DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING AND INFORMATION TECHNOLOGY
YEAR/SEM: III/VI
23CS6401/23IT6401 – COMPILER DESIGN
UNIT II
SYNTAX ANALYSIS
MODULE

Introduction-Context free grammars –Writing a grammar- Top down parser-Bottom up parser-LR parsers- Constructing an SLR parsing table-LALR parser-CLR parser- Error Handling and Recovery in Syntax Analyzer - YACC

2.1 INTRODUCTION

The parser (syntax analyzer) receives the source code in the form of tokens from the lexical analyzer and performs syntax analysis, which create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

Syntax analysis is also called parsing. A typical representation is a abstract syntax tree in which each interior node represents an operation the children of the node represent the arguments of the operation.

2.1.1 The Role of the Parser

- The parser receives a sequence of tokens from the lexical analyzer and checks whether the token string can be generated by the grammar of the source language.
- Its primary function is to detect and report syntax errors clearly and recover from common errors so that compilation can continue.
- For syntactically correct programs, the parser conceptually constructs a parse tree, which represents the grammatical structure of the program and is used for further compilation stages.
- In practice, the parse tree need not be built explicitly; instead, syntax checking and translation actions can be performed during parsing itself.

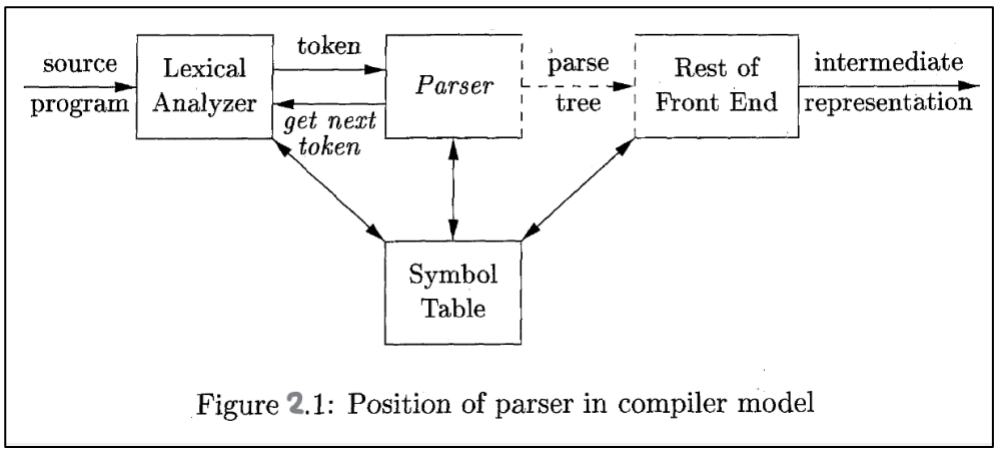


Figure 2.1: Position of parser in compiler model

There are two types of parsers for grammars: top-down and bottom-up.
a. Top-down parser: which builds parse trees from top(root) to bottom(leaves)
b. Bottom-up parser: which builds parse trees from leaves and works up to the root.

- In either case, the input to the parser is scanned from left to right, one symbol at a time.
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Need for Parser

- A parser is needed to detect syntactic errors efficiently.
- An error is detected as soon as the prefix of the input cannot be completed to form a string in the language. This process of analyzing the prefix of input is called the viable-prefix property.

Issues in Parsing

Two important issues in parsing are

- Specification of syntax
- Representation of input after parsing.

- **Specification of syntax** means how to write any programming statement in an exact, unambiguous and complete manner.
- **Representation of input after parsing** is important because all the phases of compiler take information from the parse tree being generated. Information suggested by any programming statement should not be differed after building the syntax tree.

Lexical and syntax analyzer are separated out

The lexical analyzer scans the input program and collects the token from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases.

Separating out these two phases has two advantages:

- Firstly, it speed-up the process of compilation.
- Secondly, the errors in the source input can be identified precisely.

Top-Down Parser Vs Bottom-Up Parser

Table 2.1:Top-Down Parser Vs Bottom-Up Parser

Top-down Parser	Bottom-up Parser
• A parse tree can be built from root to leaves.	• A parse tree can be built from leaves to the root.
• This is simple to implement	• This is complex to implement
• This is a less efficient parsing technique. Various problems that occur during the top-down technique are ambiguity and left recursion.	• When the bottom-up parser handles ambiguous grammar, conflicts occur in the parse table.
• It is applicable to a small class of languages.	• It is applicable to a broad class of languages.
Various parsing techniques are • Reduce descent parser • Predictive parser	Various parsing techniques are ♦ Operator precedence parsing ♦ LR parser ♦ Shift reduce

Parse Tree Vs Syntax Tree

Table 2.2: Parse tree vs Syntax tree

Parse Trees	Syntax Trees
♦ A parse tree is a graphical representation of the replacement process in a derivation.	♦ A syntax tree is nothing but the compact form of a parse tree.
♦ In parse trees, each interior node symbolizes a grammar rule, and each leaf node symbolizes a terminal.	♦ In syntax trees, each interior node symbolizes an operator & each leaf node symbolizes an operand.
♦ Parse trees provide every characteristic information from the real syntax.	♦ Syntax trees do not provide every characteristic information from the real syntax and that is the reason why they are sometimes prefixed with the term abstract. For example- no rule nodes, no parenthesis etc.
♦ For the same language construct, parse trees are comparatively less dense than syntax trees.	♦ For the same language construct, syntax trees are comparatively more denser than parse trees.

2.2 CONTEXT-FREE GRAMMARS

2.2.1 The Formal Definition of a Context-Free Grammar

A context-free grammar *G* is defined by the 4-tuples: G (V, T, P, S), where

- V is a finite set of *non-terminals(variables)*.
- T is a finite set of *terminal*s.
- P is a finite set of production rules of the form A→α.
  where A is a nonterminal and α is a string of terminals and/or non-terminals.
  P is a relation from *V* to(V∪T)*.
- S is the start symbol (variable S∈*V*)

Example:

stmt → **if** (*expr*) *stmt* **else** *stmt* *(2.1)*

Terminals:

- Terminals are basic symbols used to form strings.
- Token name is another term for a terminal.
- Tokens are produced by the lexical analyzer.
- In grammar (2.1), terminals include if, else, (and).

Nonterminals:

- Nonterminals are syntactic variables that represent sets of strings.
- In grammar (2.1), stmt and expr are nonterminals.
- Nonterminals help define the language generated by the grammar.
- They provide a hierarchical structure, which is essential for syntax analysis and translation.

Start Symbol

- One nonterminal is chosen as the start symbol of the grammar.
- The strings derived from the start symbol form the language generated by the grammar.
- By convention, productions of the start symbol are listed first.

Productions

- Productions define how terminals and nonterminals are combined to form strings.

- Each production consists of:
  - ✓ A head (left-hand side), which is a nonterminal.
  - ✓ A production symbol (→ or ::=).
  - ✓ A body (right-hand side) containing zero or more terminals and nonterminals.
- The body specifies one way to construct strings derived from the head nonterminal.

Example 2.1:
*The following grammar defines simple arithmetic expressions. In this grammar, the terminal symbols are id + - * / ( ). The nonterminal symbols are expression, term and factor, and expression is the start symbol.*
Solution:

expression	→	*expression* + *term*
expression	→	*expression* - *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

Figure 2.2: Grammar for simple arithmetic expressions

2.2.2 Notational Conventions

The following notational conventions for grammars can be used

1. These symbols are **terminals**:
   (a) Lowercase letters early in the alphabet, a, b,c..
   (b) Operator symbols such as +,*, etc..
   (c) Punctuation symbols such as parentheses, comma, etc...
   (d) The digits 0,1,. . .,9.
   (e) A single terminal symbol such as (Boldface strings) **id** or **if**.
2. These symbols are non-terminals:
   (a) Uppercase letters early in the alphabet, A, B,C..
   (b) The letter S is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.
3. Upper case letters late in the alphabet, such as *X,Y,Z*, represent either **non-terminals or terminals.**
4. Lowercase letters late in the alphabet, primarily *u, v, . . . , z*, represent strings of terminals.
5. Lowercase Greek letters, **α, β, γ**, represent a generic production can be written as $A \rightarrow \alpha$, where A is the left side of the production and α is the right side of the production.
6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …, $A \rightarrow \alpha_k$ with a common non-terminal, may be written as $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$.
7. Unless stated otherwise, the left side of the first production is the start symbol.

2.2.3 Derivations

- A parse tree can be precisely defined using a derivational view.
- In this view, productions act as rewriting rules.
- Parsing begins with the start symbol.
- Each step replaces a nonterminal with the body of one of its productions.
- This process corresponds to top-down construction of a parse tree.
- The derivational approach is also useful for understanding bottom-up parsing.
- Bottom-up parsing is related to rightmost derivations, where the rightmost nonterminal is rewritten at each step.

Definition:

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. This sequence of replacements is called derivation.

In general, a derivation step is
αAβ ⇒ αγβ is a sentential form, and if there is a production rule A→γ in our grammar.
where α and β are arbitrary strings of terminal and non-terminal symbols α1 ⇒ α2 ⇒ ... ⇒ αn (αn derives from α1 or α1 derives αn). There are two types of derivation.

1. At each derivation step, we can choose any of the non-terminals in the sentential form of G for the replacement.
2. If we always choose the left-most non-terminal in each derivation step, this derivation is called a left-most derivation.

Types of Derivation:

- **In leftmost derivations**, the leftmost nonterminal in each sentential is always chosen. If α⇒ β is a step in which the leftmost nonterminal in αis replaced, we write α $\underset{lm}{\Rightarrow}$
- **In rightmost derivations,** the rightmost nonterminal is always chosen; we write α $\underset{rm}{\Rightarrow}$ β in this case.

Example:

E → E + E | E – E | E * E | E / E | - E
E → (E)
E → id

Leftmost derivation:

E → E + E
→ E * E+E →id* E+E→id*id+E→id*id+id
The string is derived from the grammar w = id*id+id, which consists of all terminal symbols.

Rightmost derivation:

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

Given grammar G : E → E+E | E*E | (E) | - E | id
Sentence to be derived: – (id+id)

LEFTMOST DERIVATION

E → - E
E → - (E)
E → - (E+E)
E → - (id+E)
E → - (id+id)

RIGHTMOST DERIVATION

E → - E
E → - (E)
E → - (E+E)
E → - (E+id)
E → - (id+id)

Language:

L(G) is the language of G (the language generated by G), which is a set of sentences.

Sentence:

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G, then ω is a sentence of L(G) iff S ⇒ ω, where ω is a string of terminals of G.

Context-free Language:

If G is a context-free grammar, L(G) is a context-free language. Two grammars G1 and G2 are equivalent if they produce the same grammar.

Consider the production of the form S ⇒ α, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

Sentinels:

Given a grammar G with start symbol S, if S → α, where α may contain nonterminals or terminals, then α is called the sentinel form of G.

- Strings that appear in the leftmost derivation are called left sentinel forms.
- Strings that appear in the rightmost derivation are called right sentinel forms.

Yield or Frontier of the tree:

Each interior node of a parse tree is a non-terminal. The children of a node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called the yield or frontier of the tree.

4.2.4 Parse Trees and Derivations

- **A Parse tree** is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- Each interior node of a parse tree represents the application of a production.
- The interior node is labelled with the one terminal A in the head of the production; the children of the node are labelled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

- Example, the parse tree for -(id+id) in Fig. 2.3, results from the derivations

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

and

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

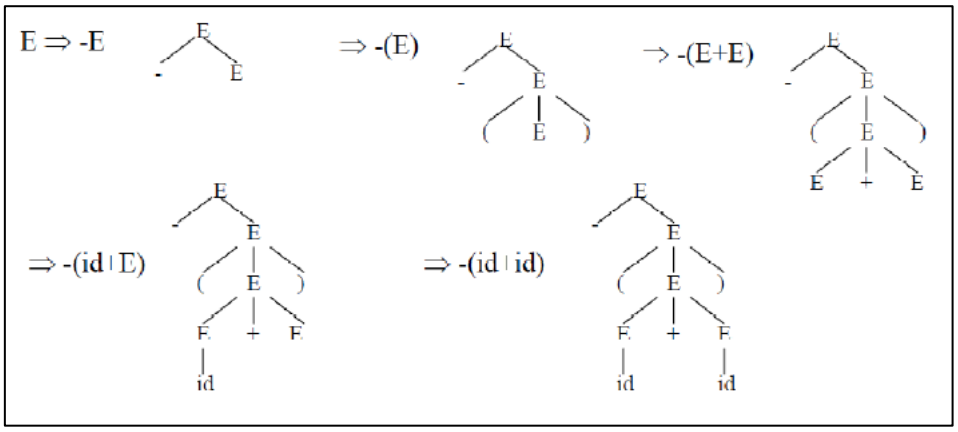


Figure 2.3: Derivation of -(id + id)

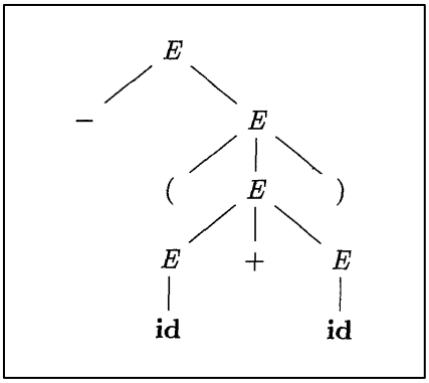


Figure 2.4: Parse tree for -(id + id)

Example 2.2:

*Consider the following grammar E→E+E|E*E|id and derive left most derivation for the string (id + id *id)*

Solution:

w=id+id*id
E→E+E
E→id+E [E→id]
E→id+E*E [E→E*E]
E→id+id*E [E→id]
E→id+id*id [E→id]

Example 2.3:

*Consider the following grammar E→E+E|E*E|idand derive right most derivation for the string (id + id *id)*

Solution:

w=id+id*id
E→E+E
E→E+E*E [E→E*E]
E→E+E*id [E→id]
E→E+id*id [E→id]
E→id+id*id [E→id]

Example 2.4:

Consider the context free grammar (CFG) G = ({S}, {a, b, c}, P, S) where P={S→SbS | ScS | a}. Derive the string "abaca" by leftmost derivation and rightmost derivation.

Solution:

Leftmost derivation for "abaca"

S ⇒ SbS
⇒abS (using rule S □a)
⇒abScS (using rule S □ScS)
⇒abacS (using rule S □a)
⇒abaca (using rule S □a)

Rightmost derivation for "abaca"

S ⇒ ScS
⇒Sca (using rule S □a)
⇒SbSca (using rule S □SbS)
⇒Sbaca (using rule S □a)
⇒abaca (using rule S □a)

Example 2.5:

Consider the following CFG grammar, S→aABe, A →Abc|b, B →d where a,b,c,d are terminals, 'S'(start symbol), A and B are non-terminals.

(a) Parse the sentence "abbcde" using right-most derivation
(b) Parse the sentence "abbcde" using left-most derivation
(c) Draw the parse tree.

Solution:

S →aABe
A →Abc | b
B →d

(a) Parse the sentence "abbcde"using right-most derivation

S → a A B e
→aA d e
→aA b c d e
→a b b c d e

(b) Parse the sentence "abbcde"using left-most derivation

S → a A B e
→aAbcBe
→abbcBe
→abbcde

(c) Draw the parse tree

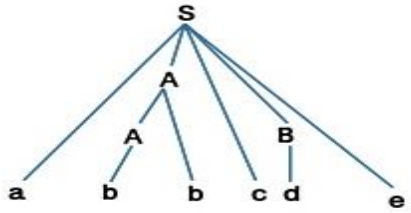


Figure 2.5: *Parse tree for the expression abbcde*

4.2.5 Ambiguity

Definition:

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*. Such a grammar is called *ambiguous grammar*.
- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous.
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous.

Example: Given grammar G : E → E+E | E*E | (E) | - E | id

- The sentence id+id*id has the following two distinct leftmost derivations:

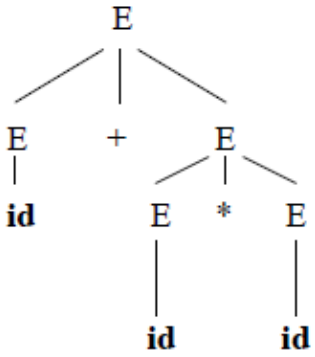
Left Most Derivation 1

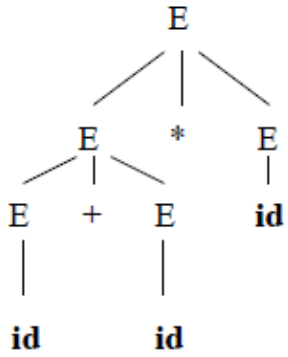
E → E+ E
E → id + E
E → id + E * E
E → id + id * E
E → id + id * id

Left Most Derivation 2

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

The two corresponding parse trees are:





- For the most parsers, the grammar must be unambiguous.
- Unambiguous grammar: Unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.
- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

2.3 WRITING A GRAMMAR

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

We then consider several transformations that could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques - left-recursion elimination and left factoring - are useful for rewriting grammars so they become suitable for top- down parsing.

2.3.1 Eliminating Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

A grammar G is said to be ambiguous if it has more than one parse tree either in LMD or in RMD for at least one string.

Example 2.6:

Consider the grammar,

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

and generate two parse trees for the string id + id – id

Solution:

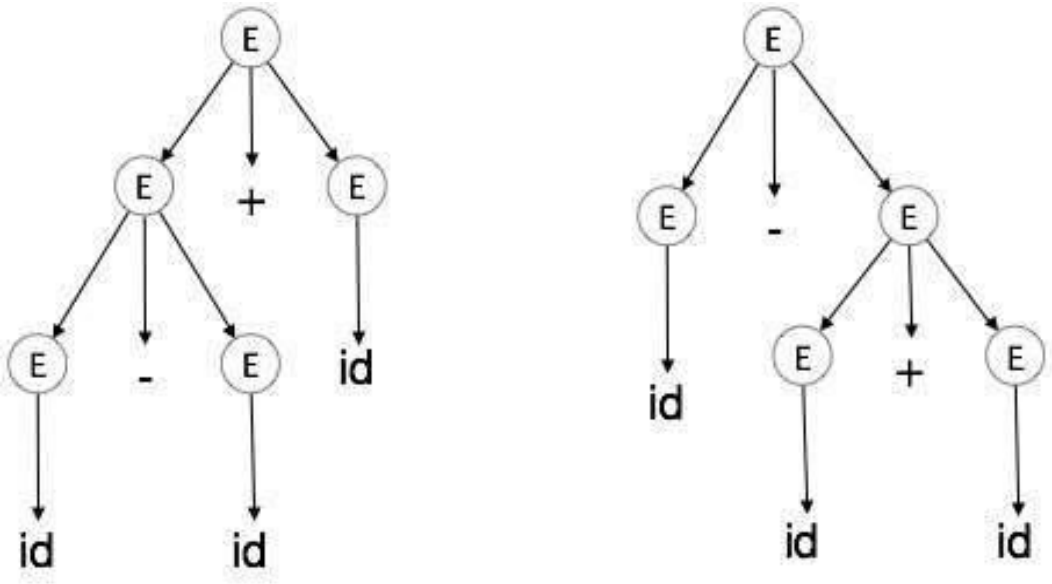


Figure 2.6: Two parse trees for the string id + id – id

Example 2.7:

Consider the grammar with the following translation rules and E as the start symbol.

*E -> E1 #T {E.value = E1.value * T.value}*

| T {E.value = T.value}

T -> T1 & F {T.value = T1.value + F.value}

|F {T.value= F.value}

F ->num {F.value = num.value}

Compute E.value for the root of the parse tree for the expression:2 # 3 & 5 # 6 &4.

a) 200

b) 180

c) 160
d) 40

Explanation:

- We can calculate the value by constructing the parse tree for the expression 2 # 3 & 5 # 6 &4.
- Alternatively, we can calculate by considering following precedence and associativity rules.
- Precedence in a grammar is enforced by making sure that a production rule with higher precedence operator will never produce an expression with operator with lower precedence.
- In the given grammar '&' has higher precedence than '#'. Left associativity for operator * in a grammar is enforced by making sure that for a production rule like S → S1 * S2 in grammar, S2 should never produce an expression with *.
- On the other hand, to ensure right associativity, S1 should never produce an expression with *.
- In the given grammar, both '#' and & are left-associative.
- So expression 2 # 3 & 5 # 6 &4 will become ((2 # (3 & 5)) # (6 & 4))
- Let us apply translation rules, we get ((2 * (3 + 5)) * (6 + 4)) = 160.

2.3.2 Left Recursion

A context free grammar is said to be left recursive if it has a non-terminal A with two productions in the following form:

$$A \rightarrow A\,\alpha \mid \beta$$

where α and β are sequences of terminals and non-terminals that do not start with A.

Left recursion in top-down parsing can enter into infinite loop. It creates serious problems, so we have avoided Left recursion.

For example, in expr →expr + term | term

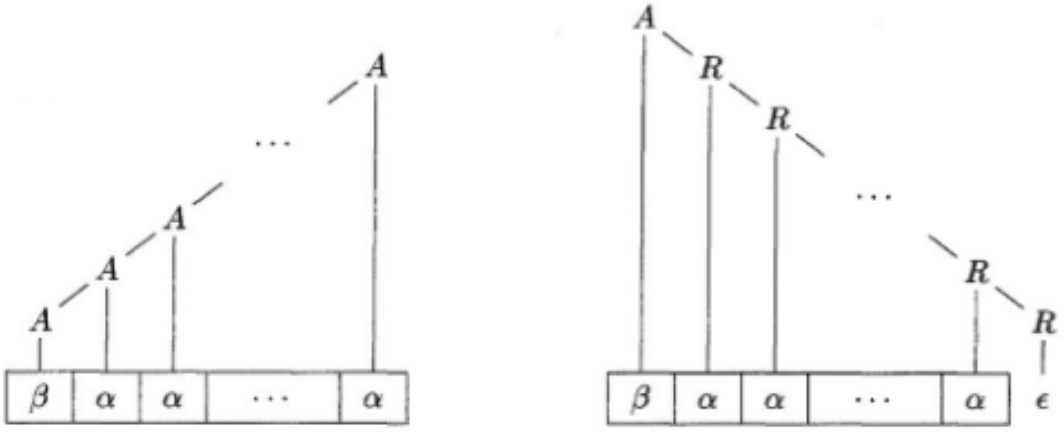


Figure 2.7: Left-recursive and Right recursive ways of generating a string

2.3.2.1 Eliminating Left Recursion

Rule

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Algorithm to eliminate left recursion from a grammar

Algorithm: Eliminating left recursion

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
                productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

Example 2.8:

Eliminate immediate left recursion for the following grammar

*E→E+T | T, T→T * F | F, F→ (E) | id.*

Solution:

The rule to eliminate the left recursion is A→Aα | β can be converted as

A → β A′
A′ → αA′
A′ → ε

Rule 1:

E → E+T | T
A A αβ

We get

E → T E′ (A→β A′)
E′→ +TE′ (A′→αA′)
E′ → ε (A′→ε)

Rule 2:

T→ T * F | F
A A α β

We get

T→FT′ (A → β A′)
T′ → *FT′ (A′ → αA′)
T′ → ε (A′ → ε)

Rule 3:

F → (E) | id is not left recursive. To summarize, we get the following set of rules after eliminating left recursion.

E → T E′
E′ → +TE′ | ε
T → FT′
T′ → *FT′ | ε

F → (E) | id

Example 2.9:
Eliminate left recursion for the following grammar
A → A B d | A a | a
B → B e | b
Solution:
The grammar without left recursion is
A →aA'
A' → B d A' | a A' | ε
B →bB'
B' → e B' |ε

Example 2.10:
Eliminate left recursion from the given grammar A→ A c | A a d | b d | b c
Solution:
After removing left recursion, the grammar becomes,
A→ b d A' | b c A'
A'→c A' | a d A'
A'→ε

2.3.3 Left factoring

- When a production has more than one alternative with common prefixes then it is necessary to make right choice on production.
- To perform Left factoring for the production, A→ $\alpha\beta_1 | \alpha\beta_2$

Rule
A → αA′
A′ → $\beta_1 | \beta_2$

Algorithm: Left factoring a grammar.
INPUT: Grammar G.
OUTPUT: An equivalent left-factored grammar.
METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If a ≠ ε - i.e., there is a nontrivial common prefix - replace all of the A-productions | A → α β_1 |α β_2|….|α β_n|γ where γ represents all alternatives that do not begin with α, by
A → αA' | γ
A' → α β_1 |α β_2|….|α β_n
Here A′ is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Examples 2.11:
Solve the grammar using Left factor: S → iEtS | iEtSeS |a; E → b.
Solution:
Left factoring is removing the common left factor that appears in two productions of the same non-terminal.
The left factored grammar is,
S → iEtSS′| a
S′→ eS | ε

E → b

Example 2.12:
Eliminate left factors from the given grammar S → T + S \| T
Solution:

After left factoring, the grammar becomes,
S → T L
L →+ S | ε

2.4 TOP-DOWN PARSER - GENERAL STRATEGIES

- The process of constructing the parse tree, which starts from the root and goes down to the leaf, is Top-Down Parsing.
- Top-down parsers are constructed from the Grammar, which is free from ambiguity. left recursion, and Left Factoring.
- Top-Down Parsers use leftmost derivation to construct a parse tree.
- Top-down parsing can be viewed as the problem of constructing a parse tree for
- the input string, starting from the root and creating the nodes of the parse tree
- in preorder.
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Classification

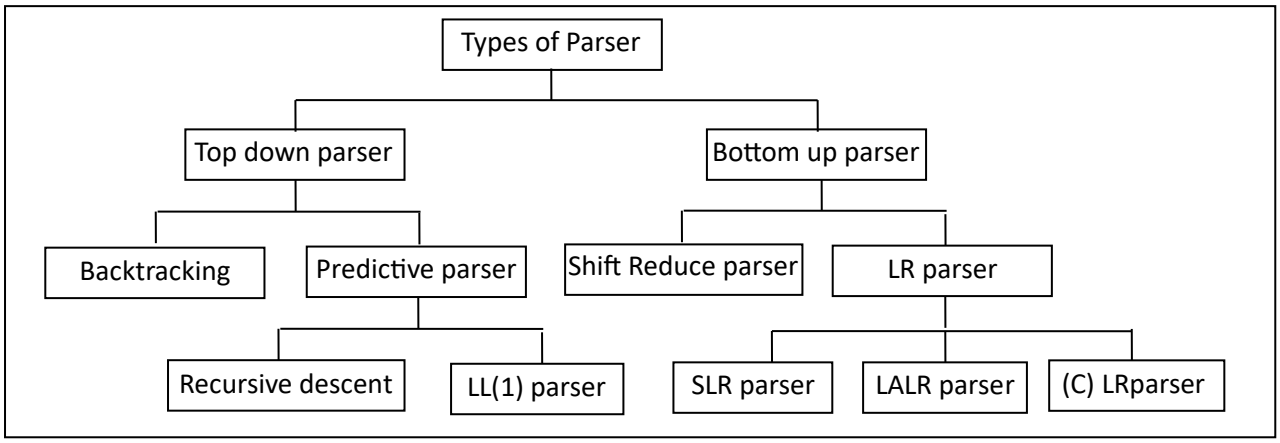


Figure 2.8: Parser Types

Factors to be considered for Top-Down Parsing

- ♦ Left recursive grammar can cause a top-down parser to go into an indefinite loop on writing procedure.
- ♦ Backtracking overhead may occur
- ♦ Due to backtracking, it may reject some valid sentences.
- ♦ Left factoring
- ♦ Ambiguity
- ♦ The order in which alternates are tried can affect the language accepted
- ♦ When failure is reported, we have very little idea where the error actually occurred

2.4.1 Backtracking

The parse tree is started from the root node and the input string is matched against the production rules for replacing them.
If the grammar is

S →aBc
B →bc|b and the input is abc

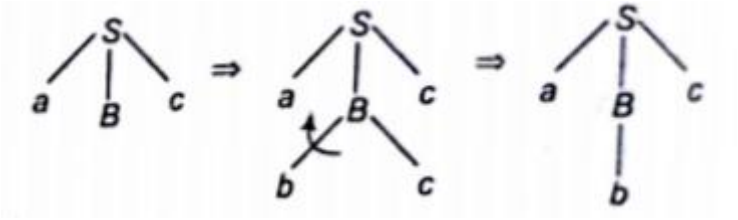


Figure 2.9: Parse tree for the above grammar

2.4.2 Predictive parser

2.4.2.1 Recursive Descent Parser Without Backtracking

Recursive descent parser without backtracking works in a similar way as that of a recursive descent parser with backtracking, with the difference that each non-terminal should be expanded by its correct alternative in the first selection itself.

When the correct alternative is not chosen, the parser cannot backtrack and resulting in a syntactic error.

Advantage

- Overhead associated with backtracking is eliminated.

Limitation

- When more than one alternative with common prefixes occurs, then the selection of the correct alternative is highly difficult.

2.4.2.2 Non-Recursive Descent Parser/ LL(1) Parser

Hence, this process requires a grammar with no common prefixes for alternatives.

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

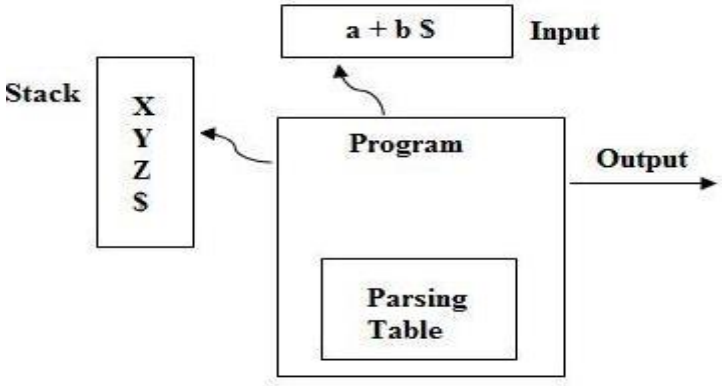


Figure 2.10: Model of Predictive parser

- The predictive parser has an input, a stack, a parsing table, and an output.
- The input contains the string to be parsed, followed by $, the right endmarker.
- The stack contains a sequence of grammar symbols, preceded by $, the bottom-of-

stack marker.

- Initially, the stack contains the start symbol of the grammar, preceded by $.
- The parsing table is a two-dimensional array M[A, a], where A is a nonterminal, and a is a terminal or the symbol $.
- The parser is controlled by a program that behaves as follows:
- The program determines X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser.
- There are three possibilities:
  1. If X = a = $, the parser halts and announces successful completion of parsing.
  2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
  3. If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.
- If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU (with U on top).
- If M[X, a] = error, the parser calls an error recovery routine.

Construction of Parsing Table

Before constructing the parsing table, two functions are to be performed to fill the entries in the table.

- FIRST( ) and FOLLOW( ) functions
- These functions will indicate proper entries in the table for a grammar G.

FIRST(X)

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

1. If X is terminal, then FIRST(X) is {X}.
2. If X is nonterminal and X → aα is a production, then add a to FIRST(X). If X → ε is a production, then add ε to FIRST(X).
3. If X → Y1, Y2, … , Yk is a production, then for all I such that all of Y1, … , Yi-1 are non-terminals and FIRST(Yj) contains ε for j = 1,2, … , i-1 (i.e., Y1, Y2, … . Yi-1 ε), add every non-ε symbol in FIRST(Yi) to FIRST(X). If ε is in FIRST(Yj) for all j = 1, 2, … , k, than add ε to FIRST(X).

FOLLOW(A)

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

1. $ is in FOLLOW(S), where S is the start symbol.
2. If there is a production A → αBβ, β ≠ ε, the everything in FIRST(β) but ε is in FOLLOW(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε (i.e., β ε), then everything in FOLLOW(A) is in FOLLOW(B).

Example 2.13:

*Eliminate the left recursion for the grammar E→ E + T | T; T → T * F | F; F → (E) | id and construct FIRST and FOLLOW for the above grammar.*

Solution:

To eliminate the left recursion for the given Grammar we get;

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

Then:

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}.
FIRST(E`) = {+, ε}
FIRST(T`) = {*, ε}
FOLLOW(E) = FOLLOW(E`) = {), $}
FOLLOW(T) = FOLLOW(T`) = {+,), $}
FOLLOW(F) = {+, *,), $}

Example 2.14:

Derive FIRST and FOLLOW for the following grammar.
S->0|1|AS0|BS0; A->ε; B->ε

Solution:

FIRST = {0,1}
FIRST = {ε}
FIRST(B) = {ε}
FOLLOW(S) ={0,$}
FOLLOW(A) = FIRST(S) = {0,1}
FOLLOW(B) = FIRST(S) ={0,1}

Example 2.15:

Consider the following CFG grammar over the non-terminals {X,Y,X} and the terminals {a,b,c} with the production below and start symbol Z.

X→a
X→Y
Z→d
Z→X Y Z
Y→c
Y→ ε

Compute the FIRST and FOLLOW sets of every non-terminal and the set of terminals are nullable.

Solution:

For this simple grammar, the "first" and "follow" sets are as follows:

FIRST(X) = {a, c, ε}
FIRST(Y) = {c, ε}
FIRST(Z) = {a, c, d, ε}
FOLLOW(X) = {a, c, d, $}
FOLLOW(Y) = {a, c, d, $}
FOLLOW(Z) = {$}

As a result, all the non-terminals are nullable, and in fact the grammar can generate the null string.

Construction of Predictive Parsing Table:

Algorithm: Constructing a predictive parsing table
Input: Grammar G

Output: Parsing table M
Method:
1. For each production A → α of the grammar, do step 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A,a].
3. If ε is in FIRST(α), add A → α to M[A,b] for each terminal b in FOLLOW(A).
   If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A,$].
4. Make each undefined entry of M error.

Problems with Top-down parsing

i) Backtracking - For expansion of non-terminal symbols, we choose on alternative, and if some mismatch occurs then we try another alternative.

ii) Left recursion–left recursive grammar is a grammar that is as follows:

$$A \overset{+}{\Rightarrow} A\alpha$$

iii) Left factoring– used when it is not clear which of the two alternatives is used to expand the non-terminal.

iv) Ambiguity– Ambiguous grammar is not desirable in top-down parsing. Hence, we need to remove it.

Example 2.16:
Construct a predictive parsing table for the following grammar. Show how the string (a, a) is parsed by the predictive parser S->a| ↑ | (T), T->T, S | S.

Solution:
Eliminate left recursion,

S→ a| ↑ | (T)
T→ ST′
T′→ ,ST′ | ε

Compute FIRST and FOLLOW:

FIRST(S) = {a, ↑, (}
FIRST(T) = {a, ↑, (}
FIRST(T′) = {, ,ε}
FOLLOW(S) = {, ,), $}
FOLLOW(T) = FOLLOW(T') = {)}

Table 2.3: Predictive Parsing table for the above grammar

Non terminals	**Input symbol**					
	a	↑	(	)	,	$
S	S → a	S → ↑	S → (T)			
T	T → ST′	T→ ST′	T → ST′			
T′′				T′→ ∈	T′→,ST′	

Table 2.4: Parse the string (a,a) using Predictive Parsing Table

State	**Input**	**Actions**
$S	(a, a)$	S → (T)

$)T(	(a, a)$	
$)T	a, a)$	T → ST′
$)T′S	a, a)$	S → a
$)T′a	a, a)$	
$)T′	,a)$	T′ → ST′
$)T′S,	a)$	
$)T′S	a)$	S → a
$)T′a	a)$	
$)T′	)$	T’ → ε
$)	)$	
$	$	ACCEPT

Example 2.17:

Construct the predictive parsing table for the given grammar S → (L) | a; L → L, S | S and show whether the following string will be accepted or not (a,(a,(a,a))).

Solution:

Given the following CFG grammar G = ({S,L}, S, {a, “(”, “)”, “,”}, P) with P:

S → (L) | a
L → L, S | S

Removing left recursion,

S → (L) | a
L → S L′
L′→ , S L′ | ε

Computing FIRST and FOLLOW:

FIRST(S) = { (, a }
FIRST(L) = { (, a }
FIRST(L’) = { , ε}

FOLLOW(L) = {) }
FOLLOW(S) = { , ,) , $ }
FOLLOW(L’) = {) }

The parsing table

Table 2.5: Predictive parsing table for the above grammar

Non terminals	**Input symbol**				
	a	**(**	**)**	**,**	**$**
S	S→a	S→(L)			
L	L→SL′	L→SL′			
L′			L′→∈	L′→,SL′	

The stack and input are as shown below using the predictive, table-driven parsing algorithm:

Table 2.6: Input string for the string (a,(a,(a,a)))

Stack	Input	Rule

$S	(a,(a,(a,a))) $	
$)L(	(a,(a,(a,a))) $	S → (L)
$)L	a,(a,(a,a))) $	
$)L′S	a,(a,(a,a))) $	L → S L′
$)L′a	a,(a,(a,a))) $	S→a
$)L′	,(a,(a,a))) $	
$)L′S,	,(a,(a,a))) $	L′ → , S L′
$)L′S	(a,(a,a))) $	
$)L′)L(	(a,(a,a))) $	S→(L)
$)L′)L	a,(a,a))) $	
$)L′)L′S	a,(a,a))) $	L→SL′
$)L′)L′a	a,(a,a))) $	S→a
$)L′)L′	,(a,a))) $	
$)L′)L′S,	,(a,a))) $	L′→,SL′
$)L′)L′S	(a,a))) $	
$)L′)L′)L(	(a,a))) $	S→L)
$)L′)L′)L	a,a))) $	
$)L′)L′)L′S	a,a))) $	L→SL′
$)L′)L′)L′a	a,a))) $	S→a
$)L′)L′)L′	,a))) $	
$)L′)L′)L′S,	,a))) $	L′→,SL′
$)L′)L′)L′S	a))) $	
$)L′)L′)L′a	a))) $	S→a
$)L′)L′)L′	))) $	
$)L′)L′)	))) $	L′→ ε
$)L′)L′	)) $	
$)L′)	)) $	L′→ ε
$)L′	) $	
$)	) $	L′→ ε
$	$	

Example 2.18:

Explain the non-recursive implementation of predictive parsers with the help of the grammar

E → E+E | T
*T→ T*F | F*
F→ (E) | id

Solution:

E → E+T
E → T
T → T*F
T → F
F → (E) | id

Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left recursive grammar (as it goes to infinite loop).

E → TE'
E' → +TE' | ε
T → FT'

T' → *FT' | ε
F → (E) | id

Remove the left recursion:

Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left recursive grammar (as it goes to infinite loop).

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

First( ):

FIRST(E) = { (, id }
FIRST (E') = { + , ∈ }
FIRST (T) = { (, id }
FIRST (T') = { * , ∈ }
FIRST (F) = { (, id }

Follow( ):

FOLLOW (E) = { $,) }
FOLLOW (E') = { $,) }
FOLLOW (T) = { + , $,) }
FOLLOW (T') = { + , $,) }
FOLLOW (F) = { + , * , $) }

Table 2.7: Predictive Parsing Table for the above grammar

Non - Terminal	id	+	*	(	)	$
E	E→ TE'			E→ T*E'*		
E′		E→ T*E'*			E′ → ∈	E′ → ∈
T	T → FT′			T → FT′		
T′		T′ → ∈	T → *FT′		T′ → ∈	T′ → ∈
F	F → id			F → €		

Table 2.8: Stack implementation using above predictive parsing table

Stack	Input	Output
$E	id+id*id $	
$E′T	id+id*id $	E → TE′
$E′T′F	id+id*id $	T → FT′
$E′T′id	id+id*id $	F → id
$E′T′	+id*id $	
$E′	+id*id $	T′ → ∈
$E′T+	+id*id $	E′ → +TE′
$E′T	id*id $	
$E′T′F	id*id $	T → FT′
$E′T′id	id*id $	F → id
$E′T′	*id $	
$E′T′F*	*id $	T′→ *FT′

$E′T′F	id $	
$E′T′id	id $	F → id
$E′T′	$	
$E′	$	T’ →∈
$	$	E’ → ∈

Example 2.19:
Is the following grammar G LL(1)? E->A|B; A->a|c.
Solution:
There is no multiple entries in the parsing table. So it is not LL(1).
A context-free grammar $G = (V_T, V_N, S, P)$ whose parsing table has no multiple entries is said to be *LL*(1). In the name *LL*(1),

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

A language is said to be *LL*(1) if it can be generated by a *LL*(1) grammar. It can be shown that *LL*(1) grammars are

- not ambiguous and
- not left-recursive.

2.5 BOTTOM-UP PARSER

- Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node.
- Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.
- Bottom-up can parse a larger set of languages than top down.

Example 2.20:
S→aAcBe
A→Ab/b
B→d
Parse the input string: abbcde using bottom-up approach.
Solution:

S→aAcBe
→aAcde
→aAbcde
→abbcde

2.5.1 Handle

- A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.
- A handle of a right – sentential form γ is a production A→β and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential

form in a rightmost derivation of γ. That is , if S =>αAw =>αβw,then A→β in the position following α is a handle of αβw.

Example 2.21:
*Consider the grammar: E → E+E, E → E*E, E → (E), E → id and the input string id1+id2*id3.*
Solution:
The rightmost derivation is :

E → **E**+**E**
→ E+**E***E
→ E+E***id3**
→ E+**id2***id3
→ **id1**+id2*id3

In the above derivation the underlined substrings are called handles.

2.5.2 Handle pruning

- A rightmost derivation in reverse can be obtained by "handle-pruning".
- The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruningi.e)start with a string of terminals w that is to parse. If w is a sentence of the grammar at hand, then w = γn, where γn is the nth right sentential form of some as yet unknown rightmost derivation.

S = γ0 =>γ1=> γ2 =>… γn-1 =>γn = w.

Example 2.22:
Construct right sentential form and handle for grammar
*E → E + E, E → E * E, E → (E), E → id*
Solution:

Table 2.9: Construction of right sentential form and handle

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
$id_1 + id_2 * id_3$	id1	E→ id
$E + id_2 * id_3$	id_2	E→ id
$E + E * id_3$	id_3	E→ id
E + E * E	E * E	E→ E * E
E + E	E + E	E→ E + E
E		

2.5.3 Shift-Reduce Parsing

- ✓ Bottom-up parsers generally operate by choosing, on the basis of the next input symbol (lookahead symbol) and the contents of the stack, whether to shift the next input onto the stack, or to reduce some symbols at the top of the stack.
- ✓ A reduce step takes a production body at the top of the stack and replaces it with the head of the production.

Rules

- Shift Reduce parser attempts to construct a parse tree from leaves to root.
- Thus, it works on the same principle as a bottom-up parser.
- The following are the two rules that can be used in shift-reduce parsing-

- If the incoming operator has more priority than the stack operator, then perform a shift.
- If the stack operator has some or less priority than the priority of the incoming operator, then perform reduce.

Actions

Shift: This involves moving of symbols from input buffer onto the stack.

Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example 2.23:

Consider the following grammar

S-> (L) | a

L->L,S | S

Parse the input string (a,(a,a)) using Shift Reduce parser.

Solution:

Table 2.10: Parse the input string (a,(a,a)) using Shift-Reduce parser

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a

$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

Example 2.24:

Construct shift-reduced parser for the following Grammar & show the moves made by the parser for input strings

*(i) id+id*id*

(ii) id+id-id

and the grammar is E->E+E

*E->E*E*

E->id

Solution:

(i) Stack implementation of shift-reduce parser id+id*id

Table 2.11: Shift reduce moves for the string id+id*id

S.No	Stack	Input string	Action
1	$	id+id*id $	Shift
2	$id	+id*id $	Reduce by E→id
3	$E	+id*id$	Shift
4	$E+	id*id$	Shift
5	$E+id	*id$	Reduce by E→id
6	$E+E	*id$	Shift
7	$E+E*	id$	Shift
8	$E+E*id	$	Reduce by E→id
9	$E+E*E	$	Reduce by E→E*E
10	$E+E	$	Reduce by E→E+E
11	$	$	The string is successfully parsed

(ii) Stack implementation of shift-reduce parser id+id-id

Table 2.12: Shift reduce moves for the string id+id-id

S. No.	Stack	Input String	Action
1	$	id+id-id$	Shift
2	$id	+id-id$	Reduce by E→id

3	$E	+id-id$	Shift
4	$E	id-id$	Shift
5	$E+id	-id$	Reduce by E→id
6	$E+E	-id$	Shift
7	$E+E-	id$	Shift
8	$E+E-id	$	Reduce by E→id
9	$E+E-E	$	The string can't be further reduced. The parsing can't be successfully done

Example 2.25:

Construct stack implementation of shift-reduce parsing for the grammar

E →E+E
*E →E*E*
E →(E)
E →id

*and the input string id1 + id2 * id3.*

Solution:

Table 2.13: Shift-reduce parsing for the input string id1 + id2 * id3

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce E→ id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce E→ id
$E+E	*id$	shift (or reduce?)
$E+E*	id$	shift
$E+E*id	$	reduce E→ id
$E+E*E	$	reduce E→E* E
$E+E	$	reduce E→E+ E
$E	$	accept

Example 2.26:

*Consider the grammar E → E+E | E*E | id and parse the input id+id*id using Shift-Reduce Conflict.*

Solution:

Table 2.14: Shift-Reduce Conflict for the string id+id*id

Stack	Input	Action
E+E	*id	reduce by E→E+E
E	*id	Shift
E*	id	Shift
E*id		reduce by E→id
E*E		reduce byE→E*E

E		

Table 2.15: After eliminating Shift-Reduce Conflict

Stack	Input	Action
E+E	*id	shift
E+E*	id	Shift
E+E*id	id	reduce by E→id
E+E*E		reduce by E→E*E
E+E		reduce by E→E+E
E		

Conflicts that occur during Shift-Reduce parsing

1. A ***shift-reduce*** conflict occurs in a state that requests both a shift action and a reduce action.
2. A ***reduce-reduce*** conflict occurs in a state that requests two or more different reduce actions.

2.6 LR PARSERS

- LR parser is an efficient bottom-up parser which can be used to parse a large class of context-free grammars. This technique is called LR(k) parsing.
  - L is left-to-right scanning of the input.
  - R is for constructing a right-most derivation in reverse.
  - k is the number of input symbols of lookahead that are used in making parsing decisions.
- There are three widely used algorithms available for constructing an LR parser:

SLR(1) - Simple LR

- Works on smallest class of grammar.
- Few number of states, hence very small table.
- Simple and fast construction.

LR(1) - LR parser

- Also called as Canonical LR parser.
- Works on complete set of LR(l) Grammar.
- Generates a large table and a large number of states.
- Slow construction.

LALR(1) - Look-ahead LR parser

- Works on intermediate levels of grammar.
- The number of states is are same as in SLR(l).

Kernel items are a collection of items S′ → •S and all the items whose dots are not at the leftmost end of R.H.S. of the rule.
Non-kernel items is the collection of all the items in which • are at the left end of the R.H.S. of the rule.
Function closure and goto are the important functions required to create a collection of a canonical set of items.

Reasons for the attractiveness of the LR parser

- LR parsers can handle a large class of context-free grammars.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method.
- An LR parser can detect the syntax errors as soon as they occur.
- LR grammars can describe more languages than LL grammars.

Drawbacks of LR parsers

- It is too much work to construct LR parser by hand. It needs an automated parser generator.
- If the grammar contains ambiguities or other constructs, then it is difficult to parse in a left-to-right scan of the input.

2.6.1 Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

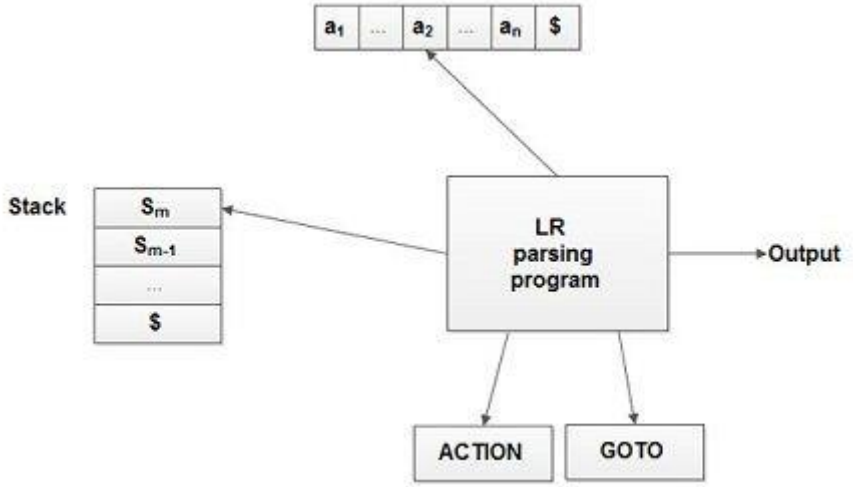


Figure 2.11: Model of LR parser

1. The ACTION function takes as arguments a state *i* and a terminal *a* (or $, the input end marker). The value of **ACTION[*i, a*]** can have one of four forms:
   (a) **Shift j**, where j is a state. The action taken by the parser effectively shifts input *a* to the stack, but uses state j to represent *a*.
   (b) **Reduce** $A \rightarrow \beta$. The action of the parser effectively reduces β on top of the stack to head *A*.
   (c) **Accept**. The parser accepts the input and finishes parsing.
   (d) **Error**. The parser discovers an error in its input and takes some corrective action.
2. We extend the GOTO function, defined on sets of items, to states:
   if **GOTO** $[I_i, A] = I_j$, then GOTO also maps a state ***i*** and a nonterminal ***A*** to state j.

2.6.2 LR-Parsing Algorithm

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in L(G), the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has so on its stack, where so is the initial state, and w$ in the input buffer.

```
let a be the first symbol of w$;
while(1)
    { /* repeat forever */
        lets be the state on top of the stack;
        if( ACTION[S, a] = shift t)
        {
            pusht onto thestack;
            let a be the next input symbol;
        }
        else if ( ACTION[S, a] = reduce A□β)
        {
            pop |β| symbols off the stack;
            let state t now be on top of
            the stack; push GOTO[t, A]
            onto the stack; output the
            production A□β;
        }
        else if ( ACTION[S, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
    }
```

2.6.3 Properties of the LR parser

LR parser is an efficient bottom-up parser. LR parser is widely used for the following reasons,

1. An LR parser can be constructed to recognize most of the programming languages for which a CFG can be written.
2. The class of grammar that can be parsed is a superset of the class of grammar that can be parsed using predictive parsers.
3. It can detect syntactic errors as soon as it is possible to do so on a left-to-right scan of the input.
4. LR parser works using non non-backtracking shift-reduce technique, yet it is an efficient one.

2.6.4 LR(0) Items

An LR(0) item of a grammar G is a production of G with a dot at some position of the body. (eg.)

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

One collection of set of LR(0) items, called the canonical LR(O) collection, provides a finite automaton that is used to make parsing decisions. Such an automaton is called an LR(O) automaton.

LR(0) Parser / SLR(1) Parser

- An LR(O)parser is a shift-reduce parser that uses zero tokens of look ahead to determine what action to take (hence the 0). This means that in any configuration of the parser, the

parser must have an unambiguous action to choose-either it shifts a specific symbol or applies a specific reduction. If there are ever two or more choices to make, the parser fails and the grammar is not LR(0).

- An LR parser makes shift-reduce decisions by maintaining states to keep track of parsing. States represent a set of *items.*

Closure of item sets

If I is a set of items for a grammar G, then *CLOSURE(I)* is the set of items constructed from I by the two rules.

- Initially, add every item I to *CLOSURE(I).*
- If A → αB β is in *CLOSURE(I)* and B → γ is a production, then add the item B → • γ to *CLOSURE(I),* if it is not already there. Apply this rule until no more items can be added to *CLOSURE (I).*

Construct canonical LR(0) collection

• Augmented grammar is defined with two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then augmented grammar G' is G with a new start symbol S' → S.

• The role of augmented production is to stop parsing and notify the acceptance of the input i.e., acceptance occurs when and only when the parser performs reduction by S' → S.

2.7 CONSTRUCTING AN SLR PARSING TABLE

Algorithm: Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct C = {$I_0, I_1, \ldots, I_n$}, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   a) If [$A \to \alpha \bullet a\beta$] is in I_i, and GOTO(I_i , a) = I_j, then set ACTION[i , a] to "*shift j*". Here a must be a terminal.
   b) If [$A \to \alpha \bullet$] is in I_i, then set ACTION[i, a] to "*reduce* $A \to \alpha$" for all a in FOLLOW(A); here A may not beS'.
   c) If [$S' \to S \bullet$] is in I_i, then set ACTION[i, $] to"*accept*".

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO(I_i, A) = I_j, then GOTO[i, A] =j.
4. All entries not defined by rules (2) and (3) are made"*error*".
5. The initial state of the parser is the one constructed from the set of items containing [$S' \to \bullet S$].

An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be SLR(1). We usually omit the "(1)" after the "SLR," since we shall not deal here with parsers having more than one symbol of lookahead.

Example 2.27:

Construct a SLR parsing table for the following grammar

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

Solution:

The given grammar is

E → E + T ----------(1)
E → T ----------(2)
T → T * F ----------(3)
T → F ----------(4)
F → (E) ----------(5)
F → id ----------(6)

Step 1: Convert the given grammar into augmented grammar

Augmented Grammar

E′→ E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2: Find the LR (0) items

Table 2.16: LR (0) items

GOTO (I0, E) **I1:** E′→E• E →E•+T	GOTO (I0, T) **I2:** E→T• T→T•*F
GOTO (I0, F) **I3:** T→F•	GOTO (I0, ( ) **I4:** F→(•E) E → •E + T E → •T T → •T * F T → •F F → •(E) F → •id
GOTO (I0, id) **I5:** F→id•	GOTO (I_1, +) **I6:** E → E+•T T→ • T * F T → • F F → • (E) F → •id
GOTO (I_2, *) **I7:** T → T * •F F → •(E) F → •id	GOTO (I4, E) **I8:** F → (E•) E → E• + T
GOTO (I4, T)	GOTO (I4, F)

I2: E → T• * F	**I3:** T → F•
GOTO (I6, T) **I9:** E → E +T• T → T• * F	GOTO (I6, F) **I3:** T → F•
GOTO (I6, ( ) **I4:** F → (•E)	GOTO (I6, id) **I5:** F → id•
GOTO (I7, F) **I10:** T → T * F•	GOTO (I7, ( ) **I4:** F → (•E) E → •E + T E → •T T → •T * F T → •F F → •(E) F → • id
GOTO (I7, id) **I5:** F → id	GOTO (I8,)) **I11:** F → (E)•
GOTO (I8, +) **I6:** E → E + • T T →• T * F T → • F F → • (E) F → • id	GOTO (I9, *) **I7:** T → T * • F F → • (E) F → • id
GOTO (I4, ( ) **I4:** F → (•E) E → •E + T E → •T T → •T * F T → •F F → •(E) F → id	

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

Table 2.17: SLR Parsing table

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I0	s5			s4			1	2	3
I1		s6				Acc			
I2		r2	s7		r2	r2			
I3		r4	r4		r4	r4			
I4	s5			s4			8	2	3
I5		r6	r6		r6	r6			
I6	s5			s4				9	3
I7	s5			s4					10
I8		s6			s11				
I9		r1	s7		r1	r1			

I10		r3	r3		r3	r3			
I11		r5	r5		r5	r5			

Note: Blank entries are error entries

Example 2.28:

*Find the SLR parsing table for the given grammar E->E+E | E*E | (E) | id and parse the sentence (a+b)*c.*

Solution:

Given grammar:

1. E→E+E
2. E→E*E
3. E→(E)
4. E→id

Augmented grammar:

E′→ E
E→ E+E
E→E*E
E→(E)
E→id

I0: E′→•E
E→•E+E
E→•E*E
E→•(E)
E→•id

I1: goto(I0, E)
E′→E•
E→E•+E
E→E•*E

I2: goto(I0, ()
E→(•E)
E→•E+E
E→•E*E
E→•(E)
E→•id

I3: goto(I0, id)
E→id•

I4: goto(I1, +)
E→E+•E
E→•E+E
E→•E*E
E→•(E)
E→•id

I5: goto(I1, *)
E→E*•E
E→•E+E
E→•E*E
E→•(E)
E→•id

I6: goto(I2, E)
E→(E•)
E→E•+E
E→E•*E

I7: goto(I4, E)
E→E+E•
E→E•+E
E→E•*E

I8: goto(I5, E)
E→E*E•
E→E•+E
E→E•*E
goto(I2, () = I2
goto(I2, id) = I3
goto(I4, () = I2
goto(I4, id) = I3
goto(I5, () = I2
goto(I5, id) = I3

I9: goto(I6,))
E→(E)•
goto(I6, +) = I4
goto(I6, *) = I5
goto(I7, +) = I4
goto(I7, *) = I5
goto(I8, +) = I4
goto(I8, *) = I5

First(E) = {(, id}
Follow(E) ={+, *,), $}

Table 2.18: SLR Parsing Table

States	**Action**						
GOTO	+	*	(	)	**id**	**$**	**E**
0			S2		S3		1
1	S4	S5				Acc	
2			S2		S3		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	S4, r1	S5, r1		r1		r1	
8	S4, r2	S5, r2		r2		r2	
9	r3	r3		r3		r3	

Parsing the sentence (a+b)*c:

0	(a+b)*c$	shift 2
0(2	a+b)*c$	shift 3
0(2a3	+b)*c$	reduce by E->id
0(2E6	+b)*c$	shift 4

0(2E6+4	b)*c$	shift 3
0(2E6+4b3	)*c$	reduce by E->id
0(2E6+4E7	)*c$	reduce by E->E+E
0(2E6	)*c$	shift 9
0(2E6)9	*c$	reduce by E->(E)
0E1	*c$	shift 5
0E1*5	c$	shift 3
0E1*5c3	$	reduce by E->id
0E1*5E8	$	reduce by E->E*E
0E1	$	accept

Example 2.29:

Construct a SLR parsing table for the following grammar.

$E \rightarrow E + T \mid T$
$T \rightarrow TF \mid F$
$F \rightarrow F^* \mid a \mid b$

Solution:

We compute the FIRST and FOLLOW for the augmented grammar (0) E' → E$
FIRST(E) = FIRST(T) = FIRST(F) = {a,b}
FOLLOW(E) = {+,$}
FOLLOW(T) = FIRST(F) + FOLLOW(E) = {a,b,+,$}
FOLLOW(F) = {*,a,b,+,$}
Consider the augmented grammars E' → E$ we compute the LR(0) set of items.
I0 = closure({[E′ → •E$]})
= E′ → •E$
E → •E + T
E → •T
T → •T F
T → •F
F → •F *
F → •a
F → •b
I1 = goto (I0, E)
= closure({[E' → E•$], [E → E• + T]})
= E' → E• $
E → E• + T
I2 = goto (I0, T)
= closure({[E → T•], [T → T• F]})
= E → E•
T → T•F
F → •F *
F → •a
F → •b
I3 = goto (I0, F)
= closure({[T → F•], [F →F• *]})
= T → F•
F →F• *
I4 = goto (I0, a)
= closure({[F → a•]})

= F → a•

I5 = goto (I0, b)

= closure({[F → b•]})

= F → b•

I6 = goto (I1, +)

= closure({[E → E+•T]})

= E → E+•T

T → •T F

T → •F

T → •F *

F → •a

F → •b

I7 = goto (I2, F)

= closure({[T → TF•], [F → F•*]})

= T → T F•

F → F•*

I8 = goto (I3, *)

= closure({[F → F*•]})

= F → F*•

I9 = goto (I6, T)

= closure({[E → E+T•], [E → T•F]})

= E → E + T•

E → T•F

F → •F *

F → •a

F → •b

goto (I9, a) = I4

goto (I9, b) = I5

goto (I9, F) = I7

Table 2.19: SLR Parsing Table

	ACTION					GOTO		
	a	**b**	**+**	*****	**$**	**E**	**T**	**F**
I0	s4	s5				goto I1	goto I2	goto I3
I1			s6		accept			
I2	s4	s5	r2		r2			goto I7
I3	r4	r4	r4	s8	r4			
I4	r6	r6	r6	r6	r6			
I5	r7	r7	r7	r7	r7			
I6	s4	s5					goto I9	goto I3
I7	r3	r3	r3	s8	r3			
I8	r5	r5	r5	r5	r5			
I9	s4	s5	r4		r4			goto I7

Example 2.30:

Given the following grammar S->AS|b, A->SA|a. Construct a SLR parsing table for the string baab.

Solution:

Given grammar:

1. S→AS
2. S→b
3. A→SA
4. A→a

Augmented grammar:

S′→S
S→AS
S→b
A→SA
A→a

I0:

S′→•S
S→•AS
S→•b
A→•SA
A→•a

I1: goto(I0, S)

S′→S•
A→S•A
A→•SA
A→•a
S→•AS
S→•b

I2: goto(I0, A)

S→A•S
S→•AS
S→•b
A→•SA
A→•a

I3: goto(I0, b)

S→b•

I4: goto(I0, a)

A→a•

I5: goto(I1, A)

A→SA•
S→A•S
S→•AS
S→•b
A→•SA
A→•a

I6: goto(I1, S)

A→S•A
A→•SA
A→•a
S→•AS
S→•b

goto(I1, a) = I4

goto(I1, b) = I3

I7: goto(I2, S)

S→AS•
A→S•A
A→•SA
A→•a
S→•AS
S→•b

goto(I2, A) = I2
goto(I2, b) = I3
goto(I2, a) = I4
goto(I5, A) = I2
goto(I5, S) = I7
goto(I5, a) = I4
goto(I5, b) = I3
goto(I6, A) = I5
goto(I6, S) = I6
goto(I6, a) = I4
goto(I6, b) = I3
goto(I7, A) = I5
goto(I7, S) = I6
goto(I7, a) = I4
goto(I7, b) = I3

First(S) = {b, a}
First(A) = {a, b}
Follow(S) ={$,a,b}
Follow(A)={a,b}

Table 2.20: SLR Parsing Table

States	Action			Goto	
	a	b	$	S	A
0	S4	S3		1	2
1	S4	S3	acc	6	5
2	S4	S3		7	2
3	r2	r2	r2		
4	r4	r4			
5	r3 s4	r3 s3		7	2
6	S4	S3		6	5
7	S4 r1	S3 r1	r1	6	5

Parsing the string baab:

0	baab$	shift 3
0b3	aab$	reduce by S→b
0S1	aab$	shift 4

0S1a4	ab$	reduce by A→a
0S1A5	ab$	reduce by A→SA
0A2	ab$	shift 4
0A2a4	b$	reduce by A→a
0A2A2	b$	shift 3
0A2A2b3	$	reduce by S→b
0A2A2S7	$	reduce by S→AS
0A2S7	$	reduce by S→AS
0S1	$	accept

Example 2.31:

Check whether the grammar is SLR (1) or not?

*S→L=R | R; L→*R | id; R→L*

Solution:

Canonical collection of sets of LR (0) items for the above grammar is

I0: S→•S
S→•L=R
S→R
L→•*R
L→•id
R→•L

I1: S•→S•

I2: S→L•=R
R→L•

I3: S→R•

I4: L→*•R
R→•L
L→•*R
L→•id

I5: L→id•

I6: S→L=•R
R→•L
L→•*R
L→•id

I7: L→*R•

I8: R→L•

I9: S→L=R•

Before we can build the parsing table, we need to compute the FOLLOW sets:

FOLLOW(S') = {$}
FOLLOW(S) = {$}
FOLLOW(L) = {$,=}
FOLLOW(R) = {$,=}

Table 2.21: Canonical collection of sets of LR (0) items

	ACTION				GOTO		
	id	=	*	$	S	L	R
I0	s3			S5	1	2	4
I1				Acc			

I2		S6/r[R→L]					
I3		r(L→id)		r(L→id)			
I4				r(S→R)			
I5	S3		S5			7	8
I6	S3		S5			7	9
I7		r(R→L)		r(R→L)			
I8		r(L→*R)		r(L→*R)			
I9				r(S→L=R)			

Consider the set of Items I2. The first item in this set makes action [2, =] be shift 6. Since Follow(R) has =, the second item sets action [2, =] to reduce R->L. Thus the entry action [2, =] is multiply defined. So it is **not SLR (1).**

Example 2.32:
Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are non-terminals, and r,s,t are terminals (A)
(i) P -> QR
(ii) P ->QsR
(iii) P -> ε
(iV) P ->QtRr

Explanation:
An operator precedence parser is a bottom-up parser that interprets an operator-precedence grammar. For example, most calculators use operator precedence parsers to convert from the human-readable infix notation with order of operations format into an internally optimized computer-readable format like Reverse Polish notation (RPN).

An operator precedence grammar is a kind of context-free grammar that can be parsed with an operator-precedence parser. It has the property that no production has either an empty () right-hand side or two adjacent nonterminals in its right-hand side. These properties allow the terminals of the grammar to be described by a precedence relation, and the a parser that exploits that relation is considerably simpler than more general-purpose parsers such as LALR parsers.

2.8 LALR PARSER

- LALR stands for *lookahead LR* parser.
- LALR parser starts with the idea of building an LR parsing table
- Tables generated by LALR parser are smaller in size as compared to that of Canonical LR (CLR) and Simple LR (SLR) techniques.
- LALR parsers are slightly less powerful than LR parsers, but still more powerful than SLR parsers.
- LALR is used by YACC and other parser generators because of its effectiveness.
- This is the extension of LR(0) items, by introducing the one symbol of lookahead on the input.
- It supports large class of grammars.
- The number of states is LALR parser is lesser than that of LR(1) parser. Hence, LALR is preferable as it can be used with reduced memory.

- Most syntactic constructs of programming language can be stated conveniently.

Steps to construct LALR parsing table

- Generate LR(l) items.
- Find the items that have same set of first components (core) and merge these sets into one.
- Merge the *goto's* of combined item sets.
- Revise the parsing table of LR(l) parser by replacing states and *goto's* with combined states and combined *goto's* respectively.

Algorithm: An easy, but space-consuming LALR table construction.
INPUT: An augmented grammar G'.
OUTPUT: The LALR parsing-table functions ACTION and GOT0 for G'.
METHOD:

1. Construct C = ($I_0, I_1, \ldots, I_n$), the collection of sets of LR(1)items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by theirunion.
3. Let C' = {$J_0, J_1, \ldots, J_m$} be the resulting sets of LR(1) items. The parsing actions for state ***i*** are constructed from J_i. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to beLALR(1).
4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \ldots \cap I_k$, then the cores of GOTO(I_1, X) , GOTO(I_2, X) , … , GOTO(I_n, X) are the same, since $I_1, I_2, \ldots, I_k$, all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X) . Then GOTO(J, X) =K.

Algorithm: Construction of the sets of LR(1) items.
INPUT: An augmented grammar *G'*.
OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'.
METHOD: The procedures CLOSURE and GOT0 and the main routine items for constructing the sets of items were given below.

```
SetOfItemsCLOSURE(I)
{
        repeat
                for( each item [A → α•Bβ, a] in I )
                        for(each production B → y in G' )
                                for(each terminal b in
                                        FIRST(βa))
                                add[B →•y, b ] to set I;
        until no more items are added to I;
        return I;
}

SetOfItemsGOTO(I, X)
{
```

```
        initialize J to be the empty set;
        for(each item [A → α•Xβ, a] in I )
            add item [A→ αX•β, a] to set J;
        return CLOSURE(J);
}
void items(G')
{
        Initialize C to CLOSURE({[S' →•S, $]});
        repeat
                for(each set of items I 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;
}
```

Conflicts In LALR Parser

- LALR Parser cannot introduce shift/reduce conflicts.
- Such conflicts arise when the look-ahead is the same as the token on which we can shift.
- They depend on the core of the item, but we merge only those rows that have common cores.
- The only way by which this conflict can arise in LALR is when the conflict is already their in the LR(1).

Advantages of LALR parser

1. A LALR parser can be automatically generated from an LALR grammar.
2. An LALR grammar can be used to define many computer languages.
3. An LALR parser is small & fast.
4. Error recovery, generalized error messages, and abstract syntax tree construction may already be built into the parser.

Disadvantages

1. Software engineers are required to use an LALR parser generator, which may or may not be user-friendly and may require some learning time.
2. Implementing meaningful error messages in the parser may be difficult or impossible.
3. Understanding the parsing algorithm is difficult.
4. If an error occurs, it may be difficult to determine whether it's in the grammar or the parser code.

Example 2.33:

Consider the following grammar

$S \rightarrow L=R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow id$
$R \rightarrow L$

Discuss the LALR parsing method for this grammar. List out canonical collections and also

parse table.

Solution:

Given grammar:

1. S→L=R
2. S→R
3. L→*R
4. L→id
5. R→L

Augmented grammar:

S• →S
S→L=R
S→R
L→*R
L→id
R→L

Canonical collection of LR(1) items

I0:

S•→•S, $
S→•L=R, $
S→•R, $
L→•*R, =
L→•id, =
R→•L, $

I1: goto(I0, S)

S→S•, $

I2: goto(I0, L)

S→L•=R, $
R→L•, $

I3: goto(I0, R)

S→R•, $

I4: goto(I0, *)

L→*•R,=
R→•L,=
L→•*R,=
L→•id,=

I5: goto(I0, id)

L→id•,=

I6: goto(I2, =)

S→L=•R,$
R→•L,$
L→•*R,$
L→•id,$

I7: goto(I4, R)

L→*R•,=

I8: goto(I4, L)

R→L•,=
goto(I4, *)=I4
goto(I4, id)=I5

I9: goto(I6, R)
S→L=R•, $
I10: goto(I6, L)
R→L•, $
I11:goto(I6, *)
L→*•R, $
R→•L, $
L→•*R, $
L→•id, $
I12: goto(I6, id)
L→id•, $
I13: goto(I11, R)
L→*R•, $
goto(I11, L) = I10
goto(I11, *) = I11
goto (I11, id) = I12

Table 2.22: LR (1) Table Construction

States	action				goto		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				Acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4						
6		s11	s12			10	9
7	r3						
8	r5						
9				r1			
10				r5			
11		s11	s12			10	13
12				r4			
13				r3			

This grammar is LR(1), since it does not produce any multi-defined entry in its parsing table.

LALR table construction:
I4 and I11 are similar. Combine them as I411 or I4:
L→*•R, =/$
R→•L, =/$
L→•*R, =/$
L→•id, =/$
I5 and I12 are similar. Combine them as I512 or I5:
L→id•, =/$
I7 and I13 are similar. Combine them as I713 or I7:
L→*R•, =/$
I8 and I10 are similar. Combine them as I810 or I8:
R→L•, =/$

Table 2.23: LALR Table Construction

States	action				goto		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				Acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s4	s5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

Table 2.24: Differentiate SLR, Canonical LR And LALR Parser

SLR parser	LALR parser	canonical LR
SLR parser is the smallest in size.	SLR and LALR have the same size.	The canonical LR parser is the largest in size.
It is the easiest method based on the FOLLOW function.	This method is applicable to a wider class than SLR.	This method is the most powerful than SLR and LALR.
This method exposes fewer syntactic features than the LR parser.	Most of the syntactic features of a language are expressed in LALR.	This method exposes fewer syntactic features than that of the LR parser.
Error detection is not immediate in SLR.	Error detection is not immediate in SLR.	Immediate error detection is done by the LR parser.
It requires less time and space complexity.	The time and space complexity is higher in LALR, but efficient methods exist for constructing an LALR parser directly.	The time and space complexity is more for the canonical LR parser.

2.9 CLR PARSING

- Even more powerful than SLR(l) is the LR(l) parsing method.
- LR(l) includes LR(O) items and a look-ahead token in item sets.
- An LR(l) item consists of,
  - o Grammar production rule.
  - o Right-hand position represented by the dot and.
  - o Lookahead token.
  - o A $\rightarrow X_1 \cdot \cdot \cdot X_i \bullet X_{i+1} \cdot \cdot \cdot X_n$, l where l is a *lookahead token*
- The • represents how much of the right-hand side has been seen,
  - o $X_1 \cdot \cdot \cdot X_i$ appear on top of the stack.
  - o $X_{i+1} \cdot \cdot \cdot X_n$ are expected to appear on the input buffer.
- The lookahead token l is expected after $X_1 \cdot \cdot \cdot X_n$ appears on stack.
- An LR(l) state is a set of LR(l) items.

Example 2.34:

Construct CLR parsing table from the grammar

$S \rightarrow AA$

$A \rightarrow Aa|b$

Solution:

While constructing the CLR parsing table, first of all, we have to build a canonical set of items with one look-ahead symbol. The canonical set of LR(1) items is as follows-

I_0: S→S,$
S→•AA,$
A→•Aa,b
A→•b,b

I_1: goto (I_0,S)
S'=S•,$

I_2: goto (I_0,A)
S→A• A,$
A→A• a,b
A→•Aa,$
A→•b,$

I_3: goto (I_0,b)
A→b•, b|a

I_4: goto (I_2,A)
S→AA•,$
A→A• a,$

I_5: goto (I_2,a)
A→Aa•,b

I_6: goto (I_2,b)
A→b•,$|a

I_7: goto (I_4,a)
A→Aa•,$

Table 2.25: Canonical Parse Table

States	Action			Goto	
	a	b	$	S	A
0		S3		1	2
1			ACCEPT		
2	S5	S6			4
3	r3	r3			
4	S7		r1		
5		r2			
6	r3		r3		
7			r2		

Example 2.35:

Consider the Context-Free Grammar (CFG) depicted below where "begin", "end" and "x" are all terminal symbols of the grammar and Stat is considered the starting symbol for this grammar. Productions are numbered in parenthesis and you can abbreviate "begin" to "b" and "end" to "e" respectively.

(1) Stat → Block

(2) Block → begin Block end
(3) Block → Body
(4) Body → x

(a) Compute the set of LR(1) items for this grammar and draw the corresponding DFA. Do not forget to augment the grammar with the initial production S → Stat $ as the production (0).
b) Construct the corresponding LR parsing table.

Solution:

(a) Compute the set of LR(1) items for this grammar and draw the corresponding DFA. Do not forget to augment the grammar with the initial production S → Stat $ as the production (0).

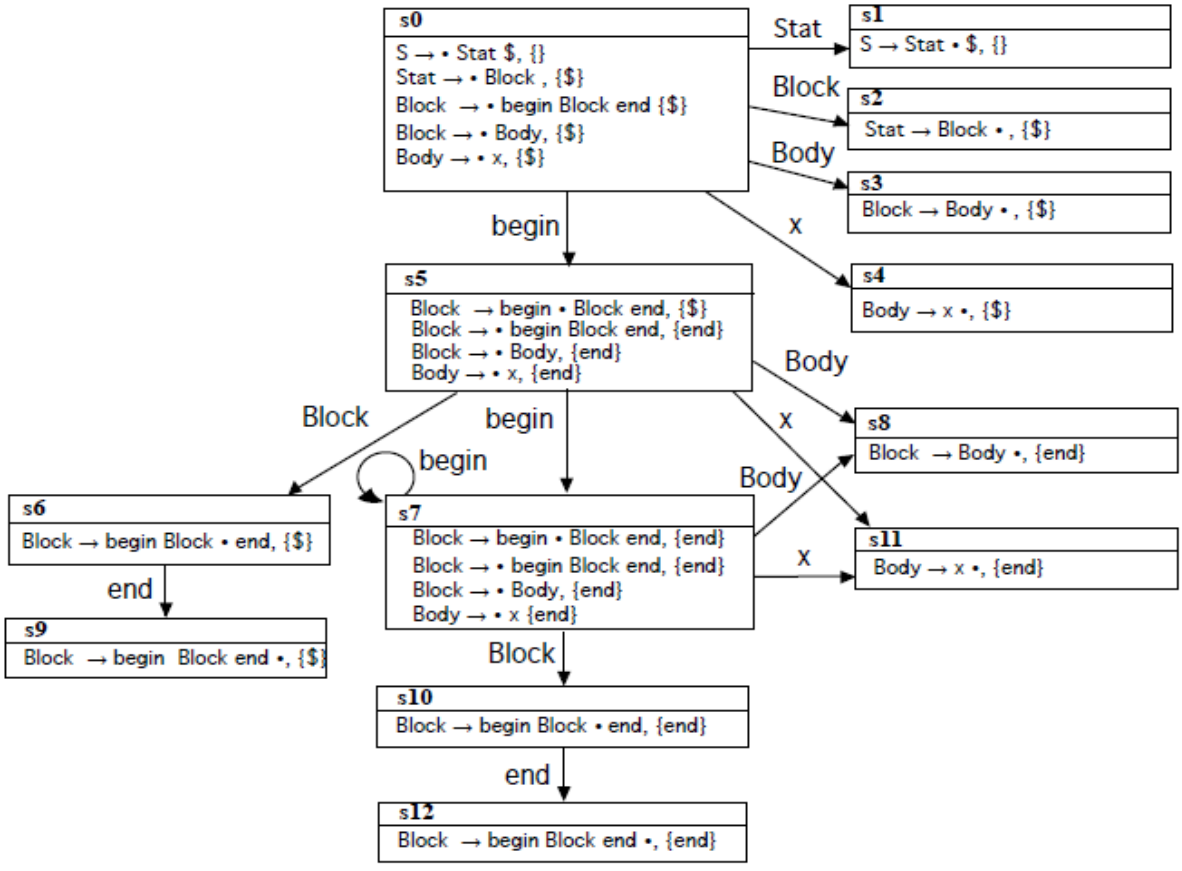


Figure 2.12: LR(1) Items

(b) Construct the corresponding LR parsing table

Table 2.26: LR Parse Table

	ACTION					GOTO		
	a	b	+	*	$	E	T	F
I0	s4	s5				gotoI1	goto I2	goto I3
I1			s6		accept			

I2	s4	s5	r2		r2			goto I7
I3	r4	r4	r4	s8	r4			
I4	r6	r6	r6	r6	r6			
I5	r7	r7	r7	r7	r7			
I6	s4	s5					goto I9	goto I3
I7	r3	r3	r3	s8	r3			
I8	r5	r5	r5	r5	r5			
I9	s4	s5	r4		r4			goto I7
I10								

Example:

CLR (1) Grammar

S → AA
A → aA
A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and
Compute the Closure I0 = Closure (S` → •S)
Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0 = S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $)
= S` → S•, $

I2 = Go to (I0, A) = closure (S → A•A, $)
Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2 = S → A•A, $
A → •aA, $
A → •b, $

I3 = Go to (I0, a) = closure (A → a•A, a/b)
Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3 = A → a•A, a/b
A → •aA, a/b
A → •b, a/b
Go to (I3, a) = closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = closure (A → b•, a/b) = (same as I4)
I4 = Go to (Io, b) = closure (A → b•, a/b) = A → b•, a/b
I5 = Go to (I2, A) = closure (S → AA•, $) = S → AA•, $
I6 = Go to (I2, a) = closure (A → a•A, $)
Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes
I6 = A → a•A, $
A → •aA, $
A → •b, $
Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)
I7 = Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8 = Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9 = Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $
DFA:

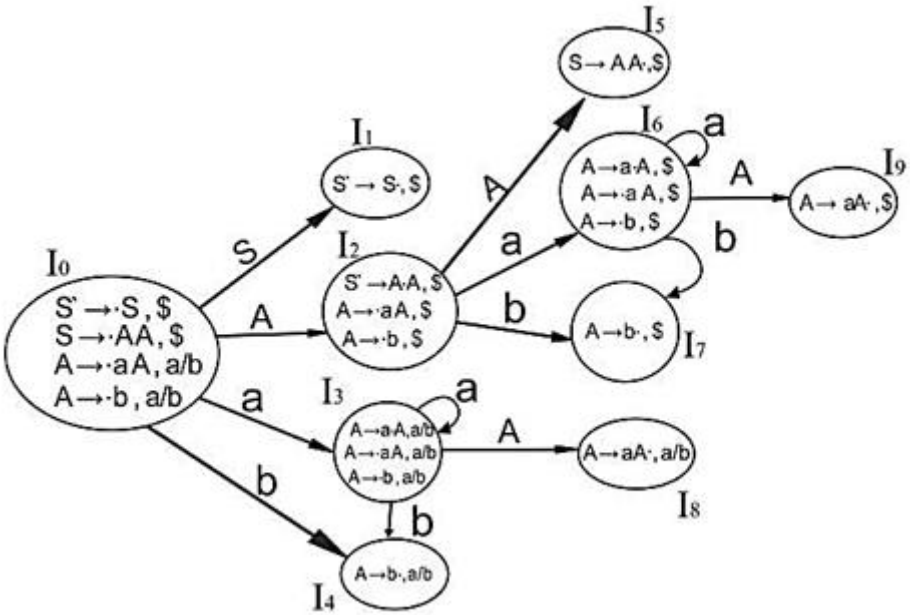


Figure 2.13: DFA diagram from CLR Parser

In the figure, I0 consists of augmented grammar.

- I0 goes to I1 when ' . ' of 0th production is shifted towards the right of S(S'->S.). This state is the accept state. S is seen by the compiler. Since I1 is a part of the 0th production, the lookahead is the same ie $
- I0 goes to I2 when ' . ' of 1st production is shifted towards right (S->A.A) . A is seen by the compiler. Since I2 is a part of the 1st production, the lookahead is the same i.e. $.
- I0 goes to I3 when ' . ' of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I3 is a part of the 2nd production, the lookahead is the same ie a|b.

- I0 goes to I4 when ' . ' of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler. Since I4 is a part of the 3rd production, the lookahead is the same i.e. a | b.
- I2 goes to I5 when ' . ' of 1st production is shifted towards right (S->AA.) . A is seen by the compiler. Since I5 is a part of the 1st production, the lookahead is the same i.e. $.
- I2 goes to I6 when ' . ' of 2nd production is shifted towards the right (A->a.A) . A is seen by the compiler. Since I6 is a part of the 2nd production, the lookahead is the same i.e. $.
- I2 goes to I7 when ' . ' of 3rd production is shifted towards right (A->b.) . A is seen by the compiler. Since I6 is a part of the 3rd production, the lookahead is the same i.e. $.
- I3 goes to I3 when ' . ' of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I3 is a part of the 2nd production, the lookahead is the same i.e. a|b.
- I3 goes to I8 when ' . ' of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler. Since I8 is a part of the 2nd production, the lookahead is the same i.e. a|b.
- I6 goes to I9 when ' . ' of 2nd production is shifted towards the right (A->aA.) . A is seen by the compiler. Since I9 is a part of the 2nd production, the lookahead is the same i.e. $.
- I6 goes to I6 when ' . ' of the 2nd production is shifted towards right (A->a.A) . a is seen by the compiler. Since I6 is a part of the 2nd production, the lookahead is the same i.e. $.
- I6 goes to I7 when ' . ' of the 3rd production is shifted towards right (A->b.) . b is seen by the compiler. Since I6 is a part of the 3rd production, the lookahead is the same ie $.

CLR (1) Parsing Table:

Table 2.27: Canonical LR

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

- $ is by default a non-terminal that takes an accepting state.
- 0,1,2,3,4,5,6,7,8,9 denotes I0, I1, I2, I3, I4, I5, I6, I7, I8, I9
- Io gives A in I2, so 2 is added to the A column and 0 row.
- Io gives S in I1, so 1 is added to the S column and 1st row.
- Similarly, 5 is written in A column and 2nd row, 8 is written in A column and 3rd row, 9 is written in A column and 6th row.
- I0 gives a in I3, so S3(shift 3) is added to a column and 0 row.
- I0 gives b in I4, so S4(shift 4) is added to the b column and 0 row.

- Similarly, S6(shift 6) is added on 'a' column and 2,6 row, S7(shift 7) is added on b column and 2,6 row, S3(shift 3) is added on 'a' column and 3 row, S4(shift 4) is added on b column and 3 row.
- I4 is reduced as '.' is at the end. I4 is the 3rd production of grammar. So write r3(reduce 3) in the lookahead columns. The look ahead of I4 are a and b, so write r3 in the a and b columns.
- I5 is reduced as ' . ' is at the end. I5 is the 1st production of grammar. So write r1(reduce 1) in the lookahead columns. The lookahead of I5 is $, so write R1 in the $ column.
- Similarly, write R2 in the a, b column and 8th row, write r2 in the $ column and 9th row. Productions are numbered as follows:

  S → AA … (1)
  A → aA … (2)
  A → b … (3)
- The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.
- I4 contains the final item which drives (A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.
- I5 contains the final item which drives (S → AA•, $), so action {I5, $} = R1.
- I7 contains the final item which drives (A → b•,$), so action {I7, $} = R3.
- I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.
- I9 contains the final item which drives (A → aA•, $), so action {I9, $} = R2.

2.10 ERROR HANDLING AND RECOVERY IN SYNTAX ANALYZER

2.10.1 Syntax Error Handling

- Syntax error handling deals with errors that violate the grammatical rules of a programming language.
- Such errors are detected during the syntax analysis (parsing) phase of the compiler.
- Since errors are common in programs, compilers must help programmers locate and correct them.
- Programming language specifications usually do not define error handling; it is the responsibility of the compiler designer.
- Proper planning of error handling simplifies compiler design and improves error reporting.
- The main objective is to detect errors, report them clearly, and continue compilation.
- Various strategies exist for recovering from syntax errors.
- Panic-mode recovery discards input symbols until a synchronizing token is found.
- Phrase-level recovery makes local corrections such as inserting, deleting, or replacing symbols.
- These strategies reduce cascading errors and allow the compiler to continue parsing.
- Programming errors can occur at different levels: **lexical, syntactic, semantic, and logical**.
- **Lexical errors** include misspelled identifiers, keywords, or operators, and missing quotation marks in strings.
- **Syntactic errors** arise from incorrect program structure, such as misplaced semicolons, missing or extra braces, or invalid statement placement.

- **Semantic errors** involve meaning-related issues, such as type mismatches between operators and operands.
- **Logical errors** result from incorrect reasoning by the programmer; the program may be syntactically correct but produce incorrect results.
- Parsing techniques like **LL and LR parsers** detect syntax errors efficiently due to the **viable-prefix property**.
- The viable-prefix property ensures that errors are detected as soon as the input cannot be extended to form a valid program.
- Many errors appear as syntactic errors during parsing, even if their actual cause is semantic or logical.
- Some semantic errors can be detected during compilation, but detecting all semantic and logical errors is difficult.
- The parser's error handler must clearly report errors, recover quickly to find additional errors, and add minimal overhead to correct programs.
- Error reporting should indicate the exact location of the error, often by displaying the source line with a pointer to the error position.

2.10.2 Error-Recovery Strategies

- After detecting a syntax error, the parser must attempt to recover and continue processing.
- No single error recovery strategy works well for all situations.
- The simplest approach is to **stop parsing after the first error** and display an informative error message.
- However, better diagnostic results are obtained if the parser can **recover and continue parsing**.
- Continuing parsing helps in detecting **additional errors** in the same program.
- If too many errors occur, the compiler should **terminate after a predefined error limit** to avoid reporting misleading or spurious errors.
- Effective error recovery aims to provide meaningful diagnostic information without overwhelming the programmer.
- Commonly used error recovery strategies include:
  - **Panic-mode recovery**
  - **Phrase-level recovery**
  - **Error productions**
  - **Global correction**

Panic-Mode Recovery

- Panic-mode recovery is a simple and commonly used syntax error recovery method.
- When an error is detected, the parser **discards input symbols one by one**.
- This continues until a **synchronizing token** is found.
- Synchronizing tokens are usually **statement delimiters** such as semicolons (;) or closing braces (}).
- These tokens have a clear and unambiguous role in the program structure.
- The compiler designer selects appropriate synchronizing tokens based on the source language.
- Panic-mode recovery may **skip large portions of input**, potentially missing some errors.
- Despite this drawback, it is **easy to implement**.

- It is **guaranteed to terminate** and does not lead to infinite loops.

Phrase-Level Recovery

- The parser performs **local corrections** on the remaining input after detecting an error.
- Common corrections include:
  - Replacing a comma with a semicolon
  - Deleting an extra semicolon
  - Inserting a missing symbol
- The choice of correction is decided by the compiler designer.
- Care must be taken to avoid **infinite loops**, such as repeatedly inserting symbols.
- This method can correct any input string.
- Its main drawback is difficulty in handling errors that occurred **before the point of detection**.

Error Productions

- The grammar is extended with productions that represent **common erroneous constructs**.
- When such a production is used during parsing, the parser recognizes a specific error.
- This allows the compiler to generate **precise and meaningful error messages**.
- Useful for handling frequently occurring syntax errors.

Global Correction

- Aims to make the **minimum number of changes** to convert an incorrect input into a valid program.
- Changes include insertion, deletion, or replacement of tokens.
- Algorithms find a corrected string with the **least overall cost**.
- These methods are **computationally expensive** in terms of time and space.
- Due to high cost, global correction is mainly of **theoretical interest**.
- Provides a benchmark for evaluating other error recovery techniques.

2.11 YACC

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided into three parts.

```
/* definitions */
 ....
%%
/* rules */
....
%%
/* auxiliary routines */
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
  %token NUMBER
  %token ID
- Yacc automatically assigns numbers for tokens, but it can be overridden by
  %token NUMBER 621
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.
- It can also include the specification of the starting symbol in the grammar:
  %start nonterminal

Input File: Rule Part:

The rules part contains grammar definition in a modified BNF form. Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

The auxiliary routines part is only C code.

- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program. The main() function must call the function yyparse().

Input File:

If yylex() is not defined in the auxiliary routines sections, then it should be included:
#include "lex.yy.c"
YACC input file generally finishes with: .y

Output Files:

- The output of YACC is a file named y.tab.c
- If it contains the main() definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function int yyparse()
- If called with the –d option in the command line, Yacc produces as output a header file y.tab.h with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the –v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Example:

Yacc File (.y)

```
%{
  #include <ctype.h>
  #include <stdio.h>
  #define YYSTYPE double /* double type for yacc stack */
%}
```

```
%%
Lines :  Lines S '\n'
{
 printf("OK \n");
}
      | S '\n'
      | error '\n'
      {
      yyerror("Error: reenter last line:");
      yyerrok;
      };
S     :  '(' S ')'
      | '[' S ']'
      |  /* empty */    ;
%%
 #include "lex.yy.c"
 void yyerror(char * s)
/* yacc error handler */
{
 fprintf (stderr, "%s\n", s);
}
int main(void)
{
 return yyparse();
}
```

Lex File (.l)

```
%{
%}
 %%
[ \t]     { /* skip blanks and tabs */ }
\n|.      { return yytext[0]; }
%%
```

For Compiling YACC Program:

- Write lex program in a file file.l and yacc in a file file.y
- Open Terminal and Navigate to the Directory where you have saved the files.
- type lex file.l
- type yacc file.y
- type cc lex.yy.c y.tab.h -ll
- type ./a.out