

Automata Theory and Compiler Design

MODULE 3

Context-Free Grammars & Parser

FSMs, regular expressions and regular grammars that offer less power and flexibility than a general purpose programming language provides. Because the frameworks were restrictive, we were not able to describe a large class of useful operations that could be performed on the languages that we defined. Context-free grammars are used to describe such constructs in programming languages.

3.1 Introduction to Rewrite Systems and Grammars

We'll begin with a very general computational model: Define a **rewrite system**, also called **production system** or a **rule-based system**) to be a list of rules and an algorithm for applying them. Each rule has a left-hand side and a right-hand side.

For example, the following could be rewrite-system rules: $S \rightarrow aSb$, $aS \rightarrow \epsilon$, $aSb \rightarrow bSabSa$.

When a rewrite system R is invoked on some initial string w, it operates as follows:

1. Set working-string = w.
2. Until told by R to halt do:
 - 2.1 Match the LHS of some rule against some part of working-string.
 - 2.2 Replace the matched part of working-string with the RHS of the rule that was matched.
3. Return working-string. A rewrite system that is used to define a language is called a grammar.

If G is a grammar, let $L(G)$ be the language that G generates. Like every rewrite system, every grammar contains a list (almost always treated as a set, i.e., as an unordered list) of rules. Also, like every rewrite system, every grammar works with an alphabet, which we can call V. In the case of grammars, we will divide V into two subsets:

- **a terminal alphabet** generally called Σ , which contains the symbols that make up the strings in $L(G)$,
- and • **a nonterminal alphabet**, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string. One final thing is required to specify a grammar. Each grammar has a unique **start symbol**, often called S.

To generate strings in $L(G)$, we invoke simple-rewrite (G, S). Simple-rewrite will begin with S and will apply the rules of G, which can be thought of (given the control algorithm we just described) as licenses to replace one string by another. At each step of one of its derivations, some rule whose left -hand

side matches somewhere in working-string is selected. The substring that matched is replaced by the rule's right hand side, generating a new value for working string.

We will use the symbol \Rightarrow to indicate steps in a derivation. So, for example suppose that G has the start symbol S and the rules $S \Rightarrow aSb \mid bSa \mid \epsilon$.

Then a derivation could begin with: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \dots\dots\dots$

At each step, it is possible that more than one rule's left-hand side matches the working string. It is also possible that a rule's left-hand side matches the working string in more than one way. In either case, there is a derivation corresponding to each alternative. It is precisely the existence of these choices that enables a grammar to generate more than one string

Continuing with our example, there are three choices at the next step:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$ (using the first rule),
 $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb$ (using the second rule).
 $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$ (using the third rule).

The derivation process may end whenever one of the following things happens:

1. The working string no longer contains any non terminal symbols (including, as a special case. when the working string is ϵ), or
2. There are nonterminal symbols in the working string but there is no match with the left-hand side of any rule in the grammar. For example, if the working string were $AaBb$, this would happen if the only left-hand side were C .

3.2 Context-Free Grammars and Language

A **Context Free Grammar** (or **CFG**) to be a grammar in which each rule must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side with any (possibly empty) sequence of symbols.

To simplify the discussion that follows, define an A rule, for any nonterminal symbol A , to be a rule whose left hand side is A . A derivation will halt whenever no rule's left-hand side matches against working-string. At every step, any rule that matches may be chosen.

Context-free grammar rules may have any (possibly empty) sequence of symbols on the right-band side. Because the rule format is more flexible than it is for regular grammars. The rules are more powerful. We will show some examples of languages that can be generated with context-free grammars but that cannot be generated with regular ones.

Ex: All of the following are allowable context-free grammar rules:

$S \rightarrow aSb$
 $S \rightarrow \epsilon$
 $T \rightarrow T$
 $S \rightarrow aSbbTT$

The following are not allowable context-free grammar rules:

$$ST \rightarrow aSb$$
$$a \rightarrow aSb$$
$$\epsilon \rightarrow a$$

The name for these grammars, "context-free," because, using these rules, the decision to replace a nonterminal by some other sequence is made without looking at the context in which the non terminal occurs.

Formal Definition of CFG

A context-free grammar G is a quadruple (V, Σ, R, S) , where:

- V is the rule alphabet, which contains **Nonterminals (or Variables)**, symbols that are used in the grammar but that do not appear in strings in the language and **Terminals**, which make up the strings, belong to the language $L(G)$.

- Σ (the set of terminals) is a subset of V

- R (the set of rules) is a finite subset of $(V - \Sigma) \rightarrow V^*$ and

- S (the start symbol) can be any element of $V - \Sigma$

Given a grammar G , define $x \Rightarrow_G y$ to be the binary relation **derives-in-one-step**, defined so that:

$\forall x, y \in V^* (x \Rightarrow_G y \text{ iff } x = \alpha A \beta, y = \alpha \gamma \beta \text{ and there exists a rule } A \rightarrow \gamma \text{ in } R_G).$

Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_n$, is called a **derivation in G** . Let \Rightarrow_G^* be

the reflexive, transitive closure of \Rightarrow_G . We'll call \Rightarrow_G^* the derives relation.

The language generated by G , denoted $L(G)$ is $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$. In other words, the language generated by G is the set of all strings of terminals that can be derived from S using zero or more applications of rules in G .

A **language L is context-free** iff it is generated by some context-free grammar G . The context-free languages (CFLs) are a **proper superset of the regular languages**.

Ex : The Balanced Parentheses Language

Consider $Bal = \{w \in \{\}, \{\}^* : \text{the parentheses are balanced}\}$. It is context-free because it can be generated by the grammar $G = (\{S, \}, \{\}, \{ \}, \{ \}, R, S)$, where: $R = \{$

$$S \rightarrow (S)$$
$$S \rightarrow SS$$
$$S \rightarrow \epsilon$$
$$\}$$

Derivations of $w = (() ())$ in G :

$S \Rightarrow (S) \Rightarrow ()$.

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()())$.

So, $S \Rightarrow^* ()$ and $S \Rightarrow^* (()())$.

Ex : Consider $A^n B^n = \{ a^n b^n, n \geq 0 \}$.

It is context-free because it can be generated by the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$R = \{ S \rightarrow aSb, S \rightarrow \varepsilon \}$.

Following two features of context-free grammars gives them the power to define languages like balanced parentheses Bal and $A^n B^n$:

- **Recursive Grammar** (consisting of recursive rules) makes it possible for a finite grammar to generate an infinite set of strings. **A grammar is recursive iff it contains at least one recursive rule.**

For example, the grammar we just presented for Bal is recursive because it contains the rule $S \rightarrow (S)$. The grammar we presented for $A^n B^n$ is recursive because it contains the rule $S \rightarrow aSb$. A grammar that contained the rule $S \rightarrow aS$ would also be recursive. So the regular grammar whose rules are $\{ S \rightarrow aT, T \rightarrow aW, W \rightarrow aS, W \rightarrow a \}$ is recursive.

- **A self-embedding CFG** : A rule in a grammar G is self-embedding iff it is of the form $X \rightarrow W_1 Y W_2$, where $Y \Rightarrow_G^* w_3 X w_4$ and both $w_1 w_3$ and $w_2 w_4$ are in Σ^+ .

A grammar is self-embedding iff it contains at least one self-embedding rule. The grammar presented for $A^n B^n$ is self-embedding because it contains the rule $S \rightarrow aSb$

Ex: Even Length Palindromes

Consider $\text{PalEven} = \{ ww^R : w \in \{a, b\}^* \}$, the language of even-length palindromes of a's and b's. It can be generated by the grammar $G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$R = \{$
 $S \rightarrow aSa$
 $S \rightarrow bSb$
 $S \rightarrow \varepsilon$
 $\}$

Ex: Equal Numbers of a's and b's

Let $L = \{ w \in \{a, b\}^* : \#a(w) = \#b(w) \}$. It can be generated by the grammar

$G = (\{S, a, b\}, \{a, b\}, R, S)$, where:

$R = \{$
 $S \rightarrow aSb$
 $S \rightarrow bSa$
 $S \rightarrow SS$
 $S \rightarrow \varepsilon$
 $\}$

This grammar can also be written as $S \rightarrow a S b \mid b S a \mid SS \mid \epsilon$

BNF (or Backus Naur form) is a widely used grammatical formalism that exploits both of these extensions. The following BNF style grammar describes a highly simplified and very small subset of Java:

```
<block> ::= { <stmt-list> } | { }
<stmt-list> ::= <stmt> | <stmt-list> <stmt>
<Stmt> ::= <block> | while (<cond>) <stmt> |
if (<cond>) <stmt> |
do <stmt> while (<cond>); | <assignment-stmt>; |
return | return <expression> |
<method-invocation>;
```

3.3 Designing Context-Free Grammars

Ex 3.5 Concatenating Independent Sublanguages

Let $L = \{ a^n b^n c^m : n, m \geq 0 \}$. Here, the c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together.

So let $G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S)$, where:

$$R = \{$$

$$\begin{aligned} S &\rightarrow NC \\ N &\rightarrow aNb \\ N &\rightarrow \epsilon \\ C &\rightarrow cC \\ C &\rightarrow \epsilon \end{aligned}$$

$$\}$$

Ex 3.6 The Kleene Star of a Language

Let $L = \{ a^{n_1} b^{n_1} a^{n_2} b^{n_2} a^{n_3} b^{n_3} \dots a^{n_k} b^{n_k} : \forall i (n_i \geq 0) \}$

For example, the following strings are in L : abab, aabbaaabbabab

We know how to produce individual elements of $\{ a^n b^n : n \geq 0 \}$, and we know how to concatenate regions together. So a solution is $G = (\{S, M, a, b\}, \{a, b\}, R, S)$ where:

$$R = \{$$

$$\begin{aligned} S &\rightarrow MS && \backslash * \text{ Each } M \text{ will generate one } \{ a_n b_n : n \geq 0 \} \\ S &\rightarrow \epsilon \\ M &\rightarrow aMb && \backslash * \text{ Generate one region.} \\ M &\rightarrow \epsilon \end{aligned}$$

$$\}$$

Every other node is labeled with some element of $V - \Sigma$, and

- If m is a nonleaf node labeled X and the children of m are labelled x_1, x_2, \dots, x_n , then R contains the rule $X \rightarrow x_1, x_2, \dots, x_n$.

Define the **branching factor of a grammar G** to be length (the number of symbols) of the longest right-hand side of any rule in G . Then the branching factor of any parse tree generated by G is less than or equal to the branching factor of G .

Terminologies:

- G 's **weak generative capacity**, defined to be the set of strings, $L(G)$, that G generates.
- G 's **strong generative capacity**, defined to be the set of parse trees that G generates.
- A **left-most derivation** is a type of derivation in which, at each step, the leftmost non terminal in the working string is chosen for expansion.
- A **right-most derivation** is one in which, at each step, the rightmost non terminal in the working string is chosen for expansion.

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:

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

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

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

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 → E + E
E → E * E
E → id
```

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

The left-most derivation is:

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

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

The right-most derivation is:

```
E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id
```

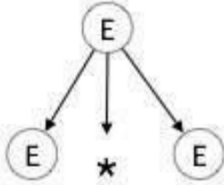
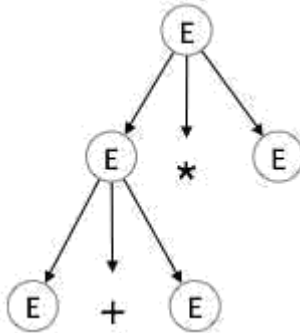
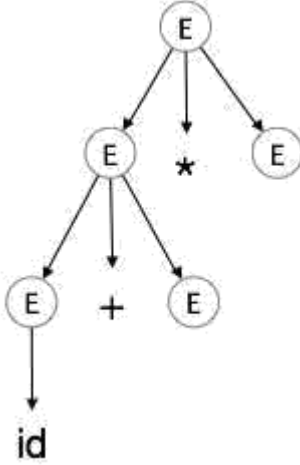
Parse Tree

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.

45

The left-most derivation is:

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

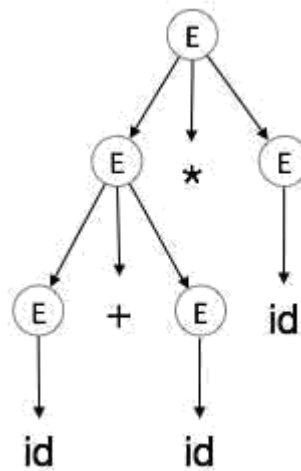
Step 1	$E \rightarrow E * E$	
Step 2:	$E \rightarrow E + E * E$	
Step 3:	$E \rightarrow id + E * E$	

Step 4:

$$E \rightarrow \text{id} + \text{id} * E$$

Step 5:

$$E \rightarrow \text{id} + \text{id} * \text{id}$$



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 may produce more than one parse tree for some (or all) of the strings it generates.

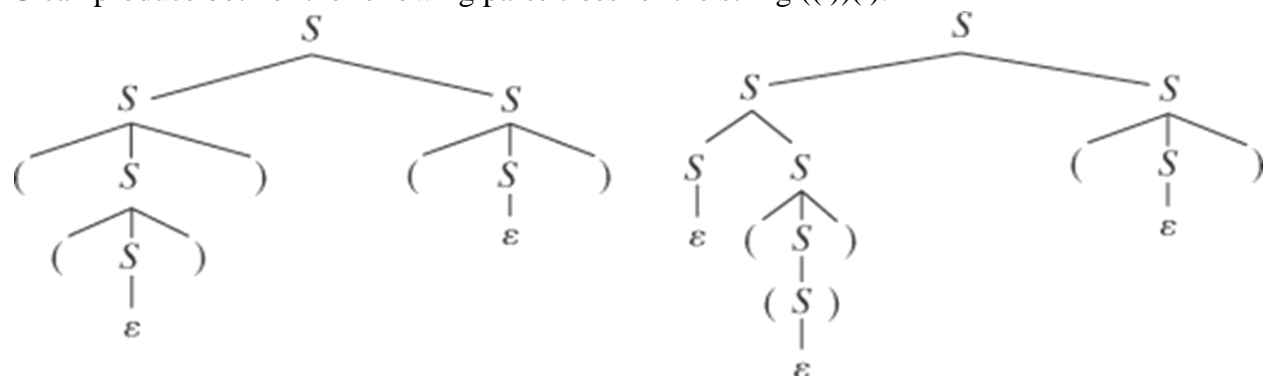
Then we say that the grammar is **ambiguous**. More precisely, a grammar G is ambiguous iff there is at least one string in $L(G)$ for which G produces more than one parse tree. It is easy to write ambiguous grammars if we are not careful.

Ex : The Balanced Parentheses Grammar is Ambiguous

Consider $Bal = \{w \in \{(), ()^*\} : \text{the parentheses are balanced}\}$ having the grammar

$$G = (\{S, \}, \{ \}, \{ \}, \{ \}, R, S), \text{ where:}$$
$$R = \{ S \rightarrow (S) \}$$
$$S \rightarrow SS$$
$$S \rightarrow \varepsilon$$
$$\}.$$

G can produce both of the following parse trees for the string $(())()$:



In fact, G can produce an infinite number of parse trees for the string $(())()$.

A grammar G is unambiguous iff, for all strings w , at every point in a leftmost or rightmost derivation of w , only one rule in G can be applied.

The two leftmost derivations of the string $(())()$ are:

- $S \Rightarrow SS \Rightarrow$
 $(S)S \Rightarrow$
 $((S))S \Rightarrow$
 $(())S \Rightarrow$
 $(())(S)$
 $\Rightarrow (())()$.

- $S \Rightarrow SS$
 $\Rightarrow SSS$
 $\Rightarrow SS$
 $\Rightarrow (S)S$
 $\Rightarrow ((S))S$
 $\Rightarrow (())S$
 $\Rightarrow (())(S)$
 $\Rightarrow (())()$.

Ex : An Ambiguous Expression Grammar

Consider E_{expr} which we'll define to be the language of simple arithmetic expressions. We can define E_{expr} with the following context-free grammar $G = (V, \Sigma, R, S)$ Where $V = \{E, \text{id}, +, *, (,)\}$, and

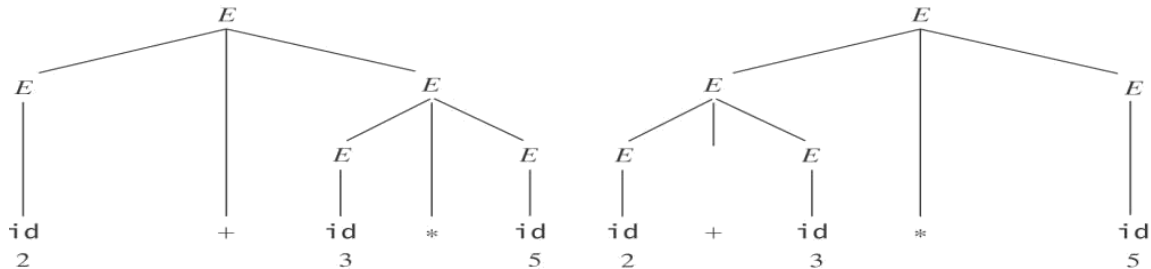
$E \rightarrow \text{id}$

}

$R = \{$
 $E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow \text{id}$
 $\}$

The terminal symbol *id* is used as shorthand for any of the numbers or variables that can actually occur as the operands in the expressions that *G* generates.

Consider the string $2 + 3 * 5$, which we will write as $id + id * id$. Using *G*, We can get two parses for this string:



There is an ambiguity in this grammar since we are able to obtain multiple parse trees for the same expression. The question here is the evaluation of this expression return 17(if multiplication done first) or 25(if addition done first).

3.8 Inherent Ambiguity

In many cases, an ambiguous grammar *G* can be converted into a new grammar *G'* that generates *L* (*G*) and that has less (or no) ambiguity. Unfortunately, it is not always possible to do this. There exist context-free languages for which no unambiguous grammar exists. We call such languages inherently ambiguous.

Ex: An Inherently Ambiguous language

Let $L = \{ a^i b^j c^k : i, j, k \geq 0, i = j \text{ or } j = k \}$.

An alternative way to describe it is $\{ a^n b^n c^m : n, m \geq 0 \} \cup \{ a^n b^m c^m : n, m \geq 0 \}$.

Every string in *L* has either (or both) the same number of a's and b's or the same number of b's and c's. *L* is inherently ambiguous. One grammar that describes it is $G = (\{ S, S1, S2, A, B, a, b, c \}, \{ a, b, c \}, R, S)$, where:

$$R = \{ S \rightarrow S1 \mid S2$$

$$S1 \rightarrow S1c \mid A$$

$$A \rightarrow aAb \mid \varepsilon \quad S2 \rightarrow S2a \mid S2b \mid S2c \mid B, B \rightarrow Bb \mid Bc \mid \varepsilon \}.$$

/* Generate all strings in $\{ a^n b^n c^m : n, m \geq 0 \}$.

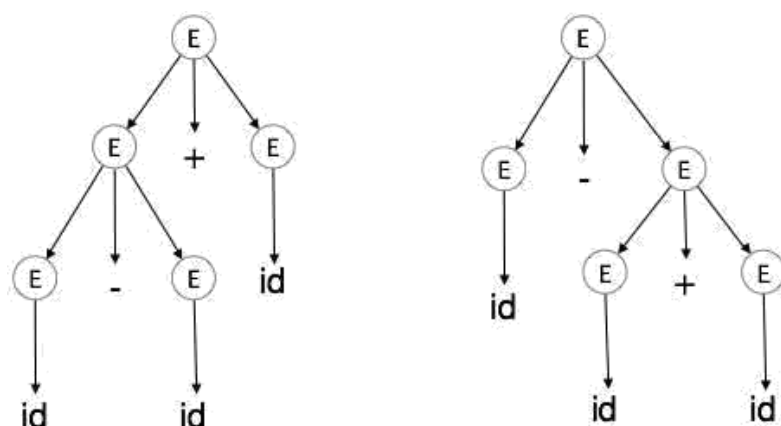
/* Generate all strings in $\{ a^n b^m c^m : n, m \geq 0 \}$.

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:

$id \text{ op } id \text{ op } id$

it will be evaluated as:

$(id \text{ op } id) \text{ op } id$

For example, $(id + id) + id$

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

$id \text{ op } (id \text{ op } id)$

For example, $id \wedge (id \wedge id)$

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.

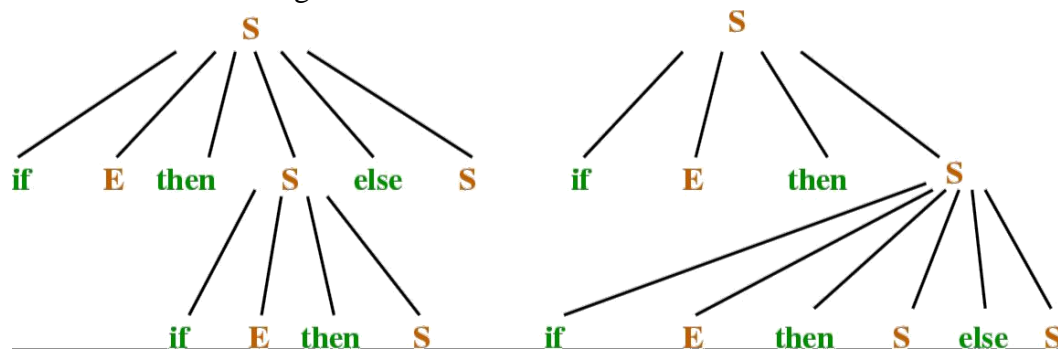
Consider $VN = \{S, E\}$, $VT = \{if, then, else\}$ and a grammar $G = (VT, VN, S, P)$ such that the following

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

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 *matched-stmt* / *unmatched-stmt*

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

matched-stmt *non-alternative-stmt*

unmatched-stmt ***if*** *expr* ***then*** *stmt*

unmatched-stmt ***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:

$E \rightarrow E \pm T \mid E - T \mid T$

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

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

$F \rightarrow (E) \mid id$

4.3.2. 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:

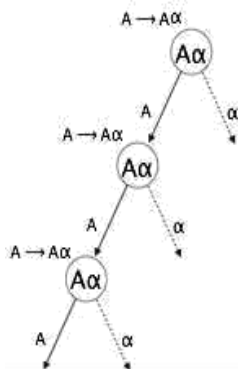
(1) $A \Rightarrow A\alpha \mid \beta$

(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

$$A \Rightarrow \beta A'$$

$$A' \Rightarrow \alpha A' \mid \varepsilon$$

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 $A_i \Rightarrow A_j$

 with $A_i \Rightarrow \delta_1 \mid \delta_2 \mid \delta_3 \mid \dots$

 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.

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

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

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

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

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

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

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

Refer class notes for examples

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

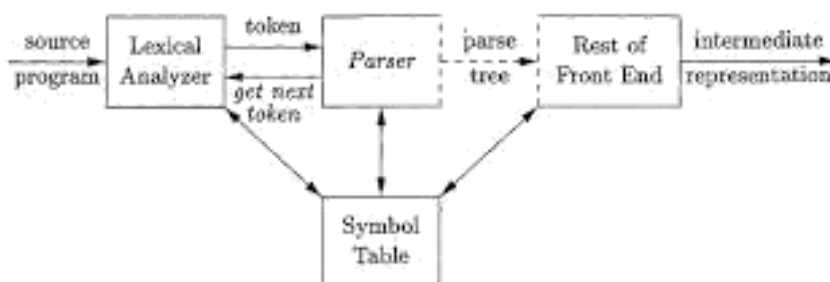
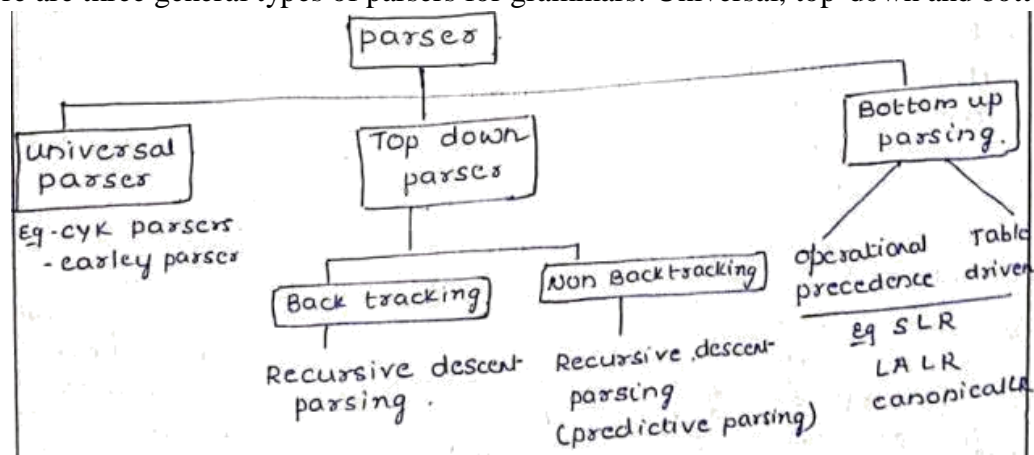


Figure 4.1: Position of parser in compiler model

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 it 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.

4.3. 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

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

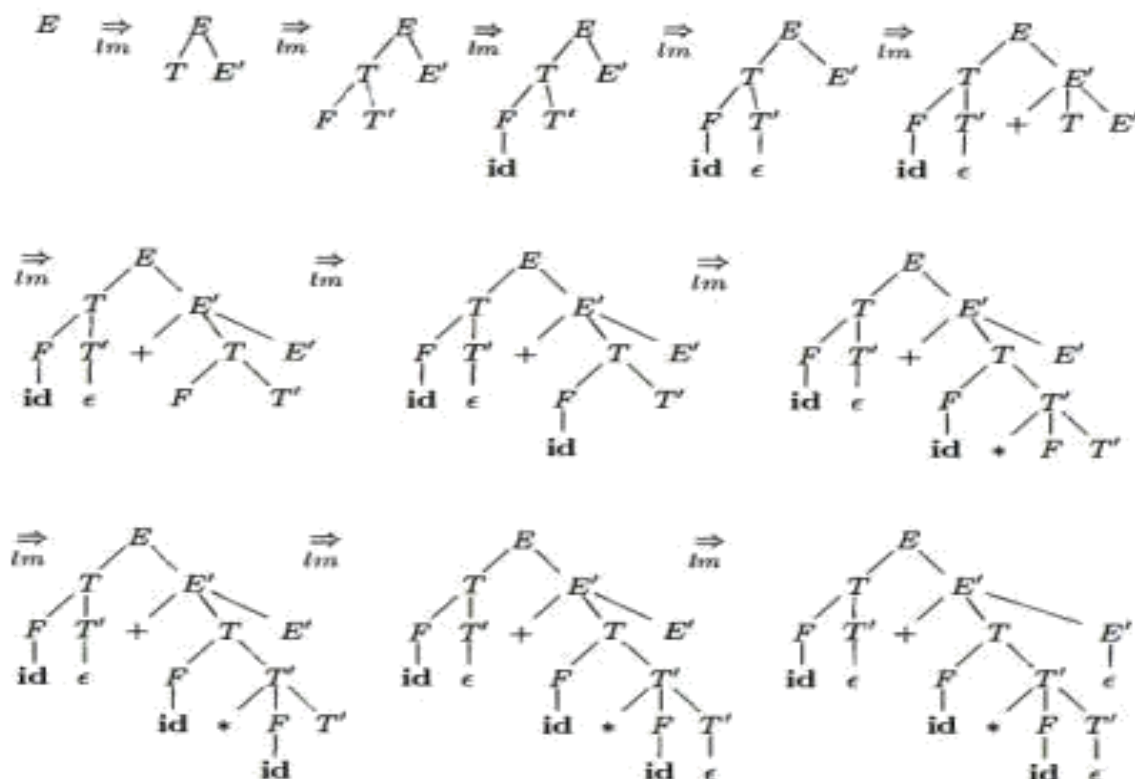


Figure 4.12: Top-down parse for `id + id * id`

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 , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form. A syntax of the form defines zero or one occurrence of the form.

A syntax of the form defines zero or more occurrences of the form .

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 en
```

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 ip symbol a)
6.         advance the ip 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

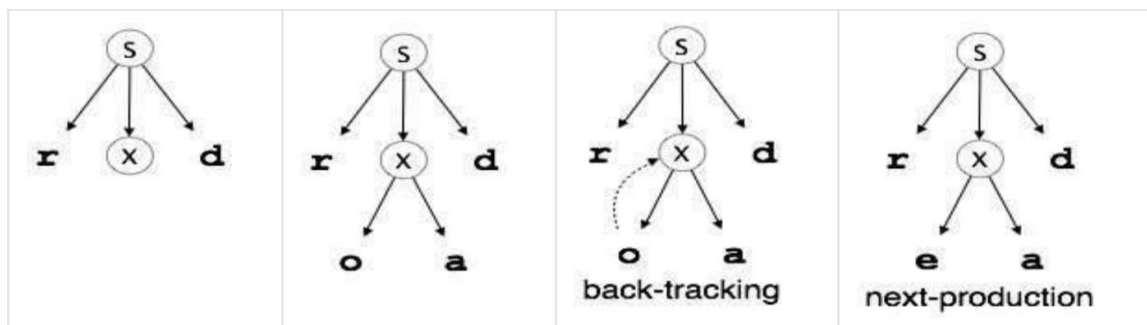
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

$S \rightarrow rXd \mid rZd$
 $X \rightarrow oa \mid ea$
 $Z \rightarrow ai$

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.

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$T \rightarrow FT'$

$T' \rightarrow * FT'$

$F \rightarrow (E) \mid id$

Solution

Procedure E ()

```
{
    T ( );
    E' ( );
}
```

Procedure E' ()

```
{
    If input symbol '='+' then
        advance ( );
    T ( );
    E' ( );
}
```

Procedure T ()

```
{
    F ( );
    T' ( );
}
```

Procedure T' ()

```
{
    If input symbol '='*' then
        advance ( );
    F ( );
    T' ( );
}
```

$T \rightarrow FT'$

$T' \rightarrow * FT'$

```

Procedure F ( )
{
    If input symbol ='id' then
        advance ( );
    else if input-symbol ='(' then
        advance ( );
        E ( );
    If input-symbol = ')'
        advance ( );
    else error ( );
    else error ( );
}

```

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

4.3.4. FIRST AND FOLLOW

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or e 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-

$E \rightarrow TE' E' \rightarrow$

$+TE'|e T \rightarrow FT'$

$T' \rightarrow *FT'|e F \rightarrow$

$(E)|id$

	E	E'	T	T'	F
FIRST	{(,id}	{+,e}	{(,id}	{*,e}	{(,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 if only if for every nonterminal A and every strings of symbols α and β 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 $\beta \xRightarrow{*} \epsilon$, then α does not derive any string beginning with a terminal in $FOLLOW(A)$.

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

if $\alpha \xRightarrow{*} \epsilon$ 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 e is in $\text{First}(a)$, add $A \rightarrow a$ to $M[A, b]$ for each terminal b in $\text{FOLLOW}(A)$. If e 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 $\text{LL}(1)$. It can be shown that the above algorithm can be used to produce for every $\text{LL}(1)$ grammar G a parsing table M that parses all and only the sentences of G . $\text{LL}(1)$ grammars have several distinctive properties. No ambiguous or left recursive grammar can be $\text{LL}(1)$ (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.

$E \rightarrow TE'$

$E' \rightarrow +TE' | e$

$T \rightarrow FT'$

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

	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 α 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.

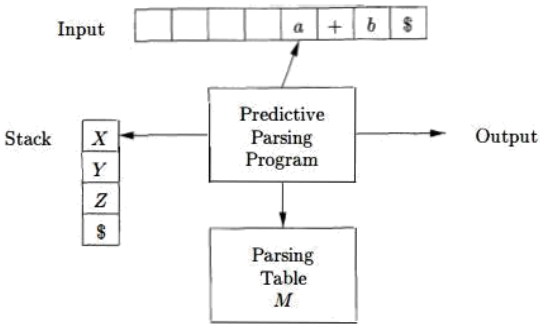


Figure 4.19: Model of a table-driven predictive parser

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;

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 nonterminals 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:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)$

$F \rightarrow \text{id}$

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

Non-Terminal	INPUT SYMBOLS					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	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	$F \rightarrow \text{id}$	SYNCH	SYNCH	$F \rightarrow (E)$	SYNCH	SYNCH

Predictive Parser table after modification for error handling is

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.

