

SYSTEM SOFTWARE and COMPILERS- 18CS61

Prof. Sampada K S, Assistant Professor
DEPT. OF CSE | RNSIT

MODULE 3

SYNTAX ANALYSIS

Syntax Analysis: Introduction, Context Free Grammars, Writing a grammar, Top Down Parsers, Bottom-Up Parsers

Text book 2: Chapter 4 4.1, 4.2 4.3 4.4 4.5

4.1. ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1. **Top down parser**: which build parse trees from top(root) to bottom(leaves)
2. **Bottom up parser**: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

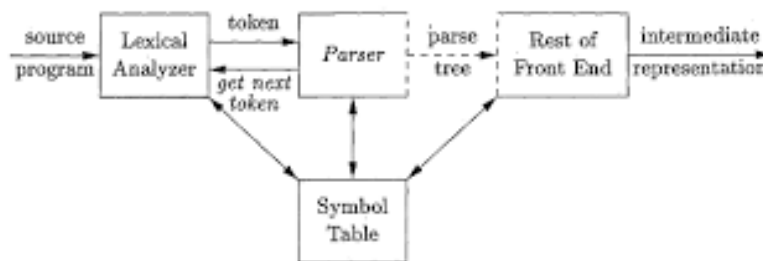
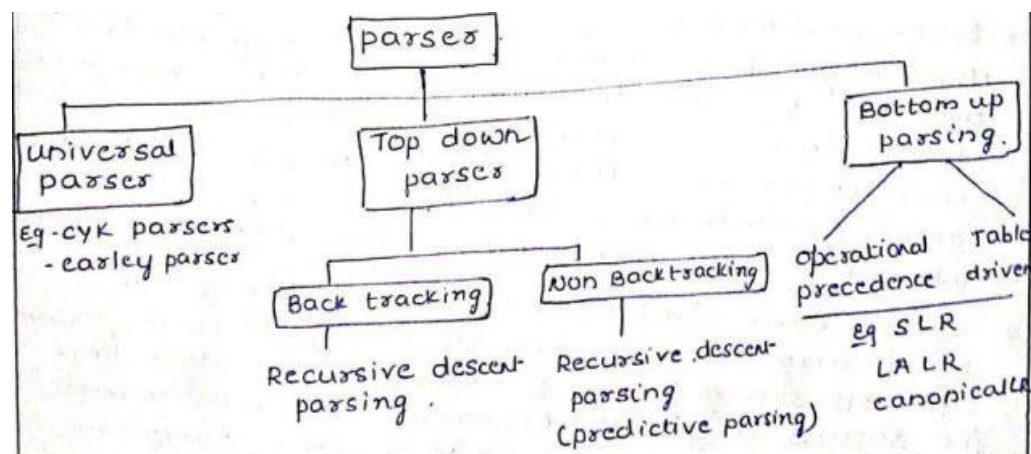


Figure 4.1: Position of parser in compiler model

There are three general types of parsers for grammars: Universal, top-down and bottom-up.



Syntax error handling:

Types or Sources of Error – There are three types of error: logic, run-time and compile-time error:

Logic errors occur when programs operate incorrectly but do not terminate abnormally (or crash). Unexpected or undesired outputs or other behavior may result from a logic error, even if it is not immediately recognized as such.

A run-time error is an error that takes place during the execution of a program and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error is an example of this. Logic errors occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.

Compile-time errors rise at compile-time, before the execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is an example of this.

Classification of Compile-time error –

- Lexical : This includes misspellings of identifiers, keywords or operators
- Syntactical : a missing semicolon or unbalanced parenthesis
- Semantical : incompatible value assignment or type mismatches between operator and operand
- Logical : code not reachable, infinite loop.

Finding error or reporting an error – Viable-prefix is the property of a parser that allows early detection of syntax errors.

Goal detection of an error as soon as possible without further consuming unnecessary input

How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

Example: for(;), this will report an error as for having two semicolons inside braces.

Error Recovery –

The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are as follows.

1. Panic mode recovery :

This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops while recovering error. The parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens (are typically the statement or expression terminators) is found. This is adequate when the presence of multiple errors in the same statement is rare. Example: Consider the erroneous expression- $(1 + + 2) + 3$. Panic-mode recovery: Skip ahead to the next integer and then continue. Bison: use the special terminal error to describe how much input to skip.

$E \rightarrow \text{int} | E + E | (E) | \text{error int} | (\text{error})$

2. Phase level recovery :

When an error is discovered, the parser performs local correction on the remaining input. If a parser encounters an error, it makes the necessary corrections on the remaining input so that the parser can continue to parse the rest of the statement. You can correct the error by deleting extra semicolons, replacing commas with semicolons, or reintroducing missing semicolons. To prevent going in an infinite loop during the correction, utmost care should be taken. Whenever any prefix is found in the remaining input, it is replaced with some string. In this way, the parser can continue to operate on its execution.

3. Error productions :

The use of the error production method can be incorporated if the user is aware of common mistakes that are encountered in grammar in conjunction with errors that produce erroneous constructs. When this is used, error messages can be generated during the parsing process, and the parsing can continue. Example: write 5x instead of 5*x

4. Global correction :

In order to recover from erroneous input, the parser analyzes the whole program and tries to find the closest match for it, which is error-free. The closest match is one that does not do many insertions, deletions, and changes of tokens. This method is not practical due to its high time and space complexity.

4.2 CONTEXT-FREE GRAMMARS:

A context-free grammar has four components:

A set of **non-terminals (V)**. Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

A **set of tokens**, known as **terminal symbols (Σ)**. Terminals are the basic symbols from which strings are formed.

A **set of productions (P)**. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or non-terminals, called the right side of the production.

One of the non-terminals is designated as the **start symbol (S)**; from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{ w \mid w = w^R \}$ is not a regular language. But it can be described by means of CFG, as illustrated below:

$G = (V, \Sigma, P, S)$

Where:

$V = \{ Q, Z, N \}$

$\Sigma = \{ 0, 1 \}$

$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \epsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1 \}$

$S = \{ Q \}$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example: Production rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: **id + id * id**

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

$E \rightarrow E + E$
 $E \rightarrow E + E * E$
 $E \rightarrow E + E * id$
 $E \rightarrow E + id * id$
 $E \rightarrow id + id * id$

Parse Tree

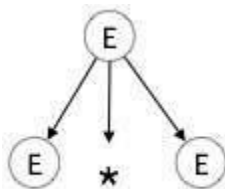
A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

The left-most derivation is:

$E \rightarrow E * E$
 $E \rightarrow E + E * E$
 $E \rightarrow id + E * E$
 $E \rightarrow id + id * E$
 $E \rightarrow id + id * id$

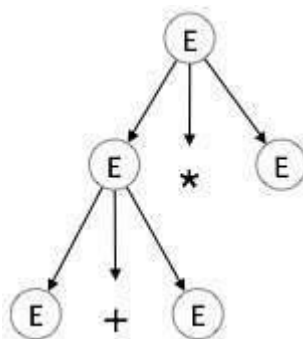
Step 1

$E \rightarrow E * E$

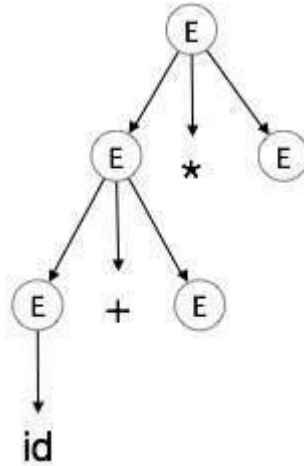


Step 2:

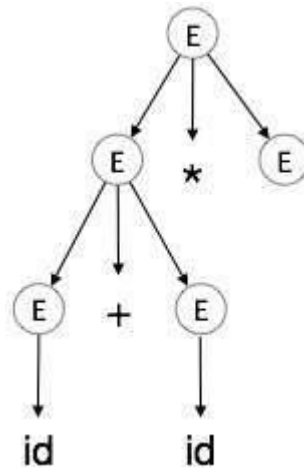
$E \rightarrow E + E * E$



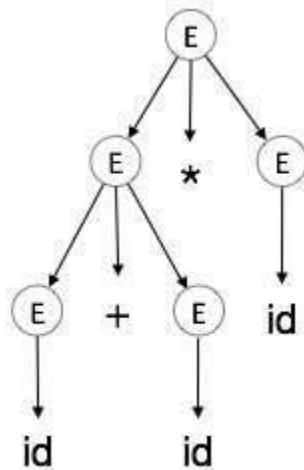
Step 3:

 $E \rightarrow id + E * E$ 

Step 4:

 $E \rightarrow id + id * E$ 

Step 5:

 $E \rightarrow id + id * id$ 

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A **parse tree depicts associativity and precedence of operators**. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

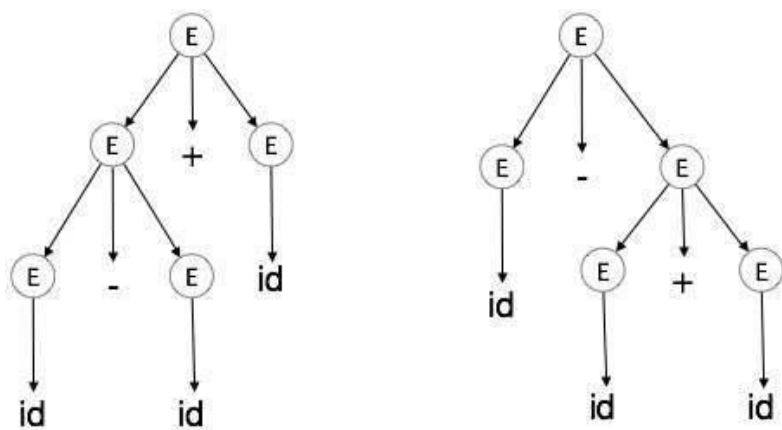
Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example

$E \rightarrow E + E$
 $E \rightarrow E - E$
 $E \rightarrow id$

For the string $id + id - id$, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as **Addition, Multiplication, Subtraction, and Division** are **left associative**. If the expression contains:

id op id op id

it will be evaluated as:

(id op id) op id

For example, (id + id) + id

Operations like **Exponentiation are right associative**, i.e., the order of evaluation in the same expression will be:

id op (id op id)

For example, id ^ (id ^ id)

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, $2+3*4$ can have two different parse trees, one corresponding to $(2+3)*4$ and another corresponding to $2+(3*4)$. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically $*$ (multiplication) has precedence over $+$ (addition), so the expression $2+3*4$ will always be interpreted as:

$2 + (3 * 4)$

These methods decrease the chances of ambiguity in a language or its grammar.

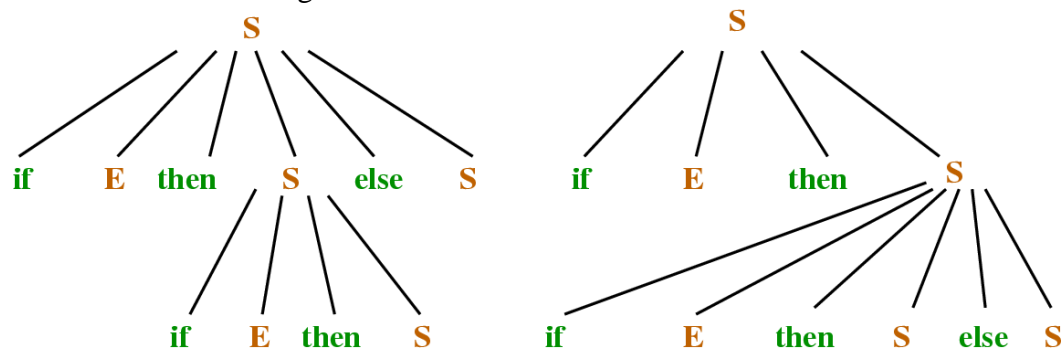
Consider $VN = \{S, E\}$, $VT = \{\text{if, then, else}\}$ and a grammar $G = (VT, VN, S, P)$ such that the following

$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S$

are productions of G . Then the string

$w = \text{if } E \text{ then if } E \text{ then } S \text{ else } S$

has two parse trees as shown on Figure 5.



Two parse trees for the same if-then-else statement.

In all programming languages with if-then-else statements of this form, the second parse tree is preferred. Hence the general rule is: match each else with the previous closest then. This disambiguating rule can be incorporated directly into a grammar by using the following observations.

- A statement appearing between a then and a else must be matched. (Otherwise there will be an ambiguity.)
- Thus statements must split into kinds: *matched* and *unmatched*.
- A *matched statement* is
 - either an if-then-else statement containing no unmatched statements

- or any statement which is not an if-then-else statement and not an if-then statement.
- Then an *unmatched statement* is
 - an if-then statement (with no else-part)
 - an if-then-else statement where unmatched statements are allowed in the else-part (but not in the then-part).

stmt \mapsto *matched-stmt* / *unmatched-stmt*

matched-stmt \mapsto **if** *expr* **then** *matched-stmt* **else** *matched-stmt*

matched-stmt \mapsto *non-alternative-stmt*

unmatched-stmt \mapsto **if** *expr* **then** *stmt*

unmatched-stmt \mapsto **if** *expr* **then** *matched-stmt* **else** *unmatched-stmt*

* Eliminating ambiguity using precedence & Associativity

eg: $E \rightarrow E * E \mid E - E$

$E \rightarrow E \wedge E \mid E / E$

$E \rightarrow E + E$

$E \rightarrow (E) \mid id$

To eliminate the ambiguity, precedence is defined.

operators	Associativity	non-terminal.
+, -	LEFT	E
*, /	LEFT	T
\wedge	RIGHT	P

unambiguous grammar:

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

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

$P \rightarrow F \wedge P \mid F$

$F \rightarrow (E) \mid id$

4.3. TOP-DOWN PARSING

A program that performs syntax analysis is called a **parser**. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. **Top-down parsers check to see if a string can be generated by a grammar by creating a parse tree** starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a

grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

Example of top-down parser:

Consider the grammar

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

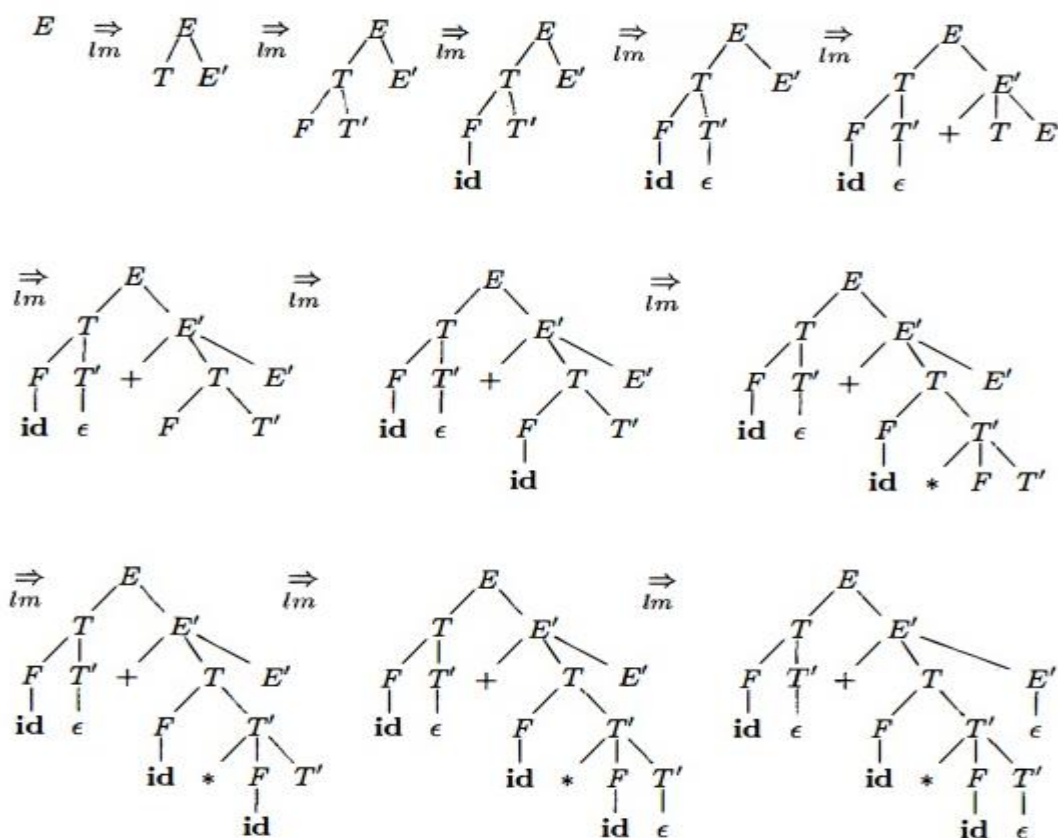


Figure 4.12: Top-down parse for `id + id * id`

4.3.1. RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form $S \rightarrow S_1 S_2 S_3$ or $S \rightarrow S_1$. A syntax of the form $S \rightarrow S_1 S_2 S_3$ defines sentences that consist of a sentence of the form S_1 followed by a sentence of the form S_2 followed by a sentence of the form S_3 . A syntax of the form $S \rightarrow S_1$ defines zero or one occurrence of the form S_1 .

A syntax of the form $S \rightarrow S_1^*$ defines zero or more occurrences of the form S_1 .

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```

proc syntaxAnalysis()
begin
initialize(); // initialize global data and structures
nextToken(); // get the lookahead token
program(); // parser routine that implements the start symbol
end;

```

Algorithm for recursive descent parsing:-

```

void A() {
1. choose an A-production,  $A \rightarrow X_1 X_2 \dots X_n$ ;
2. for ( $i = 1$  to  $n$ ) {
3.   if ( $X_i$  is a non terminal)
4.     call procedure  $X_i()$ ;
5.   else if ( $X_i =$  current i/p symbol  $a$ )
6.     advance the i/p to the next symbol;
7.   else /* an error has occurred */.
8. }
9. }

```

The above algorithm is non—deterministic. General Recursive descent may require backtracking.

Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

```

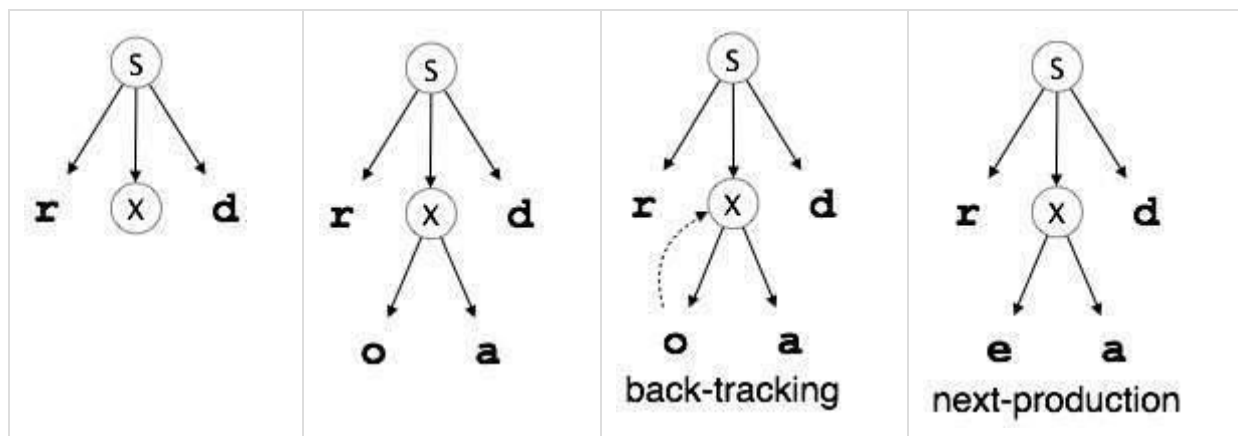
S → rXd | rZd
X → oa | ea
Z → ai

```

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S ($S \rightarrow rXd$) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left ($X \rightarrow oa$). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, ($X \rightarrow ea$).

Now the parser matches all the input letters in an ordered manner. The string is accepted.



Example – Write down the algorithm using Recursive procedures to implement the following Grammar.

$E \rightarrow TE'$

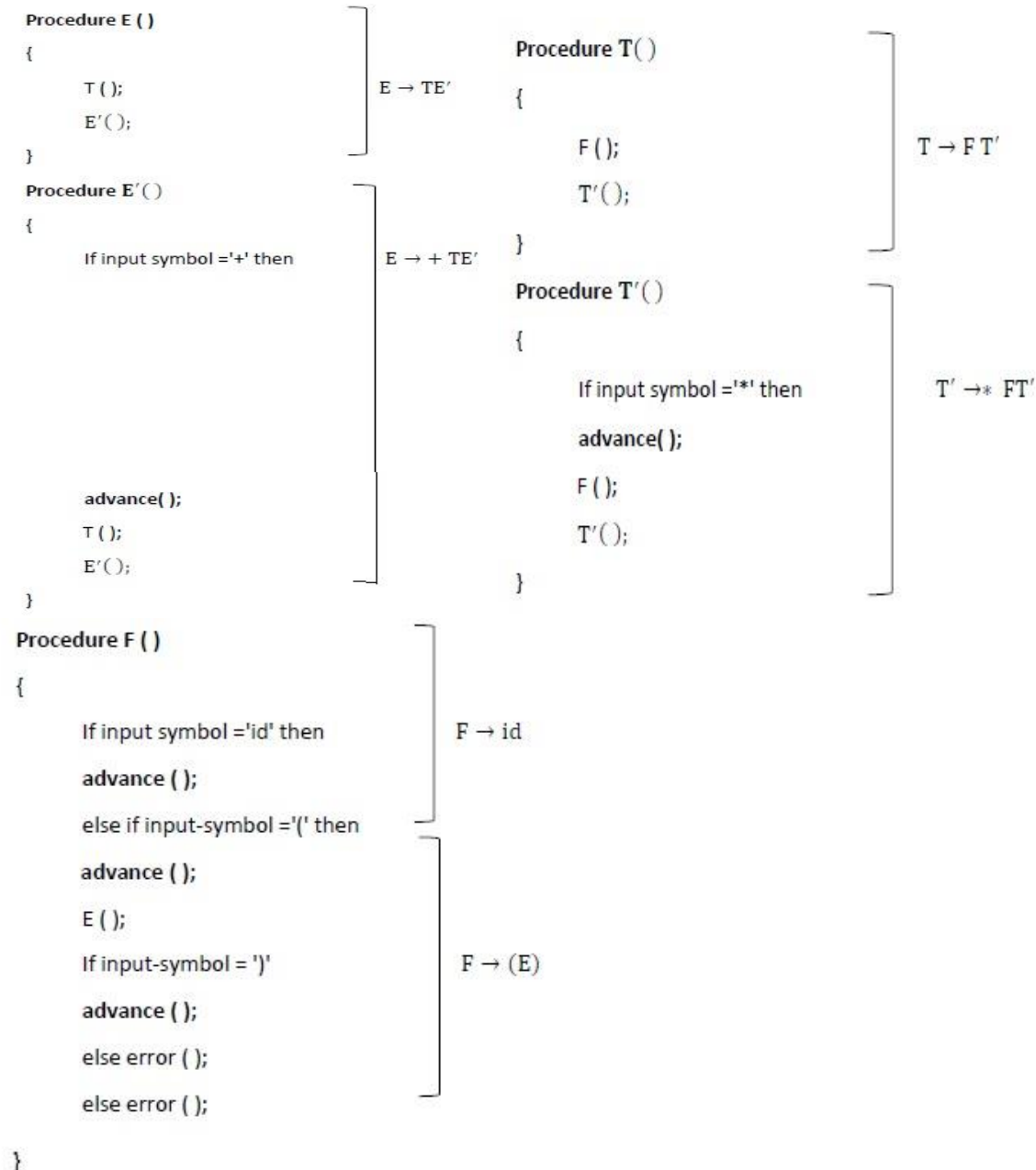
$E' \rightarrow +TE'$

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

Solution



4.3.2. Left Recursion

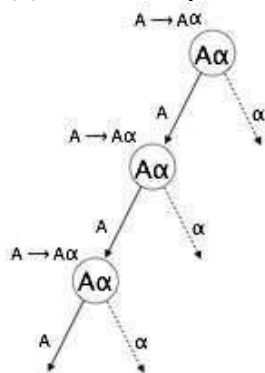
A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

Example:

- (1) $A \Rightarrow A\alpha \mid \beta$
 (2) $S \Rightarrow A\alpha \mid \beta$
 $A \Rightarrow Sd$

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

$$A \Rightarrow A\alpha \mid \beta$$

is converted into following productions

$$A \Rightarrow \beta A'$$

$$A' \Rightarrow \alpha A' \mid \epsilon$$

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

START

Arrange non-terminals in some order like $A_1, A_2, A_3, \dots, A_n$

for each i from 1 to n

```
{
    for each j from 1 to i-1
    {
        replace each production of form  $A_i \Rightarrow A_j \gamma$ 
        with  $A_i \Rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \delta_3 \gamma \mid \dots \mid \gamma$ 
        where  $A_j \Rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_n$  are current  $A_j$  productions
    }
}
```

eliminate immediate left-recursion

END

Example

The production set

$$S \Rightarrow A\alpha \mid \beta$$

$$A \Rightarrow Sd$$

after applying the above algorithm, should become

$$S \Rightarrow A\alpha \mid \beta$$

$$A \Rightarrow A\alpha d \mid \beta d$$

and then, remove immediate left recursion using the first technique.

$$A \Rightarrow \beta d A'$$

$$A' \Rightarrow \alpha d A' \mid \epsilon$$

Now none of the production has either direct or indirect left recursion.

4.3.3. Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

$$A \Rightarrow \alpha\beta \mid \alpha\gamma \mid \dots$$

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

Example

The above productions can be written as

$$A \Rightarrow \alpha A'$$

$$A' \Rightarrow \beta \mid \gamma \mid \dots$$

Now the parser has only one production per prefix which makes it easier to take decisions.

4.3.4. FIRST AND FOLLOW

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 terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then place a in FIRST(X)
4. If for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}) that is, $Y_1 \dots Y_{i-1} \Rightarrow^* \epsilon$.
5. If ϵ is in FIRST(Y_j) for all $j=1,2,\dots,k$, then add ϵ to FIRST(X).

For example, everything in FIRST(Y_j) is surely in FIRST(X). If Y_1 does not derive ϵ , then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow^* \epsilon$, then we add FIRST(Y_2) and so on.

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

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ in the input right endmarker.
2. If there is a production $A \Rightarrow aBs$ where FIRST(s) except ϵ is placed in FOLLOW(B).
3. If there is a production $A \Rightarrow aB$ or a production $A \Rightarrow aBs$ where FIRST(s) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow.

Find the first and follow of all nonterminals in the Grammar-

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

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

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with $i=1$ in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with $i=1$, the production $T \rightarrow FT'$ implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW, we put \$ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied to production $F \rightarrow (E)$, right parenthesis is also in FOLLOW(E). By rule 3 applied to production $E \rightarrow TE'$, \$ and right parenthesis are in FOLLOW(E').

Calculate the first and follow functions for the given grammar-

$S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \epsilon$

The first and follow functions are as follows-

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

4.3.5. LL(1) GRAMMAR

A context-free grammar $G = (V_T, V_N, S, P)$ whose parsing table has no multiple entries is said to be $LL(1)$. In the name $LL(1)$,

- the first L stands for scanning the input from left to right,
- the second L stands for producing a leftmost derivation,
- and the 1 stands for using **one** input symbol of lookahead at each step to make parsing action decision.

A language is said to be $LL(1)$ if it can be generated by a $LL(1)$ grammar. It can be shown that $LL(1)$ grammars are not ambiguous and not left-recursive.

Moreover we have the following theorem to characterize $LL(1)$ grammars and show their importance in practice.

A context-free grammar $G = (V_T, V_N, S, P)$ is $LL(1)$ if and only if for every nonterminal A and every strings of symbols α and β such that $A \Rightarrow^* \alpha$ and $A \Rightarrow^* \beta$ we have

1. For no terminal a do both α and β derive strings beginning with a . i.e. $FIRST(\alpha) \cap FIRST(\beta) = \Phi$,
2. At most one of α and β can derive the empty string.
3. If $\beta \Rightarrow^* \epsilon$, then α does not derive any string beginning with a terminal in $FOLLOW(A)$.

Likewise, if $\alpha \Rightarrow^* \epsilon$ then β does not derive any string with a terminal in $FOLLOW(A)$. i.e.

if $\alpha \Rightarrow^* \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \Phi$.

EXAMPLE:

examples:- Find whether the given grammar is in $LL(1)$ or not.

i). $S \rightarrow iEtSS' \mid a$
 $S' \rightarrow eS \mid \epsilon$
 $E \rightarrow b$

SOL:-

	FIRST	FOLLOW
S	{i, a}	{e, \$}
S'	{e, ε}	{e, \$}
E	{b}	{t}

i). $S \rightarrow \underbrace{iEtSS'}_{\alpha} \mid \underbrace{a}_{\beta}$

(a) $FIRST(\alpha) = \{i\}$, $FIRST(\beta) = \{a\}$. They are disjoint.

(b) α or β does not derive ϵ .

ii) $S' \rightarrow \underbrace{eS}_{\alpha} \mid \underbrace{\epsilon}_{\beta}$

(a) $FIRST(\alpha) = \{e\}$, $FIRST(\beta) = \{\epsilon\}$. They are disjoint.

(b) $\alpha \neq \epsilon$, $\beta \Rightarrow \epsilon$.

(c) $\beta \Rightarrow \epsilon$. α derives a terminal e in $FOLLOW(S')$.

\therefore condition fails. Hence grammar is not in $LL(1)$.

CONSTRUCTION OF PREDICTIVE PARSING TABLES

For any grammar G, the following algorithm can be used to construct the predictive parsing table.

The algorithm is

Input : Grammar G

Output : Parsing table M Method

1. For each production $A \rightarrow a$ of the grammar, do steps 2 and 3.
2. For each terminal a in $\text{FIRST}(a)$, add $A \rightarrow a$, to $M[A, a]$.
3. If ϵ is in $\text{First}(a)$, add $A \rightarrow a$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$. If ϵ is in $\text{FIRST}(a)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow a$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

The above algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1) (eliminate all left recursion and left factoring).

Example 4.32 : For the grammar given below, Algorithm produces the parsing table in Blanks are error entries; nonblanks indicate a production with which to expand a nonterminal.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$ $F \rightarrow (E) | id$

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

Non-Terminal	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	$F \rightarrow id$			$F \rightarrow (E)$		

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left

recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes.

The following grammar, which abstracts the **dangling-else problem**, is repeated here from Example 4.22:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

NT	FIRST	FOLLOW
S	{i, a}	{e, \$}
S'	{e, ε}	{e, \$}
E	{b}	{t}

The parsing table for this grammar appears in Fig. 4.18.

The entry for $M[S', e]$ contains both $S' \rightarrow eS$ and $S' \rightarrow \epsilon$.

The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (else) is seen.

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Figure 4.18: Parsing table M for Example 4.33

We can resolve this ambiguity by choosing $S' \rightarrow eS$. This choice corresponds to associating an **else** with the closest previous **then**.

Problems with top down parser

The various problems associated with top down parser are:

Ambiguity in the grammar, **Left recursion**, **Non-left factored grammar**, **Backtracking**

Ambiguity in the grammar: A grammar having two or more left most derivations or two or more right most derivations is called ambiguous grammar. For example, the following grammar is ambiguous:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id$$

The ambiguous grammar is not suitable for top-down parser. So, ambiguity has to be eliminated from the grammar.

Left-recursion: A grammar G is said to be left recursive if it has non-terminal A such that there is a derivation of the form:

$A \Rightarrow A\alpha$ (Obtained by applying one or more productions)

where α is string of terminals and non-terminals. That is, whenever the first symbol in a partial derivation is same as the symbol from which this partial derivation is obtained, then the grammar is said to be left-recursive grammar. For example,

consider the following grammar:

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

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid is$$

The above grammar is unambiguous but, it is having left recursion and hence, it is not suitable for top down parser. So, left recursion has to be eliminated

Non-left factored grammar: If A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A. For example, consider the following grammar that recognizes the if-statement:

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S$$

Observe the following points:

- ☐ Both productions starts with keyword if.
- ☐ So, when we get the input “if” from the lexical analyzer, we cannot tell whether to use the first production or to use the second production to expand the non- terminal S.
- ☐ So, we have to transform the grammar so that they do not have any common prefix. That is, left factoring is must for parsing using top-down parser.

A grammar in which two or more productions from every 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.

Backtracking: The backtracking is necessary for top down parser for following reasons:

- 1) During parsing, the productions are applied one by one. But, if two or more alternative productions are there, they are applied in order from left to right one at a time.
- 2) When a particular production applied fails to expand the non-terminal properly, we have to apply the alternate production. Before trying alternate production, it is necessary undo the activities done using the current production. This is possibly only using backtracking.

Even though backtracking parsers are more powerful than predictive parsers, they are also much slower, requiring exponential time in general and therefore, backtracking parsers are not suitable for practical compilers.

Nonrecursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

$$S \xRightarrow[tm]{*} w\alpha$$

The table-driven parser in Fig. 4.19 has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 4.31, and an output stream. The input buffer contains the string to be parsed, followed by the endmarker $\$$. We reuse the symbol $\$$ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $\$$.

The parser is controlled by a program that considers X , the symbol on top of the stack, and a , the current input symbol. If X is a nonterminal, the parser chooses an X -production by consulting entry $M[X, a]$ of the parsing table M . (Additional code could be executed here, for example, code to construct a node in a parse tree.) Otherwise, it checks for a match between the terminal X and current input symbol a .

The behavior of the parser can be described in terms of its *configurations*, which give the stack contents and the remaining input. The next algorithm describes how configurations are manipulated.

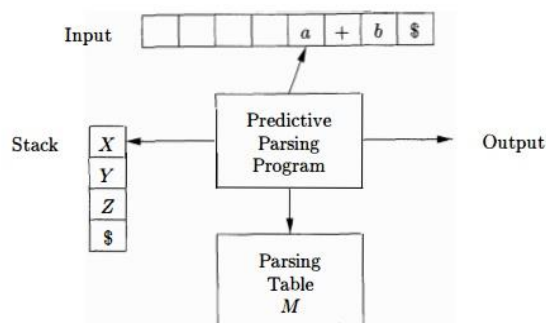


Figure 4.19: Model of a table-driven predictive parser

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.

Algorithm 4.3.4 : Table-driven predictive parsing.

INPUT : A string w and a parsing table M for grammar G .

OUTPUT : If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

1. set ip to point to the first symbol of w ;
2. set X to the top stack symbol;
3. **while** ($X \neq \$$)
 - { /* stack is not empty */
 - 3.1 **if** (X is a) pop the stack and advance ip ;
 - 3.2 **else if** (X is a terminal) error Q ;
 - 3.3 **else if** ($M[X,a]$ is an error entry) error Q ;
 - 3.4 **else if** ($M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$)
 - {
 - output the production $X \rightarrow Y_1 Y_2 \dots Y_k$;
 - pop the stack;
 - push Y_k, Y_{k-1}, \dots, Y_1 onto the stack, with Y_1 on top;
 - }
4. set X to the top stack symbol;
5. }

Example 4.35 : Consider grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$ $F \rightarrow (E) | id$

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

Non-Terminal	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	$F \rightarrow id$			$F \rightarrow (E)$		

On input $\text{id} + \text{id} * \text{id}$,

the nonrecursive predictive parser of Algorithm 4.34 makes the sequence of moves in Fig. 4.21. These moves correspond to a leftmost derivation (see Fig. 4.12 for the full derivation):

$$E \xRightarrow{lm} TE' \xRightarrow{lm} FT'E' \xRightarrow{lm} \text{id} T'E' \xRightarrow{lm} \text{id} E' \xRightarrow{lm} \text{id} + TE' \xRightarrow{lm} \dots$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\text{id} + \text{id} * \text{id}\$$	
	$TE'\$$	$\text{id} + \text{id} * \text{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\text{id} + \text{id} * \text{id}\$$	output $T \rightarrow FT'$
	$\text{id} T'E'\$$	$\text{id} + \text{id} * \text{id}\$$	output $F \rightarrow \text{id}$
id	$T'E'\$$	$+ \text{id} * \text{id}\$$	match id
id	$E'\$$	$+ \text{id} * \text{id}\$$	output $T' \rightarrow \epsilon$
id	$+ TE'\$$	$+ \text{id} * \text{id}\$$	output $E' \rightarrow + TE'$
$\text{id} +$	$TE'\$$	$\text{id} * \text{id}\$$	match $+$
$\text{id} +$	$FT'E'\$$	$\text{id} * \text{id}\$$	output $T \rightarrow FT'$
$\text{id} +$	$\text{id} T'E'\$$	$\text{id} * \text{id}\$$	output $F \rightarrow \text{id}$
$\text{id} + \text{id}$	$T'E'\$$	$* \text{id}\$$	match id
$\text{id} + \text{id}$	$* FT'E'\$$	$* \text{id}\$$	output $T' \rightarrow * FT'$
$\text{id} + \text{id} *$	$FT'E'\$$	$\text{id}\$$	match $*$
$\text{id} + \text{id} *$	$\text{id} T'E'\$$	$\text{id}\$$	output $F \rightarrow \text{id}$
$\text{id} + \text{id} * \text{id}$	$T'E'\$$	$\$$	match id
$\text{id} + \text{id} * \text{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\text{id} + \text{id} * \text{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\text{id} + \text{id} * \text{id}$

Note that the sentential forms in this derivation correspond to the input that has already been matched (in column M A T C H E D) followed by the stack contents. The matched input is shown only to highlight the correspondence. For the same reason, the top of the stack is to the left; when we consider bottom-up parsing, it will be more natural to show the top of the stack to the right. The input pointer points to the leftmost symbol of the string in the INPUT column.

ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry $M[A,a]$ is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token

in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows

- As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A.
- If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
- It is not enough to use FOLLOW(A) as the synchronizing set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions.
- A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.
- Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statement, which appear within blocks, and so on.
- We can add to the synchronizing set of a lower construct the symbols that begin higher constructs.
- For example, we might add keywords that begin statements to the synchronizing sets for the non-terminals generating expressions.
- If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.
- If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Panic Mode Recovery

This involves careful selection of synchronizing tokens for each non terminal.

Some points to follow are,

- Place all symbols in FOLLOW(A) into the synchronizing set of A. In this case parser skips symbols until a symbol from FOLLOW(A) is seen. Then A is popped off the stack and parsing continues
- The symbols that begin the higher constructs must be added to the synchronizing set of the lower constructs
- Add FIRST(A) to the synchronizing set of A, so that parsing can continue when a symbol in FIRST(A) is encountered during skipping of symbols
- If some non-terminal derives ϵ then using it can be used as default
- If a symbol on the top of the stack can't be matched, one method is pop the symbol and issue message saying that the symbol is inserted.

Example:

The Grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)$

$F \rightarrow id$

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

Predictive Parser table after modification for error handling is

Non-Terminal	INPUT SYMBOLS					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	SYNCH	SYNCH
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	SYNCH		$T \rightarrow FT'$	SYNCH	SYNCH
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	SYNCH	SYNCH	$F \rightarrow (E)$	SYNCH	SYNCH

Parsing and Error Recovery moves made by Predictive Parser

STACK	INPUT	REMARK
\$E)id* +id\$	Error, skip)
\$E	id* +id\$	id is in FIRST(E)
\$E'T	id* +id\$	
\$E'T'F	id* +id\$	
\$E'T'id	id* +id\$	
\$E'T'	*+id\$	
\$E'T'F*	*+id\$	
\$E'T'F	+id\$	Error,M[F, +] = synch
\$E'T'	+id\$	F has been popped
\$E'	+id\$	
\$E'T+	+id\$	
\$E'T	id\$	
\$E'T'F	id\$	
\$E'T'id	id\$	
\$E'T'	\$	
\$E'	\$	
\$	\$	

Phrase - level Recovery

Phrase-level error recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack. Alteration of stack symbols or the pushing of new symbols onto the stack is questionable for several reasons. First, the steps carried out by the parser might then not correspond to the derivation of any word in the language at all. Second, we must ensure that there is no possibility of an infinite loop. Checking that any recovery action eventually results in an input symbol being consumed (or the stack being shortened if the end of the input has been reached) is a good way to protect against such loops.

Bottom-Up Parsing

1 Reductions

2 Handle Pruning

3 Shift-Reduce Parsing

4 Conflicts During Shift-Reduce Parsing

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). It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree. The sequence of tree snapshots in Fig. 4.25 illustrates

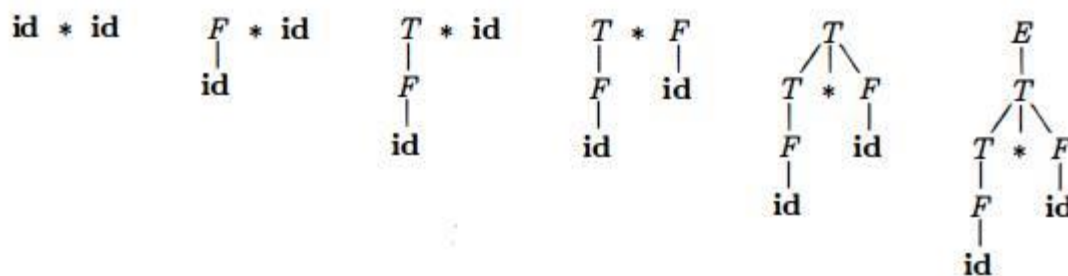


Figure 4.25: A bottom-up parse for $\text{id} * \text{id}$

a bottom-up parse of the token stream $\text{id} * \text{id}$, with respect to the expression grammar

This section introduces a general style of bottom-up parsing known as shift-reduce parsing.

1. 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.

Example 4.37 : The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

$\text{id} * \text{id}$, $F * \text{id}$, $T * \text{id}$, $T * F$, T , E

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string $id * id$. The first reduction produces $F * id$ by reducing the leftmost id to F . Using the production $F \rightarrow id$. The second reduction produces $T * id$ by reducing F to T .

Now, we have a choice between reducing the string T , which is the body of $E \rightarrow T$, and the string consisting of the second id , which is the body of $F \rightarrow id$. Rather than reduce T to E , the second id is reduced to T , resulting in the string $T * F$. This string then reduces to T . The parse completes with the reduction of T to the start symbol E .

By definition, a **reduction is the reverse of a step in a derivation** (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

This derivation is in fact a rightmost derivation.

2. Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. 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.**

For example, adding subscripts to the tokens id for clarity, the handles during the parse of $id_1 * id_2$ according to the expression grammar (4.1) are as in Fig. 4.26. Although T is the body of the production $E \rightarrow T$, the symbol T is not a handle in the sentential form $T * id_2$. If T were indeed replaced by E , we would get the string $E * id_2$, which cannot be derived from the start symbol E . Thus, the leftmost substring that matches the body of some production need not be a handle.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $id_1 * id_2$

Formally, if $S \xRightarrow{*}_{rm} \alpha A w \Rightarrow \alpha \beta w$, as in Fig. 4.27, then production $A \rightarrow \beta$ in the position following α is a *handle* of $\alpha \beta w$. Alternatively, a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ .

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we refer to the body b rather than $A \rightarrow b$ as a handle. Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of abw . If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

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

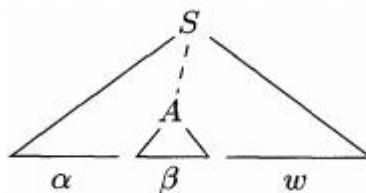


Figure 4.27: A handle $A \rightarrow \beta$ in the parse tree for $\alpha\beta w$

of the grammar at hand, then let $w = \gamma_n$, where γ_n is the n th right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \xRightarrow{rm} \gamma_1 \xRightarrow{rm} \gamma_2 \xRightarrow{rm} \cdots \xRightarrow{rm} \gamma_{n-1} \xRightarrow{rm} \gamma_n = w$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \rightarrow \beta_n$ to obtain the previous right-sentential form γ_{n-1} . Note that we do not yet know how handles are to be found, but we shall see methods of doing so shortly.

We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2} . If by continuing this process we produce a right-sentential form consisting only of the start symbol S , then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

3. Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

We use \$ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.

Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
\$	w \$

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string of grammar symbols on top of the stack. It then reduces to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$\$ S$	\$

Upon entering this configuration, the parser halts and announces successful completion of parsing. Figure 4.28 steps through the actions a shift-reduce parser might take in parsing the input string $id_1 * id_2$ according to the expression grammar (4.1).

STACK	INPUT	ACTION
\$	$id_1 * id_2$ \$	shift
$\$ id_1$	$* id_2$ \$	reduce by $F \rightarrow id$
$\$ F$	$* id_2$ \$	reduce by $T \rightarrow F$
$\$ T$	$* id_2$ \$	shift
$\$ T *$	id_2 \$	shift
$\$ T * id_2$	\$	reduce by $F \rightarrow id$
$\$ T * F$	\$	reduce by $T \rightarrow T * F$
$\$ T$	\$	reduce by $E \rightarrow T$
$\$ E$	\$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $id_1 * id_2$

While the primary operations are shift and reduce, there are actually **four possible actions** a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

- 1. Shift.** Shift the next input symbol onto the top of the stack.
- 2. Reduce.** The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
- 3. Accept.** Announce successful completion of parsing.
- 4. Error.** Discover a syntax error and call an error recovery routine.

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. This fact can be shown by considering the possible forms of two successive steps in any rightmost derivation. Figure 4.29 illustrates the two possible cases. In case (1), A is replaced by $(\beta\gamma)$, and then the rightmost nonterminal B in the body $(\beta\gamma)$ is replaced by γ . In case (2), A is again expanded first, but this time the body is a string y of terminals only. The next rightmost nonterminal B will be somewhere to the left of y .

In other words:

$$\begin{aligned} (1) \quad S &\xRightarrow{rm} \alpha A z \Rightarrow \alpha \beta B \gamma z \Rightarrow \alpha \beta \gamma z \\ (2) \quad S &\xRightarrow{rm} \alpha B x A z \Rightarrow \alpha B x y z \Rightarrow \alpha \gamma x y z \end{aligned}$$

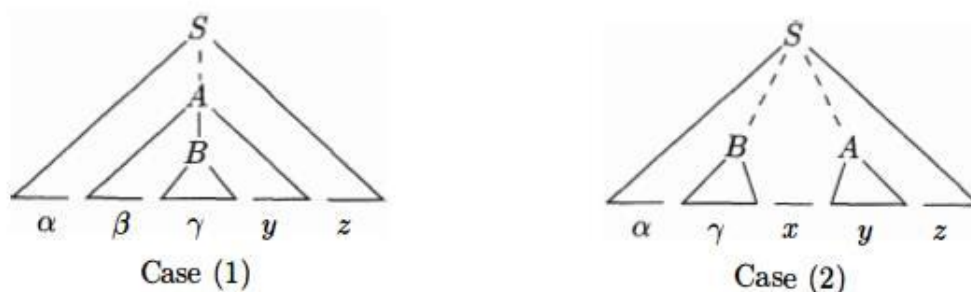


Figure 4.29: Cases for two successive steps of a rightmost derivation

Consider case (1) in reverse, where a shift-reduce parser has just reached the configuration

STACK	INPUT
$\$ \alpha \beta \gamma$	$y z \$$

The parser reduces the handle γ to B to reach the configuration

$\$ \alpha \beta B$	$y z \$$
---------------------	----------

The parser can now shift the string y onto the stack by a sequence of zero or more shift moves to reach the configuration

$\$ \alpha \beta B y$	$z \$$
-----------------------	--------

with the handle γ on top of the stack, and it gets reduced to A . Now consider case (2). In configuration

 $\$ \alpha \gamma$
 $xyz \$$

the handle γ is on top of the stack. After reducing the handle γ to B , the parser can shift the string xy to get the next handle y on top of the stack, ready to be reduced to A :

 $\$ \alpha Bxy$
 $z \$$

In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle.

Find the handles for the given RSF and construct shift-reduce parser :-

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

(a) i/p : $id + id$

RMD.	RSF	Handle.	Action.
$E \Rightarrow E + T$	$id_1 + id_2$	id_1	$F \rightarrow id$
$\Rightarrow E + F$	$F + id_2$	F	$T \rightarrow F$
$\Rightarrow E + id$	$T + id_2$	T	$E \rightarrow T$
$\Rightarrow T + id$	$E + id_2$	id_2	$F \rightarrow id_2$
$\Rightarrow F + id$	$E + F$	F	$F \rightarrow T$
$\Rightarrow id + id$	$E + T$	$E + T$	$E \rightarrow E + T$

Stack.	Input.	Action.
$\$$	$id_1 + id_2 \$$	Shift id_1
$\$ id_1$	$+ id_2 \$$	Reduce $F \rightarrow id$
$\$ F$	$+ id_2 \$$	Reduce $T \rightarrow F$
$\$ T$	$+ id_2 \$$	Reduce $E \rightarrow T$
$\$ E$	$+ id_2 \$$	Shift $+$
$\$ E +$	$id_2 \$$	Shift id_2
$\$ E + id_2$	$\$$	Reduce $F \rightarrow id$
$\$ E + F$	$\$$	Reduce $T \rightarrow F$
$\$ E + T$	$\$$	Reduce $E \rightarrow E + T$
$\$ E$	$\$$	Success.

$s \rightarrow 0s1 \mid 01$		
inp: 000111		
stack	Input	Action.
\$	000111\$	shift 0.
\$0	00111\$	shift 0.
\$00	0111\$	shift 0.
\$000	111\$	shift 1.
\$0001	11\$	reduce $s \rightarrow 01$.
\$00s	11\$	shift 1.
\$00s1	1\$	reduce $s \rightarrow 0s1$.
\$0s	1\$	shift 1.
\$0s1	\$	reduce $s \rightarrow 0s1$.
\$s	\$	success.

4. Conflicts During Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (**a shift/reduce conflict**), or cannot decide which of several reductions to make (**a reduce/reduce conflict**). We now give some examples of syntactic constructs that give rise to such grammars.

Example 4.3.8 : An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

$$\begin{array}{lcl} \text{stmt} & \rightarrow & \text{if expr then stmt} \\ & | & \text{if expr then stmt else stmt} \\ & | & \text{other} \end{array}$$

If we have a shift-reduce parser in configuration

STACK	INPUT
... if expr then stmt	else ... \$

we cannot tell whether if expr **t h e n** 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 **t h e n** stmt to stmt, or it might be correct to shift **else** and then to look for another stmt to complete the alternative if expr **t h e n** stmt else stmt.

Note that shift-reduce parsing can be adapted to parse certain ambiguous grammars, such as the if-then-else grammar above. If we resolve the shift/reduce conflict on else in favor of shifting, the parser will behave as we expect, associating each else with the previous unmatched then. We discuss parsers for such ambiguous grammars in Section 4.8.

Another common setting for conflicts occurs when we know we have a handle, but the stack contents and the next input symbol are insufficient to determine which production should be used in a reduction. The next example illustrates this situation.

Example 4.39 : Suppose we have a lexical analyzer that returns the token name *id* for all names, regardless of their type. Suppose also that our language invokes procedures by giving their names, with parameters surrounded by parentheses, and that arrays are referenced by the same syntax. Since the translation of indices in array references and parameters in procedure calls are different, we want to use different productions to generate lists of actual parameters and indices. Our grammar might therefore have (among others) productions such as those in Fig. 4.30.

A statement beginning with $p(i, j)$ would appear as the token stream *id(id, id)* to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

(1)	<i>stmt</i>	→	id (<i>parameter_list</i>)
(2)	<i>stmt</i>	→	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	→	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	→	<i>parameter</i>
(5)	<i>parameter</i>	→	id
(6)	<i>expr</i>	→	id (<i>expr_list</i>)
(7)	<i>expr</i>	→	id
(8)	<i>expr_list</i>	→	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	→	<i>expr</i>

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.

One solution is to change the token **id** in production (1) to **procid** and to use a more sophisticated lexical analyzer that returns the token name **procid** when it recognizes a lexeme that is the name of a procedure. Doing so would require the lexical analyzer to consult the symbol table before returning a token.

If we made this modification, then on processing $p(i, j)$ the parser would be either in the configuration

STACK	INPUT
... procid (id	, id) ...

or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse.

NOTE: Refer class notes for examples

MODULE-5

Syntax Directed Translation, Intermediate code generation, Code generation Text book 2: Chapter 5.1, 5.2, 5.3, 6.1, 6.2, 8.1, 8.2

SEMANTIC ANALYSIS

Semantic analysis is the third phase of the compiler which acts as an interface between syntax analysis phase and code generation phase. It accepts the parse tree from the syntax analysis phase and adds the semantic information to the parse tree and performs certain checks based on this information. It also helps constructing the symbol table with appropriate information. Some of the actions performed semantic analysis phase are:

- Type checking i.e., number and type of arguments in function call and in function header of function definition must be same. Otherwise, it results in semantic error.
- Object binding i.e., associating variables with respective function definitions
- Automatic type conversion of integers in mixed mode of operations
- Helps intermediate code generation.
- Display appropriate error messages

The semantics of a language can be described very easily using two notations namely:

- Syntax directed definition (SDD)
 - Syntax directed translation (SDT)
1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).
 2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions:

The syntax-directed definition (SDD) is a CFG that includes attributes and rules. In an augmented CFG, the attributes are associated with the grammar symbols (i.e. nodes of the parse tree). And the rules are associated with the productions of grammar.

• **Syntax Directed Definitions** are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of **Attributes**;

For example, a simple SDD for the production $E \rightarrow E_1 + T$ can be written as shown below:

Production

$$E \rightarrow E_1 + T$$
Semantic Rule

$$E.val = E_1.val + T.val$$


Attribute is a property of a programming language construct. Associated with grammar symbols.

Such formalism generates Annotated **Parse-Trees** where each node of the tree is a record with a field for each attribute. If X is a grammar symbol and 'a' is a attribute then $X.a$ denote the value of attribute 'a' at a particular node X in a parse tree.

- Ex 1: If val is the attribute associated with a non-terminal E , then $E.val$ gives the value of attribute val at a node E in the parse tree.
- Ex 2: If lexval is the attribute associated with a terminal digit, then $digit.lexval$ gives the value of attribute lexval at a node digit in the parse tree.
- Ex 3: If syn is the attribute associated with a non-terminal F , then $F.syn$ gives the value of attribute syn at a node F in the parse tree.

Typical examples of attributes are:

- The data types associated with variable such as int, float, char etc
- The value of an expression
- The location of a variable in memory
- The object code of a function or a procedure
- The number of significant digits in a number and so on.

2. Productions are associated with **Semantic Rules** for computing the values of attributes.

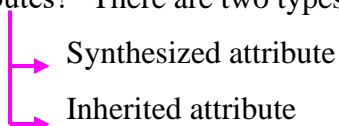
The rule that describe how to compute the attribute values of the attributes associated with a grammar symbol using attribute values of other grammar symbol is called semantic rule.

Example:

$E.val = E_1.val + T.val$ //Semantic rule

where $E.val$ on RHS can be computed using $E_1.val$ and $T.val$ on RHS

The attribute value for a node in the parse tree may depend on information from its children nodes or its sibling nodes or parent nodes. Based on how the attribute values are obtained we can classify the attributes. Now, let us see “What are the different types or classifications of attributes?” There are two types of attributes namely:

- 
- Synthesized attribute
 - Inherited attribute

Synthesized Attributes: The attribute value of a non-terminal A derived from the attribute values of its children or itself is called synthesized attribute. Thus, the attribute values of synthesized attributes are passed up from children to the parent node in bottom-up manner.

For example, consider the production: $E \rightarrow E_1 + T$. Suppose, the attribute value val of E on LHS (head) of the production is obtained by adding the attribute values $E_1.val$ and $T.val$ appearing on the RHS (body) of the production as shown below:

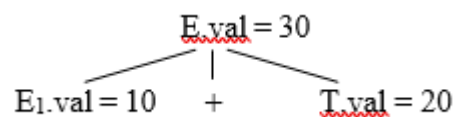
Production

$E \rightarrow E + T$

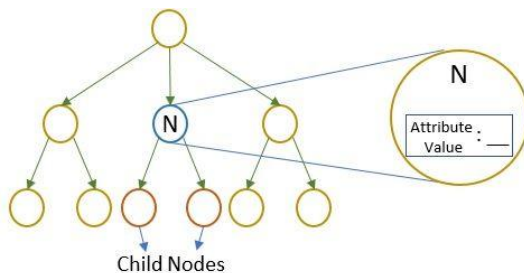
Semantic Rule

$E.val = E_1.val + T.val$

Parse tree with attribute values



Now, attribute val with respect to E appearing on head of the production is called synthesized attribute. This is because, the value of $E.val$ which is 30, is obtained from the children by adding the attribute values 10 and 20 as shown in above parse tree.



SDD with Synthesized Attributes

PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

V.inh =
T.type

Observe the following points from the above parse tree:

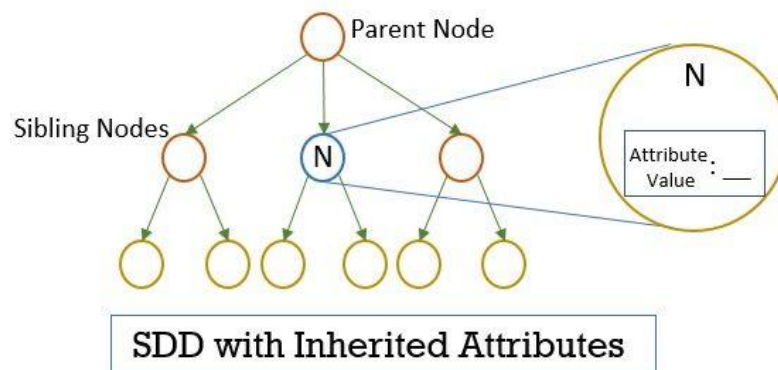
- ◆ The type `int` obtained from the lexical analyzer is already stored in `T.type` whose value is transferred to its sibling `V`. This can be done using:

$$V.inh = T.type$$

Since attribute value for `V` is obtained from its sibling, it is inherited attribute and its attribute is denoted by *inh*.

- ◆ On similar line, the value `int` stored in `V.inh` is transferred to its child `id.entry` and hence entry is inherited attribute of `id` and attribute value is denoted by `id.entry`

Note: With the help of the annotated parse tree, it is very easy for us to construct SDD for a given grammar.



Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for “type declarations”:

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in; \text{ addtype}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{ addtype}(\text{id.entry}, L.in)$

- The non terminal `T` has a synthesized attribute, `type`, determined by the keyword in the declaration.
- The production $D \rightarrow TL$ is associated with the semantic rule $L.in := T.type$ which set the inherited attribute `L.in`.
- Note: The production $L \rightarrow L_1, \text{id}$ distinguishes the two occurrences of `L`.

To evaluate translation rules, identify the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals (because SDT rules don't impose any specific order on evaluation)

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

A parse tree showing the attribute values of each node is called **annotated parse tree**. The terminals in the annotated parse tree can have only synthesized attribute values and they are obtained directly from the lexical analyzer. So, there are no semantic rules in SDD (short form **Syntax Directed Definition**) to get the lexical values into terminals of the annotated parse tree. The other nodes in the annotated parse tree may be either synthesized or inherited attributes. **Note:** Terminals can never have inherited attributes

Consider the SDD

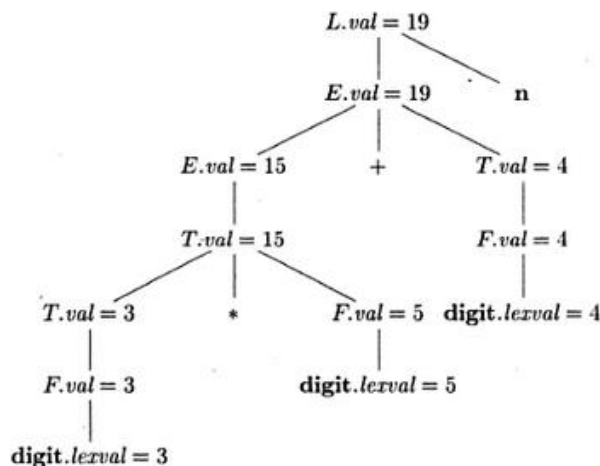
$L \rightarrow En$ where n represent end of file marker

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{digit}$

Here we can see the production rules of grammar along with the semantic actions. And the input string provided by the lexical analyzer is $3 * 5 + 4 n$.



the final SDD along with productions and semantic rules is shown below:

Productions

$L \rightarrow En$
 $E \rightarrow E + T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow F$
 $F \rightarrow (E)$
 $F \rightarrow \text{digit}$

Semantic Rules

$L.val = E.val$
 $E.val = E_1.val + T.val$
 $E.val = T.val$
 $T.val = T_1.val * F.val$
 $T.val = F.val$
 $F.val = E.val$
 $F.val = \text{digit.lexval}$

Dependency Graph

A dependency graph is used to represent the flow of information among the attributes in a parse tree. In a parse tree, a dependency graph basically helps to determine the evaluation order for the attributes. The main aim of the dependency graphs is to help the compiler to check for various types of dependencies between statements in order to prevent them from being executed in the incorrect sequence, i.e. in a way that affects the program's meaning. This is the main aspect that helps in identifying the program's numerous parallelizable components.

Example of Dependency Graph:

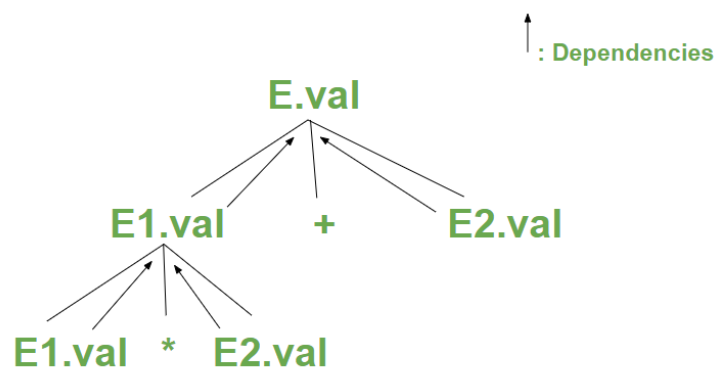
Design dependency graph for the following grammar:

$E \rightarrow E1 + E2$

$E \rightarrow E1 * E2$

PRODUCTIONS	SEMANTIC RULES
$E \rightarrow E1 + E2$ $E \rightarrow E1 * E2$	$E.val \rightarrow E1.val + E2.val$ $E.val \rightarrow E1.val * E2.val$

Required dependency graph for the above grammar is represented as –



Evaluation Orders for SDD

There can be two classes of syntax-directed translations S-attributed translation and L-attributed translation.

6.3 S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

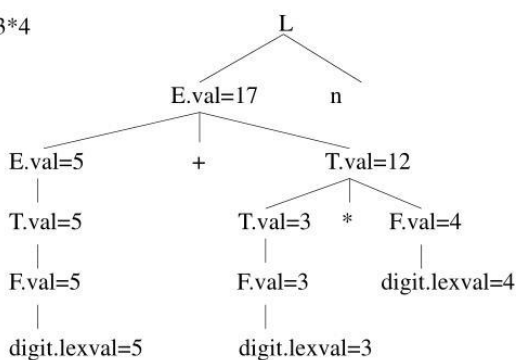
- **Example.** The arithmetic grammar is an example of an S-Attributed Definition.

PRODUCTION	SEMANTIC RULE
$L \rightarrow E n$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

The annotated parse-tree for the input 5+3*4 is:

Annotated Parse Tree -- Example

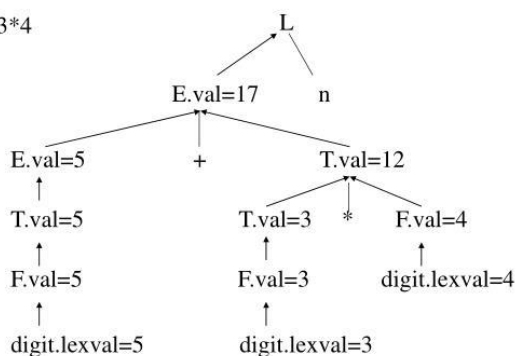
Input: 5+3*4



12

Dependency Graph

Input: 5+3*4



13

6.4 L-attributed definition

Definition: A SDD is *L-attributed* if each inherited attribute of X_i in the RHS of $A \rightarrow X_1 \dots X_n$ depends only on

1. $X_1 \dots X_{i-1}$

2. attributes of X_1, X_2, \dots, X_{i-1} (symbols to the left of X_i in the RHS)

3. inherited attributes of A.

Restrictions for translation schemes:

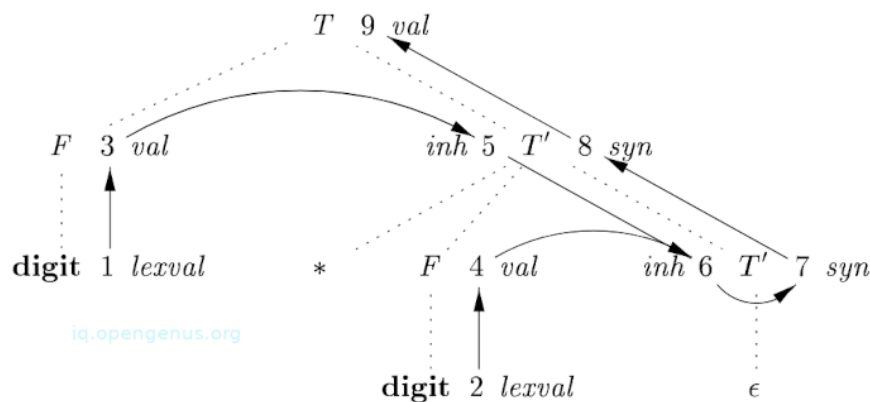
1. Inherited attribute of X_i must be computed by an action before X_i .

2. An action must not refer to synthesized attribute of any symbol to the right of that action.

3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

and the SDD:

PRODUCTION	SEMANTIC RULES
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow * FT_1'$	$T_1'.inh = T'.inh \times F.val, T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$



SDT Example

In this scheme, we place the grammar symbol (node) along with its attribute onto the stack. Thus it becomes easy to retrieve attributes and symbols together when the reduction of the corresponding node occurs. As the stack operates in the first-in-last-out mode.

3. SDT's With Action Inside Productions

In one of the methods, you can even place the semantic action at any position within a production body. The action takes place just after processing all the grammar symbols on the left side of the action.

Such as consider the production $B \rightarrow X \{a\} Y$;

Here we can perform the semantic action 'a' after the processing grammar symbol X (in case X is a terminal).

Or after processing all the symbols derived from X (in case the X is a non-terminal).

4. Eliminating Left Recursion from SDT's

It is not possible to parse the grammar with left recursion in a top-down fashion. Thus, we must eliminate the left recursion from the grammar.

5. SDT's for L-Attributed Definition

To translate the L-attributed SDD into the SDT we have to follow:

1. First, perform the semantic action evaluating the value of the inherited attribute.
2. Secondly, perform the semantic action evaluating the value of the synthesized attribute.

We can relate these steps for the SDT with the L-attributed translation.

Applications of Syntax Directed Translation

- SDT is useful in evaluating an arithmetic expression
- It helps in converting infix to postfix or converting infix to prefix.
- The syntax-directed translation is helpful in converting binary to decimal.
- SDT facilitates help in creating the syntax tree.
- The syntax-directed translation can help in generating the intermediate code and even in type checking.
- SDT stores the type info in the symbol table.

So this is all about syntax-directed translation. We have learnt about its implementation with the help of an example. We have also discussed some SDT schemes and their applications. The SDT also helps in type checking and even in generating intermediate code. The translation technique also helps in implementing little languages for some specialized tasks.