UNIT 2: SYNTAX ANALYSIS

4.1 INTRODUCTION:

➢ **Parser or Syntax Analysis:**

- Parser or syntax analysis basically checks for the syntax of the language.
- A parser or syntax analysis is a process which takes the inputs (as tokens) and produce either parse tree (Syntactic structure) or generates error report.

4.1.1 Role of The Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer (as shown in Fig. 4.1,) and verifies that the string of token names can be generated by the grammar for the source language.

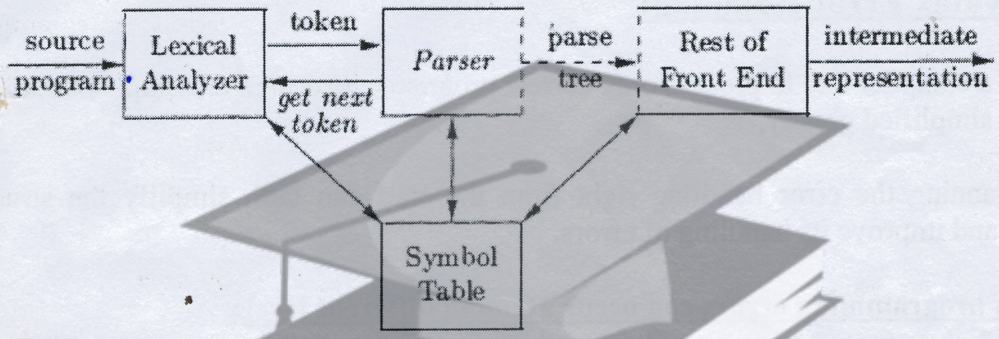


Figure 4.1: Position of parser in compiler model

In the process of compilation the parser and lexical analyzer works together. That means, when parser requires string of tokens t invokes lexical analyzer. In turn, the lexical analyzer supplies tokens to parser

The parser call the lexical analyzer (by the getNextToken command), causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.

There are three general types of parsers for grammars:

1. **Universal Parser:**

   Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar .These general methods are, however, too inefficient to use in production compilers.

2. **Top-Down Parser:**

Top-down methods build parse trees from the top (root) to the bottom (leaves).
Ex: LL Parser

3. **Bottom-Up Parser:**

Bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.
Ex: LR Parser

4 .1.3 Syntax Error Handling:

If a compiler had to process only correct programs, its design and implementation would be simplified greatly.

Planning the error handling right from the start can both simplify the structure of a compiler and improve its handling of errors.

Common programming errors can occur at many different levels.

• **Lexical errors-**

It include misspellings of identifiers, keywords, or operators .
e.g., Use of an identifier ***arunkmar*** instead of ***arunkumar***
Missing quotes around text intended as a string.

• **Syntactic errors** - include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, the appearance of a case statement without an enclosing switch is a syntactic error.

• **Semantic errors:** include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.

• **Logical errors :** It can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

The gaols of error handler in a parser:

The error handler in a parser has goals that are simple to state but challenging to realize:

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly enough to detect subsequent errors.
3. Add minimal overhead to the processing of correct programs.

4.1.4 Error-Recovery Strategies:

The following activities are performed whenever errors are detected by the parser:

- Detected the syntax errors accurately and produce appropriate error message so that the programmer can correct the program.
- It has to recover from the errors quickly and detect subsequent errors in the program.
- Error handler should take all the actions very fast and should not slowdown the compilation.

Panic Mode Recovery:

- It is the simplest and most popular error recovery method.
- When error is detected, the parser discarded symbols one at a time until next synchronizing token (valid token) is found.
- The synchronizing (valid) tokens are usually delimiters, such as semicolon or}, whose role in the source program is clear and unambiguous.

For example consider the erroneous expression:

(x+yz)+k**

- The parser scans the input from left to right and finds no errors after reading (, x, +, y, and * .
- After reading the second *, it knows that no expression has consecutive * operators and it display an error ***"Extra * in the input expression".***
- In panic mode recovery, it skips all input symbols till the next valid token found (i.e 'z'). Here z is the synchronizing token.

Phrase-Level Recovery:

- On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.

- A typical local correction is
  - To replace a comma by a semicolon,
  - Delete an extraneous semicolon, or insert a missing semicolon.

Disadvantages of this phase are:

- Very difficult to implement.
- Slows down the parsing of correct programs.
- Proper care must be taken while choosing replacement as they may lead to infinite loops.

Error production:

- The error productions specify commonly known mistakes in the grammar.

- When we implement the parser, when an error production is used, it displays the appropriate error message.
- For example, consider the expression 10x. Mathematically it means multiply 10 with x. But in programming language we should write 10 * x. such errors can be identified very easily by error production.

Global Correction:

- The global correction method replaces an incorrect input with correct input using ***least cost correction method.***
- These algorithm take an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.
- These methods are costly to implement in terms of time and space and hence are only of theoretical interest.

4 . 2 Context-Free Grammars:

4.2. 1 The Formal Definition of a Context-Free Grammar (C.F.G):

- A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.
- G= (V, T, P, S) Where G is grammar, T-terminals, V-non-terminals, S-start symbol, and P-productions.
- Consider the grammar

E→ E+T|T

T→ T*F|F

F→ id

1. **Terminals:**
   - Terminals are the basic symbols from which strings are formed.
   - The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal.
   - In above grammar terminals are **+ , * and 'id'** .
   -
2. **Nonterminals:**
   - Nonterminals are syntactic variables that denote sets of strings.
   - **E, T and F** are nonterminals in above grammar.
   - The sets of strings denoted by nonterminals help define the language generated by the grammar.
3. **start symbol:**
   - The first no-terminal of the grammar is the start symbol.

- **E** is the start symbol.

The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Each production consists of:
(a) A nonterminal called the head or left side of the production;
(b) The symbol → . Sometimes : : = has been used in place of the arrow.
(c) A body or right side consisting of zero or more terminals and nonterminals.

.

S → E+T|T

(head) → (body)

4.2.2 Notational Convent ions:

1. These symbols are terminals:
   (a) Lower case letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as +, *, and so on.
   (c) Punctuation symbols such as parentheses, comma, and so on.
   (d) The digits 0, 1, . . . , 9.
   (e) **Boldface** strings such as **id** or **if,** each of which represents a single terminal symbol.

2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr or stmt.*
   (d) When discussing programming constructs, uppercase letters may be used t o represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F,respectively.

P.T.O

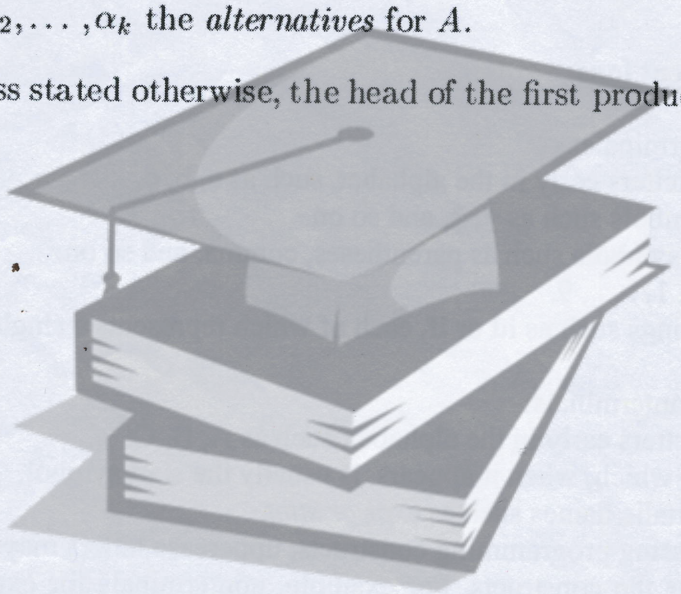
3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \to \alpha$, where A is the head and α the body.

6. A set of productions $A \to \alpha_1$, $A \to \alpha_2, \ldots, A \to \alpha_k$ with a common head A (call them *A-productions*), may be written $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.



4.2.3 Derivations:

- The process of starting with non-terminal (start symbol) of a grammar and successively replacing it by the body of one of its productions is called a derivation.

Two Types of Derivations:

1. Leftmost Derivation:
2. Rightmost Derivation.

- If $A \to \alpha B \gamma$ and $B \to \beta$ are the productions, the string $\alpha\beta\gamma$ can be obtained from A-production.

$A \Rightarrow \alpha B \gamma$ (Apply the production $A \to \alpha B \gamma$)

$\Rightarrow \alpha \beta \gamma$ (Replace B by β using the production $B \to \beta$)

The above derivation can also be written as,

$A \overset{*}{\Rightarrow} \alpha\beta\gamma$

- The symbol "$\Rightarrow$" means "derives in one step".

Ex:

- $A \Rightarrow \alpha\beta$ Represents that $\alpha\beta$ is derived from A in single step.
  (or) A derives $\alpha\beta$ in one step.

- Sequence of derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \alpha_n$ Rewrites as α_1 to α_n. (α_1 derives α_n)

- $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ ($\overset{*}{\Rightarrow}$ derives in zero or more steps)

- $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$ ($\overset{+}{\Rightarrow}$ derives in one or more steps)
  α_1 derives α_n in one or more steps.

Sentence and Sentential form of Grammar (G):

If G is the grammar and S is the start symbol then

$S \overset{*}{\Rightarrow} \alpha$ (S derives α in zero or more steps)

→ α is Sentential form of G (if α contains both terminals and non-terminals)

→ α is Sentence of G (if α contains only terminals)

Ex:

$S \to E+T$
$E \to E+E$
$T \to (E)$
$E \to id \mid a$ } Sentential – Sentence

$S \to AaB$
$A \to aaB$ } Sentential
$B \to a \mid b$ } Sentence.

1. Leftmost derivation: (LMD)

- The leftmost nonterminal in each Sentential is always chosen.
- The process of obtaining a string of terminals from a Seavence of Replacement such that only leftmost non-terminal is Replaced at each and every step is called Leftmost derivation.

we write $\alpha \underset{lm}{\Rightarrow} \beta$

2. Rightmost derivation: (RMD)

- The rightmost nonterminal in each Sentential is always chosen.
- The process of obtaining a string of terminals from a Seavence of Replacement such that only rightmost non-terminal is Replaced at each and every step is called Rightmost derivation.

We write $\alpha \underset{rm}{\Rightarrow} \beta$.

Consider the following Grammar:

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

The leftmost derivation for string id+id*id can be obtained as shown below,

$$E \underset{lm}{\Rightarrow} E+E$$
$$\underset{lmd}{\Rightarrow} id+E$$
$$\underset{lmd}{\Rightarrow} id+E*E$$
$$\underset{lmd}{\Rightarrow} id+id*E$$
$$\underset{lmd}{\Rightarrow} id+id*id$$

The Rightmost derivation for string id+id*id can be obtained as shown below.

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

Left Sentential and Right Sentential

- If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then we say that α is a left-sentential form of the grammar
- If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, then we say that α is a right-sentential form of the grammar.

4.2.4 Parse Trees and Derivations:

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non-terminals.
- Interior node of Parse tree represents the application of a production
- Nodes → Represents the Non-terminals
- Leaves → Represents the terminals.

① consider the context Free Grammar and the string aa+a*.

$S \rightarrow SS+ \mid SS* \mid a$

(a) Give a LMD for string

(b) Give a RMD for string

(c) Give a parse tree for string

Soln:

LMD

$S \underset{lm}{\Rightarrow} \underline{S}S*$

$\Rightarrow \underline{SS+}S*$

$\Rightarrow a\underline{S+}S*$

$\Rightarrow aa+\underline{S}*$

$\Rightarrow aa+a*$

RMD

$S \underset{rm}{\Rightarrow} S\underline{S}*$

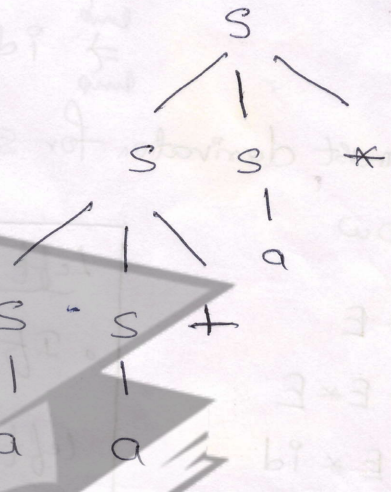
$\Rightarrow Sa*$

$\Rightarrow S\underline{S+}a*$

$\Rightarrow \underline{S}a+a*$

$\Rightarrow aa+a*$

Parse tree



② consider the C.F.G and string id+id*id.

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

(a) Give LMD and Parse tree

(b) Give RMD and Parse tree.

Soln: Read class notes.

4.2.5 Ambiguity:

* A grammar that produce more than one parse tree, or equivalent two or more LMD or two or more RMD for some string is said to be ambiguous grammar.

① Consider Grammar and prove that it is ambiguous.

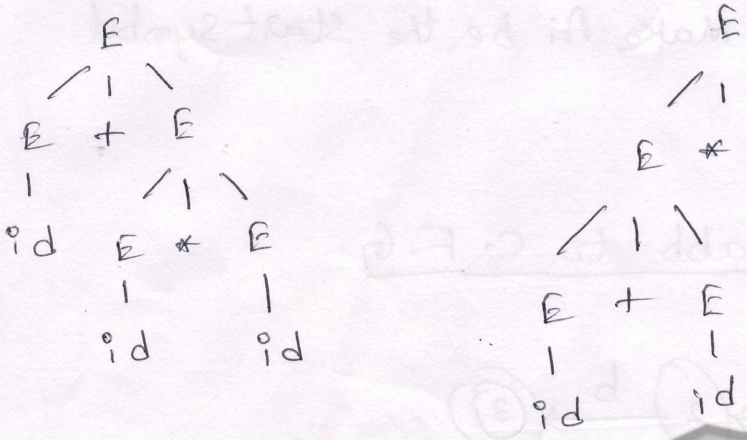
$E \rightarrow E*E$

$E \rightarrow E+E$

$E \rightarrow id$

Soln: Consider the string (Sentence) id + id * id.

using the Grammar we can generate two different parse trees as shown below, Since the two parse trees are different for the same string (sentence), the grammar is ambiguous.



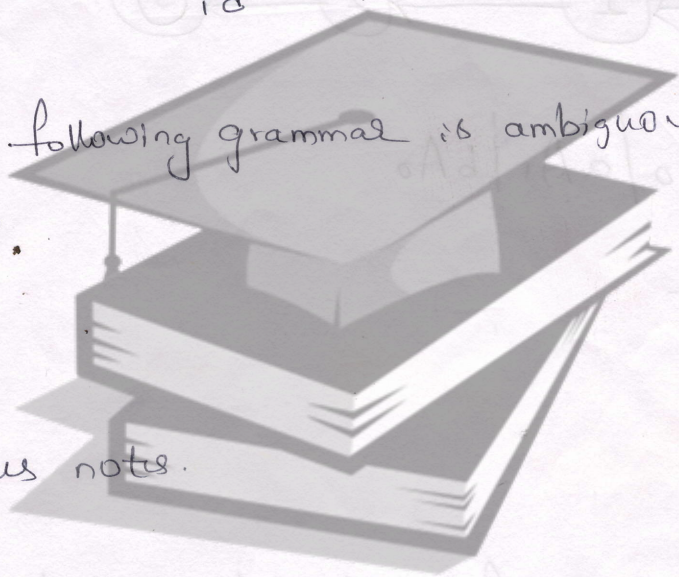
(2) prove that the following grammar is ambiguous?

$S \rightarrow AB \mid aaB$

$A \rightarrow a \mid Aa$

$B \rightarrow b$

Soln: Read class notes.



4.2.7 C.F.G v/s R.E

- Grammars are more powerful notation than R.E, ~~but not Vice Versa~~
- Every construct that can be described by a R.E can be described by a Grammar, but not Vice-Versa.

Procedure to convert R.E to C.F.G:

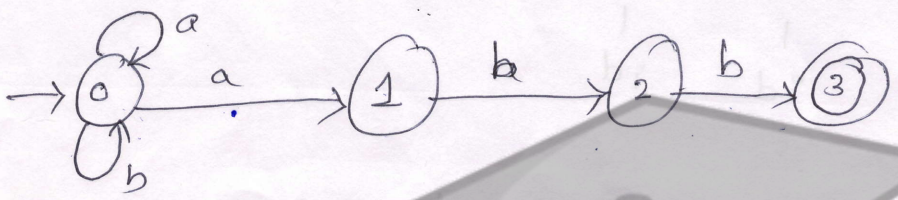
1. For each state i of the NFA, create a Non-terminal A_i
2. If i (state) is an accepting state add $A_i \rightarrow \varepsilon$

3. If state i has transition to state j on input 'a', add the production $A_i \longrightarrow a A_j$

If state i goes to state j on input ε, add the production $A_i \longrightarrow A_j$

4. If i is the start state, Make A_i be the start symbol of the grammar.

convert R.E (a|b)* abb to C.F.G



$A_0 \longrightarrow aA_0 | aA_1 | bA_0$

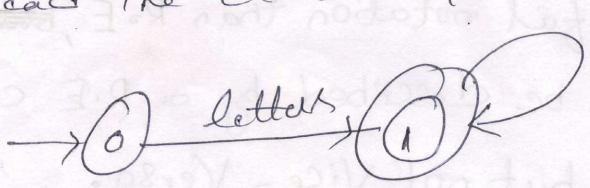
$A_1 \longrightarrow bA_2$

$A_2 \longrightarrow bA_3$

$A_3 \longrightarrow \varepsilon$

(2) write the R.E for identifiers and Convert it into C.F.G

Soln: Read the class notes



$A_0 \longrightarrow$ letters A_1

$A_1 \longrightarrow$ letters A_1 | digits A_1 | ε.

4.3 Writing a Grammar:

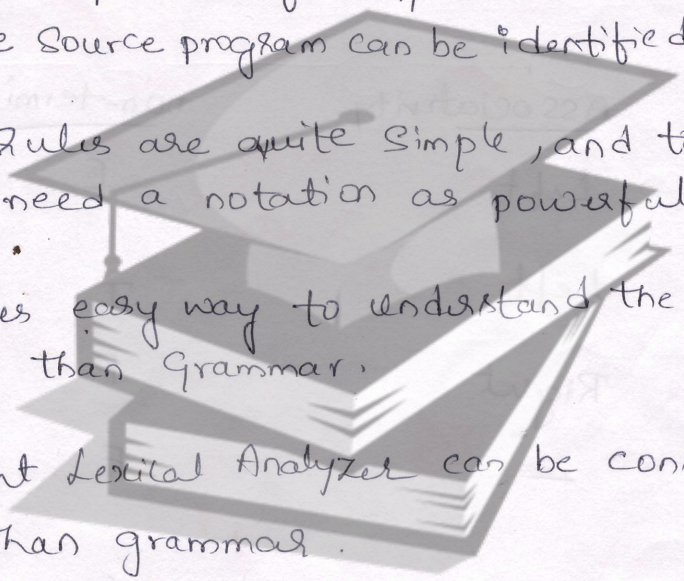
4.3.1 Lexical Versus Syntatic Analysis:

- Everything that can be described (defined) by a R.E can also be described by a grammar, then

"Why use R.E to define the lexical syntax of a language?"

There are Several Reasons:

1. Separating the lexical and non-lexical parts (in compilation process) provides the convenient (easy) way to implement the front-end of the compiler.
2. Accelerates the process of compilation.
   Errors in the source program can be identified easily.
3. The lexical Rules are quite simple, and to describe them we do not need a notation as powerful as grammars.
4. R.E provides easy way to understand the notation for tokens than Grammar.
5. More efficient Lexical Analyzer can be constructed automatically from R.E than grammar.



4.3.2 Eliminating Ambiguity:

- Some grammar that are ambiguous can be converted into unambiguous. This can be done using two methods.
  1. Using the Precedence and Associativity of operators.
  2. Dis-ambiguity Rule.

1. Precedence and Associativity of operators:

① Convert the following ambiguous grammar into Unambiguous grammar.

$E \rightarrow E * E \mid E - E$

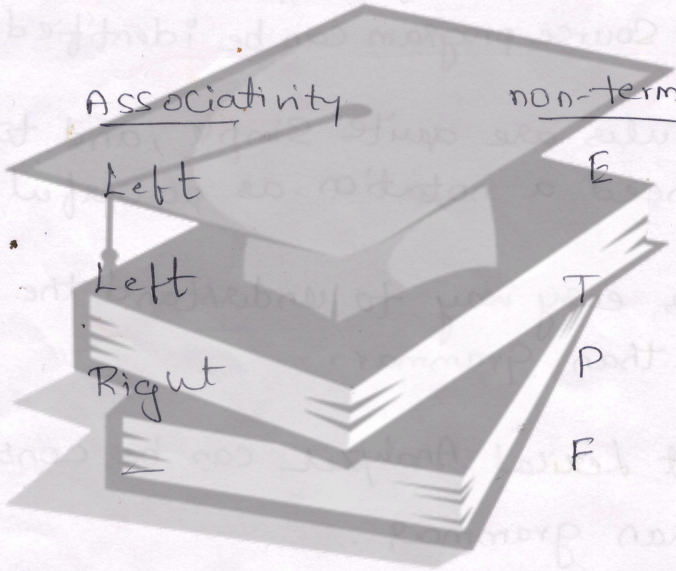
$E \rightarrow E \hat{} E \mid E / E$

$E \rightarrow E + E$

$E \rightarrow (E) \mid id$

Soln: Arrange the operators in increasing order of the precedence along with associativity as shown below.

operators	Associativity	non-terminal
+, −	Left	E
*, /	Left	T
^	Right	P
Basic unit (E), id	—	F



Unambiguous grammar:

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

$T \rightarrow T * P \mid T / P \mid P$

$P \rightarrow F \hat{} P \mid F$

$F \rightarrow (E) \mid id$

only $E \rightarrow E+T$ is possible so unambiguous

② Convert the following ambiguous grammar into Unambiguous grammar

$E \rightarrow E+E \mid E-E$

$E \rightarrow E*E \mid E/E$

$E \rightarrow E \wedge E$

$E \rightarrow (E) \mid id$

by considering * and − operators lowest priority and they are left associative, / and + operators have the highest priority and are right associative and ∧ operator has precedence in between and it is left associative.

Soln:

Precedence	operators	Associativity	non-terminals
(lowest)	*, −	Left	E
(in between)	∧	Left	P
(highest)	/, +	Right	T
Basic unit	(E), id	−	F

$E \rightarrow E*P \mid E-P \mid P$

$P \rightarrow P \wedge T \mid T$

$T \rightarrow F+T \mid F/T \mid F$

$F \rightarrow (E) \mid id.$

2. DIS-Ambiguity Rule:

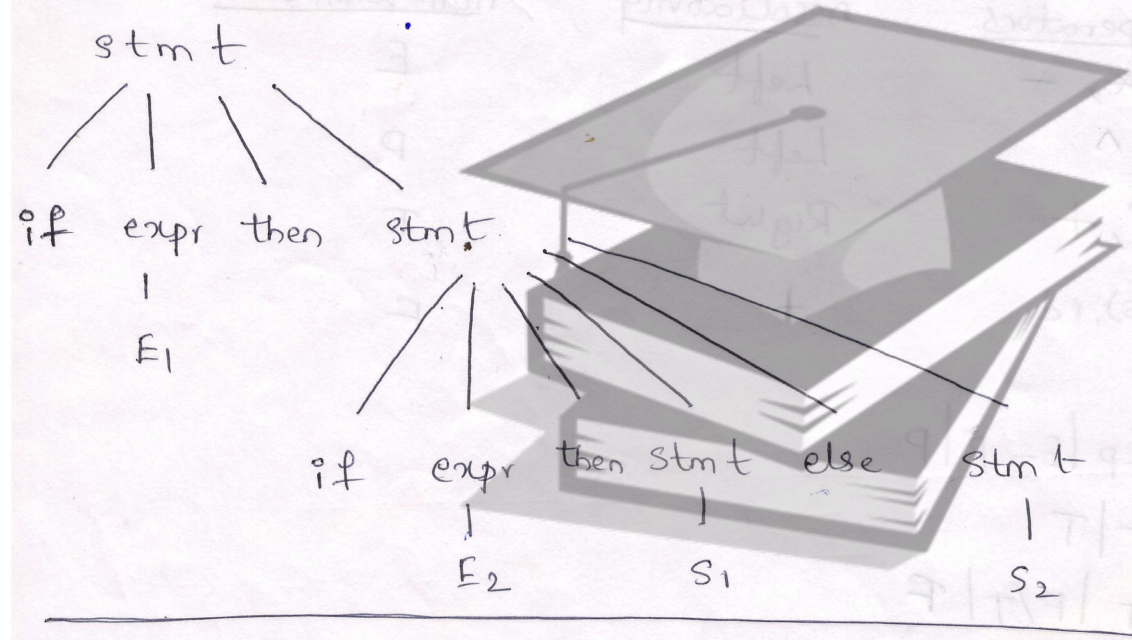
consider Dangling-else Grammar. and prove that it is ambiguous.

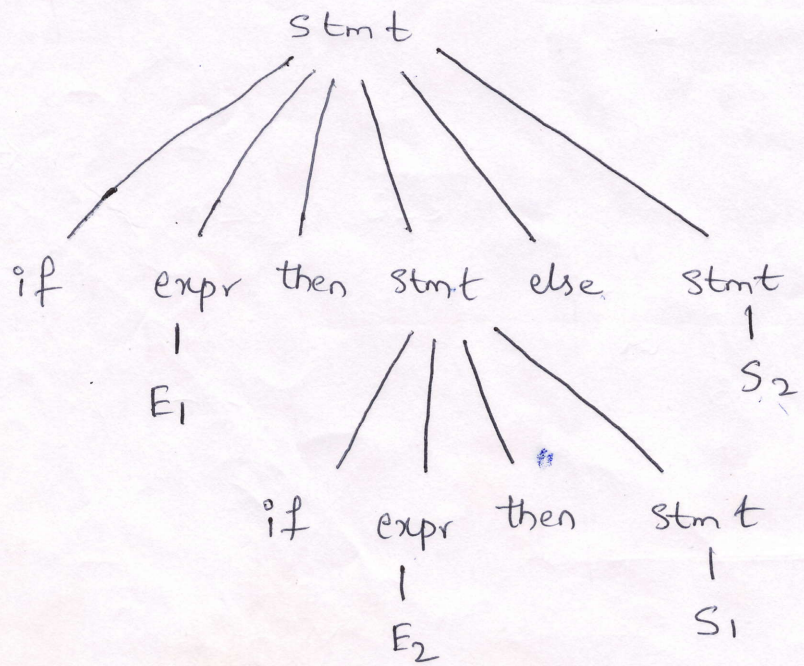
stmt → if expr then stmt
| if expr then stmt else stmt
| other.

*Soln:

consider the string of terminals as shown below.

if E_1 then if E_2 then S_1 else S_2.





The above dangling-else grammar is ambiguous, as its generates two different parse trees for same string (Sentence).

The idea to convert ambiguous to Un-ambiguous is

- Statement appearing between a then and an else must be "matched".
- it means, the interior statement must not end with an unmatched or open then.

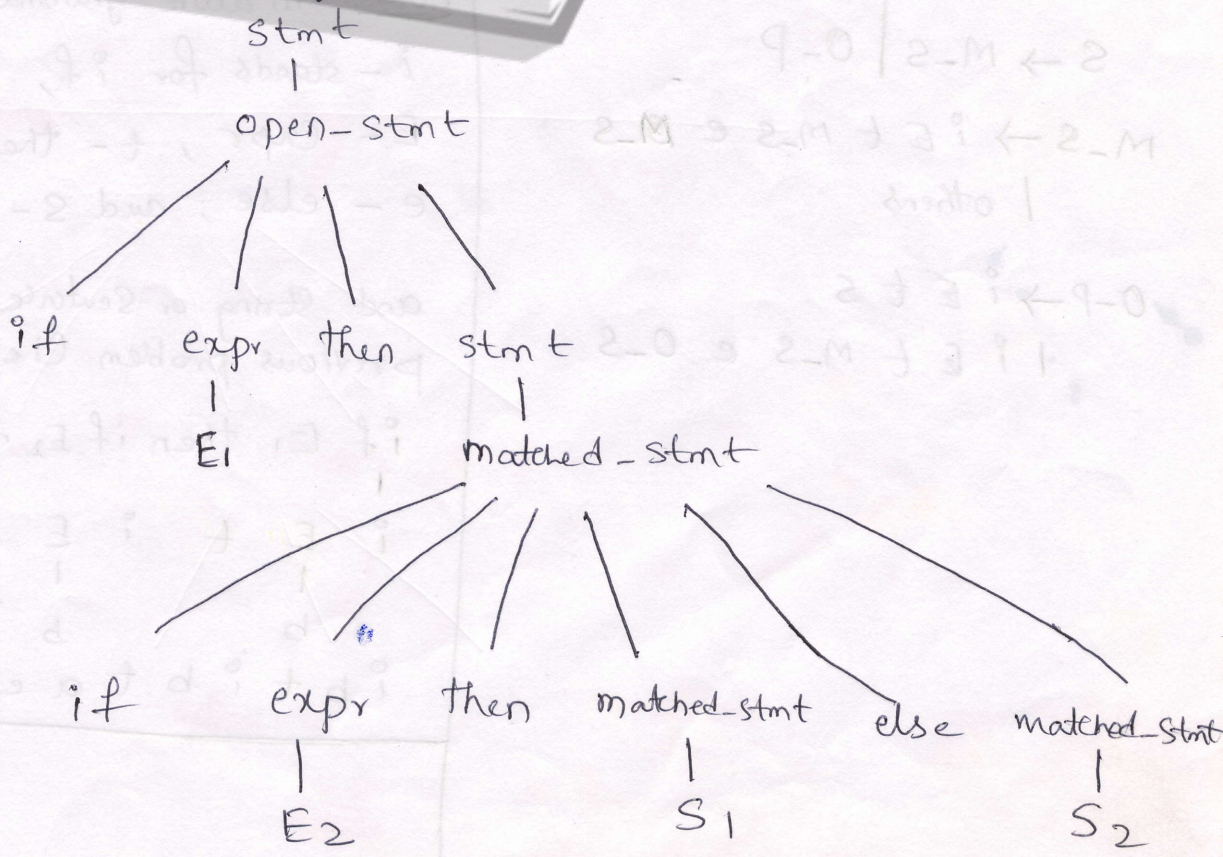
Unambiguous Grammar.

stmt → matched_stmt | open_stmt

matched_stmt → if expr then matched_stmt else matched
| other

open_stmt → if expr then stmt
| if expr then matched_stmt else open_stmt

Parse tree for string if E_1 then if E_2 then S_1 else S_2

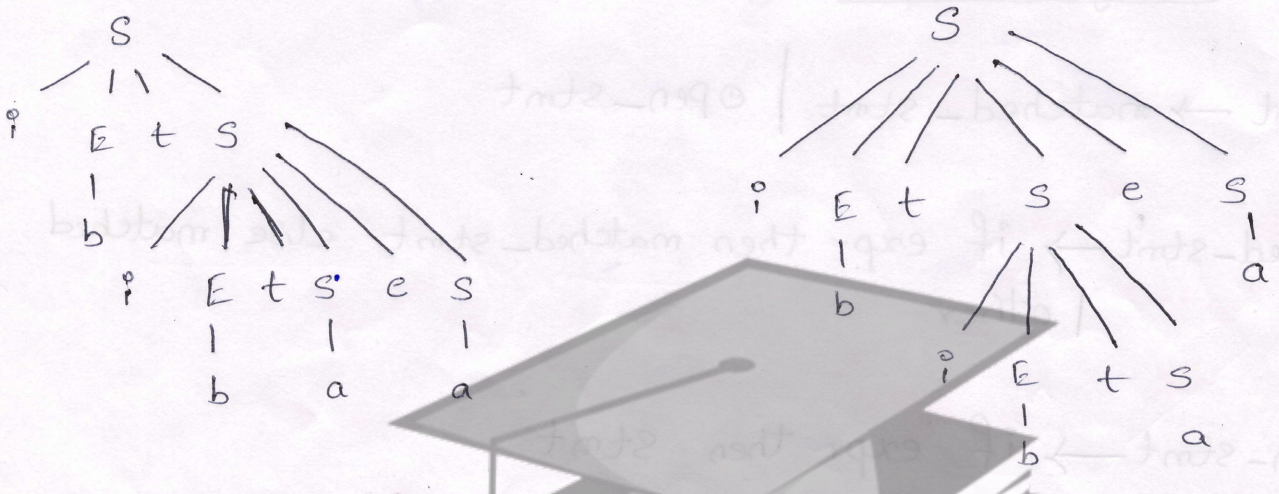


(2) consider "Dangling - else grammar" shown below and prove it is ambiguous grammar. and convert it into unambiguous.

$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$

$E \rightarrow b$

Soln: consider the string (sentence) i b t i b t a e a



For the same string we have generated the two different parse tree, so the above Grammar is ambiguous.

Unambiguous Grammar:

$S \rightarrow M\_S \mid O\_P$

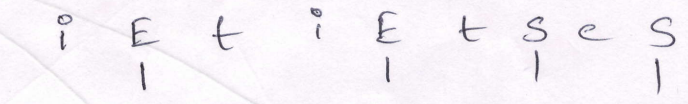
$M\_S \rightarrow i\ E\ t\ M\_S\ e\ M\_S$
$\mid$ others

$O\_P \rightarrow i\ E\ t\ S$
$\mid i\ E\ t\ M\_S\ e\ O\_S$

note: in above grammar.
i - stands for if,
E - expr, t - then
e - else, and S - stmt.

and string or sentence is same as previous problem (i.e)

if E_1 then if E_2 then S_1 else S_2



i b t i b t a e a.

4.3.3 Elimination of Left Recursion:

Left Recursion:

The First symbol on the right hand side of the production is same as the Symbol on the left hand side of the production.

General form:

$A \rightarrow A\alpha \mid \beta$

Exs:

$E \rightarrow E+T \mid T$

$A \rightarrow Aa \mid b$.

Right Recursion:

The ~~Right most~~ ~~First~~ Symbol on the right side of the product is same as the symbol on the left hand side of the production.

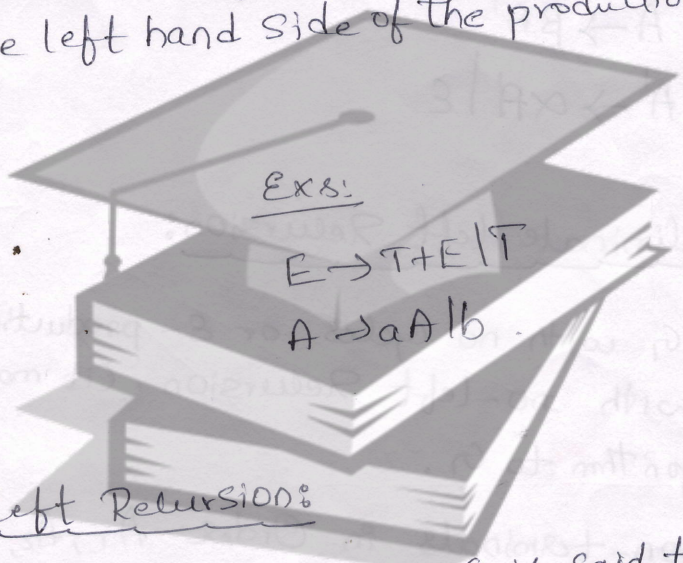
General form

$A \rightarrow \alpha A \mid \beta$

Exs:

$E \rightarrow T+E \mid T$

$A \rightarrow aA \mid b$.



Two types of Left Recursion:

1. **Immediate left Recursion:** A Grammar G is said to have immediate left Recursion if it has a production of the form

$A \rightarrow A\alpha$.

In this grammar first two productions are immediate left Recursion.

Exs:

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

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$.

2. **Indirect left Recursion:**

A left Recursion involving derivation of two or more steps so that the first symbol on the right hand side of the partial derivation is same as the symbol from which the derivation started is called Indirect left Recursion.

Ex: $E \rightarrow T$

$T \rightarrow F$

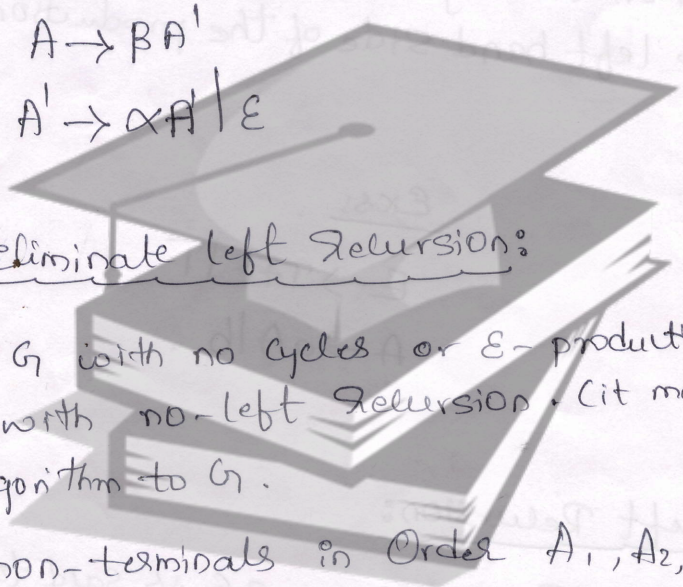
$F \rightarrow E+T \mid id$

Problems in left-Recursion Grammar:

- A grammar is left Recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods cannot handle left-Recursive grammar (because there may be chance of entering into infinite loop), so a transformation is needed to eliminate left Recursion.

General form of left Recursive Grammar is $A \to A\alpha \mid \beta$, it can be converted to non-left Recursive Grammar using procedure as shown below.

- $A \to \beta A'$
  $A' \to \alpha A' \mid \varepsilon$



Algorithm to Eliminate left Recursion:

Input: Grammar G with no cycles or ε-production
output: Grammar with no-left Recursion. (it may contain ε-production)
Method: Apply Algorithm to G.

1. Arrange non-terminals in Order $A_1, A_2, \ldots A_n$.
2. for (each i from 1 to n) {
3. ~~Replace~~ for (each j from 1 to i-1) {
4. Replace $A_i \to A_j\gamma$ by productions
5. $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_n$
5. }
6. eliminate the immediate Left Recursion among A_i productions (A_i-productions)
7. }

④ Eliminate the left Recursion from the following grammars.

① $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Soln:
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
~~$T' \to *FT' \mid \varepsilon$~~ $T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

② $S \to Sab \mid cd$

soln:
$S \to cdS'$
$S' \to abs' \mid \varepsilon$

③ $S \to Sab \mid \varepsilon$

soln:
$S \to S'$
$S' \to abs' \mid \varepsilon$

④ $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon$

soln:
$S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \varepsilon$

non-left Grammar
$S \to Aa \mid b$
$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \varepsilon$.

4.3.4 Left Factoring:

- A grammar in which two or more productions from a non-terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called left factored grammar.

General form of non-left factoring Grammar

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

⇛

Equivalent left factoring Grammar

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

Ex:

① $S \to abA \mid abB$
$A \to aa$
$B \to ab$

soln:
$S \to abS'$
$S' \to A \mid B$
$A \to aa$
$B \to ab$

(2) consider the following non-leftoring Grammar and Convert it into equivalent left factor Grammar.

$S \rightarrow iEts \mid iEtses \mid a$

$E \rightarrow b$

Soln:

$S \rightarrow iEtsS' \mid a$

$S' \rightarrow \varepsilon \mid es$

$E \rightarrow b$

Algorithm for left-Factoring:

Input: Grammar G (non-Left-factor Grammar)

output: An equivalent left-factored Grammar.

Method:

1. For each non-terminal A, Find the largest prefix α common to two or more alternatives.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots \mid \alpha\beta_n \mid \gamma$

$A \rightarrow \alpha A' \mid \gamma$

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

Repeatedly apply this transformation until no two alternatives for a non-terminal have common prefix.

4.4 TOP-down Parser:

- In Top-down Parsing the parse tree is generated from top to bottom (i.e from Root to leaves).
- In this approach, we start with start-Symbol, we Keep on desiving the parse tree (by Replacing the non-terminal using production) until unless we get the given input string.

4.4.1 Recursive-Descent Parsing:

- A Recursive-descent parsing program consists of a set of procedure, one for each non-terminal.
- Executions begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire string else sent error Report.

Algorithm for Recursive-Descent Parsing:

```
Void A()
{
    choose an A-production, A -> X1 X2 --- Xk;
    for (i = 1 to k){
        if (Xi is a non terminal)
            call procedure Xi();
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
        else
            error();
    }
}
```

Note: working of Recursive Descent parsing.

1. If Symbol is non-terminal then apply production or procedure
2. If Symbol is terminal compare it with input-pointer value if it's matched increment the input pointer.

Two types of Recursive Descent Parsing:

1. Recursive Descent parsing with backtracking
2. ——— " ——— without backtracking (Predictive Parser)

Recursive Descent Parsing with back tracking:

Backtracking: It is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occur then we try another alternative if any.

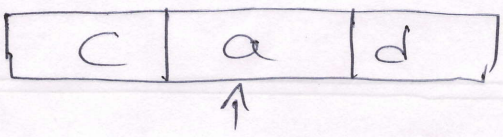
① consider the grammar given below and string $w = cad$ and trace it using Recursive Descent parsing.

$S \rightarrow cAd$

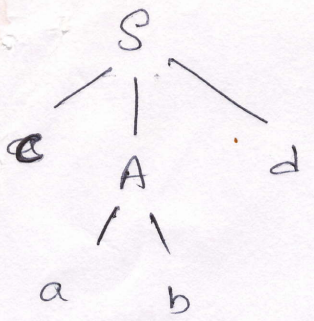
$A \rightarrow ab \mid a$

Soln:

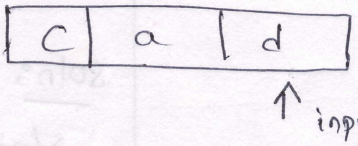
- We start we start symbol S, S has only one production, so we expand S as shown above, The input_pointer initial point to the starting symbol of input string.
- The leftmost leaf of parse tree (i.e c) matches the first symbol of input string (i.e w=cad), so we advance input_pointer to next symbol i.e a,



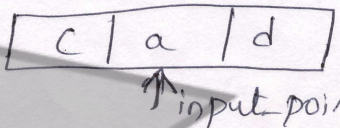
- Next leaf of tree is A, we expand A using first alternative $A \rightarrow ab$



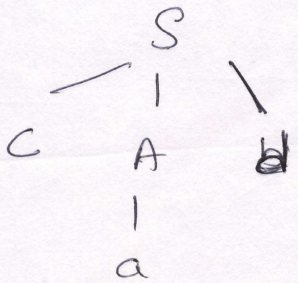
- Now second leaf of Parse tree is matched with the input pointer value, so we advance the input pointer to d.



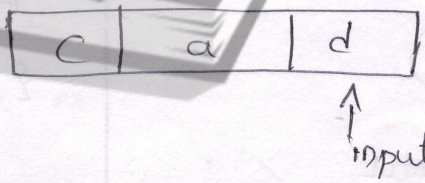
- Since the b is does not match d, we report failure and we go back to A (non-terminal) to see whether there is another alternative for A that has not been tried.
- When we go back to A, we must reset the input pointer to position 2. (ie at 'a').



- Second Alternative of A is A → a,



- Leaf 'a' matches with the input pointer value, so we advance the input pointer to d.



- Next leaf of Parse tree is d and it matches the third symbol of input string (i.e d). we halt and announce successful completion of Parsing.

(1) Write the Recursive Descent Parsing for the Grammar

$E \rightarrow E+T$

Soln:

```
Void E()
{
  E();
 if(input symbol==+')
  { advance the input pointer;
   T();
  }
 else
   error();
}
```

(2) $F \rightarrow (E) \mid id$

Soln:

```
Void F()
{
 if(input symbol=='(')
  {advance input pointer
   E();
  if(input symbol==')')
   return true
  advance input pointer;
  else
    error();
  }
 else if(input symbol=='id')
   advance input pointer;
                return true
 else
   error();
}
```

(3) $S \rightarrow cAd$

$A \rightarrow ab \mid a$

Soln:

```
Void S()
{
 if(input symbol=='c')
  { advance input pointer;
    A();
    if(input symbol=='d')
    advance input pointer;
                actual true
  }
 else
   error();
}

A()
{
 if(input symbol=='a')
  {
    advance input symbol;
   if(input symbol=='b')
    advance input symbol;
                return true.
  }
 else
   error();
}
```

UNIT-II (continue)

FIRST AND Follow

FIRST(α) : It is defined as a set of terminals that appears in the begging of derivation derived from α.

Algorithm:

1. If X is a terminal, then FIRST(X) = $\{x\}$
2. If $X \to \varepsilon$ is a production, then add ε to FIRST(X)
3. If X is a non-terminal and $X \to Y_1 Y_2 Y_3 \cdots Y_n$ and if $Y_1 Y_2 Y_3 \cdots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$ then
   FIRST(X) $\leftarrow$ non ε Symbols in FIRST(Y_i)

   If $X \to Y_1 Y_2 Y_3 \cdots Y_n$ and $Y_1 Y_2 \cdots Y_n \overset{*}{\Rightarrow} \varepsilon$ then FIRST(X) $\leftarrow \varepsilon$

Follow(α): For non-terminal α is defined as the set of terminals that will appear immediately to the right of α in some Sentential form.

Algorithm:

1. place \$ in Follow(S), where S is the start symbol
2. If $A \to \alpha B \beta$, then everything in FIRST(β) except ε is in Follow(B). (i.e Follow(B) = FIRST(β) except ε)
3. If $A \to \alpha\beta$ or $A \to \alpha B \beta$, where FIRST(β) contains ε, then everything in Follow(A) is in Follow(B). (i.e Follow(B) = Follow(A))

① Find the FIRST and FOLLOW for the grammar.

$E \to TE'$

$E' \to +TE' | \varepsilon$

$T \to FT'$

$T' \to *FT' | \varepsilon$

$F \to (E) | id$

Soution

Symbols	FIRST	Follow
E	(id	) $
E'	+ ε	) $
T	(id	+) $
T'	* ε	+) $
F	(id	+ *) $

② $S \to i C t S S' | a$

$S' \to \varepsilon | eS$

$C \to b$

Soution

Symbols	FIRST	Follow
S	a i	$ e
S'	ε ε	$ e
C	b	t

③

$S \to iS | iSeS | a$

Symbols	FIRST	Follow
S	i a	e $

④ $S \to CC$

$C \to cC | d$

Symbols	FIRST	Follow
S	c d	c d
C	c d	c d $

⑤ $S \to ABCd$

$A \to +B | \varepsilon$

$B \to *B | \varepsilon$

$C \to \%B | \varepsilon$

Symbols	FIRST	Follow
S	+ * % d	$
A	+ ε	* % d
B	* ε	* % d
C	% ε	d

LL(1) parser or Predictive parser:

- LL(1) parser is one of the type of Top-down parser.
- First L - stands for Left to right scanning
- Second L - stands for Left most derivation
- 1 - stands for using one symbol as look ahead.
- The grammar from which a predictive parser, that is, Recursive descent parser without backtracking is constructed is called LL(1) grammar.

* The following Grammars are not LL(1)

1. Ambiguous grammar is not LL(1).
2. Left Recursive grammar is not LL(1).
3. The grammar which is not left factored is also not LL(1).
4. The grammar that results in multiple entries in the parsing table is not LL(1).

How to check whether a given grammar is LL(1) or not without constructing the Predictive parser table:

- For every production of the form $A \to \alpha_1 | \alpha_2 | \dots | \alpha_n$

  $FIRST(\alpha_1) \cap FIRST(\alpha_2) \cap \dots \cap FIRST(\alpha_n) = \phi$

- For every non-terminal A such that FIRST(A) contain ε.

  $FIRST(A) \cap Follow(A) = \phi$

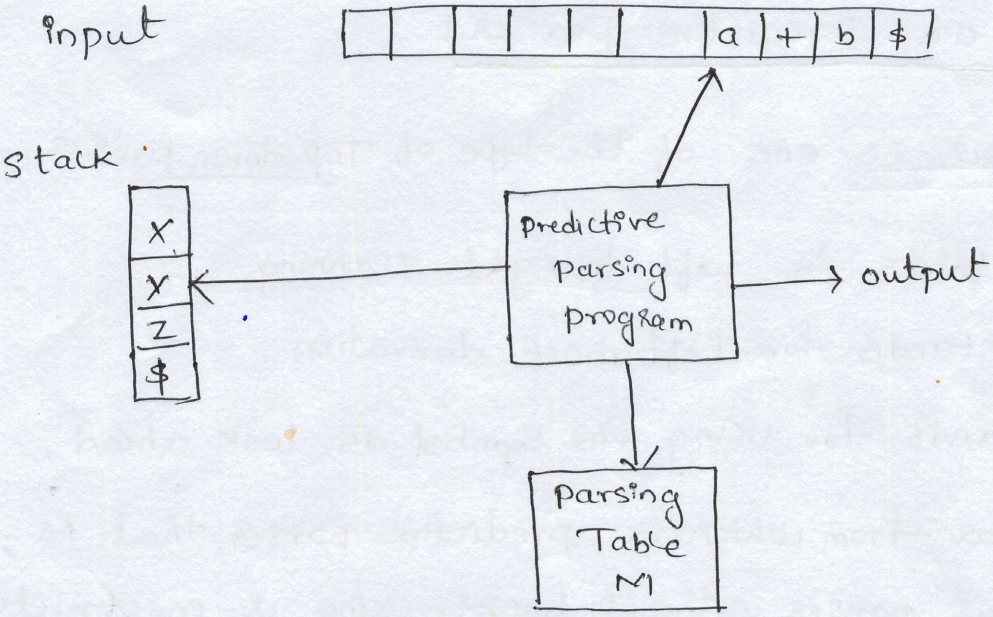


fig: Model of table-driven predictive parser

A Predictive parser Consists of

- Input Buffer: It contains the string to be parsed followed by $
- stack: It contains grammar symbols followed by a $
- The parsing Table: It is a 2D array of the form M[A,a] where A is a non-terminal and 'a' is terminal.

LL(1)

(1) obtain the predictive parsing table for the following grammar, and shows the moves made by predictive parser on the input id+id.

$E \to TE'$

$E' \to +TE' | \varepsilon$

$T \to FT'$

$T' \to *FT' | \varepsilon$

$F \to (E) | id$

solution: First find the FIRST And Follow.

Symbols	FIRST	Follow
E	(id	) $
E'	+ ε	) $
T	(id	+) $
T'	* ε	+) $
F	(id	+ *) $

Parsing table

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Moves made by predictive parser on the input id+id.

Matched	stack	input	Actions
	$E\$$	$id + id\$$	
	$TE'\$$	$id + id\$$	output $E \to TE'$
	$FT'E'\$$	$id + id\$$	output $T \to FT'$
	$idT'E'\$$	$id + id\$$	output $F \to id$
id	$T'E'\$$	$+id\$$	match id
id	$E'\$$	$+id\$$	output $T' \to \varepsilon$
id	$+TE'\$$	$+id\$$	output $E' \to +TE'$
id +	$TE'\$$	$id\$$	matched +
id +	$FT'E'\$$	$id\$$	output $T' \to FT'$
id +	$idT'E'\$$	$id\$$	output $F \to id$
id+id	$T'E'\$$	$\$$	matched id
id+id	$E'\$$	$\$$	$T' \to \varepsilon$
id+id	$\$$	$\$$	$E' \to \varepsilon$

Note: Read all class note problems.

(2) Consider the grammar

$E \rightarrow 5+T \mid 3-T$

$T \rightarrow V \mid V*V \mid V+V$

$V \rightarrow a \mid b$

(a) is the grammar suitable for predictive parsing?

(b) what is the use of left Factoring? Do necessary changes

(c) Compute First and Follow

(d) without constructing parse tree, check whether the grammar is LL(1) or not

(e) By constructing parse tree ——— " ———

Soln:

(a) The above grammar is not left factor Grammar. The second production contains the common prefix.

$T \rightarrow V \mid V*V \mid V+V$

(b) To make it suitable for LL(1) parser, we need to do left factoring.

General form:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

$T \rightarrow V \mid V*V \mid V+V$

$T \rightarrow VT'$

$T' \rightarrow \varepsilon \mid *V \mid +V$

Resultant Grammar.

$E \rightarrow 5+T \mid 3-T$

$T \rightarrow VT'$

$T' \rightarrow \varepsilon \mid *V \mid +V$

$V \rightarrow a \mid b$

(c) FIRST and Follow:

Symbols	FIRST	Follow
E	5 3	$
T	a b	$
T'	* + ε	$
V	a b	* + $

(d) The grammar to be LL(1) the following two conditions must be satisfied

1. Consider the production.

$E \to 5+T \mid 3-T$ FIRST(5+T) ∩ FIRST(3-T)

{5} ∩ {3} = ϕ.

$V \to a \mid b$ FIRST(a) ∩ FIRST(b)

{a} ∩ {b} = ϕ.

First condition satisfied.

2. Consider the production

$T' \to \varepsilon \mid * V \mid + V$ FIRST(T') ∩ Follow(T')

{* + ε} ∩ {$} = ϕ

Second condition satisfied.

Hence the resultant grammar is LL(1).

(e) Parsing table:

	5	3	a	b	*	+	$
E	$E \to 5+T$	$E \to 3-T$					
T			$T \to VT'$	$T \to VT'$			
T'					$T' \to *V$	$T' \to +V$	$T' \to \varepsilon$
V			$V \to a$	$V \to b$			

(2) Error Recovery in predictive Parsing:

* An error is detected during predictive parsing when the following two situations occur.
  - The terminal on top of stack doesnot matches with the next input symbol
  - when non-terminal A is on top of stack, a is the next input symbol and M[A,a] has blank entry (blank entry denoted an error).

Error Recovery is done using Panic mode and pharse level Recovery.

Panic Mode:

In this approach, error Recovery is done by skipping symbols from the input until a token matches with Synchronizing token, the Synchronizing tokens are selected such that the parser should quickly Recover from the errors that are likely to occur in practice.

Some of the Recovery techniques are shown below

1) For a non-terminal A, consider the symbols in Follow(A). These symbols can be considered as Synchronizing tokens and are added into parsing table Replacing only blank entries.

2) For a non-terminal A, consider the symbol in FIRST(A). These symbols can also be considered characters and add to the passing table Replacing only blank entries.

3) If a non-terminal can generate the empty string, then the production deriving ε, can be used as default.

4) If a terminal on top of stack cannot be matched, pop the terminals from the stack and issue "Error message" and insert the Corresponding terminal and continue passing.

phrase level Recovery:

This Recovery method is implemented by filling the blank entries in the predictive parsing table with pointer to error Routines.

These routines may change, insert, Replace or delete symbols from the input and issue appropriate error message. They may also pop from the stack.

Panic Mode problems:

1. Consider the following grammar.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

(a) Construct the parsing table,

(b) Add the Synchronizing tokens for the passing table.

(c) Shows the Sequence of moves made by parser for string ")id*+id"

(a)

Soln:

Symbols	FIRST	Follow
E	(id	) $
E'	+ ε	) $
T	(id	+) $
T'	* ε	+) $
F	(id	+ *) $

Parsing table

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

(b) Parsing table after adding the Synchronizing tokens

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

(c) Sequence of moves made by parser for string) id * + id

Matched	stack	input	Action
	$E\$$	) id * + id $	error, skip-Remove) from input
	$E\$$	id * + id $	output $E \to TE'$
	$TE'\$$	id * + id $	o/p $T \to FT'$
	$FT'E'\$$	id * + id $	o/p $F \to id$
	$id\ T'E'\$$	id * + id $	Match id
id	$T'E'\$$	* + id $	o/p $T' \to *FT'$
id	$*FT'E'\$$	* + id $	Match *
id*	$FT'E'\$$	+ id $	error skip, pop + from input
id*	$FT'E'\$$	id $	o/p $F \to id$
id*	$id\ T'E'\$$	id $	Match id
id*	$T'E'\$$	$	o/p $T' \to \varepsilon$
id * id	$E'\$$	$	o/p $E' \to \varepsilon$
id * id	$\$$	$	

Algorithm : construction of a predictive parsing Table

Input : Grammar G.

output : parsing Table M.

Method :- for each production $A \to \alpha$ of the grammar, do the following.

1. for each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a]

2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A, b]. If ε is in FIRST(α) and \$ is in FOLLOW(A), $A \to \alpha$ to M[A, \$] as well.

- If, after performing the above, there is no production at all in M[A, a], then Set M[A, a] to error (empty entry in the table)