIRST(A) = { d } U FIRST(B) = { d, g , h, Є }
FIRST(B) = { g , Є }
FIRST(C) = { h , Є }
```

Example 3

Grammar

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \in$

$D \rightarrow EF$

$E \rightarrow g \mid \in$

$F \rightarrow f \mid \in$

First Functions-

First(S) = { a }

First(B) = { c }

First(C) = { b , ∈ }

First(D) = { First(E) - ∈ } ∪ First(F) = { g , f , ∈ }

First(E) = { g , ∈ }

First(F) = { f , ∈ }

Rules to Compute FPLLOW Set

- Compute FOLLOW set for every non-terminal using the RHS of the production rules of the grammar
  - Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form
  - If X is the starting symbol of a grammar, then include $ in the FOLLOW(X) such as FOLLOW(X) = {$}
  - If there is a production A -> α Bβ, then everything in FIRST(β), except for ε, is placed in FOLLOW(B)
  - If there is a production A => α Bβ where FIRST(β) contains ε (i.e., β => ε), then everything in FOLLOW(β) is in FOLLOW(B) Such FOLLOW(B) = {First(β)- ε} U FOLLOW(β)
  - If there is a production A => α B then include everything in FOLLOW(A) in the FOLLOW(B) such that FOLLOW(B) = FOLLOW(A)

Example 1

```
Production Rules:
E -> TE’
E’ -> +T E’|Є
T -> F T’
T’ -> *F T’ | Є
F -> (E) | id

FIRST set
FIRST(E) = FIRST(T) = { ( , id }
FIRST(E’) = { +, Є }
FIRST(T) = FIRST(F) = { ( , id }
FIRST(T’) = { *, Є }
FIRST(F) = { ( , id }

FOLLOW Set
FOLLOW(E)  = { $ , ) }  // Note  ')' is there because of 5th rule
FOLLOW(E’) = FOLLOW(E) = {  $, ) }  // See 1st production rule
FOLLOW(T)  = { FIRST(E’) – Є } U FOLLOW(E’) = { + , $ , ) }
FOLLOW(T’) = FOLLOW(T) =      { + , $ , ) }
FOLLOW(F)  = { FIRST(T’) –  Є } U FOLLOW(T’) = { *, +, $, ) }
```

Example 2

S => A a

A => B D

B => b | ε

D => d | ε

First(S) = {b, d, ε}

First(A) = {b, d, ε}

First(B) = {b, ε}

First(D) = {d, ε }

Follow(S) = {$}

Follow(A) = {a}

Follow(B) = {d, a}

Follow(D) = {a}

Example 3

Grammar	**Follow Functions-**
$S \rightarrow aBDh$	Follow(S) = { \$ }
$B \rightarrow cC$	Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
$C \rightarrow bC \mid \in$	Follow(C) = Follow(B) = { g , f , h }
$D \rightarrow EF$	Follow(D) = First(h) = { h }
$E \rightarrow g \mid \in$	Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
$F \rightarrow f \mid \in$	Follow(F) = Follow(D) = { h }

- End of Chapter # 5