UNIT -3

SYNTAX ANALYSIS

3.1 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

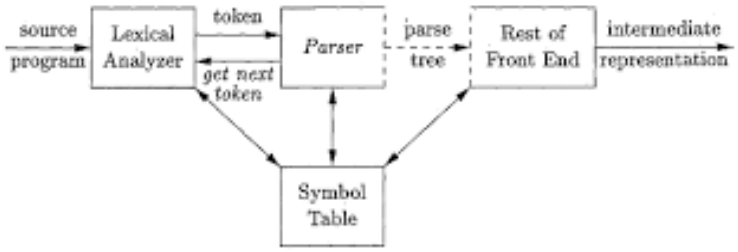

Figure 4.1: Position of parser in compiler model

3.2 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-downparsers 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.

3.3RECURSIVE 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
end;
```

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->e is a production, then add e to FIRST(X).
3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is, Y1.....Yi-$_1$=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

To compute the FIRST(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=>aBs where FIRST(s) except e is placed in FOLLOW(B).

3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, 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 -> TE'

E'-> +TE'|e

T -> FT'

T'-> *FT'|e

F -> (E)|id

Then:

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

FIRST(E')={+,e}

FIRST(T')={*,e}

FOLLOW(E)=FOLLOW(E')={),$}

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

FOLLOW(F)={+,*,),$}

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 -> 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 toproduction F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', $ and right parenthesis are in FOLLOW(E').

3.5 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. 1.For each production A-> a of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(a), add A->a, to M[A,a].
3. If e is in First(a), add A->a to M[A,b] for each terminal b in FOLLOW(A). If e is in FIRST(a) and $ is in FOLLOW(A), add A->a to M[A,$].
4. Make each undefined entry of M be error.

3.6.LL(1) GRAMMAR

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). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

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. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control

constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

3.7.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 synchronizingset for A. Fo 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 hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,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 generaitn 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 e 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.