MODULE -3

Syntax Analysis

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation,

Grammars offer significant benefits for both
language designers and compiler writers.
• A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
• From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program.
As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.
• The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.
• A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

The Role of the Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names
can be generated by the grammar for the source language

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parse

There are three general types of parsers for grammars: universal, top-down, and bottom-up. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These general methods are, however, too inefficient to use in production compilers.
The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while 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.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars,

are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars;

Syntax Error Handling

Common programming errors can occur at many different levels.

• *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier e l i p s e S i z e instead of e l l i p s e S i z e — and missing quotes around text intended as a string.

• *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, "{" or " } . " As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

• *Semantic* errors include type mismatches between operators and operands. An example is a r e t u r n statement in a Java method with result type void.

• *Logical* errors 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 program containing = may be well formed; however, it may not reflect the programmer's intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect

an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the *viable-prefix property,* meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue.

Error-Recovery Strategies

Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability. The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information. If errors pile up, it is better for the compiler to give up after exceeding some error limit than to produce an annoying avalanche of "spurious" errors

Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer

must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking

it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop

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. The choice of the local correction is left to the compiler designer. Of course, we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.

Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Productions

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Global Correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given 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. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

Context-Free Grammars

Grammars were introduced in Section 2.2 to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable *stmt* to denote statements and variable *expr* to denote expressions, the production

stmt —> if (*expr*) *stmt* else *stmt*

The Formal Definition of a Context-Free Grammar

context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. *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 when it is clear that we are talking about just

the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords if and else and the symbols "(" and ") ."
2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the *start symbol,* and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
4. 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; this production defines some of the strings denoted by the head.
(b) The symbol -K Sometimes : : = has been used in place of the arrow.
(c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Derivations
The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

Derivation Example

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

OR

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

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen;

Analogous definitions hold for rightmost derivations. Rightmost derivations are sometimes called *canonical* derivations

Left-Most and Right-Most Derivations

Left-Most Derivation

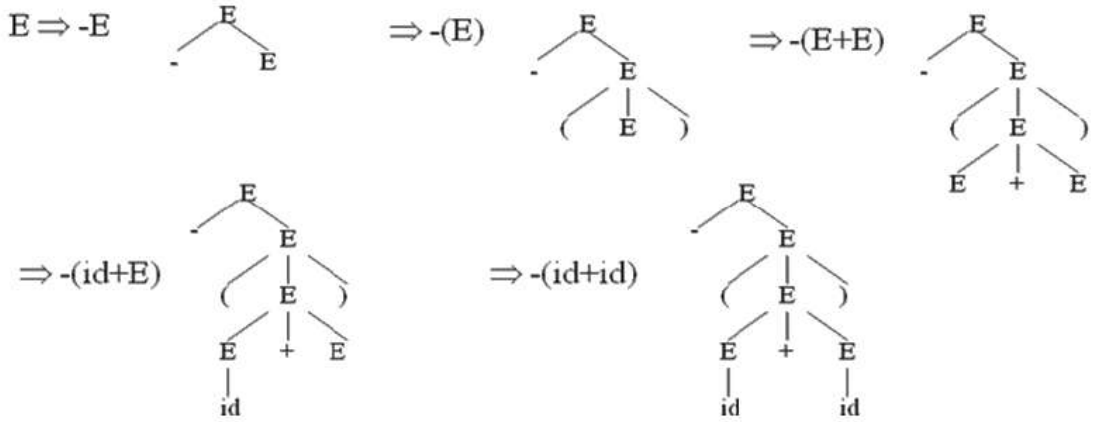
$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Right-Most Derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

Parse Trees and Derivations

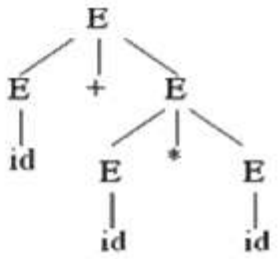
A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal *A* in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this *A* was replaced during the derivation.



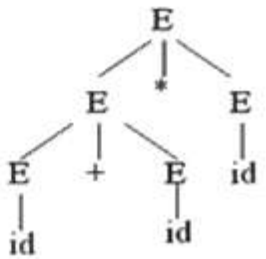
Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

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



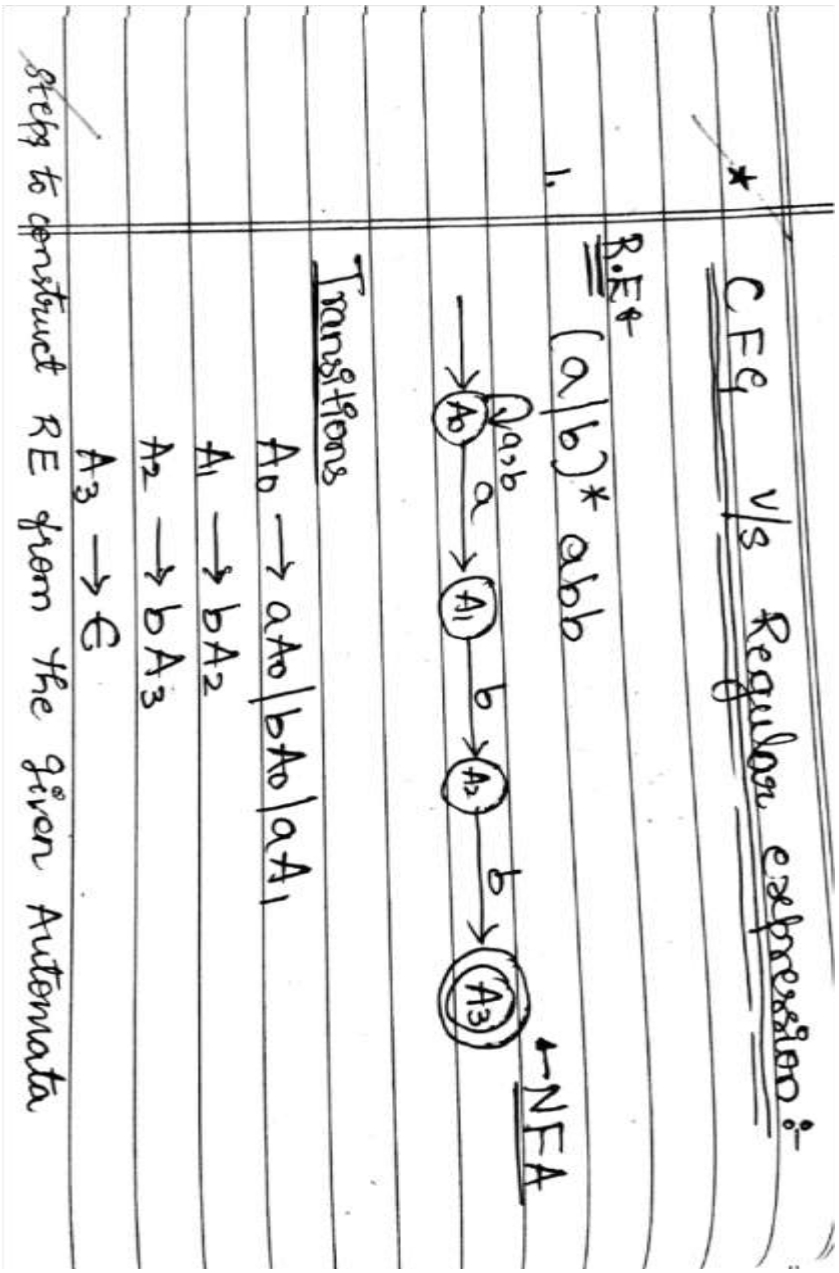
$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$



Context-Free Grammars Versus Regular Expressions

Before leaving this section on grammars and their properties, we establish that grammars are a more powerful notation than regular expressions. Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa. Alternatively, every regular language is a context-free language, but not vice-versa.

For example, the regular expression **(a|b)*abb** and the grammar



describe the same language, the set of strings of a's and 6's ending in *abb*.

We can construct mechanically a grammar to recognize the same language as a nondeterministic finite automaton (NFA). The grammar above was constructed from the NFA , using the following construction:

1. For each state *i* of the NFA, create a nonterminal *Ai.*
2. If state *i* has a transition to state *j* on input a, add the production *Ai* —>

aAj. If state *i* goes to state *j* on input *e*, add the production *Ai* —• *Aj*.
3. *Hi* is an accepting state, add *Ai* —> *e*.
4. If *i* is the start state, make *Ai* be the start symbol of the grammar.

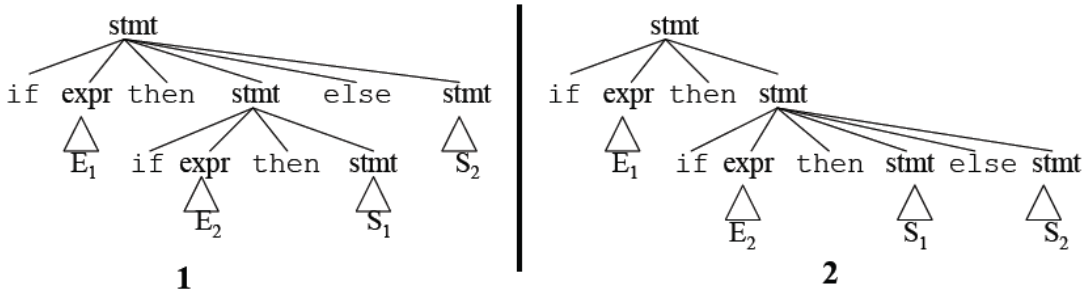
Lexical Versus Syntactic Analysis
E everything that can be described by a regular
expression can also be described by a grammar. We may therefore reasonably ask: "Why use regular expressions to define the lexical syntax of a language?" There are several reasons.

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

Eliminating Ambiguity
Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "danglingelse" grammar:

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

```
if E1 then  if E2 then S1 else  S2
```



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 -> matchedstmt | unmatchedstmt

matchedstmt → if expr then matchedstmt else matchedstmt | otherstmts
unmatchedstmt → 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 → E+E | E*E | E^E | id | (E)

disambiguate the grammar

precedence: ^ (right to left)

* (left to right)

+ (left to right)

E → E+T | T

T → T*F | F

F → G^F | G

G → id | (E)

Left Recursion

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

$$A \overset{+}{\Rightarrow} A\alpha \qquad \text{for some string } \alpha$$

• 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 leftrecursion*), or may appear in more than one step of the derivation.

To eliminate left recursion use the equations

A→βA’

A’→αA’|ε

Eliminate Left-Recursion – Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
  - **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
      $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$
      where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$
      }
    - eliminate immediate left-recursions among A_i productions
  }

Left-Factoring

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

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the

A'-> $\beta_1 | \beta_2$

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

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Recursive-Descent Parsing

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.

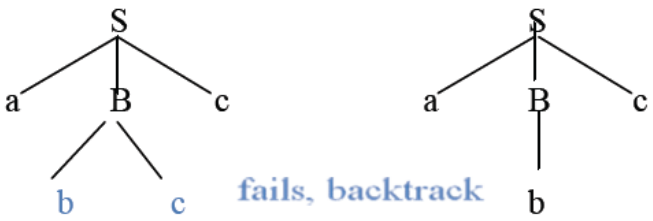
```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

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

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW,** associated with a grammar *G*. During topdown parsing, **FIRST** and **FOLLOW** allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by **FOLLOW** can be used as synchronizing tokens.

Define ***FIRST(a),*** where ***a*** is any string of grammar symbols, to be the set of terminals that begin strings derived from ***a.***

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

Define ***FOLLOW(A),*** for nonterminal *A,* to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form;

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in $\text{FOLLOW}(S)$, where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is in $\text{FOLLOW}(B)$.
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ, then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Identify FIRST and FOLLOW by eliminating left recursion in the grammar

E → E + T | F ; T → T * F | F; F → (E) | id

$$E \longrightarrow E+T \mid T$$
$$T \longrightarrow T*F \mid F$$
$$F \longrightarrow id \mid (E)$$

$\Downarrow$ eliminate immediate left recursion

$$E \longrightarrow T\,E'$$
$$E' \longrightarrow +T\,E' \mid \varepsilon$$
$$T \longrightarrow F\,T'$$
$$T' \longrightarrow *F\,T' \mid \varepsilon$$
$$F \longrightarrow id \mid (E)$$

	FIRST	FOLLOW
E	**id,(**	**$,)**
E'	**+,ε**	**$,)**
T	**id,(**	**+,$,)**
T'	***,ε**	**+,$,)**
F	**id,(**	***,+,$,)**

Predictive Parser

• 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.

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.

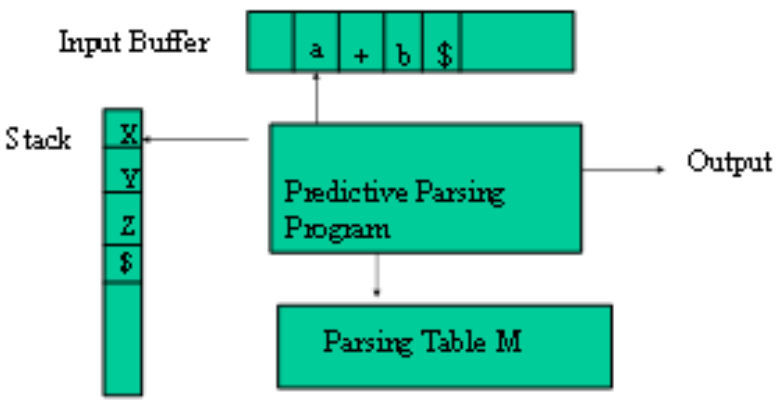


Fig: Model Of Non-Recursive predictive parsing

LL(1) grammar

The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets. The third condition is equivalent to stating that if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ϵ is in FIRST(α).

LL(1) Parser
input buffer
– our string to be parsed. We will assume that its end is marked with a special symbol $.
output
– a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.
stack
– contains the grammar symbols
– at the bottom of the stack, there is a special end marker symbol $.
– initially the stack contains only the symbol $ and the starting symbol S. $S initial stack
– when the stack is emptied (ie. only $ left in the stack), the parsing is completed.
parsing table
– a two-dimensional array M[A, a]
– each row is a non-terminal symbol
– each column is a terminal symbol or the special symbol $
each entry holds a production rule.

Algorithm 4.31: 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 $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to$ $M[A, \$]$ as well.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

For grammar

E->TE’

E’->+TE’|ε

T->FT’

T’->*FT’| ε

F->(E)|id

LL(1) parsing table is

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Parsing moves for id+id

Matched	Stack	Input	Action
	E $	id + id $	
	TE' $	id + id $	o/p E → TE'
	FT'E' $	id + id $	o/p T → FT'
id	id T'E' $	id + id $	o/p F → id
id	T'E' $	+ id $	Match id
	E' $	+ id $	o/p T' → ϵ
	+ TE' $	+ id $	o/p E' → + TE'
id +	TE' $	id $	Match +
	FT'E' $	id $	o/p T → FT'
	id T'E' $	id $	o/p F → id
id + id	T'E' $	$	Match id
	E' $	$	o/p T' → ϵ
	$	$	o/p E' → ϵ
			Accept
id + id $	$	$	Match $

Examples

1.

$S \rightarrow Aa \mid b$

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

- Order of non-terminals: A, S

for A:

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

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

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

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

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

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

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

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

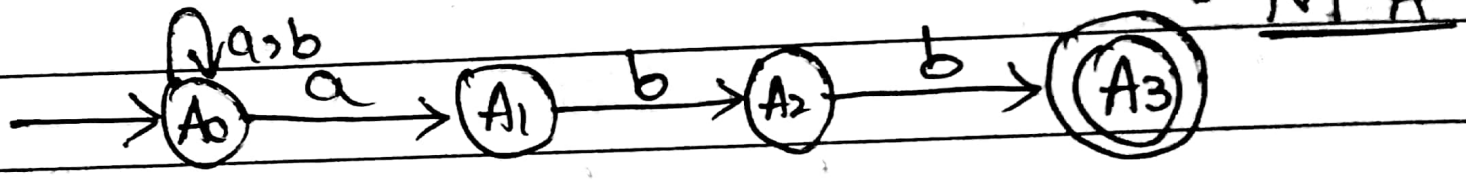
$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \varepsilon$

★ CFG V/S Regular expression :-

R.E :-

1. $(a|b)^* abb$



← NFA

Transitions

$A_0 \rightarrow aA_0 | bA_0 | aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \epsilon$

Steps to construct RE from the given Automata

Ex: 1. $S \to (L) \mid a$

$L \to L, S \mid S$

Sol^n

$A \to A\alpha \mid \beta \quad \to \quad L \to L, S \mid S$

$A \to \beta A' \quad \to \quad L \to SL'$

$A' \to \alpha A' \mid \epsilon \quad \to \quad L' \to , SL' \mid \epsilon$

$S \to (L) \mid a$

$L \to SL'$

$L' \to , SL' \mid \epsilon$

2. $S \to A$

$A \to Ad \mid Ae \mid aB \mid ac$

$B \to bBc \mid f$

$C \to g$

2018

$A \rightarrow Ad \mid Ae \mid aB \mid aC$

$\alpha_1 = d, \alpha_2 = e$
$\beta_1 = aB, \beta_2 = aC$

$A \rightarrow aBA' \mid aCA'$
$A' \rightarrow dA' \mid eA' \mid \varepsilon$

∴ resulting grammar

$S \rightarrow A$
$A \rightarrow aBA' \mid aCA'$
$A' \rightarrow dA' \mid eA' \mid \varepsilon$
$B \rightarrow bBc \mid f$
$C \rightarrow g$

3. $S \rightarrow aBDh$
$B \rightarrow Bb \mid C$

2018

$B \rightarrow Bb \mid C$

$\alpha = b, \beta = C$

$B \rightarrow CB'$
$B' \rightarrow bB' \mid \epsilon$

∴ resulting grammar

$S \rightarrow aBDh$
$B \rightarrow CB'$
$B' \rightarrow bB' \mid \epsilon$

4. $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

soln

$E \rightarrow E+T \mid T$

$\alpha = +T$, $\beta = T$

$E \rightarrow TE'$

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

$T \rightarrow T*F \mid F$

$\alpha = *F$, $\beta = F$

$T \rightarrow FT'$

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

∴ resulting grammar

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

* Left Factoring :-

→ If there is a common prefix in all the production of a same non-terminal such as

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$ → Left factor

eliminating

$A \rightarrow \alpha A' \mid \gamma$ → No left factor

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

Ex 1. $S \to iEtS \mid iEtSeS \mid a$
$E \to b$

Find the longest common prefix.

Soln $S \to \underbrace{iEtS}_{\alpha} \mid \underbrace{iEtSeS}_{\alpha\beta} \mid \underbrace{a}_{\gamma}$

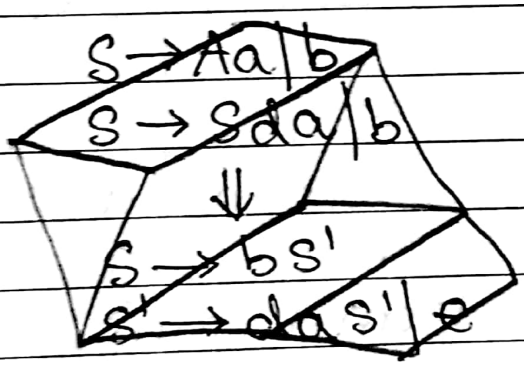
$S \to iEtSS' \mid a$
$S' \to \epsilon \mid eS$
$E \to b$

20/02/17

* Eliminate left recursion from the grammar

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid \epsilon$

⇒



Substitute S in A production

$A \to Ac \mid Sd \mid \epsilon$
$A \to Ac \mid Aad \mid bd \mid \epsilon$
$\alpha_1 \quad \alpha_2 \quad \beta_1 \quad \beta_2$

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

iii) $A \to BF \mid cd$ if $B = \epsilon$ then $A \to F \mid cd$

$B \to b \mid \epsilon$

$F \to eg$

since there is an ϵ production in B consider F.

$FIRST(A) = \{FIRST(B), c\} \cup FIRST(F)$

$FIRST(B) = \{b, \epsilon\}$

$FIRST(F) = \{e\}$

$FIRST(A) = \{b, c, e, \epsilon\}$

iv) $E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

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

$FIRST(E) = \{FIRST(T)\} = \{FIRST(F)\} = \{(, id\}$

$FIRST(E') = \{+, \epsilon\}$

$FIRST(T') = \{*, \epsilon\}$

Non-terminal	FIRST
E	{(, id}
E'	{+, ϵ}
T	{(, id}
T'	{*, ϵ}
F	{(, id}

v) $S \to i\,E\,t\,S\,S' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

$FIRST(S) = \{i, a\}$

$FIRST(S') = \{e, \epsilon\}$

$FIRST(E) = \{b\}$

vi) stmt_seq → stmt stmt_seq'
stmt_seq' → ; stmt_seq | ϵ
stmt → s

FIRST(stmt_seq) = {FIRST(stmt)}
FIRST(stmt_seq) = { s }
FIRST(stmt_seq') = { ; , ϵ }
FIRST (stmt) = { s }

vii) S → L = R | R
L → * R | id
R → L

FIRST (S) = {FIRST(L), FIRST(R)} = {*, id}
FIRST (R) = FIRST(L) = {*, id}

Nonterminal	FIRST
S	{*, id}
L	{*, id}
R	{*, id}

viii) stmt → if_stmt | other
if_stmt → if (expr) stmt else_part
else_part → else stmt | ϵ
expr → 0 | 1

FIRST(stmt) = { FIRST(if_stmt), other } = {if, other}
FIRST(if_stmt) = {if}
FIRST(else_part) = {else, ϵ}
FIRST(expr) = {0, 1}

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

FOLLOW (E) = { $,) }

FOLLOW (T) = { + } ∪ FOLLOW(E)

FOLLOW(T) = { +, $,) }

FOLLOW(E') = { $,) }

FOLLOW(T') = FOLLOW(T) = { +, $,) }

FOLLOW(F) = FIRST(T') = { *, ∈ }

= FIRST(T') ∪ FOLLOW(T)

= { *, $, +,) }

Nonterminal	FOLLOW
E	{ $,) }
T	{ +, $,) }
E'	{ $,) }
T'	{ +, $,) }
F	{ *, +, $,) }

vi) $S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \epsilon$

$E \rightarrow b$

FOLLOW(S) = { $, FIRST(S'), FOLLOW(S') }

= { $, e }

FOLLOW(S') = FOLLOW(S) = { $, e }

FOLLOW(E) = { t }

vii) stmt_seq → stmt stmt_seq'

stmt_seq' → ; stmt seq | ∈

stmt → s

FOLLOW(stmt_seq) = { $ } ∪ FOLLOW(stmt_seq')

FOLLOW(stmt_seq') = { FOLLOW(stmt_seq) }

FOLLOW(stmt) = { FOLLOW FIRST(stmt_seq') }

FOLLOW(stmt) = { ; , FOLLOW(stmt_seq) }
FOLLOW(stmt) = { ; , $ }

Nonterminal	FOLLOW
stmt_seq	{ $ }
stmt	{ ; , $ }
stmt_seq'	{ $ }

viii) $S \rightarrow L = R \mid R$ S { $ }
$L \rightarrow * R \mid id$ L { = , $ }
$R \rightarrow L$ R { = , $ }

FOLLOW(S) = { $ }
FOLLOW(L) = { = , FOLLOW(R) } = { = , $ }
FOLLOW(R) = FOLLOW(S) ∪ FOLLOW(L)
FOLLOW(R) = { $, = }

Non terminal	FOLLOW
S	{ $ }
L	{ = , $ }
R	{ $, = }

ix) stmt → if_stmt | other
if_stmt → if (expr) stmt else_part | ϵ
else_part → else stmt | ϵ
expr → 0 | 1

FOLLOW(stmt) = { $, FOLLOW(else_part) , FIRST(else_part) } = { $, else }
FOLLOW(if_stmt) = FOLLOW(stmt) = { $, else }
FOLLOW(else_part) = FOLLOW(if_stmt) = { $, else }
FOLLOW(expr) = {) }

$S \rightarrow ACB \mid CbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

⇒ **FIRST**

$FIRST(S) = \{FIRST(A) \cup FIRST(C) \cup FIRST(B)\}$

$= \{d, g, \epsilon, h, b, a\}$

$FIRST(A) = \{d, FIRST(B)\} = \{d, g, \epsilon\} \cup FIRST($

$= \{d, g, \epsilon, h\}$

$FIRST(B) = \{g, \epsilon\}$

$FIRST(C) = \{h, \epsilon\}$

FOLLOW

FOLLOW(S) = {$}

FOLLOW(A) = FIRST(C) = {h} ∪ FIRST(B)
= {h, g} ∪ FOLLOW(S)
= {h, g, $}

FOLLOW(C) = {FIRST(B), b, FOLLOW(A)}
= {g, $, b, h}

FOLLOW(B) = FOLLOW(S), FIRST(C), a
= {$, h, a, FOLLOW(A)}

FOLLOW(B) = {$, h, a, g}

Non-teminal	FOLLOW
S	{$}
A	{h, g, $}
B	{$, h, a, g}
C	{g, $, b, h}

Non-teminal	FIRST
S	{a, b, d, g, h, ϵ}
A	{d, g, ϵ, h}
B	{g, ϵ}
C	{h, ϵ}

Predictive parser / Top down parsing / LL parsing

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

⇒ Eliminate left recursion

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

Non-terminal	FIRST	FOLLOW
E	{(, id}	{$,)}
E'	{+, ϵ}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ϵ}	{+, $,)}
F	{(, id}	{*, +, $,)}

Nonterminal	Input symbols					
	+	*	(	)	id	$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Parsing Moves

Input id+id

Matched	Stack	Input	Action
	$E\$$	$id+id\$$	
	$TE'\$$	$id+id\$$	o/p $E \to TE'$
	$FT'E'\$$	$id+id\$$	o/p $T \to FT'$
id	$idT'E'\$$	$id+id\$$	o/p $F \to id$
id	$T'E'\$$	$+id\$$	Match id
	$E'\$$	$+id\$$	o/p $T' \to \epsilon$
	$+TE'\$$	$+id\$$	o/p $E' \to +TE'$
id+	$TE'\$$	$id\$$	Match +
	$FT'E'\$$	$id\$$	o/p $T \to FT'$
	$idT'E'\$$	$id\$$	o/p $F \to id$
id+id	$T'E'\$$	$\$$	Match id
	$E'\$$	$\$$	o/p $T' \to \epsilon$
	$\$$	$\$$	o/p $E' \to \epsilon$
			Accept
id+id$	$	$	Match $

The string is successfully parsed.

2. $S \to a \mid (L)$
$L \to L, S \mid S$

$\Rightarrow$ $S \to a \mid (L)$
$L \to SL'$
$L' \to ,SL' \mid \epsilon$

Non terminal	FIRST	FOLLOW
S	{a, (}	{$, , ,)}
L	{a, (}	{)}
L'	{, , ϵ}	{)}

Non terminal	Input symbols				
	a	(	)	,	$
S	S→a	S→(L)			
L	L→SL'	L→SL'			
L'			L'→ϵ	L'→,SL'	

Parsing moves Input (a, a)

Matched	Stack	Input	Action
	S$	(a,a)$	
	(L)$	(a,a)$	o/p S→(L)
(	L)$	(a,a)$	Matched (
	SL')$	(a,a)$	o/p L→SL'
	aL')$	(a,a)$	o/p S→a
(a	L')$	(a,a)$	Matched a
	,SL')$	(a,a)$	o/p L'→,SL'
(a,	SL')$	(a,a)$	Matched ,
	aL')$	(a,a)$	o/p S→a
(a,a	L')$	(a,a)$	Matched a
	)$	(a,a)$	o/p L'→ϵ
(a,a)	$	$	Accept

*

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

$T \to V \mid V * V \mid V+V$

$V \to a \mid b$

Input : $5 \to a+b$

⇒

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

$T \to VT'$

$T' \to *V \mid +V \mid \epsilon$

$V \to a \mid b$

Nonterminal	FIRST	FOLLOW
E	{5, 3}	{$}
T	{a, b}	{$}
T'	{*, +, ε}	{$}
V	{a, b}	{$, *, +}

Nonterminal	input symbols						
	−	5	3	*	+	a	b	$
E		E→5+T	E→3−T					
T						T→VT'	T→VT'	
T'		N/A		T'→*V	T'→+V			T'→ε
V						V→a	V→b	

Matched	Stack	Input	Action
	E $	5+a*b $	
	5+T$	5+a*b$	E→5+T
5	+T$	+a*b$	Matched 5
5+	T$	a*b$	Matched +
	VT'$	a*b$	T→VT'
	aT'$	a*b$	V→a
5+a	T'$	*b$	Matched a
	*V$	*b$	T'→*V
5+a*	V$	b$	Matched *
	b$	b$	V→b
5+a*b	$	$	Matched b
5+a*b	$	$	Accept.

03/17* **Error recovery in predictive parsing:-**

1. **Panic Mode:** Skip the input symbols until the synchronizing pattern is found. 'synch' indicates synchronizing tokens obtained from the follow set of the non-terminal.

To recover from error

i) If the parser refers to the blank entry in the table, the input symbol is skipped.

ii) If the parser refers to synch entry in the table the non terminal on the top of the stack is popped in an attempt to resume parsing.

Ex#

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

	FOLLOW	FIRST
E	{$,)}	{(, id}
E'	{$,)}	{+, ϵ}
T	{+, $,)}	{(, id}
T'	{+, $,)}	{*, ϵ}
F	{*, +, $,)}	{(, id}

Non-terminals	Input Symbols					
	(	)	+	*	id	$
E	E→TE'	synch			E→TE'	synch
E'		E'→ϵ	E'→+TE'			E'→ϵ
T	T→FT'	synch	synch		T→FT'	synch
T'		T'→ϵ	T'→ϵ	T'→*FT'		T'→ϵ
F	F→(E)	synch	synch	synch	F→id	synch

Match	Stack	Input	Action
	E$	*id*+id$	Error, skip *
	E$	id*+id$	
	TE'$	id*+id$	E→TE'
	FT'E'$	id*+id$	T→FT'
	idT'E'$	id*+id$	F→id
id	T'E'$	*+id$	Matched id
	*FT'E'$	*+id$	T'→*FT'
id*	FT'E'$	+id$	Matched *
	T'E'$	+id$	M[F,+]=synch: pop F
	E'$	+id$	T'→ϵ
	+TE'	+id$	E'→+TE'

id*+	TE'$	id$	Matched +
	FT'E'$	id$	T→FT'
	id T'E'$	id$	F→id
id*+id	T'E'$	$	Matched id
	E'$	$	T'→ϵ
	$	$	E'→ϵ
id*+id	$	$	Accept.

★ A grammar G is LL(1) if whenever $A \to \alpha | \beta$ are two distinct productions of G, the following conditions hold:

i) For no terminal 'a' do both α and β derive strings beginning with a 'a'.

ii) Atmost one of $\alpha | \beta$ can derive the empty string.

iii) If $\beta \overset{*}{\Rightarrow} \epsilon$ then α does not derive any string beginning with the terminal in FOLLOW(A). like wise, $\alpha \overset{*}{\Rightarrow} \epsilon$ then β does not derive any string beginning with a terminal in FOLLOW(A).

- i) and ii) means FIRST(α) and FIRST(β) should be disjoint sets.
- iii) means if $\beta \overset{*}{\Rightarrow} \epsilon$ then FIRST(α) and FOLLOW(A) should be disjoint sets and if $\alpha \overset{*}{\Rightarrow} \epsilon$ then FIRST(β) and FOLLOW(A) should be disjoint sets.

★ Check whether the grammar is in LL(1):

i) Table construction method P.T.O

ii) Without constructing the table.

b. $S \to iEtSS' | a$
$S' \to eS | \epsilon$
$E \to b$

Table Construction Method ↓

i)

Non-terminal	FIRST	FOLLOW
S	{i, a}	{$, e}
S'	{e, ϵ}	{$, e}
E	{b}	{t}

Non-terminal	Input symbols					
	i	a	e	b	t	$
S	$S \to iEtSS'$	$S \to a$				
S'			$S' \to eS$ $S' \to \epsilon$			$S' \to \epsilon$
E				$E \to b$		

Since there are multiple entries for the same Non-terminal and terminal.

$S' \to eS$

$S' \to \epsilon$

Hence, the grammar is not LL(1).

ii) W/o Table construction

$S \to iEtSS' | a$
$\alpha \quad \beta$

→ If there is more than one production then only we can apply this method.

$FIRST(\alpha) \cap FIRST(\beta) = \phi$

$\{i\} \cap \{a\} = \phi$

$S' \to eS | \epsilon$
$\alpha \quad \beta$

$FIRST(\alpha) \cap FOLLOW(S') = \phi$

$\{e\} \cap \{\$, e\} \neq \phi$

∴ The grammar is not in LL(1).

2. $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

q)

Non-terminal	FIRST	FOLLOW
S	{a, b}	{$}
A	{ϵ}	{a, b}
B	{ϵ}	{b, a}

Non-terminal	Input Symbols		
	a	b	$
S	S→AaAb	S→BbBa	
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

Since there are no multiple productions ~~S→AaAb and S→BbBa~~

∴ Grammar is ~~not~~ in LL(1).

pp)

$S \to AaAb \mid BbBa$

$\alpha \quad \beta$

$FIRST(\alpha) \cap FIRST(\beta) = \phi$

$\{a\} \cap \{b\} = \phi$

∴ Grammar is in LL(1).

3. stmt → if_stmt | other
if_stmt → if (expr) stmt else_part
else_part → else stmt | ϵ
expr → 0 | 1

Ans

Non-terminal	FIRST	FOLLOW
stmt	{if, other}	{$, else}
if_stmt	{if}	{$, else}
else_part	{else, ϵ}	{$, else}
expr	{0, 1}	{)}

Non-terminal	Input symbols							
	if	else	0	1	(	)	$	other
stmt	stmt → if_stmt							stmt → other
if_stmt	if_stmt → if (expr) stmt else_part							
else_part		else_part → else stmt else_part → ϵ					else_part → ϵ	
expr			expr → 0	expr → 1				

else_part → else stmt | ϵ
∴ Grammar is not in LL(1)

Ans

stmt → if_stmt | other
α β
FIRST(α) ∩ FIRST(β) = ϕ
{if} ∩ {other} = ϕ

else_part → else stmt | ϵ
α β
FIRST(α) ∩ FOLLOW(else_part) = ϕ
{else} ∩ {$, else} ≠ ϕ
∴ Grammar is not in LL(1)

4. Check whether the grammar is LL(1)

$S \to 1AB \mid \epsilon$

$A \to 1AC \mid 0C$

$B \to 0S$

$C \to 1$

⇒

Non terminal	FIRST	FOLLOW
S	{1, ε}	{$}
A	{1, 0}	{0, 1}
B	{0}	{$}
C	{1}	{0, 1}

$S \to 1AB \mid \epsilon$

$\alpha \quad \beta$

FIRST(α) + FIRST(β)

$\{1\} \cap \{\$\} = \phi$

$S \to 1AC \mid 0C$

$\{1\} \cap \{0\} = \phi$

∴ The grammar is LL(1).

5. Eliminate left recursion

$A \to Ba \mid Aa \mid c$

$B \to Bb \mid Ab \mid d$

⇒

$A \to BaA' \mid cA'$

$A' \to aA' \mid \epsilon$

$B \to bB' \mid aA'bB'$

$B' \to cA'bB' \mid dB' \mid \epsilon$

6. ~~Eliminate~~ left factoring the grammar

stmt → assign_stmt | call_stmt | other
assign_stmt → id := exp
call_stmt → id (exp_list)

⇒ stmt → id := exp | id (exp_list) | other
assign_stmt → id := exp
call_stmt → id (exp_list)

stmt → id stmt' | other
stmt' → := exp | (exp_list) | ∈

Bottom up parsing

Introduction

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). Given a grammar and a sentence belonging to that grammar, if we have to show that the given sentence belongs to the given grammar, there are two methods.

1. Leftmost – Derivation
2. Rightmost – Derivation

In left most derivation we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence. The parsing methods based on this are known as top-down methods and we have already discussed this in previous chapter.

In rightmost derivation we start from the given sentence and using various productions, we try to reach the start symbol. The parsing method based on this concept are known as bottom-up method and are generally used in compilers generated using various tools like LEX and YACC. The bottom up parsing methods is more general and efficient. They can find syntactic error as soon as they occur.

We will be studying the following methods of parsing in this chapter

1. LR(0) Parsing
2. SLR (1) Parsing
3. LALR (1) Parsing Methods

Where LR stands for Left to Right scan and Rightmost derivation in reverse. SLR Stands for simple LR and LALR for Look Ahead LR and 1 in brackets indicate the number of look aheads.

LR (0) stands for no look ahead and LR (1) for one look ahead symbol. In general there can be LR (K) parsers with 'K' lookahead symbols.

Reductions

We can think of bottom-up parsing as the process of "reducing" a string *w* to the start symbol of the grammar. At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in :

E => *T* => *T* * *F* => *T* *** id** => *F* *** id** => **id* id**

Handle Pruning
Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the
reverse of a rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
idi * id$_2$	idi	F - ^ i d
F * id$_2$	*F*	T -> *F*
T * i d$_2$	id$_2$	F - » i d
T * *F*	T * F	*E*^*T* * *F*

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals *w* to be parsed.

Bottom up parsing

Bottom-up parsers in general use explicit stack. They are also known as shift reduce parsers.

Example: Consider a grammar S → a S b c and a sentence a a c b b. The parser put the sentence to be recognized in the input buffer appended with end of input symbol $ and bottom of stack has also $.

Step	Start	Input buffer	Action
1	$	a a c b b $	Shift
2	$ a	a a b b $	Shift
3	$ a a	c b b $	Shift
4	$ a a c	b b $	Reduce using S → c
5	$ a a S	b b $	Shift
6	$ a a S b	b $	Reduce using S → a S b
7	$ a S	b $	Shift
8	$ a S b	$	Reduce using S → a S b
9	$ S	$	Accept.

The parser consults a table indexed by two parameters to be discussed later. The parameters what is on the top of stack and input character pointed by input buffer pointer. Assume for the time being that the table tells the parser to do one of the following activities.

1.	Shift	–	Shift the symbol to stack
2.	Reduce	–	Pop some symbols in the stack and replace by a non-terminal
3.	Accept	–	Input is syntactically correct, therefore accept
4.	Error	–	Input is syntactically not correct.

The table formation will be discussed later. From the figure above, after 3 shifts the table tells the parser to reduce in step-4 that is pop c and replace by (i.e., push) 'S'. In the 5^{th} Step again shift is executed. In 6^{th} step the parser is told to pop 3 symbols and replace it by S, in 7^{th} step again parser is told to shift. In 8^{th} step the parser pops 3 symbols & replaces it by S i.e., reduce action takes place. When top of stack is start symbol i.e., S in this case and input buffer is empty i.e., input buffer is pointing to $ (this indicates there is no more input available). The parser is told to carryout accept action. The parser is able to recognize that aacbb is indeed a valid sentence of the given grammar. We shall see later on how shift reduce and accept actions are indicated.

Example:

Consider another example of parsing using the grammar

S → a A c B e
A → A b b
B → d

Recognize abbcde

Start	Input buffer	Action
$	a b b c d e $	Shift
$ a	b b c d e $	Shift
$ a b	b c d e $	Reduce A → b
$ a A	b c d e $	Shift
$ a A b	c d e $	Reduce A → A b
$ a A	c d e $	Shift
$ a A c	d e $	Shift
$ a A c d	e $	Reduce B → d
$ a A c B	e $	Shift
$ a A c B e	$	Reduce S → A c B e
$ S	$	Accept

Rightmost Derivation

S ⇒a A c B e
⇒a A c d e replace B → d
⇒ a A b c d e replace A → A b
⇒ a b b c d e replace A → b

Example 4.38: An ambiguous grammar can never be LR. For example, c
sider the dangling-else grammar (4.14) of Section 4.3:

Example 4.38: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

Example 4.38: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

stmt -» if *expr* **then** *stmt*
| if *expr* **then** *stmt* **else** *stmt*
I **other**

If we have a shift-reduce parser in configuration

STACK	INPUT
• • • if *expr* **then** *stmt*	**else** • • $

Shift/reduce conflict

Example 4.38: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

stmt -» if *expr* **then** *stmt*
| if *expr* **then** *stmt* **else** *stmt*
I **other**

If we have a shift-reduce parser in configuration

STACK	INPUT
• • • if *expr* **then** *stmt*	**else** • •$

we cannot tell whether if *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the **else** on the input, it might be correct to reduce if *expr* **then** *stmt* to *stmt,* or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative if *expr* **then** *stmt* **else** *stmt.*

Reduce/Reduce conflict

(1)	*stmt*	->	**id** (*parameterJist)*
(2)	*stmt*		expr := errpr
(3)	*parameter-list*		*parameterJist* , *parameter*
(4)	*parameter-list*	->	*parameter*
(5)	*parameter*		**id**
(6)	*expr*		**id** (*exprJist)*
(?)	*expr*		**id**
(8)	*exprJist*		*exprJist* , *expr*
(9)	*exprJist*	->	*expr*

Figure 4.30: Productions involving procedure calls and array references

STACK	INPUT
• • • **id (id**	**, id)** • • •

It is evident that the **id** on top of the stack must be reduced, but by which production? The correct choice is production (5) if p is a procedure, but production (7) if p is an array. The stack does not tell which; information in the symbol table obtained from the declaration of p must be used.