Unit – 2

Chapter 4

Syntax Analysis – 1

4.1 Introduction

Every programming language has rules that prescribe the syntactic structure of programs. The syntax of programming language can be described by context-free grammars. A grammar gives precise, easy-to-understand, syntactic specification of a programming language. We can automatically construct an efficient parser and also determine undetected errors, syntactic ambiguities in the initial design phase. A grammar gives structure of a programming language which is useful in translation of source program to object code and for the detection of errors.

The role of the parser

The input for the parser is a stream of tokens from lexical analysis and output is a parse tree, in which tokens are grouped hierarchically with collective meaning. It should report: any syntax errors, recover from commonly occurring errors, collecting information about various tokens, performing type checking, generating intermediate code, etc.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars; for example, the predictive-parsing approach works for LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

Types of parsers for grammar are: top-down: build parse tree from root to the leaves and bottom-up: build parse tree from leaves to the root. Input the parser is scanned from left to right, one symbol at a time.

Syntax Error Handling

Common programming errors can occur at many different levels.

• *Lexical* errors include misspellings of identifiers, keywords, or operators-e.g., the use of an identifier *elipseSize* instead of *ellipseSize* and missing quotes around text intended as a string. Misspelling an identifier, keyword, and operator are basically addressed.

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

• *Semantic* errors include type mismatches between operators and operands. An example is a r e t un statement in a Java method with result type void. Operator on an incompatible operand is an example.

• *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent. Infinite recursive calls are considered as logical errors.

Usually, error detection and recovery is centered on the syntax analysis phase because of two reasons: Many errors are syntactic in nature and many tokens may disobey the grammatical rules. They can detect the presence of syntactic errors in programs very efficiently. Therefore, parser should report the presence of errors clearly and accurately. Recover from errors quickly so that it is able to detect subsequent errors. It should not slow down the processing of correct programs. Several parsing methods, LL and LR, detect an error as soon as possible. They have viable-prefix property, by which they detect error as soon as they see a prefix of the input that is not a prefix of any string in the language.

The error handler in a parser has goals that are simple to state but challenging to realize:
• Report the presence of errors clearly and accurately.
• Recover from each error quickly enough to detect subsequent errors.
• Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies

The simplest approach is for the parser to quit with an informative error message when it detects the first error. Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information. If errors increase, it is better for the compiler to give up after exceeding some error limit.

Panic-Mode Recovery

With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of *synchronizing tokens* is found. The synchronizing tokens are usually delimiters, such as semicolon or}, whose role in the source program is clear and unambiguous. The compiler designer must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an infinite loop.

Phrase-Level Recovery

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer. Of course, we must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol. Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Productions

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

Global Correction

Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string *x* and grammar *G,* these algorithms will find a parse tree for a related string *y,* such that the number of insertions, deletions, and changes of tokens required to transform *x* into *y* is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest. Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.

4.2 Context-free grammar

Consider a conditional statement
If S1 and S2 are statements and E is an expression, then

"if E then S1 else S2"

We know that, regular expression can specify the lexical structure of tokens. Using some syntactic variable, stmt, we can specify grammar production

$$Stmt \rightarrow if\ expr\ then\ stmt\ else\ stmt$$

A context-free grammar consists of terminals, nonterminals, start symbol and set of productions.

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. The terminals are the keywords if and else and the symbols "(" and ") ."

2. *Nonterminals* are syntactic variables that denote sets of strings. *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. In a grammar, one nonterminal is distinguished as the *start symbol,* and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each *production* consists of:
   (a) A nonterminal called the *head* or *left side* of the production; this production defines some of the strings denoted by the head.
   (b) The symbol : = has been used in place of →

(c) A *body* or *right side* consisting of zero or more terminals and nonterminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Consider the grammar for simple arithmetic expressions as follows:

expr → expr + term | expr-term | term
term → term * factor | term / factor | factor
factor →(expr) | id
E →E + T | E – T | T
T →T * F | T / F | F
F →(E) | id

Notational conventions

1. These symbols are terminals:
   (a) Lowercase letters, early in the alphabet, such as *a, b, c.*

   (b) Operator symbols such as +, *, and so on.

   (c) Punctuation symbols such as parentheses, comma, and so on.

   (d) The digits 0 , 1 , . . . ,9.

   (e) Boldface strings such as id or if, each of which represents a single terminal symbol.

2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as *A, B, C.*

   (b) The letter *S,* which, when it appears, is usually the start symbol.

   (c) Lowercase, italic names such as *expr* or *stmt.*

   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by *E, T,* and *F,* respectively.

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

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

5. Lowercase Greek letters represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as *A → a,* where *A* is the head and *a* is the body.

6. A set of productions with a common head *A* may be written with the *alternatives* for A.

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

Now, consider an example:

E →E + T | E – T | T

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

Derivations

The objective is that a production is treated as a rewriting rule in which the nonterminal on the left is replaced by the string on the right side of the production. The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree, but the precision afforded by derivations will be especially helpful when bottom-up parsing is discussed.

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$ is a grammar

$E \rightarrow -E$ means, we can replace E by –E and represented by $E \Rightarrow -E$ means E derives –E

$E*E \Rightarrow (E)*E \mid E*(E)$

We can apply productions in any order

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$ derivation of –(id) from E
$\alpha A\beta \Rightarrow \alpha\delta\beta$ if $A \rightarrow \delta$ is a production

$\alpha 1 \Rightarrow \alpha 2 \Rightarrow \ldots \Rightarrow \alpha n$ we say $\alpha 1 \Rightarrow \alpha n$

$\Rightarrow$ derives in one step

$\overset{*}{\Rightarrow}$ derives in zero or more steps

$\overset{+}{\Rightarrow}$ derives in one or more steps

Eg: $\alpha \overset{*}{\Rightarrow} \alpha$ for any string α

If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \delta$ then $\alpha \Rightarrow \delta$

Grammar G with start symbol S
L(G) language generated by G
A string of terminals 'w' is in L(G) if and only if
$S \overset{+}{\Rightarrow} w$. string w is called a sentence of G
L(G) is context-free language

If two grammars generate the same language, the grammars are said to be equivalent

Sentential form of G

At each step in a derivation, there are two choices to be made. If $\alpha \Rightarrow \beta$ by a step in which left most non terminal in α is replaced, we write

$$\alpha \Rightarrow \beta$$

lm
so we rewrite left sentential form of the grammar.
Similarly, right most derivations/canonical derivations also exist.

Parse trees and Derivations

A Parse tree is a graphical representation for a derivation. A parse tree ignores variations in the order in which symbols in sentential forms are replaced. '*' will have higher priority than '+' in the sentence id+id*id

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. For certain types of parsers, it is desirable that the grammar be made unambiguous. For instance, two parse trees exist for id+id*id

Verifying the Language Generated by a Grammar

Although compiler designers rarely do so for a complete programming-language grammar, it is useful to be able to reason that a given set of productions generates a particular language. Troublesome constructs can be studied by writing a concise, abstract grammar and studying the language that it generates. We shall construct such a grammar for conditional statements below. A proof that a grammar *G* generates a language *L* has two parts: show that every string generated by *G* is in *L,* and conversely that every string in *L* can indeed be generated by *G.*

Lexical Versus Syntactic Analysis

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

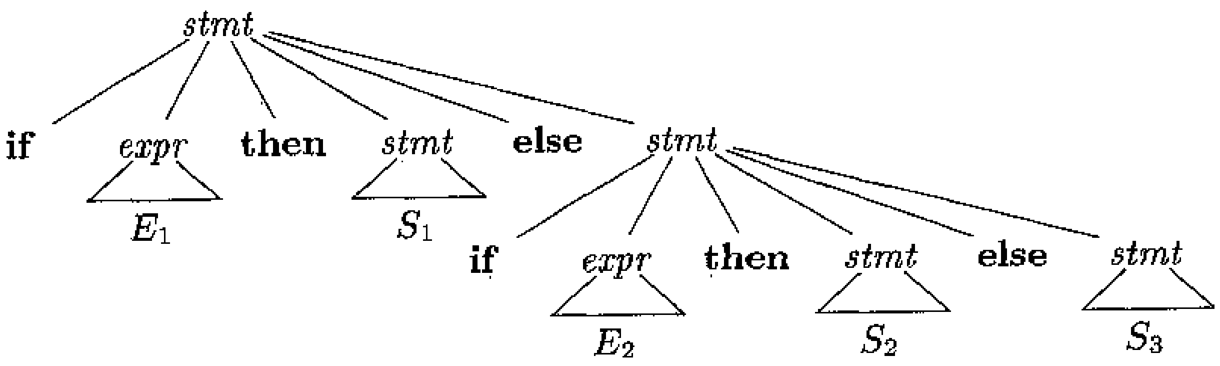
Eliminating ambiguity

Consider the grammar

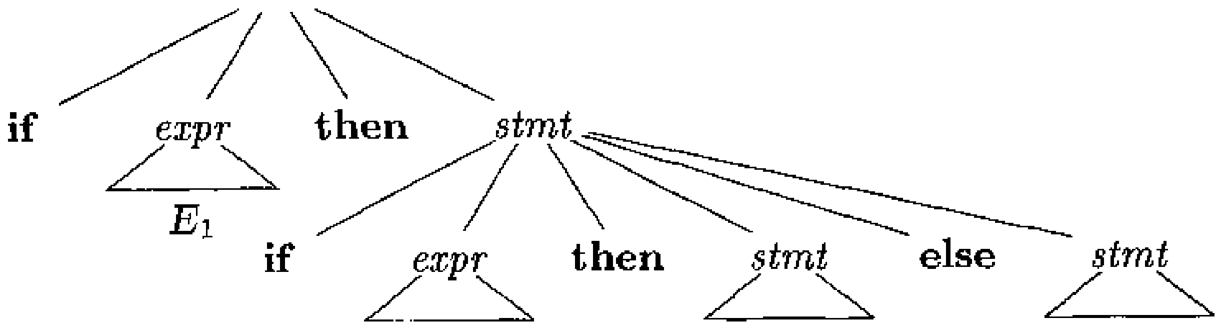
stmt → if E then S | if E then S else S | other

For the sentence

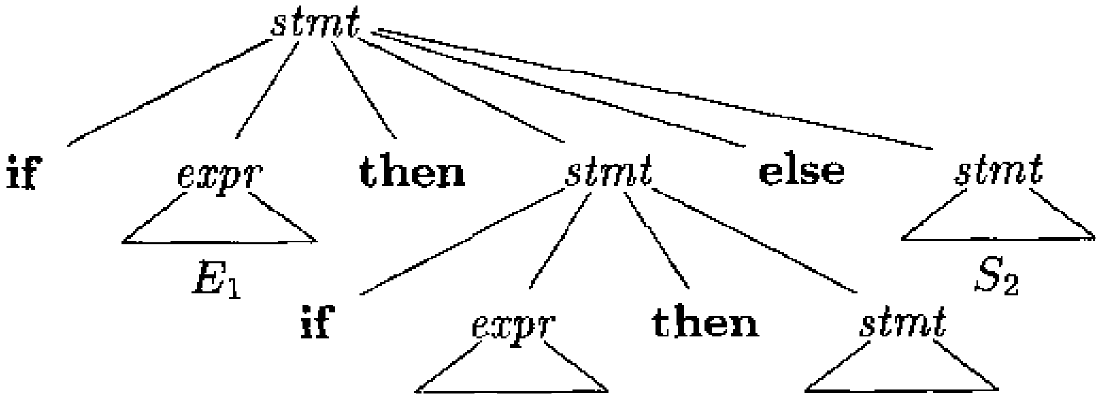
If E then S else if E then S else S



• For the sentence
If E then if E then S else S



and



The general rule to eliminate ambiguity is:

"match each else with the closest previous unmatched then"

Unambiguous grammar is:

stmt → matched-stmt | unmatched-stmt
matched-stmt →if E then matched-stmt else matched-stmt | other
unmatched-stmt →if E then stmt | if E then matched-stmt else unmatched-stmt

Elimination of left recursion

A grammar is left recursive if it has a nonterminal A, such that there is a derivation A => Aα for some string α. Top-down parsing cannot handle left recursive grammars. So elimination is required

Algorithm: Eliminating left recursion

INPUT: Grammar *G* with no cycles or Є-productions.
OUTPUT: An equivalent grammar with no left recursion.
METHOD: Apply the algorithm to the grammar *G.* Note that the resulting non-left-recursive grammar may have Є-productions.

```
  arrange the nonterminals in some order $A_1,A_2,... ,An$.
  for(each $i$ from 1 to $n$){
        for(each $j$ from 1 to $i$-1){
              replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta 2\gamma \mid \ldots \mid \gamma$.
        where $A_j$ —> $Si$ I $S2$ I • • • I $Sk$ are all current Aj-productions}
  eliminate the immediate left recursion among the ^-productions}
```

Algorithm to eliminate left recursion from a grammar

Consider again the grammar,

E →E + T I E – T I T
T →T * F I T / F I F
F → (E) I id

The modified grammar after the elimination of left recursion is:

E -> T E'
E' ^ + T E'
T ^ FT'
*T' -> * F T*
F->(E) **I id**

Left factoring

It is a grammar transformation suitable for predictive parsing. Basic idea is: when it is not clear which of the two productions to expand, we can rewrite the productions so to make the right choice.

The following grammar abstracts the "dangling-else" problem:

S → i E t S I i E t S e S I a
E → b

Here, i, *t,* and e stand for if, then, and else; *E* and S stand for "conditional expression" and "statement." After left-factoring this grammar, we have:

S → i E t S S' I a
S' → e S I Є
E → b

Top-down parsing

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

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say *A*. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

To find a left-most derivation for an input string, To construct parse tree for the input starting from the root and creating the nodes of the parse tree in preorder, Predictive parser is non-backtracking form of top-down parser, Predictive parsers is a special case of recursive-descent parsing, General form of top-down parsing is recursive-descent, that may involve backtracking

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. By eliminating left recursion from a grammar, left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parser without backtracking – predictive parsing. The proper alternative must be detectable by looking at only the first symbol it derives FIRST and FOLLOW – two functions associated with a grammar G, in construction of a predictive parser.

Recursive-Descent Parsing

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

A typical procedure for a nonterminal in a top-down parser

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Pseudo code for a typical nonterminal is shown above. Note that this pseudo code is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

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

To allow backtracking, the code needs to be modified. First, we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try do we declare that an input error has been found. In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

FIRST and FOLLOW

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

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

1. If *X* is a terminal, then FIRST(X) = *{X}.*

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

3. If $X \rightarrow$ Є is a production, then add Є to FIRST(X).

To compute 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 $ is the input right end marker.

2. If there is a production *A→ aB/3,* then everything in FIRST(/3) except Є is in FOLLOW(B).

3. If there is a production *A→aB,* or a production *A→ aB/3,* where FIRST(/3) contains Є, then everything in FOLLOW(A) is in FOLLOW(B) .

Consider the grammar, (after the elimination of left recursion and doing left factoring)

E→TE`
E`→+TE`|Є
T→FT`
T`→*FT`| Є
F→(E)|id

FIRST(E)=FIRST(T)=FIRST(F)={(,id}
FIRST(E`)={+, Є}
FIRST(T`)={*, Є}

FOllOW(E)=FOLLOW(E`)={),$}
FOllOW(T)=FOLLOW(T`)={+,),$}
FOLLOW(F)={*,+,),$}

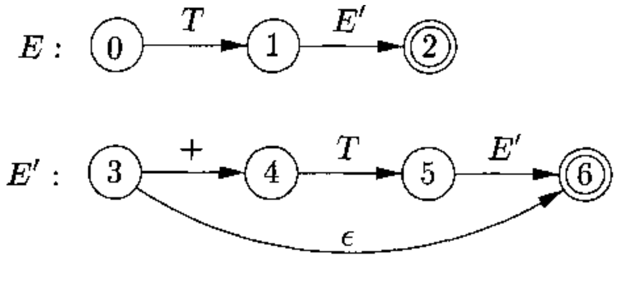
LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL (1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of look ahead at each step to make parsing action decisions.

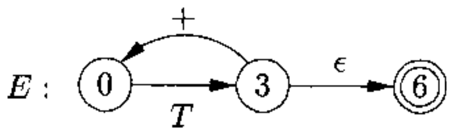
Transition diagrams are useful for visualizing predictive parsers. For example, the transition diagrams for nonterminals *E* and *E'* of above grammar appear in below figure. To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar. Then, for each nonterminal *A,*

1. Create an initial and final (return) state.

2. For each production $A \rightarrow X_1, X_2, \ldots, X_k$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_k$, If $A \rightarrow \epsilon$, the path is an edge labeled ϵ.



(a)



(b)

Construction of predictive parsing table, M

INPUT: Grammar *G.*
OUTPUT: Parsing table *M.*
METHOD: For each production A→α of the grammar, do the following:

1. For each terminal *a* in FIRST(α), add A→α to M[A, *a].*

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

If, after performing the above, there is no production at all in *M[A,* a], then set *M[A, a]* to error. Error entries are represented by blanks in the table.

For the above expression grammar, the algorithm produces the parsing table shown below. Blanks are error entries; nonblanks indicate a production with which to expand a nonterminal.

Nonterminal	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

Parsing table *M*

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while (X !=$ ) { /* stack is not empty */
        if (X is a ) pop the stack and advance ip;
        else if (X is a terminal ) error();
        else if (M[X,a] is an error entry ) error();
        else if (M[X,a] = X→ Y1Y2 •••Yk) {
                output the production X -> Y1 Y2 • • • Yk;
                pop the stack;
                push Yk, YK-1,... ,Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Predictive parsing algorithm

Matched	Stack	Input	Action
	E$	id + id * id$	
	TE'$	id + id * id$	output E -> TE'
	FT'E'$	id + id * id$	output T -+ FT'
	id T'E'$	id + id * id$	output F -> id
id	T'E'$	+ id * id$	match id
id	E'$	+ id * id$	output T" -> Є
id	+ TE'$	+ id * id$	output E' -+ + TE'
id+	TE'$	id * id$	match +
id+	FT'E'$	id * id$	output T FT'
id+	T'E'$	id * id$	output F -» id
id+id	T'E'$	*id$	match id
id+id	* FT'E'$	*id$	output T" -» * FT'
id+id*	FT'E'$	id$	match *
id+id*	T'E'$	id$	output F -» id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output T' -+ e
id+id*id	$	$	output E' -+ e

Move made in stack by predictive parser on input string

LL (1) grammar

It could be possible to have more than one entry in M [nonterminal, terminal] in parsing table. A grammar whose parsing table has no multiply-defined entries is said to be LL (1) where,
'L' means left to right scanning
'L' means producing left-most derivation
'1' means using one input symbol

Error recovery in predictive parsing

Panic mode recovery

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.

Synchronizing tokens are added in parsing table (enter "synch" for FOLLOW={set of terminals})

Nonterminal	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E→TE'			E→TE'	Synch	Synch
E'		E'→+TE'			E'→Є	E'→Є
T	T→FT'	Synch		T→FT'	Synch	Synch

T’		T’→Є	T’→*FT’		T’→Є	T’→Є
F	F→id	Synch	Synch	F→(E)	Synch	Synch

Moves made in stack by predictive parser on wrong input string with respect to the table entries

Stack	Input	Remark
E$	)id*+id$	Error, skip)
E$	id*+id$	id is in FIRST(E)
TE'$	id*+id$	
FT'E'$	id*+id$	
idT'E'$	id*+id$	
T'E'$	*+id$	
* FT'E'$	*+id$	
FT'E'$	+id$	Error, M[F,+]=synch
T'E'$	+id$	F has been popped
E'$	+id$	
+TE'$	+id$	
TE'$	id$	
FT’E’$	id$	
idT’E’$	id$	
T’E’$	$	
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.