



COMPILER DESIGN

SUBJECT CODE: 203105351

Pratibha Shakya, Assistant Professor
Computer Science & Engineering



CHAPTER-2

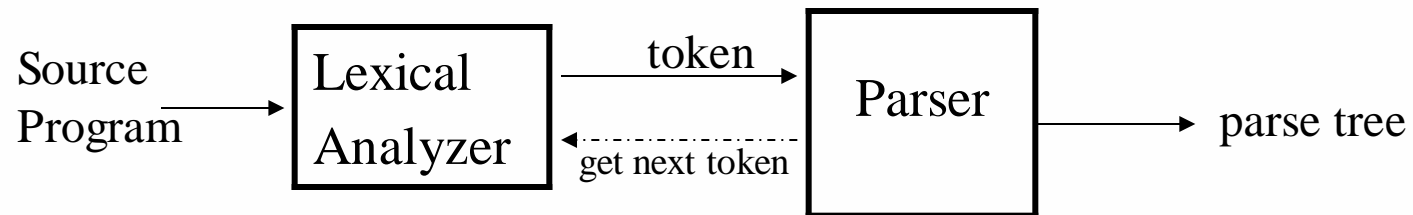
Introduction to syntax analysis

Syntax Analyzer

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
 - If it satisfies, the parser creates the parse tree of that program.
 - Otherwise the parser gives the error messages.
- A context-free grammar
 - gives a precise syntactic specification of a programming language.
 - the design of the grammar is an initial phase of the design of a compiler.
 - a grammar can be directly converted into a parser by some tools.

Parser

- Parser works on a stream of tokens.
- The smallest item is a token.



Parsers (cont.)

- We categorize the parsers into two groups:
 1. **Top-Down Parser**
 - the parse tree is created top to bottom, starting from the root.
 2. **Bottom-Up Parser**
 - the parse is created bottom to top; starting from the leaves
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
 - LL for top-down parsing
 - LR for bottom-up parsing

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
 - A finite set of terminals (in our case, this will be the set of tokens)
 - A finite set of non-terminals (syntactic-variables)
 - A finite set of productions rules in the following form
 - $A \rightarrow \alpha$ where A is a non-terminal and
 α is a string of terminals and non-terminals (including the empty string)
 - A start symbol (one of the non-terminal symbol)
- Example:
 - $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$
 - $E \rightarrow (E)$
 - $E \rightarrow \text{id}$

Derivations

$E \Rightarrow E+E$

- $E+E$ derives from E

- we can replace E by $E+E$
- to be able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of $id+id$ from E .

- In general a derivation step is

$\alpha A \beta \Rightarrow \alpha \gamma \beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar
where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

- \Rightarrow : derives in one step
- \Rightarrow : derives in zero or more steps
- \Rightarrow : derives in one or more steps

CFG - Terminology

- $L(G)$ is *the language of G* (the language generated by G) which is a set of sentences.
- A *sentence of $L(G)$* is a string of terminal symbols of G .
- If S is the start symbol of G then
 ω is a sentence of $L(G)$ iff $S \Rightarrow \omega$ where ω is a string of terminals of G .
- If G is a context-free grammar, $L(G)$ is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow \alpha$ - If α contains non-terminals, it is called as a *sentential form* of G .
 - If α does not contain non-terminals, it is called as a *sentence* of G .

Derivation Example

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

OR

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

Right-Most Derivation

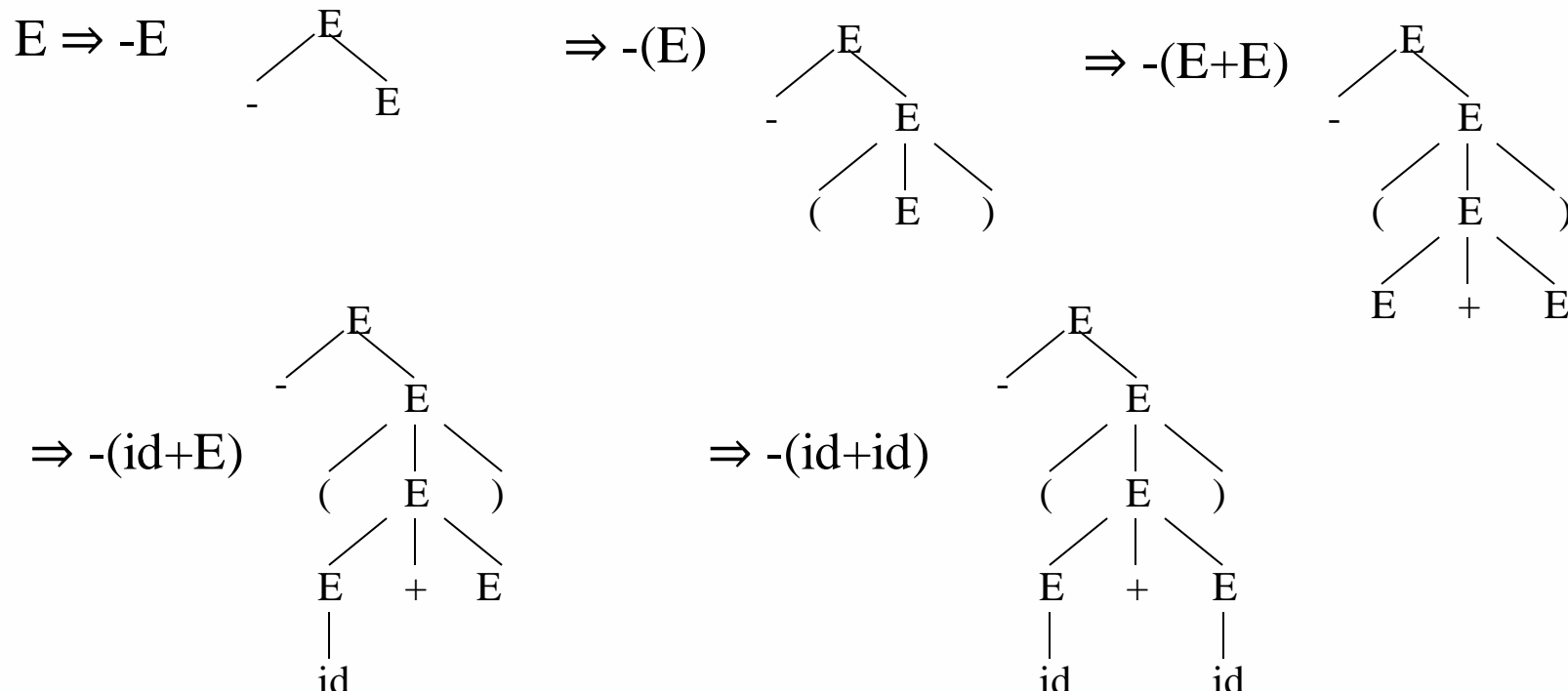
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

- We will see that the top-down parsers try to find the left-most derivation of the given source program.
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.



Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

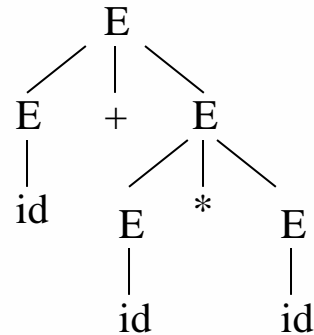




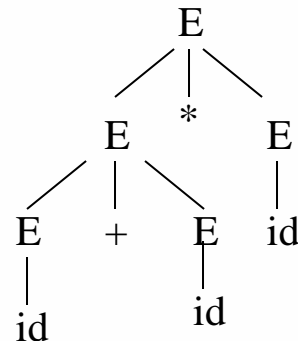
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an **ambiguous** grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E * E$
 $\Rightarrow id+id * E \Rightarrow id+id * id$



$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E$
 $\Rightarrow id + id * E \Rightarrow id + id * id$



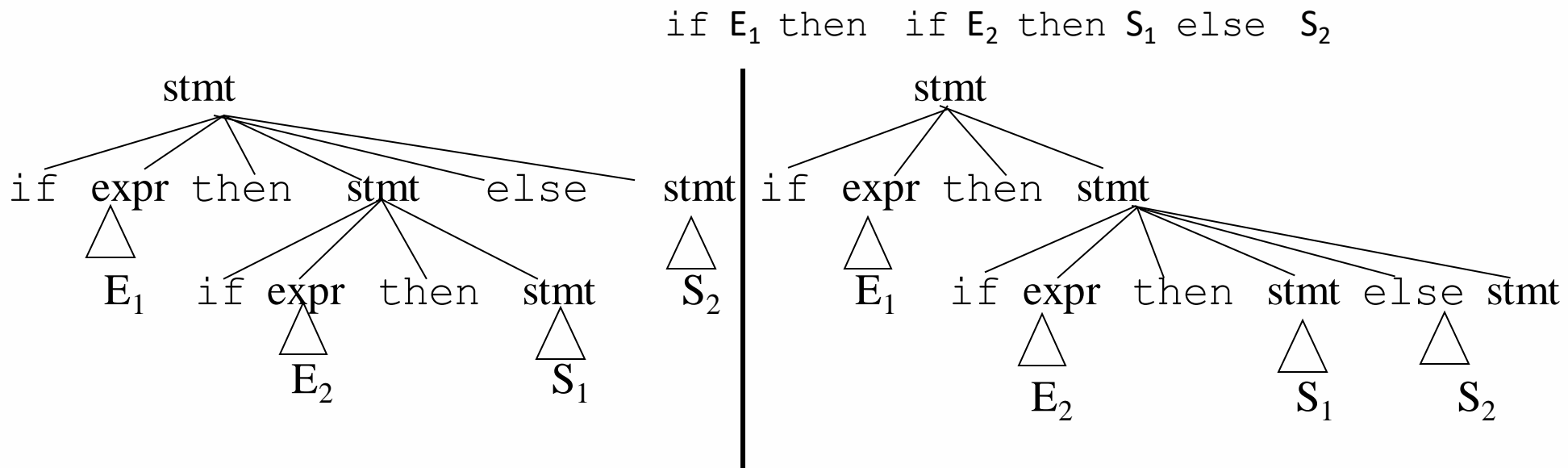
Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar
 - unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.



Ambiguity (cont.)

stmt \rightarrow if expr then stmt |
if expr then stmt else stmt | otherstmts



Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.
- The unambiguous grammar will be:

`stmt` \rightarrow `matchedstmt` | `unmatchedstmt`

`matchedstmt` \rightarrow `if expr then matchedstmt else matchedstmt` | `otherstmts`

`unmatchedstmt` \rightarrow `if expr then stmt` |
 `if expr then matchedstmt else unmatchedstmt`

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

$$E \rightarrow E+E \mid E^*E \mid E^E \mid id \mid (E)$$

disambiguate the grammar

precedence:

\wedge (right to left)

$*$ (left to right)

$+$ (left to right)

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T^*F \mid F$$
$$F \rightarrow G^F \mid G$$
$$G \rightarrow id \mid (E)$$


Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

$$A \Rightarrow A\alpha \quad \text{for some string } \alpha$$

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.



Immediate Left-Recursion

$A \rightarrow A \alpha \mid \beta$ where β does not start with A



eliminate immediate left recursion

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$ an equivalent grammar

In general,

$A \rightarrow A \alpha_1 \mid \dots \mid A \alpha_m \mid \beta_1 \mid \dots \mid \beta_n$ where $\beta_1 \dots \beta_n$ do not start with A



eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

an equivalent grammar



Immediate Left-Recursion -- Example

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow \text{id} \mid (E)$$


eliminate immediate left recursion

$$E \rightarrow T E'$$
$$E' \rightarrow +T E' \mid \varepsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow *F T' \mid \varepsilon$$
$$F \rightarrow \text{id} \mid (E)$$


Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sc \mid d$$

left-recursive,

This grammar is not immediately

but it is still left-recursive.

$$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$$

$$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$$

or

causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar



Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \dots A_n$
- **for** i **from** 1 **to** n **do** {
 - **for** j **from** 1 **to** $i-1$ **do** {
 - replace each production
$$A_i \rightarrow A_j \gamma$$
by
$$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$$
where $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$
- eliminate immediate left-recursions among A_i productions



Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
 $A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A
$$\begin{aligned} A &\rightarrow bdA' \mid fA' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$
 $A \rightarrow bdA' \mid fA'$
 $A' \rightarrow cA' \mid adA' \mid \epsilon$



Eliminate Left-Recursion – Example2

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \epsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$
So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \epsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \epsilon$

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \epsilon$



Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

grammar[?] a new equivalent grammar suitable for predictive parsing

```
stmt → if expr then stmt else stmt |  
      if expr then stmt
```

- when we see `if`, we cannot now which production rule to choose to re-write *stmt* in the derivation.



Left-Factoring (cont.)

- In general,

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols
of β_1 and β_2 (if they have one) are different.

- when processing α we cannot know whether expand

A to $\alpha\beta_1$ or

A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$



Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_n$$



Left-Factoring – Example1

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cde}B \mid \underline{cdf}B$$
$$A' \rightarrow bB \mid B$$

$$A \rightarrow aA' \mid cdA''$$
$$A' \rightarrow bB \mid B$$
$$A'' \rightarrow g \mid eB \mid fB$$


Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$



$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$



$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$



Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.
- $L1 = \{ \omega c \omega \mid \omega \text{ is in } (a|b)^* \}$ is not context-free
 $\boxed{?}$ declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).
- $L2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ is not context-free
 $\boxed{?}$ declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.



Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
 - Recursive-Descent Parsing
 - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
 - It is a general parsing technique, but not widely used.
 - Not efficient
 - Predictive Parsing
 - no backtracking
 - efficient
 - needs a special form of grammars (LL(1) grammars).
 - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
 - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

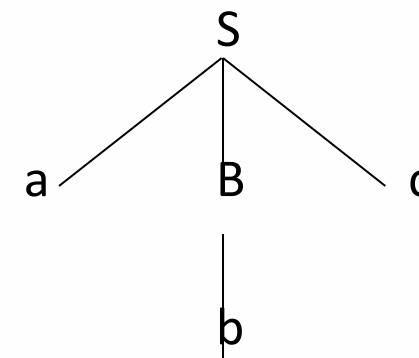
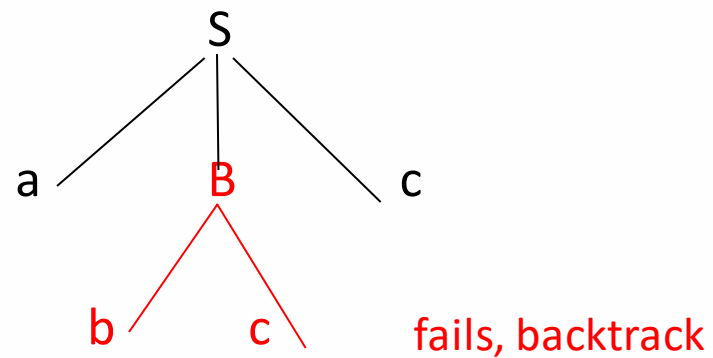
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc





Predictive Parser

a grammar

?

eliminate

left recursion factor

?

left

a grammar suitable for predictive
parsing (a LL(1) grammar)
no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$

input: ... a

current token



Predictive Parser (example)

```
stmt →      if .....      |  
          while .....      |  
          begin .....      |  
          for .....
```

- When we are trying to write the non-terminal *stmt*, if the current token is *if* we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).



Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {  
    - match the current token with a, and move to the next token;  
    - call 'B';  
    - match the current token with b, and move to the next token;  
}
```



Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {  
    case of the current token {  
        'a': - match the current token with a, and move to the next token;  
              - call 'B';  
              - match the current token with b, and move to the next token;  
        'b': - match the current token with b, and move to the next token;  
              - call 'A';  
        - call 'B';  
    }  
}
```



Recursive Predictive Parsing (cont.)

- When to apply ϵ -productions.

$$A \rightarrow aA \mid bB \mid \epsilon$$

- If all other productions fail, we should apply an ϵ -production. For example, if the current token is not a or b, we may apply the ϵ -production.
- Most correct choice: We should apply an ϵ -production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).





Recursive Predictive Parsing (Example)

• Content inside

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \epsilon$

$C \rightarrow f$

```
proc A {
  case of the current token {
    a:      - match the current token with a,
            - and move to the next token;
            - call B;
            - match the current token with e,
            - and move to the next token;
    c:      - match the current token with c,
            - and move to the next token;
            - call B;
            - match the current token with d,
            - and move to the next token;
    f:      - call C
  }
}
```

first set of C

```
proc C {      match the current token with f,
              and move to the next token; }
```

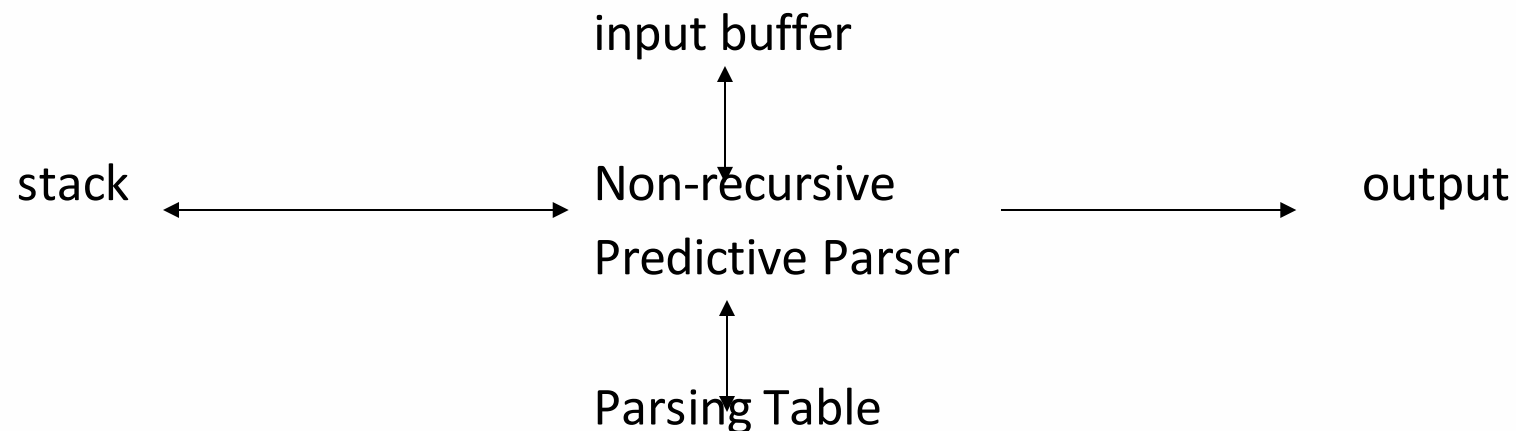
```
proc B {
  case of the current token {
    b:      - match the current token with b,
            - and move to the next token;
            - call B
    e,d: do nothing
  }
}
```

follow set of B



Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



× ○ DIGITAL LEARNING CONTENT



Parul[®] University



www.paruluniversity.ac.i

n

