Unit 2
Chapter 2
Syntax Analysis

Structure of the Unit

1. The Role of the Parser
2. Context-free Grammars
3. Writing a Grammar
4. Top-down Parsing
5. Bottom-up Parsing
6. Operator-Precedence Parsing
7. LR Parsers
8. Using ambiguous grammars
9. Parser Generators

2.1 Course Outcomes

After the successful completion of this unit, the student will be able to:

1. Memorize the steps involved in the syntax analysis phase.
2. Describe the role of a parser in syntax analysis and its relationship with lexical analysis.
3. Utilize a parsing algorithm to analyze the syntax of a program and identify syntax errors.
4. Analyze a grammar and identify any ambiguities or conflicts that might arise during parsing.
5. Design and implement a custom grammar for a small programming language.

2.2 Compilers

Syntax analysis, also known as **parsing**, is an essential phase in the compilation process of programming languages. It involves analyzing the structure of a given sequence of tokens (lexical units) and determining whether it follows the rules of the grammar. The primary goal of syntax analysis is to check the syntax validity of the input program and generate a parse tree or syntax tree that represents its structure.

The syntax analysis phase is typically the second step in the compilation process. Its main purpose is to ensure that the program adheres to the syntactic rules specified by the programming language grammar.

The syntax analysis process involves the following key components and techniques:

Grammar: A formal grammar is a set of rules that defines the syntax of a programming language. It describes the valid combinations and ordering of tokens to form correct statements or expressions. Commonly used grammar is Context Free Grammar(CFG) or BNF(Backus Naur Form). The grammar gives a precise, easy to understand syntactic specification of a programming language.

Parser: The parser is the core component of the syntax analysis phase. It takes a stream of tokens generated by the lexical analyzer and verifies whether they follow the grammar rules. The parser constructs a parse tree or a syntax tree as a *hierarchical representation* of the program's syntactic structure.

Top-down Parsing: This parsing technique starts with the root of the parse tree and recursively applies grammar rules to derive the input tokens. The most widely used top-down parsing algorithm is *recursive descent parsing*, which typically implements a separate function for each non-terminal symbol in the grammar.

Bottom-up Parsing: Unlike top-down parsing, bottom-up parsing starts with the input tokens and works upwards to construct the parse tree. A popular bottom-up parsing algorithm is the *LR (left-to-right, rightmost derivation)* parsing technique, which uses a table-driven approach to handle the grammar rules.

Ambiguity and Conflict Resolution: Sometimes, a grammar may have ambiguous productions or conflicting rules that can lead to multiple parse trees or parsing conflicts. Techniques like operator precedence and associativity rules or resolving shift-reduce and reduce-reduce conflicts help ensure deterministic parsing.

Error Handling: During syntax analysis, the parser identifies syntax errors when the input program does not conform to the grammar rules. Error recovery techniques, such as *panic-mode recovery* , *phrase level error recovery*, or *error productions*, aim to detect and report as many errors as possible while continuing the parsing process.

Overall, syntax analysis phase plays a vital role in the compilation process by verifying the correctness of the program's structure according to the specified grammar. Its output, the *parse tree* or *syntax tree*, serves as a foundation for subsequent phases like semantic analysis and code generation, enabling further analysis and translation of the program.

2.3 Role of the Parser

In the compiler model that we are discussing, the parser examines the sequence of tokens provided by the lexical analyzer as shown in the Fig 2.1 and validates that the string can be generated by the grammar. In case the parser fails to validate the string, it must report errors precisely. Also, the parser must recover quickly from the errors, so that it can continue parsing the remaining part of the input string.

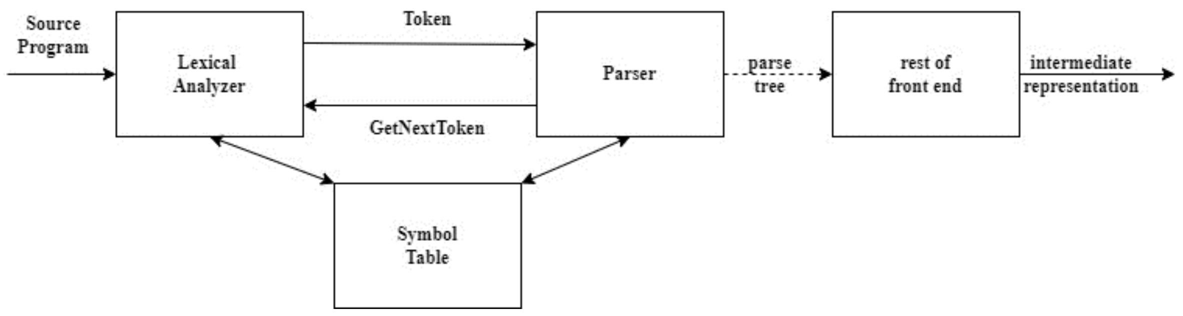


Fig 2.1 Parser lexer interaction

The output of the parser is some representation of the parse tree. Other tasks performed by the parser are as follows:

Collecting information about various tokens and storing in the symbol table.

Performing type checking.

Generating intermediate code.

2.3.1 Syntax Error Handling

A good compiler must assist the programmer in identifying errors. Planning the error handling from the beginning, simplifies the structure of the compiler and improves its response to errors. Programs contain errors at different levels. For example, errors can be

- **Lexical:** Misspelled identifier, keyword or operator.
- **Syntactic:** Expressions with mismatching pairs of parentheses or braces, missing semicolon, consecutive operators or operands etc.
- **Semantic:** Operators applied to an incompatible operand, e.g., array index with a real operand.
- **Logical:** Infinite recursive calls, infinite looping constructs etc.

Often most of the error detection and recovery in a compiler is centered around the syntax analysis phase. This is because many errors are syntactic in nature and detected when the string of tokens from the lexer is not according to the grammar. The other reason is the precision of the modern parsing techniques. In this section we shall discuss few error recovery strategies.

The goals of the error handler in a parser are:

- It should recover from the error quick enough so that the subsequent errors can be detected.
- It should not slow down the processing of correct programs.
- It should report the presence of errors clearly and precisely

The error handler must report the position in the source program where the error has occurred. Many compilers print the line number and the position in the line with the error.

2.3.2 Error Recovery Strategies

The parser must not quit or hang after detecting an error. Instead, it should continue to read the remaining portion of the input string and report all possible errors. Parser must recover to a state, where processing of the remaining input string continues. There are many general strategies the parser may employ to recover from an error. They are:

- Panic mode recovery
- Phrase level recovery
- Using error productions
- Global correction

Panic mode recovery: This is the simplest method to implement. On detecting an error, the parser skips the input symbols, till one of the synchronizing tokens is found. Synchronizing tokens are usually delimiters.

Phrase level recovery: On detecting an error, the parser may perform local correction on the remaining input. The local correction may be,

- Replace a symbol by another symbol
- Delete an extra character
- Insert a missing character
- Interchange adjacent characters etc.

The choice of the local correction depends on the compiler designer.

Using error productions: This method is suitable, if the compiler designer has a good guess of the common errors made by the programmers. In this this method the grammar is augmented with additional rules(productions) , one each for a guessed error.

Global correction: Ideally the compiler should make as few corrections as possible while processing the input string. There are few algorithms to choose minimal corrections. These algorithms are too costly in terms of time and space. Hence, they are studied only theoretically.

2.4 Context Free Grammar

In the realm of compiler design, context-free grammars play a fundamental role in describing the syntax of programming languages. A context-free grammar provides a formal framework to define the structure and syntax rules of a language, facilitating the development of compilers and parsing algorithms.

A context-free grammar consists of a set of non-terminals, terminals, production rules and a start symbol.

- **Non-terminals:** These symbols represent syntactic categories, such as expressions, statements, or declarations.
- **Terminals:** These symbols represent the actual tokens or lexemes that make up the language, like keywords, identifiers, or operators.
- **The production rules:** These define how the non-terminals and terminals can be combined to form valid program constructs. They dictate how valid sentences or programs in the language can be constructed. Each rule specifies the formation of a non-terminal symbol by replacing it with a sequence of terminal and/or non-terminal symbols.
- **Start symbol:** It is a special non-terminal representing the main language, whereas the other non-terminals represent supporting languages.

By defining a context-free grammar, a compiler designer establishes the syntax rules of a language, enabling the implementation of parsing algorithms that can analyze and understand the structure of programs written in that language. These algorithms parse the input program based on the context-free grammar, ensuring that it adheres to the defined syntax rules.

Overall, context-free grammars serve as a foundation for the design and development of compilers, enabling the transformation of high-level programming languages into machine-executable code by ensuring the syntactic correctness of programs.

2.4.1 Notational Conventions

1. *Terminals* are represented by
    a. Lower case letters at the beginning of the alphabet like a, b, c.
    b. Operator symbols like $+, -, *$ etc.
    c. Punctuation symbols like comma, parentheses, semicolon etc.
    d. Digits $0, 1, \cdots, 9$.
2. *Non-Terminals* are represented by
    a. Upper case letters early in the alphabet like A, B, C.
    b. The letter S is normally the start symbol.
3. Upper-case letters late in the alphabet like X, Y, Z, represent the *grammar-symbols*, i.e., either a terminal or a non-terminal.
4. Lower-case letters late in the alphabet like x, y, z, represent string of zero or more terminals.
5. Lower-case Greek letters α, β, γ, represent the sentential forms, i.e., string of zero or more terminals and/or non-terminals.

2.4.2 Derivations and Parse Trees

A *derivation* is a step-by-step process of applying production rules in a context-free grammar to transform a start symbol into a specific string belonging to the language defined by the grammar. Each step in the derivation involves replacing a non-terminal symbol with one of its production expansions until only terminal symbols remain.

Example 1: Consider the following CFG with the start symbol S and productions:

1. $S \rightarrow AB$
2. $A \rightarrow a$
3. $B \rightarrow b$

To derive the string "ab," we can use the following derivation:

$S \rightarrow AB$ (Using rule 1)
$\rightarrow aB$ (Using rule 2)
$\rightarrow ab$ (Using rule 3)

Example 2: Consider the following CFG representing the arithmetic expressions:

1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow E * E$
4. $E \rightarrow E/E$
5. $E \rightarrow (E)$
6. $E \rightarrow -E$
7. $E \rightarrow \boldsymbol{id}$

In the above grammar E is the only non-terminal, whereas $+, -, *, /, (,)$ and $\boldsymbol{id}$ are terminal symbols.

1. To derive the terminal string $id + id$ the derivation is
   i. $E \Rightarrow E + E$ (Applying rule1, $\Rightarrow$ indicates a derivation step)
   ii. $\Rightarrow id + E$ (Applying rule 7)
   iii. $\Rightarrow id + id$ (Applying rule 7)
2. To derive the terminal string $-(id + id)$ the derivation is
   i. $E \Rightarrow -E$ (Applying rule 6)
   ii. $\Rightarrow -(E)$ (Applying rule 5)
   iii. $\Rightarrow -(id + E)$ (Applying rule 7)
   iv. $\Rightarrow -(id + id)$ (Applying rule 7)

Left Most Derivation(LMD): In each derivation step the left most non-terminal is replaced by the body of the production.

Right Most Derivation(RMD): In each derivation step the right most non-terminal is replaced by the body of the production.

Parse trees:

A parse tree is a graphical representation of a derivation in a context-free grammar. It represents the hierarchical structure of the generated string, where each node in the tree corresponds to a non-terminal symbol, and the edges represent the production rules used to generate the string. The leaves of the parse tree represent terminal symbols. The start symbol is the root of the tree. The concatenation of the leaf nodes from left to right in the parse tree is called the ***yield*** of the parse tree and it represents the generated string.

The parse tree for the derivation in example 1 above is shown in Fig 2.2

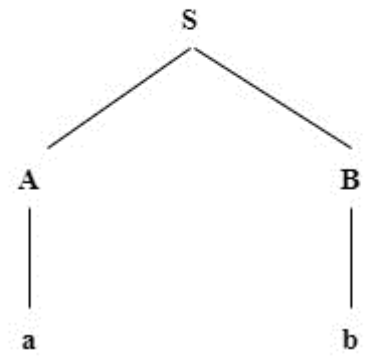
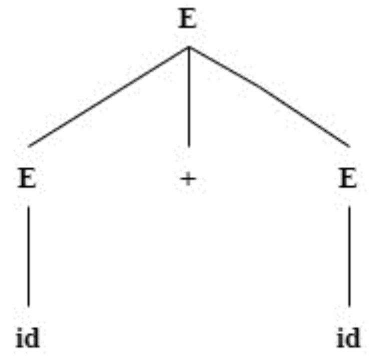


Fig 2.2 parse tree for the string *ab*

The parse trees for the derivations in example 2 above are shown in Fig 2.3(a) and Fig 2.3(b)



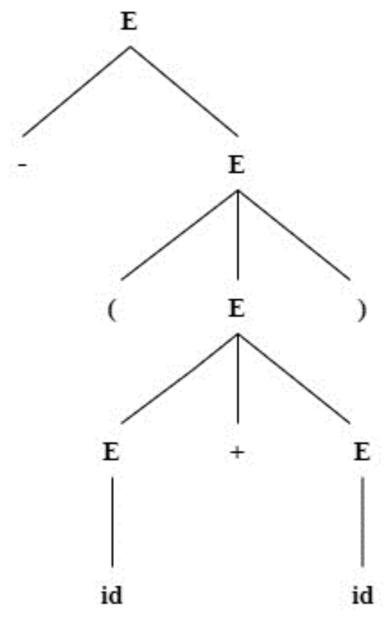


Fig 2.3(a) parse tree for the string $id + id$ **Fig 2.3(b)** parse tree for the string $-(id + id)$

2.4.3 Ambiguous Grammars

A grammar that produces two or more left most derivations or two or more right most derivations or two or more parse trees for any valid string in the language is said to be an ambiguous grammar. Ambiguous grammars pose challenges because they can result in various interpretations of the same input. Therefore, ambiguous grammars are not suitable for compilers.

Eliminating Ambiguity:

Ambiguous grammar needs to be rewritten to eliminate the ambiguity. For example, consider the below grammar for the if statement:

$stmt \rightarrow \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt$

$\quad |\ \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt\ \boldsymbol{else}\ stmt$

$\quad |\boldsymbol{other}$

(Note: In the above grammar the tokens in boldface are terminals whereas the other tokens are non-terminals)

Consider the statement **if** E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3, the parse tree is shown in Fig 2.4

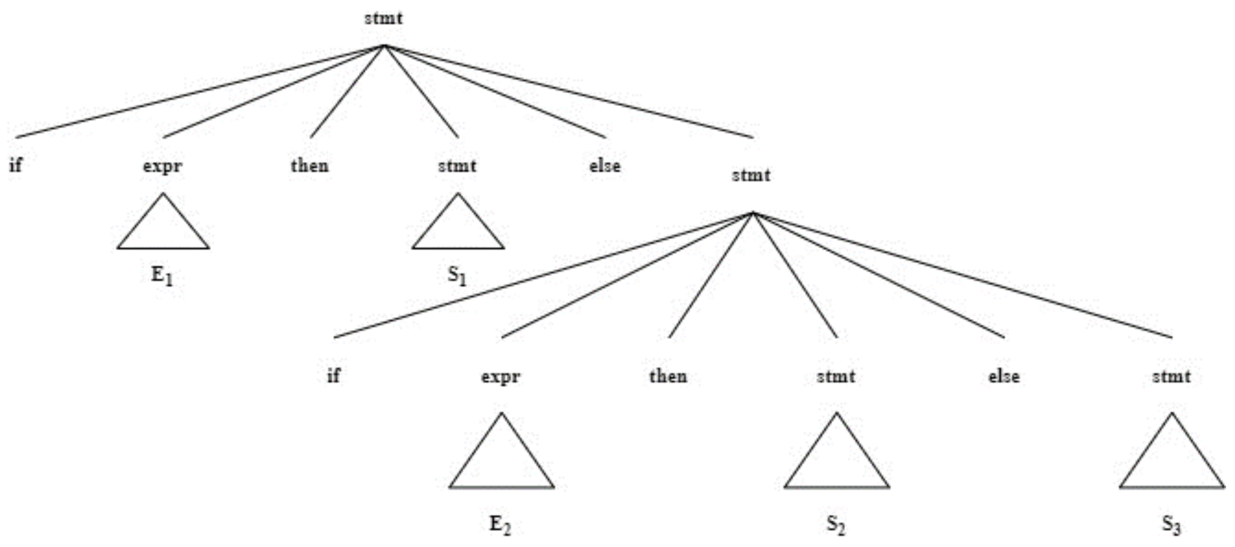


Fig 2.4 parse tree for the if statement

The statement **if** E_1 **if** E_2 **then** S_1 **else** S_2 has two parse trees shown in Fig 2.5(a) and (b).

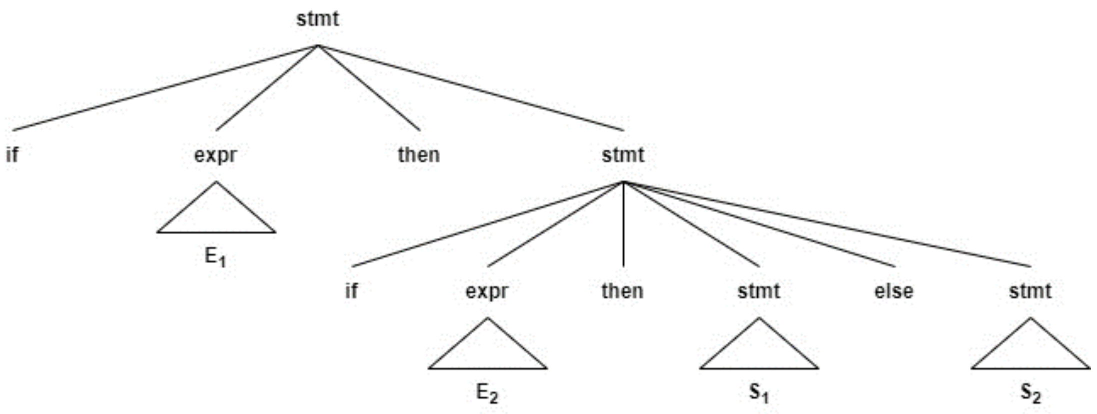


Fig 2.5(a) Parse tree for the if statement

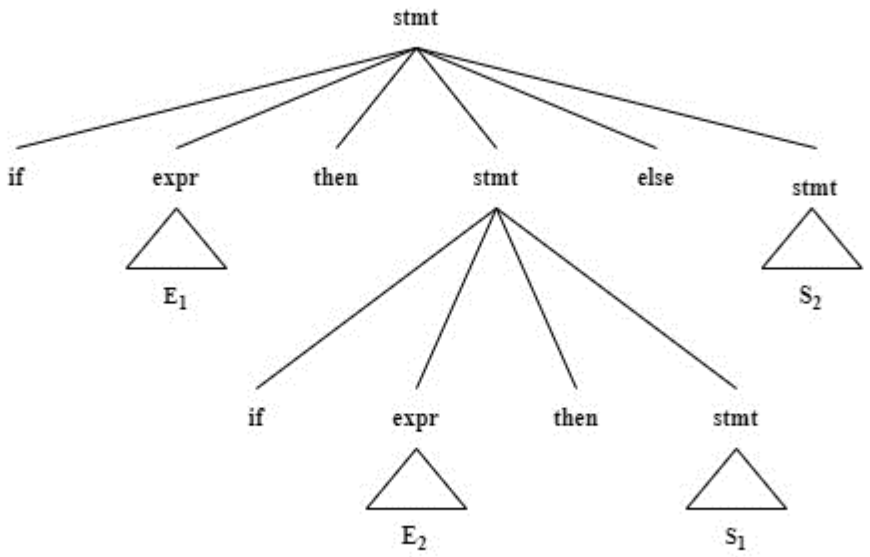


Fig 2.5(b) Parse tree for the if statement

Since the statement **if** E_1 **if** E_2 **then** S_1 **else** S_2 has two parse trees the grammar is ambiguous.

In the "if" statement, the "else" must always correspond to the nearest "if." This critical rule has not been addressed in the preceding grammar, resulting in ambiguity. To ensure clarity, an unambiguous grammar for the "if" statement is crafted by accounting for this fact.

$$\begin{aligned} stmt &\rightarrow matched\_stmt \\ &\quad | \, unmathed\_stmt \\ matched\_stmt &\rightarrow \textbf{\textit{if}}\, expr\ \textbf{\textit{then}}\ matched\_stmt\ \textbf{\textit{else}}\ matched\_stmt \\ unmatched\_stmt &\rightarrow \textbf{\textit{if}}\ expr\ \textbf{\textit{then}}\ stmt \\ &\quad | \, \textbf{\textit{if}}\ expr\ \textbf{\textit{then}}\ matched\_stmt\ \ \textbf{\textit{else}}\ unmatched\_stmt \end{aligned}$$

The above grammar generates unique parse trees for all the valid strings. Hence it is an unambiguous grammar. (Note: In the above grammar the tokens in boldface are terminals whereas the other tokens are non-terminals)

2.4.4 Elimination of Left Recursion

A grammar is left recursive if it has productions of the form $A \rightarrow A\alpha$ or derivations of the form $A \overset{+}{\Rightarrow} A\alpha$ for some string α. If the grammar has productions of the form $A \rightarrow A\alpha$ it is said to have *direct* or *immediate left recursion*. On the other hand, if has derivations of the form $A \overset{+}{\Rightarrow} A\alpha$ it is said to have *indirect left recursion*.

Left recursive grammars are not suitable for top-down parsing as they can lead to infinite loops during parsing.

Eliminating immediate(direct) left recursion:

Consider the production of the form $A \rightarrow A\alpha \mid \beta$ where A is the non-terminal and α and β are sentential forms(strings of non-terminals and/or terminals). These productions are replaced by

$A \rightarrow \beta A'$ and

$A' \rightarrow \alpha A' \mid \epsilon$

Example: Eliminate the left recursion from the grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

The grammar after eliminating left recursion is:

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

Immediate left recursion is eliminated by the following technique:

Replace the production of the form:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ **by**

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$ and

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A'$

Eliminating indirect left recursion:

Algorithm 2.1 below eliminates indirect left recursion from the grammar.

```
Arrange the non-terminals in some order $A_1, A_2, \cdots, A_n$.
for each ( i from 1 to n) {
    for (each j from 1 to i-1) {
        Replace each production of the form $A_i \to A_j\gamma$ by the productions
            $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$  where
                $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are the $A_j$ productions.
    }
    Eliminate immediate left recursion among the $A_i$ productions
}
```

Algorithm 2.1 Algorithm to eliminate left recursion from the grammar

Example: Check if the grammar given below is left recursive. If it is left recursive eliminate left recursion from the grammar.

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

Solution:

The above grammar has both immediate and indirect left recursion. Second rule has direct left recursion. The derivation $S \Longrightarrow Aa \Longrightarrow Sd$ shows that the grammar has indirect recursion as well. Using the algorithm 2.1, the grammar is rewritten as

Step 1:

$$S \to Aa \mid b$$

$A \to Ac \mid Aad \mid bd \mid \epsilon$ (non-terminal S replaced by its body)

Step 2: Eliminating direct recursion in the second rule above we get

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

2.4.5 Left factoring

When a grammar contains common prefixes in the right-hand sides of its productions, it can lead to ambiguity during parsing. Left factoring helps to resolve this issue by factoring out the common prefixes, making the grammar more concise and easier to parse. In other words, it is the grammar transformation producing grammar suitable for top-down parsing.

Consider the following grammar for the if statement:

$$stmt \rightarrow \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt$$
$$\mid \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt\ \boldsymbol{else}\ stmt$$

The non-terminal has two production bodies having the common prefix "$\boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt$". When the next input to the parser is the token $\boldsymbol{if}$ it cannot determine which alternative to choose to parse the input string. In general, when the grammar has productions of the form $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ and the input begins with α, the parser cannot decide whether to expand the non-terminal A to $\alpha\beta_1$ or $\alpha\beta_2$. This can be avoided by removing the common prefixes i.e., left factoring the grammar.
The original productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ after left factoring become

$A \rightarrow \alpha A'$ and

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

In general, the productions of the form

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ are rewritten as

$A \rightarrow \alpha A' \mid \gamma$ and

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

Example: The grammar for if statement

$$stmt \rightarrow \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt$$
$$\mid \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt\ \boldsymbol{else}\ stmt$$

After left factoring looks like

$$stmt \rightarrow \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt\ stmt'$$
$$stmt' \rightarrow \epsilon \mid \boldsymbol{else}\ stmt$$

2.5 Top-Down Parsing

Top-down parsing is a fundamental technique used in compilers to analyze and understand the structure of source code written in a programming language. The key advantage of top-down parsing is that it closely resembles how humans read and understand code, making it easier to design and maintain. In this section, we'll explore the principles and methodologies behind top-down parsing, highlighting its role in converting human-readable source code into machine-executable instructions.
Top-down parsing tries to find the leftmost derivation for a given input string. It constructs the parse tree from top to bottom, i.e., starting from the root and creating the nodes towards the leaves. There are two forms of top-down parsing, namely;

- Recursive descent parser with backtracking and
- Recursive descent parser without backtracking(predictive parsing).

2.5.1 Recursive descent parser with backtracking:

A Recursive Descent Parser with Backtracking is a top-down parsing technique used in compilers to analyze the syntax of a programming language. It employs a series of recursive procedures, each corresponding to a non-terminal symbol in the grammar. Execution begins with the procedure for the start symbol. The procedure for the start symbol halts and announces successful completion of parsing if it scans the entire input string. Algorithm 2.2 presents the pseudocode that illustrates the implementation of the procedure for a non-terminal.

```
void A(){
    choose a production $A \rightarrow X_1X_2 \cdots X_k$;
    for( i = 1 to k ){//For each grammar symbol $X_i$ on the RHS of the production
        if ( $X_i$ is a non-terminal)
            call procedure $X_i()$;
        else
            if ( $X_i$ equals the current input symbol $a$)
                advance the input pointer to the next symbol;
            else
                /* error */
}
```

Algorithm 2.2 Procedure for a non-terminal in recursive descent parsing

If the non-terminal A has two or more bodies i.e., the A production is of the type

$$A \rightarrow X_1X_2 \cdots X_k \mid Y_1Y_2 \cdots Y_k \mid \cdots$$

then the parser cannot determine which alternative to choose in the first step. The parser has to try the first alternative and if it fails to generate the input string, backtrack and try the next alternative and so on. To allow backtracking the above procedure needs to be modified. However, this method is time consuming as the parser has to try different alternatives for each of the productions.

For example, consider the grammar

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

To Construct the parse tree for the string $w = cad$ the steps are shown in Fig 2.6

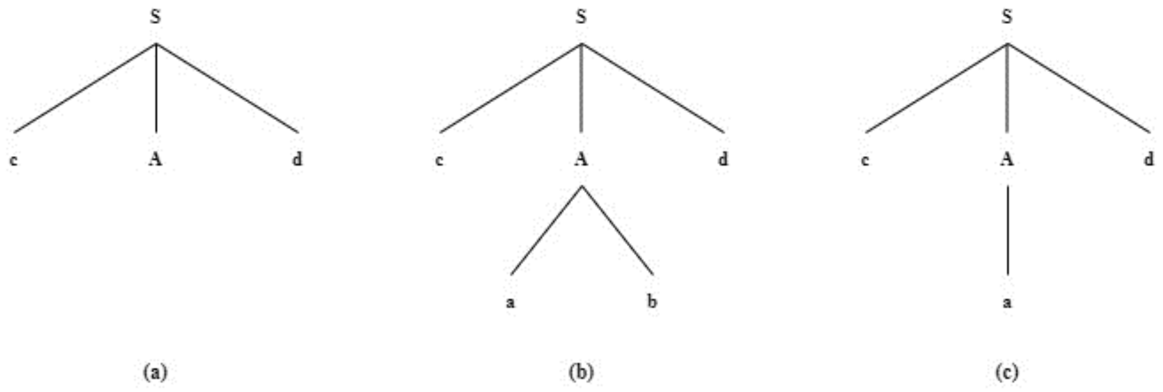


Fig 2.6 Steps in top-down parsing

The parser applies rule 1 to generate the sentential form cAd(Fig 2.6(a)). In the next step the parser applies the rule $A \rightarrow ab$ to generate the parse tree of Fig 2.6(b). But the yield of this parse tree is $cabd$, which does not match the input string. In other words, the parser fails to generate the input string cad. Hence, it backtracks and applies the production $A \rightarrow a$, generating the parse tree in Fig 2.6(c), which represents the input string.

2.5.2 FIRST and FOLLOW

The first and follow functions are very useful in the construction of both top-down and bottom-up parsers. During top-down parsing they assist in choosing the correct production based on the next input symbol. The follow set is used as the set of synchronizing tokens in the panic mode error recovery.

$\boldsymbol{FIRST(\alpha)}$, where α is a string of grammar symbols, is defined as the set of terminals that begin the strings derived from α.

Computing $\boldsymbol{FIRST(X)}$ for a grammar symbol $\boldsymbol{X}$:

1. If X is a terminal, then $FIRST(X) = \{X\}$.
2. If X is a non-terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production in the grammar then
    a. Add $FIRST(Y_1)$ to $FIRST(X)$
    b. Add $FIRST(Y_i)$ to $FIRST(X)$ if all of $FIRST(Y_1), FIRST(Y_2), \cdots, FIRST(Y_{i-1})$ contain ϵ, i.e., $Y_1 Y_2 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$.
    c. If $\epsilon \in FIRST(Y_i)$, for $1 \leq i \leq k$ then add ϵ to $FIRST(X)$.
3. If $X \rightarrow \epsilon$ is a production in the grammar then add ϵ to $FIRST(X)$.

Computing $FIRST$ for the string $X_1X_2 \cdots X_n$:

1. Add all non-ϵ symbols of $FIRST(X_1)$ to $FIRST(X_1X_2 \cdots X_n)$.
2. Add all non-ϵ symbols of $FIRST(X_2)$ to $FIRST(X_1X_2 \cdots X_n)$ if $FIRST(X_1)$ contains ϵ.
3. Add all non-ϵ symbols of $FIRST(X_3)$ to $FIRST(X_1X_2 \cdots X_n)$ if both $FIRST(X_1)$ and $FIRST(X_2)$ contain ϵ.
4. Add ϵ to $FIRST(X_1X_2 \cdots X_n)$ if all of $FIRST(X_i)$ for $1 \leq i \leq n$ contain ϵ.

Computing $FOLOW(A)$ for a non-terminal A:

1. Add \$ to $FOLLOW(S)$, where S is the start symbol and \$ marks the end of input.
2. If $A \to \alpha B\beta$ is a production in the grammar then add all non-ϵ symbols of $FIRST(\beta)$ to $FOLLOW(B)$.
3. If there is a production $A \to \alpha B$ or a production $A \to \alpha B\beta$ and $FIRST(\beta)$ contains $\epsilon(\beta \overset{*}{\Rightarrow} \epsilon)$ then add all the symbols of $FOLLOW(A)$ to $FOLLOW(B)$.
4. Apply rules 2 and 3 till no new terminals are added to follow sets.

Example 1: Construct the FIRST and FOLLOW sets for the following grammar.

$$A \to BC$$
$$B \to b \mid \epsilon$$
$$C \to c \mid \epsilon$$

Solution:

Constructing the FIRST set:

Consider the production $B \to b \mid \epsilon$

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

Consider the production $C \to c \mid \epsilon$

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

Consider the production $A \to BC$

Add $non - \epsilon$ symbols of $FIRST(B)$ to $FIRST(A)$, i.e., b is added to $FIRST(A)$.

$$\therefore FIRST(A) = \{b\}.$$

Since $FIRST(B)$ contains ϵ add $non - \epsilon$ symbols of $FIRST(C)$ to $FIRST(A)$, i.e., c is added to $FIRST(A)$.

$$\therefore FIRST(A) = \{b, c\}.$$

Since $FIRST(B)$ contains ϵ and also $FIRST(C)$ contains ϵ, add ϵ to $FIRST(A)$.

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

Constructing the follow set:

Apply rule 1 and add \$ to $FOLLOW$ of start symbol A.

$$\therefore FOLLOW(A) = \{\$\}$$

Rule 2 and 3 are applied to productions of the form $A \rightarrow \alpha B\beta$ or $A \rightarrow \alpha B$, i.e., there must be at least one non-terminal in the production body. Hence in the above example rule 2 and 3 are not applicable to the 2nd and 3rd productions.

Applying rule 2 to the production $A \rightarrow BC$, add $non - \epsilon$ symbols of $FIRST(C)$ to $FOLLOW(B)$.

$$\therefore FOLLOW(B) = \{c\}$$

Applying Rule 3 to the production $A \rightarrow BC$

Add $non - \epsilon$ symbols of $FOLLOW(A)$ to $FOLLOW(C)$.

$$\therefore FOLLOW(C) = \{\$\}$$

$\because FIRST(C)$ contains ϵ $non - \epsilon$ symbols of $FOLLOW(A)$ to $FOLLOW(B)$.

$$\therefore FOLLOW(B) = \{\$, c\}$$

The $FIRST$ and $FOLLOW$ sets for the non-terminals in the example grammar are shown in table 2.1.

Non-Terminal	FIRST	FOLLOW
A	$\{b, c, \epsilon\}$	$\{\$\}$
B	$\{b, \epsilon\}$	$\{\$, c\}$
C	$\{c, \epsilon\}$	$\{\$\}$

Table 2.1 First and Follow sets

Example 2: Construct the FIRST and FOLLOW sets for the following grammar.

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

Solution:

Constructing the FIRST set:

1. $First(F) = \{(, id\}$ as production bodies for F start with (or id.
2. $First(T) = FIRST(F) = \{(, id\}$ as the first symbol in the production $T \rightarrow FT'$ is F and $FIRST(F)$ does not contain ϵ.

3. $FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}$ as the first symbol in the production $E \rightarrow TE'$ is T and $FIRST(T)$ does not contain ϵ.

4. $FIRST(E') = \{+, \epsilon\}$ as production for the non-terminal E' starts with $+$ or ϵ.

5. $FIRST(T') = \{*, \epsilon\}$ as production for the non-terminal T' starts with $*$ or ϵ.

Constructing the follow sets:

1. Add \$ to $FOLLOW(E)$ as E is the start symbol.

2. It is evident from the production $F \rightarrow (E)$,) belongs to $FOLLOW(E)$.

3. Consider the production $E \rightarrow TE'$. Non-ϵ symbols of $FIRST(E')$ are added to $FOLLOW(T)$, i.e., $+$ is added to $FOLLOW(T)$. Also, $FIRST(E')$ contains ϵ, hence all the symbols in $FOLLOW(E)$ are added to $FOLLOW(T)$ i.e., \$ and) are added to $FOLLOW(T)$.

4. Consider the production $T \rightarrow FT'$. Non-ϵ symbols of $FIRST(T')$ are added to $FOLLOW(T)$, i.e., $*$ is added to $FOLLOW(T)$. Also, $FIRST(T')$ contains ϵ, hence all the symbols in $FOLLOW(T)$ are added to $FOLLOW(F)$ i.e., \$ and) are added to $FOLLOW(F)$.

The rules 2 and 3 are applied to all the productions till no new terminals are added to the follow sets. The first and follow sets for all the non-terminals in the grammar of example 2 is shown in table 2.2.

Non-Terminal	FIRST	FOLLOW
E	$\{(, id\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, id\}$	$\{), \$, *\}$
T'	$\{*, \epsilon\}$	$\{), \$, *\}$
F	$\{(, id\}$	$\{), \$, +, *\}$

Table 2.2 First and Follow sets

FIRST and FOLLOW sets enable efficient predictive parsing by allowing the parser to make decisions based on the current non-terminal being expanded and the next input symbol. By consulting the FIRST and FOLLOW sets, the parser can predict which production to use and how to handle different input sequences, leading to a faster and more accurate parsing process.

2.5.3 Recursive descent parser without backtracking(predictive parsing).

A predictive parser, also known as a recursive descent parser without backtracking, is a top-down parsing technique that uses a set of recursive procedures to parse the input based on a given grammar. Unlike backtracking parsers, which may explore multiple alternatives during parsing, predictive parsers make a parsing decision based solely on the current input symbol, without having to backtrack.

The predictive parser makes use of a table called predictive parsing table. It is a two-dimensional array $M[A, a]$, where A is a non-terminal and a is a terminal. Non-terminals are the rows in the table and terminal symbols are the columns. The matrix entry $M[A, a]$ determines the parsing decisions. Algorithm 2.3 below uses the information from the first and follow sets and constructs the parse table.

```
INPUT: Grammar $G$
OUTPUT: Parsing table $M$
METHOD:
For each production $A \rightarrow \alpha$ in the grammar, do the following
    1. For each terminal $a \in FIRST(A)$ add $A \rightarrow \alpha$ to $M[A, a]$.
    2. If $\epsilon \in FIRST(A)$ then for each $b \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$.
After performing the above procedure if $M[A, a]$ is blank set it to **error**.
```

Algorithm 2.3 Constructing the predictive parse table M

General procedure to construct the predictive parse table:

1. Eliminate ambiguity from the grammar.
2. Eliminate left recursion.
3. Left factor the grammar.
4. Construct first and follow sets.
5. Use algorithm 2.3 to make entries in the parse table.

Example 3: Construct the predictive parse table for the expression grammar shown below.

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

Solution:

- The grammar after eliminating left recursion is:

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

- Left factoring is not required as the alternatives for the non-terminals do not have common prefixes.
- First and Follow sets are shown in the table 2.2.

Following the steps in the algorithm 2.3

1. Consider the production $E \rightarrow TE'$

   $FIRST(TE') = FIRST(T) = \{(, id\}$. $FIRST(T)$ does not contain ϵ, hence only the symbols in $FIRST(T)$ are added to $FIRST(TE')$. $\because$ $(, id \in FIRST(TE')$, set $M[E, (\,] = E \rightarrow TE'$ and $M[E, id] = E \rightarrow TE'$.

2. Consider the production $E' \rightarrow +TE'$

   $FIRST(+TE') = \{+\}$. $\because$ $+ \in FIRST(+TE')$, set $M[E, +\,] = E' \rightarrow +TE'$.

3. Consider the production $E' \rightarrow \epsilon$

   The right-hand side of the production derives ϵ and $FOLLOW(E') = \{), \$\}$. Hence set $M[E',)] = E' \rightarrow \epsilon$ and $M[E', \$] = E' \rightarrow \epsilon$.

4. Consider the production $T \rightarrow FT'$

   $FIRST(FT') = FIRST(F) = \{(, id\}$. $FIRST(F)$ does not contain ϵ, hence only the symbols in $FIRST(F)$ are added to $FIRST(FT')$. $\because$ $(, id \in FIRST(FT')$, set $M[T, (\,] = T \rightarrow FT'$ and $M[F, id] = T \rightarrow FT'$.

5. Consider the production $T' \rightarrow * FT'$

   $FIRST(* FT') = \{*\}$. $\because$ $* \in FIRST(* FT')$, set $M[T, *\,] = T' \rightarrow * FT'$.

6. Consider the production $T' \to \epsilon$

   The right-hand side of the production derives ϵ and $FOLLOW(T') = \{*,), \$\}$. Hence set $M[T',*] = T' \to \epsilon,\ M[T',)] = T' \to \epsilon$ and $M[T', \$] = T' \to \epsilon$.

7. Consider the production $F \to (E)$

   $FIRST(\,(E)\,) = \{\,(\,\}.\ \therefore M[F, (\,] = F \to (E)$

8. Consider the production $F \to id$

   $FIRST(id) = \{id\}. \therefore M[F, id] = F \to id$

Table 2.3 below shows the predictive parse table for the expression grammar.

Non-Terminal	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to * FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Table 2.3 Predictive Parse Table

Non-recursive predictive parsing:

The method uses the predictive parse table to choose a correct production always. The predictive parser mimics the left most derivation. This table-driven parser is depicted in Fig 2.7. The table-driven parser has a stack containing the grammar symbols, an input buffer containing the string to be parsed and the predictive parsing table M. The input string ends with the end marker \$. Also, \$ is used to mark the bottom of the stack. The stack initially contains the start symbol of the grammar. The parsing algorithm makes the decision based on the grammar symbol X at the stack top and the current input symbol a. Algorithm 2.4 demonstrates the predictive parsing algorithm.

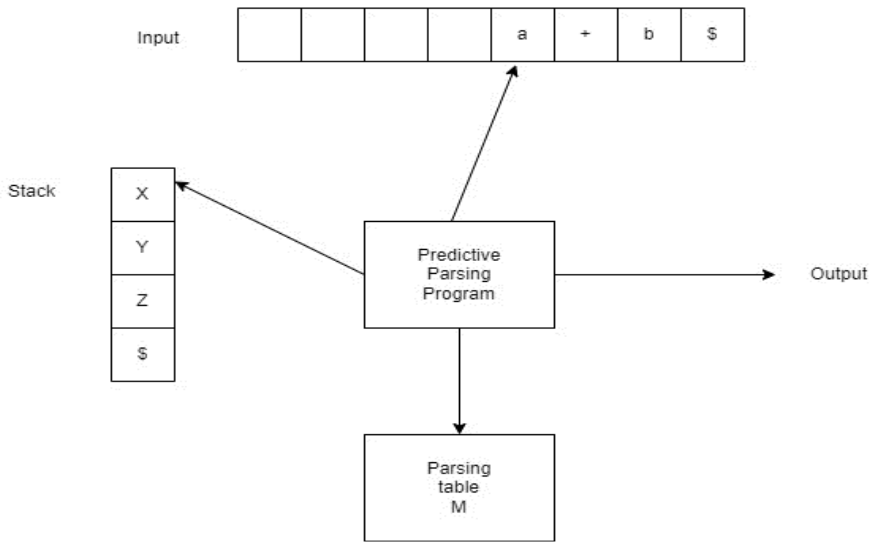


Fig 2.7 Table driven predictive parsing algorithm

```
INPUT: The string $w$ to be parsed and the predictive parse table $M$.
OUTPUT: if $w$ is in the language of the grammar $G$, i.e., $w \in L(G)$, then leftmost
derivation of $w$; otherwise, an error indication.
METHOD: Initially
   ▪ Input buffer contains the string $w\$$ and
   ▪ The stack contains the start symbol $S$ on top of $\$$.
Set $ip$ to point to the first symbol of $w$. // $ip$ is the pointer to the symbol in
the buffer.
Let $X$ be the symbol on the stack top.
Let $a$ be the next input symbol.
while $(X \neq S)$ {
  if $(X = a)$ pop the stack and advance the input pointer $ip$;
  else if $(X \text{ is a terminal})$ error();
       else if $(M[X, a] \text{is blank})$ error();
            else if $M[X, a] = X \rightarrow Y_1Y_2 \cdots Y_k$){
                   output the production $X \rightarrow Y_1Y_2 \cdots Y_k$;
                   pop the stack;
                   push the symbols $Y_kY_{k-1} \cdots Y_1$ on the stack, with $Y_1$ on the top;
                  }
                  set $X$ to the top of the stack;
}
```

Algorithm 2.4 Predictive Parsing Algorithm

Example 4: Parse the string $\boldsymbol{id + id * id}$ using the parse table 2.3.

Stack	Input	Action
E$	**id**+**id***id**$	
TE'$	**id**+**id*****id**$	Output $E \rightarrow TE'$
FT'E'$	**id**+**id*****id**$	Output $T \rightarrow FT'$
idT'E'$	**id**+**id*****id**$	Output $F \rightarrow id$
T'E'$	+**id*****id**$	Match **id**
E'$	+**id*****id**$	Output $T' \rightarrow \epsilon$
+TE'$	+**id*****id**$	Output $E' \rightarrow +TE'$
TE'$	**id*****id**$	Match +
FT'E'$	**id*****id**$	Output $T \rightarrow FT'$
idT'E'$	**id*****id**$	Output $F \rightarrow id$
T'E'$	***id**$	Match **id**
*FT'E'$	***id**$	Output $T' \rightarrow * FT'$
FT'E'$	**id**$	Match *
idT'E'$	**id**$	Output $F \rightarrow id$
T'E'$	$	Match id
E'$	$	Output $T' \rightarrow \epsilon$
$	$	Output $E' \rightarrow \epsilon$

Table 2.4 Parsing the string **id+id*id$**

The productions output int the third column, i.e., action column, correspond to the leftmost derivation of the string.

The grammar suitable for predictive parsing is also called LL(1) grammar. The first "L" indicates the input is scanned from left to right. The second "L" indicates the parser is using the productions corresponding to the leftmost derivation. The number "1" in parentheses indicates that the parsing decisions are based on the next input symbol(one symbol).

Example 5: Construct the predictive parse table for the grammar

$S \rightarrow iEtSeS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

The above grammar is for the if statement after left factoring. The token "i" is "if", "t" is "then" and "e" is else. They are terminals. The symbols $S, S', and\ E$ are the non-terminals.

Solution:

First and follow sets are shown in table 2.5

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

Table 2.5 First and Follow sets

The predictive parse table M is shown in table 2.6

Non-terminal	i	e	t	a	b	$
S	$S \rightarrow iEtSeS'$			$S \rightarrow a$		
S'		$S' \rightarrow eS$ $S' \rightarrow \epsilon$				$S' \rightarrow \epsilon$
E					$E \rightarrow b$	

Table 2.6 Predictive parse table M

In the above example the parse table entry $M[S', e]$ has two productions, namely $S' \rightarrow eS$ and $S' \rightarrow \epsilon$. During parsing if S' is the symbol on the stack top and e is the current input symbol, the parser cannot decide which production to use. This is because the grammar is not suitable for predictive parsing, i.e., it is not a LL(1) grammar.

Properties of LL(1) grammar:

1. The grammar must be unambiguous.
2. It should not be left recursive.
3. Whenever the grammar has distinct productions of the form $A \rightarrow \alpha \mid \beta$, the following conditions must hold.
    a. Both α and β must not derive strings starting with the same terminal, i.e., $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$.
    b. At most one of α or β can derive ϵ.
    c. If $\alpha \overset{*}{\Rightarrow} \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \emptyset$, and if $\beta \overset{*}{\Rightarrow} \epsilon$ then $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$.

2.5.4 Error Recovery in Predictive Parsing

An error in predictive parsing is detected when

1. The terminal on the stack top does not match the current input symbol OR
2. The current top of stack is A, the next input symbol is a and the parse table entry $M[A, a]$ is blank.

The parser must report the presence of error precisely and recover from it, to continue parsing the rest of the input. Predictive parser uses panic mode and phrase level error recovery modes.

Panic mode error recovery:

In this method the parser skips the input till it finds a token in the set of synchronizing tokens. The effectiveness of this method depends on the choice of synchronizing set. The set is chosen so that the parser recovers quickly from the error and continues parsing the remaining input. Few guidelines for choosing the synchronizing set are:

1. $Follow(A)$ may be used as a synchronizing set for the non-terminal A. Skip symbols till we get a symbol in $Follow(A)$ when A is on the stack top and pop A from the stack.
2. If a token in the $Follow(A)$ itself is missing in the input, using $Follow(A)$ as synchronizing set may not be enough. The tokens that start the higher-level constructs may be used as the synchronizing tokens for the lower-level constructs.
3. If the symbols in $first(A)$ are added to the synchronizing set for the non-terminal A, it may be possible to resume parsing.

4. If there is a production of the form $A \rightarrow \epsilon$, then this production may be used to pop the non-terminal from the stack.
5. If the terminal on the stack dos not match the current input symbol, pop the terminal, issue an error message “the terminal was inserted” and continue parsing.

Phrase level error recovery:

This strategy is implemented by inserting pointers to error routines in the blank entries of the parse table. These routines may insert, change, delete symbols in the input and issue proper error messages.

2.6 Bottom-Up Parsing

Bottom-up parsing is a technique used in computer science to analyze the structure of a given input string based on a formal grammar. The primary goal of bottom-up parsing is to build a parse tree from the input by applying grammar rules in a "bottom-up" fashion.

In this parsing approach, the parser starts with the individual tokens (words or symbols(terminal)) of the input string and proceeds by repeatedly combining adjacent tokens into larger syntactic structures. It continues this process until it constructs the complete parse tree, representing the hierarchical structure of the input according to the rules of the grammar.

Bottom-up parsing has advantages over top-down parsing approaches as it can handle a broader class of grammars and provides more flexibility in dealing with ambiguity in the input. However, it can be more complex to implement compared to top-down methods, especially due to the construction and maintenance of parser tables.

Some well-known bottom-up parsing algorithms include LR(0), SLR(1), LALR(1), and LR(1). These algorithms differ in terms of the types of grammars they can handle and their efficiency.

Overall, bottom-up parsing is a crucial component in the design and implementation of compilers, interpreters, and other language processing systems, enabling them to understand and process the structure of input languages efficiently and accurately.

Consider the grammar

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

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

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

Figure 2.8 illustrates the steps in the construction of the parse tree in using bottom-up approach for the expression $id * id$.

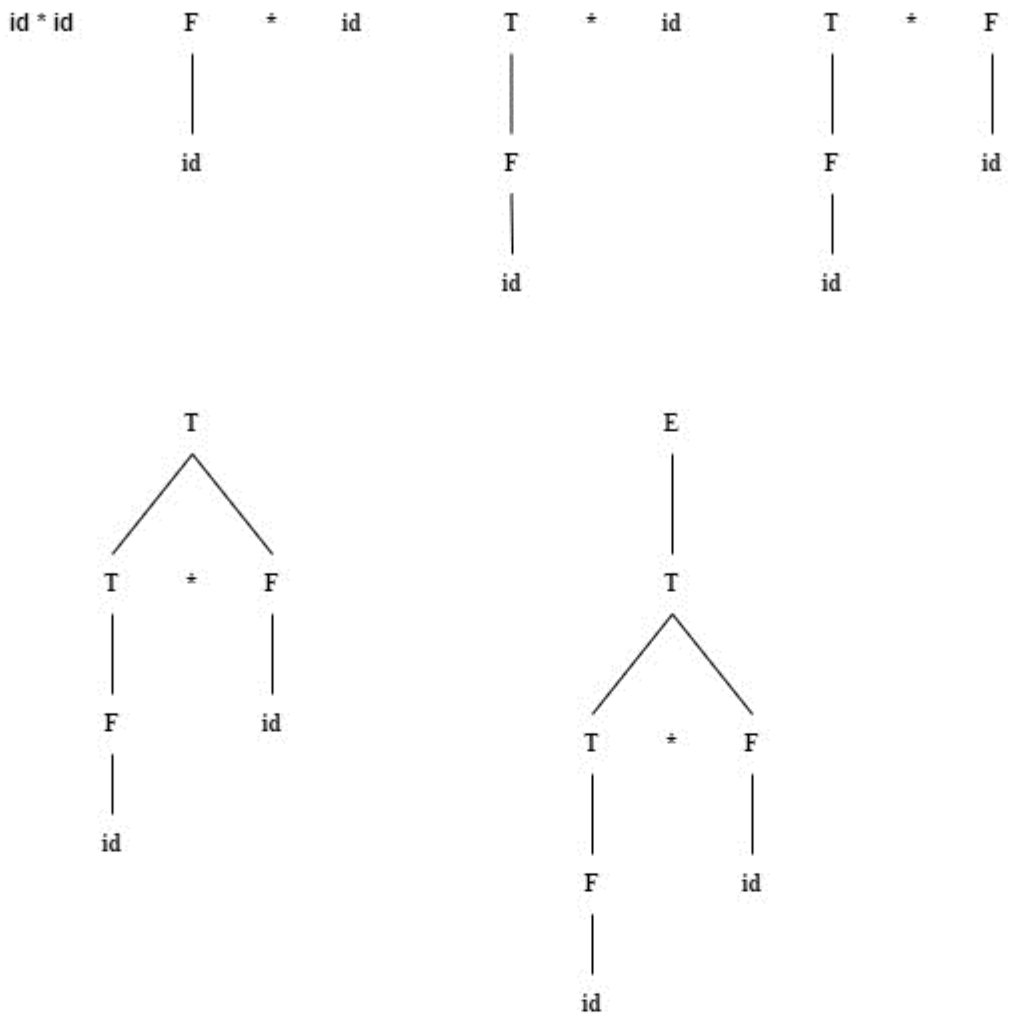


Fig 2.8 Bottom-up parsing for the string $id * id$

2.6.5 Reductions

Bottom-up parsing is the process of reducing the string of terminal symbols "w" to the start symbol of the grammar. At each reduction step a substring matching the body (right hand side) of the production is replaced by the non-terminal on the left-hand side(head) of the production. The key decisions in bottom-up parsing are when to reduce and by which production to reduce.

The steps in the bottom-up parsing in Fig 2.8 illustrate reductions. The reductions correspond to the following sequence of strings.

$$id * id, F * id, T * id, T * F, T, E$$

2.6.6 Handle Pruning

"Handle" is a substring that matches the body of a production and its reduction to the non-terminal on the left of the production represents one step along the reverse of a rightmost derivation. Bottom-up parsing during a left to right scan of the input string constructs the right most derivation in reverse.

The rightmost derivation in reverse can be obtained by handle pruning. Start with a string of terminals $w(\beta_n)$ to be parsed. Locate the handle in the string and replace it by the non-terminal on the left-hand side of the production to get the previous right sentential form β_{n-1}. Repeat this process reducing β_{n-1} to β_{n-2} and so on till the start symbol is reached.

Example 1: Consider the grammar $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$ and the input string $\boldsymbol{id_1 + id_2 * id_3}$. The reduction steps are shown in table 2.7.

Right-Sentential Form	Handle	Reducing Production
$\boldsymbol{id_1 + id_2 * id_3}$	$\boldsymbol{id_1}$	$E \rightarrow id$
$\boldsymbol{E + id_2 * id_3}$	$\boldsymbol{id_2}$	$E \rightarrow id$
$\boldsymbol{E + E * id_3}$	$\boldsymbol{id_3}$	$E \rightarrow id$
$\boldsymbol{E + E * E}$	$\boldsymbol{E * E}$	$E \rightarrow E * E$
$\boldsymbol{E + E}$	$\boldsymbol{E + E}$	$E \rightarrow E + E$

Table 2.7 Reductions to parse the string $\boldsymbol{id_1 + id_2 * id_3}$

2.6.7 Shift Reduce Parsing

Shift-reduce parsing is a type of bottom-up parsing technique used in natural language processing and compiler construction to analyze and parse a sequence of tokens into a valid syntactic structure, often represented by a parse tree or an abstract syntax tree. It is commonly used to implement LR parsers, where "L" stands for left-to-right scanning of the input, and "R" denotes rightmost derivation.

The shift-reduce parsing algorithm maintains a stack and an input buffer. The stack is used to keep track of the partial parse tree, and the input buffer contains the remaining tokens to be processed. The parsing process involves four basic actions: "shift" , "reduce", “accept” and “error”.

1. **Shift:** In a shift operation, the parser moves the next input token from the buffer onto the top of the stack.
2. **Reduce:** If the top elements of the stack match the right-hand side of a production rule, the parser replaces these elements with the non-terminal symbol (left-hand side of the production). This step is done to simplify the stack by collapsing a portion of the parse tree into a single non-terminal node.
3. **Accept:** Announce successful completion of parsing. This happens when the stack contains only the start symbol of the grammar and the input buffer is empty.
4. **Error:** Detect a syntax error and invoke the error recovery routine.

The parsing process continues until the entire input is consumed, and a valid parse tree is constructed at the top of the stack. If the parsing process completes successfully, the input is recognized as a valid sentence according to the grammar rules.

One common way to guide the shift-reduce parser is to use a parsing table, known as an LR table. This table is generated based on the grammar of the language. The table provides instructions to the parser about which action to take (shift or reduce) based on the current state and the next input token.

Shift-reduce parsing is efficient and can handle a wide range of context-free grammars. However, constructing the parsing table can be complex, and the parser may require additional conflict resolution strategies (e.g., precedence rules) to handle ambiguous grammars correctly. Many parser generators, such as Yacc/Bison and ANTLR, use shift-reduce parsing to automatically generate parsers based on a given grammar specification.

Example 2: Illustrate the working of Shift-Reduce(SR) parser to parse the string $\boldsymbol{id_1} + \boldsymbol{id_2} * \boldsymbol{id_3}$, using the grammar,

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

S. No	STACK	INPUT	ACTION
1	$\$$	$\boldsymbol{id_1 + id_2 * id_3}\$$	Shift
2	$\$\boldsymbol{id_1}$	$+\boldsymbol{id_2 * id_3}\$$	Reduce by the production $\boldsymbol{F \rightarrow id}$
3	$\$\boldsymbol{F}$	$+\boldsymbol{id_2 * id_3}\$$	Reduce by the production $\boldsymbol{T \rightarrow F}$
4	$\$\boldsymbol{T}$	$+\boldsymbol{id_2 * id_3}\$$	Reduce by the production $\boldsymbol{E \rightarrow T}$
5	$\$\boldsymbol{E}$	$+\boldsymbol{id_2 * id_3}\$$	Shift
6	$\$\boldsymbol{E} +$	$\boldsymbol{id_2 * id_3}\$$	Shift
7	$\$\boldsymbol{E + id_2}$	$* \boldsymbol{id_3}\$$	Reduce by the production $\boldsymbol{F \rightarrow id}$
8	$\$\boldsymbol{E + F}$	$* \boldsymbol{id_3}\$$	Reduce by the production $\boldsymbol{T \rightarrow F}$
9	$\$\boldsymbol{E + T}$	$* \boldsymbol{id_3}\$$	Shift
10	$\$\boldsymbol{E + T} *$	$\boldsymbol{id_3}\$$	Shift
11	$\$\boldsymbol{E + T * id_3}$	$\$$	Reduce by the production $\boldsymbol{F \rightarrow id}$
12	$\$\boldsymbol{E + T * F}$	$\$$	Reduce by the production $\boldsymbol{T \rightarrow T * F}$
13	$\$\boldsymbol{E + T}$	$\$$	Reduce by the production $\boldsymbol{E \rightarrow E + T}$
14	$\$\boldsymbol{E}$	$\$$	Accept

Table 2.8 Shift-Reduce parsing

2.6.8 Conflicts in Shift-Reduce Parsing

In shift-reduce parsing, conflicts can arise when the parser encounters an ambiguous grammar or an input string that could be parsed in multiple ways, i.e., the current top of the parser stack and the next input symbol are unable to decide which action to perform. There are two main types of conflicts that can occur in shift-reduce parsing: ***shift-reduce conflicts*** and ***reduce-reduce conflicts***.

1. **Shift-Reduce Conflicts:**

Shift-reduce conflicts occur when the parser faces a choice between shifting the next input symbol onto the stack or reducing a portion of the stack to a higher-level construct based on the grammar rules. For example, the ninth action in the table 2.8(S. No: 9), the parser action may be to shift the next input symbol $*$ or reduce by the production $\boldsymbol{E \rightarrow E + T.}$

2. **Reduce-Reduce Conflicts:**

Reduce-reduce conflicts occur when the parser has multiple options to reduce the current stack configuration to different higher-level constructs based on the grammar rules. Precisely the parser has to choose the correct production for the reduce operation among the various alternatives available. For example, the 12th action in table 2.8(S. No: 12) may be to reduce by the production $\boldsymbol{T \rightarrow F}$ or $\boldsymbol{T \rightarrow T * F.}$

2.7 LR Parsing

LR parsing is a type of bottom-up parsing technique used in compilers to analyze the syntax of a given programming language. It stands for "Left-to-right, Rightmost derivation" parsing. The "L" indicates that the input symbols are read from left to right, and the "R" indicates that the parser builds a rightmost derivation in reverse during parsing.

There are different types of LR parsers, such as SLR (Simple LR), LALR (Look-Ahead LR), and LR(1) parsers, which differ in their abilities to handle different types of grammars and the size of their parsing tables. Among them, LALR parsers are the most commonly used in practice as they strike a balance between parsing power and table size.

LR parsers are table driven like LL(predictive) parsers. A grammar for which LR parsing table can be constructed is called ***LR grammar***.

There are several reasons why LR parsing is appealing.

1. LR parsing can be constructed to recognize virtually all programming language constructs.
2. It is the most general shift reduce parsing method.
3. LR parsers detect errors quickly.
4. LR grammars is a super set of LL grammars.

2.7.5 LR(0) items and Automaton

A shift reduce parser maintains the states to keep track of the parsing process. These states help the parser to choose its actions, i.e., shift, reduce, accept or error. A state represents a set of *items*.

LR(0) item:

An LR(0) item is a production in the grammar with a dot(.) at some position in the production body. The following are the LR(0) items corresponding to the production $A \rightarrow XYZ$.

$$A \rightarrow .XYZ$$
$$A \rightarrow X.YZ$$
$$A \rightarrow XY.Z$$
$$A \rightarrow XYZ.$$

Similarly, the production $A \rightarrow a$ yields following LR(0) items.

$$A \rightarrow .a$$
$$A \rightarrow a.$$

A LR(0) item indicates how much of the input is seen by the parser. For example, the item $A \rightarrow .XYZ$ indicates that the parse has to read the input derivable from XYZ. The LR(0) item $A \rightarrow XY.Z$ indicates that the parser has seen the input derivable from XY and is expecting to read the string derivable from Z. Likewise, the LR(0) item $A \rightarrow XYZ.$, indicates the parser has seen the input derivable from XYZ and there is nothing to read next and it is time to reduce XYZ to A.

Set of similar LR(0) items is a state in the parser. For example, the items $A \rightarrow .BC$, $B \rightarrow .b$ are similar. The item $A \rightarrow .BC$ indicates the parser has to read the string derived from BC, i.e., parser is expecting a string derived from B followed by a string derived from C. The string derived from B is the terminal b. Hence the item $B \rightarrow .b$ is identical to the item $A \rightarrow .BC$.

Collection of sets of LR(0) items is called the *canonical* LR(0) collection. It is the set of states for the deterministic finite automaton used to make the parsing decisions. This finite automaton is called the *LR(0) automaton.* To construct the *canonical* LR(0) collection or *LR(0) automaton* , we define the following.

1. Augmented grammar
2. CLOSURE function and
3. GOTO function.

Augmented grammar:

If G is a grammar and S is the start symbol we obtain the augmented grammar G' by adding a new start symbol S' and a new production $S' \rightarrow S$ to G. The purpose of augmenting the grammar is to determine when to stop parsing. The parser stops and announces successful completion of parsing when its action is to reduce by the production $S' \rightarrow S$. The LR(0) item $S' \rightarrow S.$ indicates that the parser has seen the string derived from the start symbol S, i.e., it has read the entire input string. The parser action is "***accept***", only when it is reducing by the production $S' \rightarrow S$.

CLOSURE function:

Let I be the set of LR(0) items. CLOSURE(I) is the set of LR(0) items constructed from I by the following rules.

1. Add every item in I to CLOSURE(I).
2. If $A \rightarrow \alpha.B\beta \in$ CLOSURE(I) and $B \rightarrow \gamma$ is a production in the grammar, then add $B \rightarrow .\gamma$ to CLOSURE(I) if it is not already there. Apply this rule till no new items are added to CLOSURE(I). $A \rightarrow \alpha.B\beta$ indicates that the parse is expecting to read the string derived from $B\beta$, i.e., a string derived from B followed by the string derived from β. Since B derives γ ($B \rightarrow \gamma$), the parser is expecting to read the string derived from γ, followed by the string derived from β. Hence the item $B \rightarrow .\gamma$ is added to CLOSURE(I).

Example 1: Consider the grammar

$E' \rightarrow E$ (Augmenting the grammar)

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

If $I = \{ [E' \rightarrow .E] \}$, construct CLOSURE($I$).

Solution:

1. Add the item $[E' \rightarrow .E]$ to CLOSURE(I). (Rule 1)
2. **a)** Since $E' \rightarrow .E$ is in CLOSURE(I) and $E \rightarrow E + T$ and $E \rightarrow T$ are the productions in the grammar add the items $E \rightarrow .E + T$ and $E \rightarrow .T$ to CLOSURE(I). (Rule2)

   b) Since E$\rightarrow .T$ is in CLOSURE(I) and $T \rightarrow T * F$ and $T \rightarrow F$ are the productions in the grammar add the items $T \rightarrow .T * F$ and $T \rightarrow .F$ to CLOSURE(I).

   c) Since T$\rightarrow .F$ is in CLOSURE(I) and $F \rightarrow (E)$ and $F \rightarrow id$ are the productions in the grammar add the items $F \rightarrow .(E)$ and $F \rightarrow .id$ to CLOSURE(I).

∴ CLOSURE(I) = { $[E' \rightarrow .E]$, $[E \rightarrow .E + T]$, $[E \rightarrow .T]$, $[T \rightarrow .T * F]$, $[T \rightarrow F]$, $[F \rightarrow .(E)]$, $[F \rightarrow .id]$ }

Algorithm 2.5 computes the CLOSURE(I).

```
SetOfItems CLOSURE($I$) {
     $J = I$;
     repeat
          for ( each item $A \rightarrow \alpha.B\beta$ in $J$)
               for(each production $B \rightarrow \gamma$ of $G$)
                    if ($B \rightarrow .\gamma$ is not in $J$)
                         add $B \rightarrow .\gamma$ to $J$;
     until no more items can be added to $J$;
     return $J$;
}
```

Algorithm 2.5 Computing CLOSURE(I)

GOTO function:

Let I be a set of LR(0) items and X be a grammar (terminal or non-terminal) symbol. The function GOTO(I, X) is defined as follows;

$$\text{if } A \rightarrow \alpha.X\beta \in I, \text{ then } GOTO(I, X) = CLOSURE([A \rightarrow \alpha X.\beta])$$

Example 2: If $I = \{ [E' \rightarrow E.], [E \rightarrow E.+T] \}$ then GOTO($I, +$) is defined as

$GOTO(I, +) = CLOSURE([E \rightarrow E+.T])$

$= \{ [E \rightarrow E+.T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id] \}$

We are now ready to construct the LR(0) automaton, i.e., the collection of sets of LR(0) items, also called the canonical collection of sets of LR(0) items. This automaton directs the actions of a LR parser, i.e., indicates when to shift, when to reduce and by what production to reduce.

```
void items($G'$){
     $C = CLOSURE([S' \rightarrow .S])$;
     repeat
          for (each set of items in $C$)
               for (each grammar symbol $X$)
                    if ($GOTO(I,X)$ is not empty and not in $C$)
                         add $GOTO(I,X)$ to $C$;
     until no new sets of items are added to $C$;
}
```

Algorithm 2.6 Computing the canonical collection of sets of LR(0) items

Example 3: Construct the Canonical collection of sets of LR(0) items, i.e., the LR(0) automaton for the below grammar.

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

Solution:

Augment the grammar, by adding a new non-terminal E' and a new production $E' \to E$. The grammar is

$E' \to E$ (Augmenting the grammar)

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

1. The start state of the automaton is $CLOSURE([E' \to .E])$. This state is represented by I_0. It corresponds to the parser configuration in which it has not seen the input, i.e., it has to read the string derivable from E.

   $\therefore I_0 = CLOSURE([E'.E])$

   $= \{[E' \to .E], [E \to .E + T], [E \to .T], [T \to .T * F], [T \to .F], [F \to .(E)], [F \to .id]\}$

2. In state I_0, the automaton is expecting to read the strings derivable from the grammar symbols $\boldsymbol{E}, \boldsymbol{T}, \boldsymbol{F}$, (and $\boldsymbol{id}$. Hence, we compute the states reachable from I_0 on these grammar symbols by computing $GOTO(I_0, \boldsymbol{E}), GOTO(I_0, \boldsymbol{T}), GOTO(I_0, \boldsymbol{F}), GOTO(I_0, (\,), GOTO(I_0, \boldsymbol{id})$. The $GOTO$ function may compute new states or the already existing states. New states are added to the $LR(0)$ automaton.

3. Repeat step 2 for each of the states in the $LR(0)$ automaton till no new states are added.

The resulting $LR(0)$ automaton is shown in Fig 2.9.

The $LR(0)$ automaton helps in making the shift-reduce decisions.

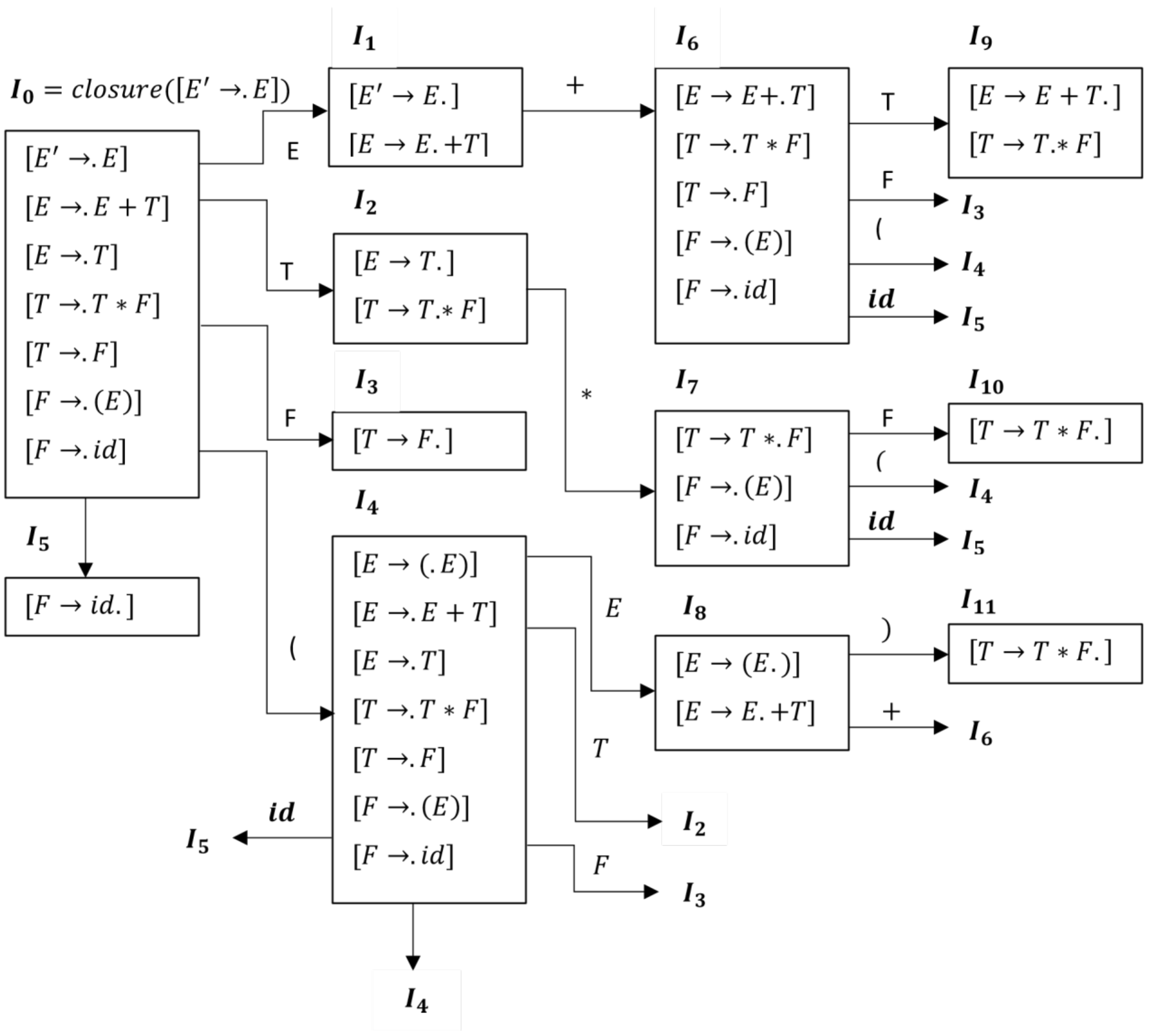


Fig 2.9 LR(0) Automaton for the expression grammar

2.7.2 The LR parsing Algorithm

Fig 2.10 shows the schematic of the LR parser. It consists of an input, an output, a stack, the parsing algorithm and the parsing table. The parsing algorithm is the same for all the LR parsers. Only the parsing table is different for different parsers.

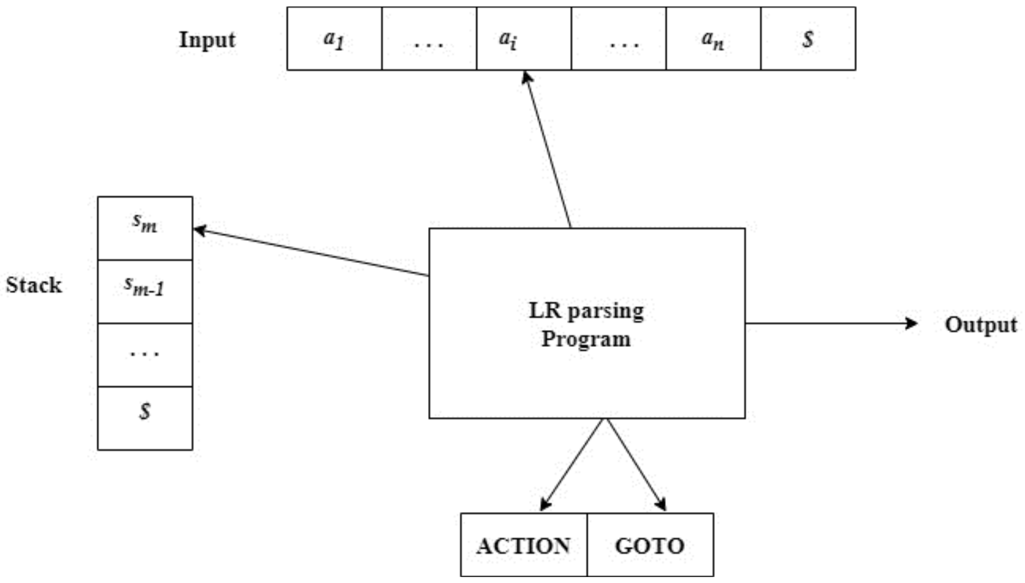


Fig 2.10 The LR parser

The input is the string $a_1a_2 \cdots a_n$ to be parsed. Stack holds a sequence of states $s_0s_1 \cdots s_m$, with s_m at the top. Initially the stack is empty, i.e., it contains only the symbol $, that marks the bottom of the stack. When the shift reduce parser has to shift a symbol the LR parser will shift a state on the stack. In an LR parser, states represent the progress of the parser as it scans the input string.

2.7.3 Structure of the LR Parsing Table

The parsing table consists of two parts.

1. The parsing-action function ACTION:

   The ACTION function takes the state $\boldsymbol{i}$ and the input symbol(a terminal) $\boldsymbol{a}$ as inputs. The value of $ACTION[i, a]$ is one of the following

   a. *Shift* $\boldsymbol{j}$: The parser shifts the state $\boldsymbol{j}$ on to the stack, i.e., in effect push the symbol $\boldsymbol{a}$.

   b. *Reduce* by the production $\boldsymbol{A} \rightarrow \boldsymbol{\beta}$: The parser reduces the string $\boldsymbol{\beta}$ on the stack top to the non-terminal $\boldsymbol{A}$, i.e., pop the string $\boldsymbol{\beta}$ and push the non-terminal $\boldsymbol{A}$.

   c. *Accept*: The parser accepts the input string and stops parsing.

   d. *Error*: The parser detects an error in the input.

2. The go-to function GOTO: It is defined on sets of items and represents a transition on a non-terminal. If $GOTO[I_i, A] = I_j$ then the automaton makes a transition from state I_i to state I_j on the non-terminal A.

2.7.4 Constructing the SLR Parsing Table

The procedure to construct SLR parsing table is as follows

1. Augment the given grammar G, by adding a new start symbol S' and a new production $S' \rightarrow S$. Let the augmented grammar be G'.

2. Construct C, the canonical collection of sets of $LR(0)$ items, i.e., the $LR(0)$ automaton.

3. The $ACTION$ and $GOTO$ entries in the parse table are constructed using the following algorithm(algorithm 2.7).

```
INPUT: The augmented grammar $G'$.
OUTPUT: The SLR parsing table functions ACTION and GOTO for $G'$.
METHOD:
    1. Construct $C = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of $LR(0)$ items.
    2. State $i$ is constructed from $I_i$. The parsing actions for state $i$ are determined
       as follows:
        a. If $[A \rightarrow \alpha . a\beta] \in I_i$ and $GOTO(I_i, a) = I_j$, then set $ACTION[i, a] = shift\, j$.
        b. If $[A \rightarrow \alpha .] \in I_i$, then set $ACTION[i, a] = reduce\ by\ A \rightarrow \alpha$, for all $a \in$
           $FOLLOW(A)$.
        c. If $[S' \rightarrow S.] \in I_i$ then set $ACTION[i, \$] = "accept"$.
       If any conflicting actions result by following the above rules the grammar
       is not SLR(1).
    3. The GOTO part of the parse table is defined as: if $GOTO(I_i, A) = I_j$, then set
       $GOTO[i, A] = j$.
    4. Entries bot defined by rules 2 and 3 above are marked “error”.
    5. The initial state of the parser is $I_0 = CLOSURE([S' \rightarrow .S])$.
```

Algorithm 2.7 Constructing the SLR(1) parsing table

Example 4: Construct the SLR parsing table for the expression grammar

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

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

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

Solution: The SLR parse table(Table 2.9) is constructed for the above grammar using algorithm 2.7.

STATE	ACTION						GOTO		
	id	$+$	$*$	$($	$)$	$\$$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				$accept$			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	4
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Table 2.9 The SLR parsing table for the expression grammar

3.7.4 LR-Parser Configuration

The behavior of an LR parser is described by its state. The state consists of the current contents of the stack and the remaining input. Thus, the configuration of an LR parser is a pair:

$$(s_0 s_1 \cdots s_m, a_i a_{i+1} \cdots a_n \$)$$

Where the first component is the stack contents and the second component is the remaining input. The string $s_0 s_1 \cdots s_m$ on the stack is the sequence of the states that are shifted to the stack during parsing, with s_0 at the bottom and s_m at the top. Instead of grammar symbols the parser shifts the states on the stack. However, the grammar symbols can be inferred from the states. We use the notation X_i to represent the grammar symbol corresponding to the state s_i. The state s_0, the start state does not represent a grammar symbol.

The configurations after the parser actions are as follows:

1. If ACTION$[s_m, a_i] = shift\ s$ then the parser shifts the state s on the stack and the resulting configuration is:

$$(s_0 s_1 \cdots s_m s, a_{i+1} \cdots a_n \$)$$

   (It corresponds to shifting the current input symbol a_i on the stack)

2. If ACTION$[s_m, a_i] = reduce\ by\ A \to \beta$, then the parser pops the states corresponding to the string β and pushes the state corresponding to the non-terminal A. If the length of the string β is r, i.e., $|\beta| = r$, then the parser

   a. Pops r states from the stack. Now the stack contents are $s_0 s_1 \cdots s_{m-r}$ with s_{m-r} at the top

   b. Computes $s = GOTO[s_{m-r}, A]$ and

   c. Shifts the new state s(computed in step 2a above) onto the stack.

   The parser configuration now is:

$$(s_0 s_1 \cdots s_{m-r} s, a_i a_{i+1} \cdots a_n \$)$$

3. If ACTION$[s_m, a_i] = accept$, parsing is completed.

4. If ACTION$[s_m, a_i] = error$, the parser has detected an error and invokes an error recovery routine.

```
The LR Parsing algorithm:
INPUT: Input string $w$, LR parsing table.
OUTPUT: If $w \in L(G)$, then reduction steps else an error.
Method:
initially the parser stack contains $s_0$
input buffer contains the string $w\$$
let $a$ be the first symbol of $w\$$
while(1) {
     let $s$ be the state on the stack top;
     if ( $ACTION[s,a] = shift\ t$ ) then {
          shift t onto the stack;
          let $a$ be the next input symbol;
     }
     else if ($ACTION[s,a] = reduce\ by\ A \to \beta$) then {
            {
                let $r = |\beta|$, i.e., length $\beta$;
                pop $r$ symbols from the stack;
                let $t$ be the state on the stack top;
                push $GOTO[t, A]$ on to the stack;
                output the production $A \rightarrow \beta$;
            } else if $ACTION[s, a] = accept$) break;
                    else call error recovery routine;
}
```
Algorithm 2.8 The LR parsing algorithm

Example 5: Parse the string $id * id + id$ using the SLR parsing algorithm(algorithm 2.8) and the SLR parse table(Table 2.9).

Solution:

	STACK	SYMBOLS (Corresponding to states on stack)	INPUT	ACTION
1.	0		$id * id + id\$$	$ACTION[0, id] = s_5$, shift and goto state 5
2.	0 5	id	$* id + id\$$	$ACTION[5,*] = r_6$, i.e., reduce by $F \rightarrow id$
3.	0 3	F	$* id + id\$$	$ACTION[3,*] = r_4$, i.e., reduce by $T \rightarrow F$
4.	0 2	T	$* id + id\$$	$ACTION[2,*] = s_7$, i.e., shift and goto state 7
5.	0 2 7	$T *$	$id + id\$$	$ACTION[7, id] = s_5$, i.e., shift and goto state 5
6.	0 2 7 5	$T * id$	$+id\$$	$ACTION[5, +] = r_6$, i.e., reduce by $F \rightarrow id$
7.	0 2 7 10	$T * F$	$+id\$$	$ACTION[10, +] = r_3$, i.e., reduce by $T \rightarrow T * F$
8.	0 2	T	$+id\$$	$ACTION[2, +] = r_2$, i.e., reduce by $E \rightarrow T$
9.	0 1	E	$+id\$$	$ACTION[1, +] = s_6$, i.e., shift and goto state 6
10.	0 1 6	$E +$	$id\$$	$ACTION[6, id] = s_5$, i.e., shift and goto state 5
11.	0 1 6 5	$E + id$	$\$$	$ACTION[5, \$] = r_6$, i.e., reduce by $F \rightarrow id$
12.	0 1 6 3	$E + F$	$\$$	$ACTION[3, \$] = r_4$, i.e., reduce by $T \rightarrow F$
13.	0 1 6 9	$E + T$	$\$$	$ACTION[9, \$] = r_1$, i.e., reduce by $E \rightarrow E + T$
14.	0 1	E	$\$$	$ACTION[1, \$] = accept$, accept the input string

Table 2.10 Moves of the SLR parser

Example 6: Construct SLR parse table for the below grammar

$$S \to L = R \mid R$$
$$L \to * R \mid id$$
$$R \to L$$

Solution:

Augment the grammar

$$S' \to S$$
$$S \to L = R \mid R$$
$$L \to * R \mid id$$
$$R \to L$$

Construct the canonical collection of sets of LR(0) items

$\boldsymbol{I_0}$:

$$S' \to .S$$
$$S \to .L = R$$
$$S \to .R$$
$$L \to .* R$$
$$L \to .id$$
$$R \to .L$$

$\boldsymbol{I_1}$:

$$S' \to S.$$

$\boldsymbol{I_2}$:

$$S \to L. = R$$
$$R \to L.$$

$\boldsymbol{I_3}$:

$$S \to R.$$

$\boldsymbol{I_4}$:

$$L \to *.R$$
$$R \to .L$$
$$L \to .* R$$
$$L \to .id$$

$\boldsymbol{I_5}$:

$$L \to id.$$

$\boldsymbol{I_6}$:

$$S \to L =.R$$
$$R \to .L$$
$$L \to .* R$$
$$L \to .id$$

$\boldsymbol{I_7}$:

$$L \to * R.$$

$\boldsymbol{I_8}$:

$$R \to L.$$

$\boldsymbol{I_9}$:

$$S \to L = R.$$

$\therefore C = \{I_0, I_1, \cdots, I_9\}$

Consider the state I_2:

a. The item $S \rightarrow L. = R$ is in I_2.

   $\therefore ACTION[2, =] = shift\ 6$

b. The item $R \rightarrow L.$ Is in I_2 and $FOLLOW(R) = \{\$, =\}$

   $\therefore ACTION[2, =] = reduce\ by\ R \rightarrow L.$

When the parser is in state 2 and the next input symbol is $=$, it may either shift or reduce by the production $R \rightarrow L$. Hence the parser is in a shift reduce conflict. $\therefore$ The grammar is not SLR(1).

2.8 More Powerful LR Parsers

These parsers can handle a wider range of grammars. However, the construction of parsing tables can be computationally expensive and may lead to large parsing tables. There are two methods:

1. ***The canonical LR parser:*** This method makes use of the lookahead symbols.
2. ***The Look Ahead or LALR parser:*** This method also makes use of the lookahead symbols and has fewer states compared to the canonical parser.

2.8.1 Canonical LR(1) items

Canonical LR(1) items are used in the construction of Canonical LR(1) parsers. They represent the states of the parsing automaton and capture the progress of parsing. Each item consists of a production rule, a dot indicating the current position in the rule, and a lookahead token that specifies the context in which the production might be applied.

The general form of an LR(1) item is $[A \rightarrow \alpha.\beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal symbol. The 1 in LR(1) refers to the length of the second component, called the lookahead. The first component $A \rightarrow \alpha.\beta$ is same as in SLR parser. It represents the parser state indicating the parser has seen the string derived from α and is expecting to read the string derivable from β.

- If $\beta \neq \varepsilon$ the lookahead(second component) has no effect.
- If $\beta = \varepsilon$, i.e., the item is of the from $[A \rightarrow \alpha., a]$, then the parser should reduce by the production $A \rightarrow \alpha$, if the next input symbol is a.

2.8.2 Constructing LR(1) sets of items(LR(1) automaton)

The method is similar to constructing the canonical collection of sets of LR(0) items(LR(0) automaton).

We need to only modify the procedures CLOSURE and GOTO.

```
SetOfItems $CLOSURE(I)$ {
    repeat
        for ( each item $[A \rightarrow \alpha.\beta, a]$ in $I$)
            for ( each production $B \rightarrow \gamma$ in $G'$ )
                for ( each terminal $b$ in $FIRST(\beta a)$ )
                    add $[B \rightarrow .\gamma, b]$ to $I$;
    until no more items are added to $I$;
return($I$);
}
```

Algorithm 2.9 CLOSURE function for LR(1) items

```
SetOfItems $GOTO(I, X)$ {
    initialize $J = \emptyset$, i.e., an empty set;
    for ( each item $[A \rightarrow \alpha.X\beta, a]$ in $I$ )
        add item $[A \rightarrow \alpha X.\beta, a]$ to $J$;
    return $CLOSURE(J)$;
}
```

Algorithm 2.10 GOTO function for LR(1) items

```
void $items(G')$ {
    initialize $C$ to $CLOSURE(\{[S' \rightarrow .S, \$]\})$;
    repeat
        for ( each set of items $I$ in $C$ )
            for ( each grammar symbol $X$ in $G$ )
                if ( $GOTO(I, X)$ is not empty and not in $C$ )
                    add $GOTO(I, X)$ to $C$;
    until no new sets of items are added to $C$;
}
```

Algorithm 2.11 Constructing LR(1) automaton

Example 1: Construct the LR(1) automaton for the below grammar.

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

Solution:

Augment he grammar

$$S' \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

The Fig 2.11 shows the LR(1) automaton.

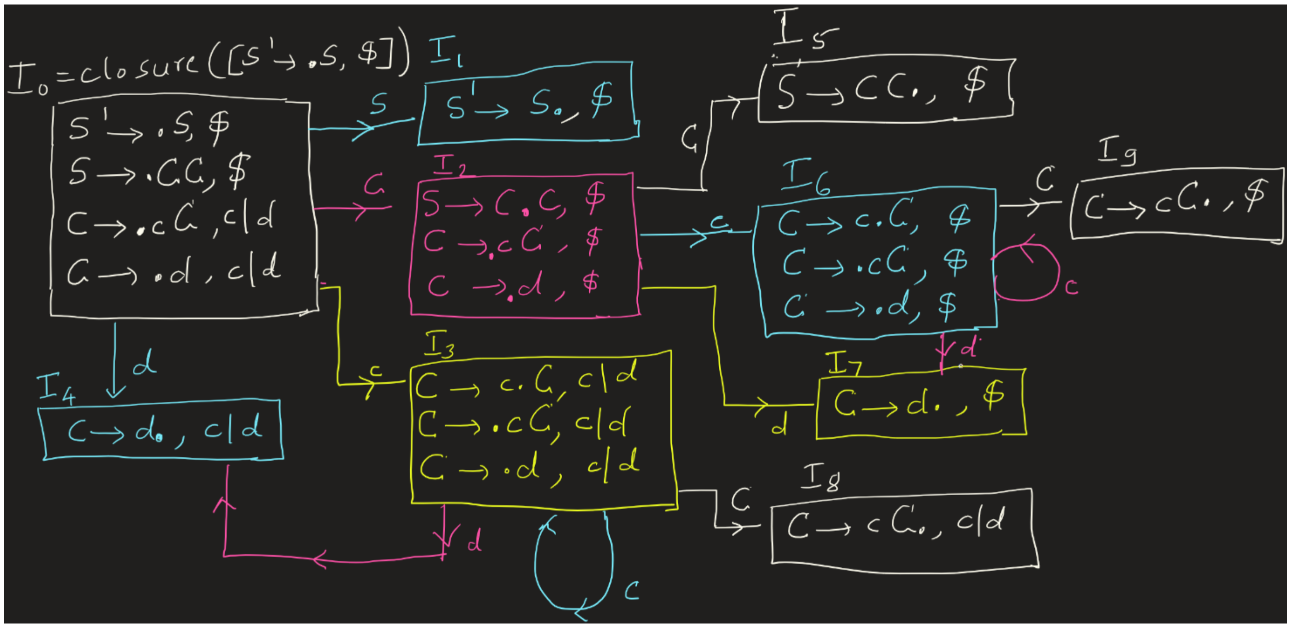


Fig 2.11 LR(1) automaton

2.8.3 Constructing Canonical LR(1) Parsing Table

The structure of the LR(1) parse table is same as that of SLR parse table. Also, the parsing procedure is same as SLR parsing. However, the procedure to construct the table makes use of the LR(1) automaton.

INPUT: The augmented Grammar.

OUTPUT: The canonical LR parsing table functions ACTION and GOTO.

METHOD:

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the canonical collection of sets of LR(1) items.
2. State i of the parser(LR(1) automaton) corresponds to the set I_i. The parsing actions for state i are defined as follows.
    a. If $[A \rightarrow \alpha.a\beta, b] \in I_i$ then set $ACTION[i,a] = shift\, j$ where $I_j = GOTO(I_i, a)$.
    b. If $[A \rightarrow \alpha., a] \in I_i$, then set $ACTION[i,a] = reduce\ by\ A \rightarrow \alpha$.
    c. If $[S' \rightarrow S., \$] \in I_i$ then set $ACTION[i,\$] = accept$.

    If any conflicting actions result from these rules, the grammar is not LR(1).
3. If $GOTO(I_i, A) = I_j$ then set $GOTO[i,A] = j$.
4. All entries not defined by rules 2 and 3 are marked as "error".
5. The initial state is $I_0 = CLOSURE[S' \rightarrow .S, \$]$.

Algorithm 2.12 Constructing LR(1) parse table

Example 2: Construct the LR(1) parse table for the augmented grammar

$S' \rightarrow S$

1) $S \rightarrow CC$

2) $C \rightarrow cC$

3) $C \rightarrow d$

The LR(1) parse table(Table 2.11) is constructed using the LR(1) automaton(**Fig 2.11**) and algorithm 2.12.

STATE	ACTION			GOTO	
	c	d	$\$$	S	C
0	s_3	s_4		1	2
1			$accept$		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Table 2.11 Canonical parse table (LR(1))

The entries r_i in the table parse table 2.11 indicate reduce by the production number i. For example, $ACTION[8, d] = r_2$, i.e., reduce by production 2($C \rightarrow cC$).

2.8.4 Constructing LALR parser

LALR (Look-Ahead LR) parser is a type of bottom-up parser often used in practice since the tables obtained by this method are considerably smaller than the canonical LR tables. Most common syntactic constructs of programming languages can be parsed by LALR parser. The SLR parsing tables also have a smaller number of states but cannot handle few programming language constructs.

For example, the SLR and LALR parsers have several hundreds of states for a language like ‘C’, whereas the canonical parser has several thousands of states.

Consider the LR(1) automaton(Fig 2.11) for example 2 in the section 2.8.3. The states I_4 and I_7 look similar. Both these states have the same first component $C \rightarrow d.$, but differ in the second components. The second component for the state I_4 is $c|d$, i.e., in state I_4 the lookahead is c or d, whereas the second component for the state I_7 is \$(look ahead operator). The states I_4 and I_7 can be therefore replaced by their union I_{47}. The state I_{47} consists of the items $[C \rightarrow d., c|d|\$]$. In the parse table all references to the states I_4 and I_7 are to be replaced by the new state I_{47}. Also, the states I_3 , I_6 can be merged into a new state I_{36} and the states I_8 , I_9 can be merged into a new state I_{89}.

LALR Table construction:

INPUT: An augmented Grammar G'

OUTPUT: The LALR parsing table functions $ACTION$ and $GOTO$

METHOD:

1. Construct $C = \{I_0, I_1, \cdots, I_n\}$ the canonical collection of sets of items.
2. For each core(first component of the LR(0) item) among the set of LR(1) items, find all sets having the same core. Replace these sets by their union.
3. Let $C' = \{J_0, J_1, \cdots, J_m\}$ be the resulting set of LR(1) items. The parsing actions for state i are determined as in canonical LR parser.
4. The $GOTO$ function is defined as in canonical LR parser.

Algorithm 2.13 Constructing LALR parse table.

The table 2.12 shows the LALR parse table for example 2 in the section 2.8.3.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s_{36}	s_{47}		1	2
1			$accept$		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Table 2.12 LALR parse table

2.9 Using Ambiguous Grammars

The ambiguous grammars are not LR grammars. However, some ambiguous grammars are useful in the implementation of programming languages. For example, the ambiguous grammar for arithmetic expression provides a shorter and more natural specification. Also, the ambiguous grammars are useful in separating the common syntactic structures for optimization purposes. Although the grammars are ambiguous disambiguating rules may be added so that the grammar generates unique parse tree for any valid sentence.

2.10 Parser Generators

Parser generators are software tools that automate the process of generating parsers for parsing formal languages. These tools take a formal grammar(CFG) or a specification of a language as input and produce the parser. In this section we shall discuss the LALR parser generator YACC(yet Another Compiler- Compiler).

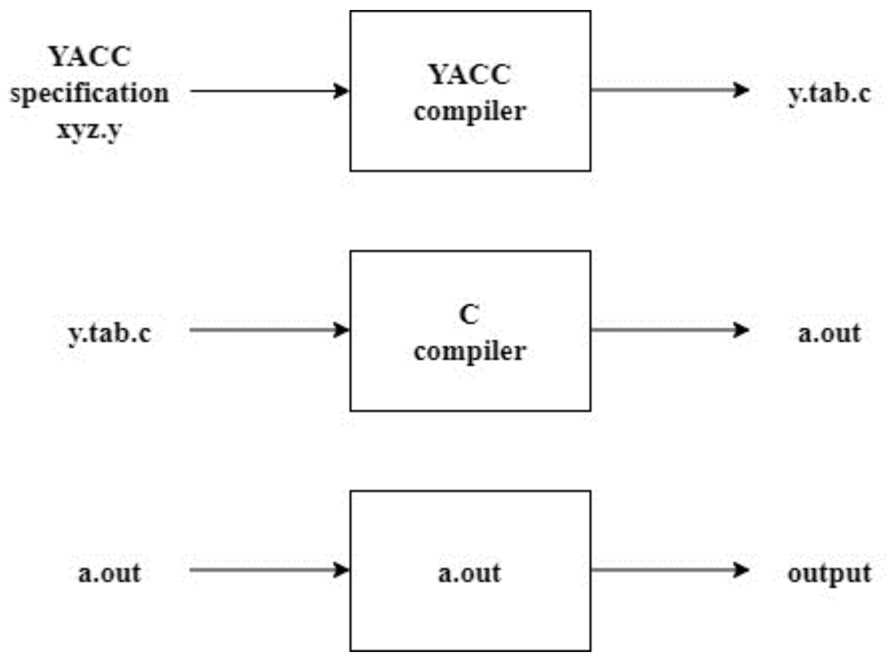


Fig 2.xx Creating the parser with YACC

The file **xyz.y** contains the YACC specification, i.e., the grammar rules along with the actions to be performed. The command `yacc xyz.y` transforms the file **xyz.y** into a c program called **y.tab.c**(the LALR parser). The command `cc y.tab.c -ly` compiles the 'C' program y.tab.c generating the object program **a.out**. The program **a.out** performs the specified operation .

The YACC source program has three parts:

```
declarations
%%
translation rules
%%
supporting C routines
```

Declaration part:

It consists of two sections:

- First section has 'C' declarations, delimited by **%{** and **%}**. It consists of declarations of temporaries and header file inclusion directives.
- Second section has declaration of grammar tokens. The format is %token token-name.

The declaration part ends with **%%** .

The translation Rules Part:

This section contains the translation rules. Each translation rule consists of a production and an optional semantic action part. A production of the form $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ is written as:

$$\begin{aligned} A :\ & \alpha_1 \ \{\langle semantic\ action\rangle_1 \} \\ \mid\ & \alpha_2 \ \{\langle semantic\ action\rangle_2 \} \\ \vdots\ & \\ \mid\ & \alpha_n \ \{\langle semantic\ action\rangle_n \} \\ ;\ & \end{aligned}$$

The $\langle semantic\ action\rangle_i$ is a set of 'C' statements. The symbol $$ refers to the attribute of the non-terminal which the head of the production. The symbol $$i$ refers to the attribute value of the i^{th} grammar symbol in the production body. This second section ends with **%%** .

The Supporting 'C' Routines Part:

This section consists of the supporting 'C' routines. The lexical analyzer by the name `yylex()` must be provided. This routine produces tokens consisting of a token name and an attribute for the token. The token name must be specified in the first section of a YACC program. The attribute value of the token is communicated through the global variable `yylval`.

2.12 Multiple Choice Questions

Q1. Primary purpose of error handling in the syntax analysis phase of a compiler is:

a. To correct all detected errors automatically
b. To report and recover from errors gracefully
c. To terminate the compilation process immediately
d. To ignore errors and proceed with code generation

Q2. The purpose of error recovery in syntax analysis to

a. To correct all syntax errors automatically
b. To continue parsing after detecting an error
c. To terminate the compilation process immediately
d. To ignore errors and proceed with code generation

Q3. The role of a panic mode recovery in syntax analysis is:

a. Ignore all errors and proceed with code generation
b. Discard the erroneous part and resume parsing at a predefined point
c. Terminate the compilation process immediately
d. Correct syntax errors automatically

Q4. The primary purpose of Context-Free Grammars (CFGs) in the context of programming languages is to:

a. Specify lexical rules
b. Define the syntax of programming languages
c. Describe the meaning of programming constructs
d. Handle runtime errors

Q5. Which of the following represents a terminal symbol in a CFG?

a. A symbol that can be replaced by a sequence of other symbols
b. A symbol that cannot be expanded further
c. A symbol that represents a variable or placeholder
d. A symbol that appears only at the beginning of a production rule

Q6. What is the purpose of non-terminal symbols in a CFG?

a. Represent actual values in the programming language

b. Indicate the end of a production rule

c. Represent variables or placeholders for language constructs

d. Define the order of execution of statements

Q7. Which of the following statements is true about the start symbol in a CFG?

a. It must be a terminal symbol
b. It is the symbol used to begin a production rule
c. It represents the initial state of the parsing process
d. It is optional and can be omitted in CFGs

Q8. In the context of CFGs, what does a derivation represent?

a. A sequence of production rules used to generate a string
b. The process of replacing non-terminal symbols with terminal symbols
c. The order of execution of statements in a program
d. The set of all valid strings in the language

Q9. The parse tree in the context of compiler design is:

a. A tree data structure used for code generation
b. A tree structure representing the syntactic structure of a program
c. A tree structure used for lexical analysis
d. A tree structure representing semantic errors

Q10. In a parse tree, what do the leaves represent?

a. Non-terminal symbols
b. Terminal symbols
c. Production rules
d. Semantic actions

Q11. The root of a parse tree is:

a. The first leaf node
b. The last leaf node
c. The node with the maximum number of children

d. The node representing the start symbol

Q12. An ambiguous grammar is:

a. A grammar that is difficult to understand
b. A grammar with multiple parse trees for a single string
c. A grammar that generates only ambiguous strings
d. A grammar without production rules

Q13. Why is ambiguity in grammars a concern for compiler design?

a. It leads to increased parsing complexity
b. It causes issues in code generation
c. It may result in multiple interpretations of a program
d. It hinders lexical analysis

Q14. Left-recursive grammar is:

a. A grammar that only generates strings with leftmost derivations
b. A grammar where the start symbol is a terminal
c. A grammar where the production rules contain left parenthesis
d. A grammar where a non-terminal can directly derive itself from the left side

Q15. Which of the following is an example of a left-recursive production rule?

a. $A \rightarrow BC$
b. $A \rightarrow aB$
c. $A \rightarrow Aa$
d. $A \rightarrow Ba$

Q16. Left factoring is:

a. A technique to eliminate right recursion in grammars
b. A technique to eliminate left recursion in grammars
c. A technique to simplify grammars by removing the common prefixes
d. A technique to optimize code generation in compilers

Q17. A recursive descent parser is a parser that

a. Only handles left-recursive grammars
b. Recursively descends through the parse tree
c. Uses backtracking for error recovery

d. Generates code without recursion

Q18. Which of the following statements is true about the First set of a grammar symbol?

a. The First set is always a subset of the Follow set
b. The First set is always equal to the Follow set
c. The First set may contain epsilon (ε) if the symbol derives epsilon
d. The First set is only applicable to terminal symbols

Q19. Which of the following is included in the Follow set of a non-terminal A for a grammar production B → αAβ?

a. First(α)
b. First(β)
c. Follow(B)
d. Both First(α) and Follow(B)

Q20. Predictive parser is a parser that

a. Uses lookahead symbols
b. Automatically corrects syntax errors
c. Handles left recursion effectively
d. Generates code without any lookahead

Q21. Which parsing technique is commonly associated with predictive parsers?

a. LL parsing
b. LR parsing
c. SLR parsing
d. LALR parsing

Q22. What is the distinguishing feature of LL(1) parsers?

a. They use left recursion
b. They require exactly one lookahead symbol
c. They use right recursion
d. They do not use any lookahead symbols

Q23. In LL(1) parsing, how is the correct production to apply determined?

a. By choosing the production with the longest right-hand side
b. By choosing the production with the fewest non-terminals in the right-hand side
c. By consulting the parse table based on the current non-terminal and lookahead symbol
d. By using the production that appears first in the grammar

Q24. What does the term "predictive" refer to in the context of predictive parsers?

a. The ability to predict the future behaviour of the program
b. The ability to predict the correct production without backtracking
c. The ability to predict runtime errors in the program
d. The ability to predict the number of parsing steps required

Q25. Which of the following is a requirement for a grammar to be LL(1)?

a. It should be left-recursive
b. It should be right-recursive
c. It should not be ambiguous
d. It should not have epsilon productions

Q26. What is a parse table used for in predictive parsing?

a. To store the intermediate values during parsing
b. To store the symbols in the input string
c. To store the production rules of the grammar
d. To determine the next production to apply based on the current state and lookahead symbol

Q27. In shift-reduce parsing, what does the "shift" operation involve?

a. Moving the input symbol to the output
b. Moving a symbol from the input to the top of the parse stack
c. Reducing the size of the parse stack
d. Removing symbols from the parse stack

Q28. What is the purpose of the "reduce" operation in shift-reduce parsing?

a. To remove symbols from the parse stack
b. To eliminate left recursion in the grammar
c. To replace a sequence of symbols on the parse stack with a non-terminal

d. To discard symbols from the input

Q29. In a shift-reduce parser, when is the "reduce" operation applied?

a. When there are no more symbols in the input string
b. When a specific sequence of symbols on the parse stack matches the right-hand side of a production
c. When there is a conflict in the grammar
d. When the parse stack is empty

Q30. What is the primary challenge in shift-reduce parsing?

a. Handling left recursion in the grammar
b. Determining when to shift and when to reduce
c. Dealing with ambiguity in the grammar
d. Managing the parse stack efficiently

2.13 Summary

The parser, a critical component in compiler design, analyzes program syntax and constructs a parse tree or abstract syntax tree. This tree serves as an intermediary representation, facilitating subsequent compiler phases. Context-free grammars provide a formal description of programming language syntax through production rules, enabling parsers to recognize and generate correct language structures.

Constructing a grammar involves defining language rules, considering constructs, hierarchical structures, and terminal/non-terminal symbols. Top-down parsing begins at the parse tree's top, applying production rules recursively, with algorithms like Recursive Descent and LL(1) suitable for LL(1) grammars.

In contrast, bottom-up parsing builds the parse tree from leaves to root, identifying and reducing input segments that match production rules. LR parsers, including LR(0), SLR(1), LR(1), and LALR(1), efficiently handle various grammars through a shift-reduce approach.

Handling ambiguous grammars, which produce multiple parse trees for the same input, is vital in parser design. Techniques like operator precedence and associativity rules resolve ambiguities, ensuring a unique parse tree.

Parser generators automate parser creation from formal grammar descriptions. The tools YACC takes high-level grammars as input, generating parser code in a specific language. The generator simplifies the implementation of parsers for diverse languages and grammars.

2.13 Keywords

- Parser
- Error Recovery
- Panic Mode Error Recovery
- Phrase Level Error Recovery
- Context Free Grammars
- Derivations
- Parse trees
- Ambiguous Grammars
- Left Recursion
- Left Factoring
- FIRST and FOLLOW sets
- Top-Down Parser
- Recursive Descent Parsing
- Predictive Parsing
- LL(1) Grammars
- Bottom-Up Parsing
- Handle
- SR parsers
- LR Parsers

1.10 Recommended Learning Resources

1. Ellis Horowitz, Sartraj Sahni, S Rajasekaran ., 1998, Fundamentals of Computer Algorithms, 2ndEd., Galgotia Publication.
2. Anany Levitin., 2011, Introduction to The Design and analysis of algorithms, 3rd Ed., Pearson India Publishers.

3. Time and Space, Tab No. 3, Video Lecture URL: https://nptel.ac.in/courses/106106131