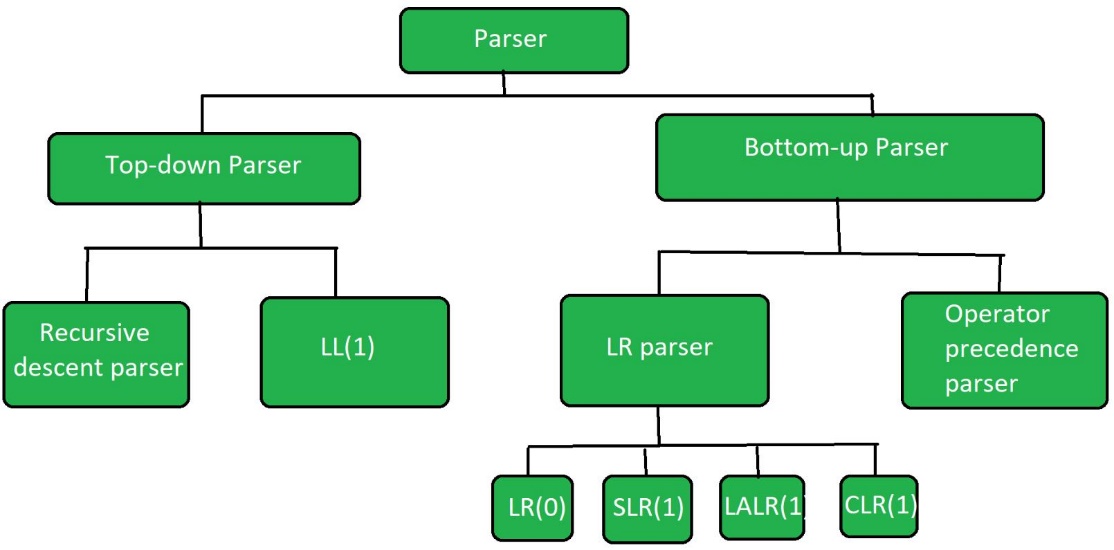
# \***Top-Down Parser**

**Parser** is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.



**Types of Parser:**  
Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser. These are explained as following below.

**1.**[**Top-down Parser**](https://www.geeksforgeeks.org/compiler-design-classification-top-parsers/)**:**  
Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.  
Further Top-down parser is classified into 2 types: Recursive descent parser, and Non-recursive descent parser.

**(i).**[**Recursive descent parser**](https://www.geeksforgeeks.org/compiler-design-recursive-descent-parser/)**:**  
It is also known as **Brute force parser** or the with **backtracking parser**. It basically generates the parse tree by using brute force and backtracking.

**(ii). Non-recursive descent parser:**  
It is also known as **LL(1)** **parser** or predictive parser or without backtracking parser or dynamic parser. **It uses parsing table** to generate the parse tree instead of backtracking.

**2.**[**Bottom-up Parser**](https://www.geeksforgeeks.org/parsing-set-2-bottom-up-or-shift-reduce-parsers/)**:**  
Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. **it starts from non-terminals and ends on the start symbol.** It uses reverse of the right most derivation.

Further Bottom-up parser is classified into **2 types**: LR parser, and Operator precedence parser.

**(i). LR parser:**  
LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar. It follow reverse of right most derivation.  
LR parser is of 4 types:

**(a).** LR(0)

**(b).** SLR(1)

**(c).** LALR(1)

**(d).** CLR(1)

**(ii).**[**Operator precedence parser**](https://www.geeksforgeeks.org/theory-computation-operator-grammar-precedence-parser/)**:**

It generates the parse tree form given grammar and string but the **only condition is two consecutive non-terminals and epsilon never appear** in the right-hand side of any production.

### **\*Recursive Descent Parsing**

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

A *Predictive Parser* is a special case of Recursive Descent Parser, where no Back Tracking is required.

**Example:**

| **BEFORE REMOVING LEFT RECURSION** | | **AFTER REMOVING LEFT RECURSION** | |
| --- | --- | --- | --- |
| **E –> E + T | T T –> T \* F | F F –> ( E ) | id** | | **E –> T E’ E’ –> + T E’ | e T –> F T’ T’ –> \* F T’ | e F –> ( E ) | id** | |

\*\*Here **e is Epsilon**

For Recursive Descent Parser, we are going to write one program for every variable.

**Example:**

Grammar: E --> i E'

E' --> + i E' | e

### **Back-tracking**

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).

To understand this, take the following example of CFG:

**S → rXd | rZd**

**X → oa | ea**

**Z → ai**

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So, the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

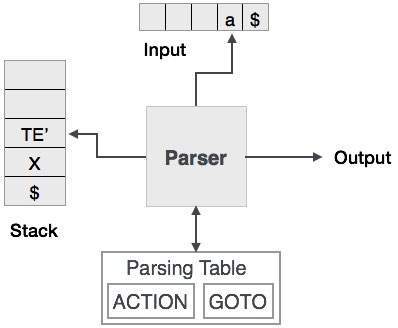
Now the parser matches all the input letters in an ordered manner. The string is accepted.

|  |  |  |  |
| --- | --- | --- | --- |
| Back Tracking | Back Tracking | Back Tracking | Back Tracking |

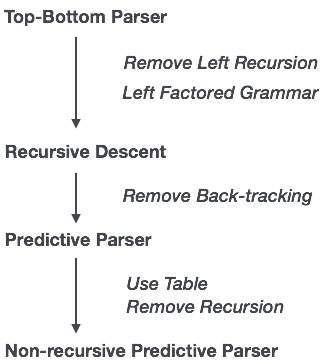
### **Predictive Parser**

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contain an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

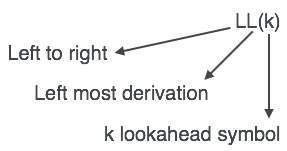


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

### **\*LL Parser**

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The **first L** in LL(k) is **parsing the input from left to right**, the **second L** in LL(k) **stands for left-most derivation** and **k itself represents the number of look aheads.** Generally k = 1, so LL(k) may also be written as LL(1).



A grammar G is LL(1) if A → α | β are two distinct productions of G:

* for no terminal, both α and β derive strings beginning with a.
* at most one of α and β can derive empty string.
* if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

**\*Non-recursive predictive parsing**

The Predictive [parsing](https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/) is a special form of recursive descent parsing, where no backtracking is required, so this can predict which production to use to replace the input string.

Non-recursive predictive parsing or table-driven is also known as LL(1) parser. This parser follows the leftmost derivation (LMD).

The main problem during predictive parsing is that of determining the production to be applied for a non-terminal.This non-recursive parser looks up which production to be applied in a parsing table.

A LL(1) parser has the following **components:**

(1) buffer: an input buffer which contains the string to be passed

(2) stack: a pushdown stack which contains a sequence of grammar symbols

(3) A parsing table: a 2d array M[A, a]

where

A->non-terminal, a->terminal or $

(4) output stream:

end of the stack and an end of the input symbols are both denoted with $

**Algorithm for non-recursive Predictive Parsing:**  
The main Concept With the help of [FIRST() and FOLLOW() sets](https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/), this parsing can be done using a stack which avoids the recursive calls.

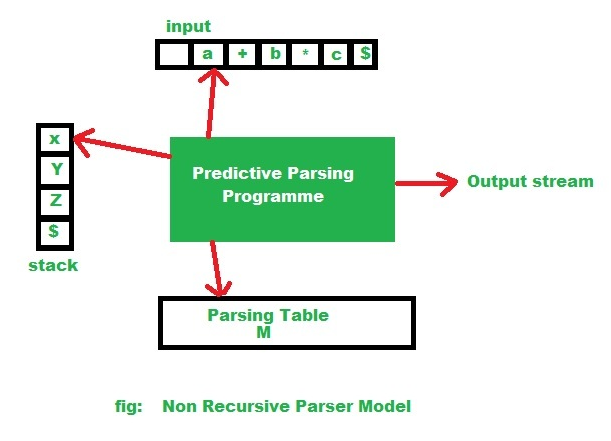
For each rule, A->x in grammar G:

1. For each terminal ‘a’ contained in [FIRST(A)](https://www.geeksforgeeks.org/first-set-in-syntax-analysis/) add A->x to M[A, a] in parsing table if x derives ‘a’ as the first symbol.
2. If FIRST(A) contain null production for each terminal ‘b’ in [FOLLOW(A)](https://www.geeksforgeeks.org/follow-set-in-syntax-analysis/), add this production (A->null) to M[A, b] in parsing table.

**The Procedure:**

1. In the beginning, the pushdown stack holds the start symbol of the grammar G.
2. At each step a symbol X is popped from the stack:  
   if X is a terminal then it is matched with the lookahead and lookahead is advanced one step,  
   if X is a nonterminal symbol, then using lookahead and a parsing table (implementing the FIRST sets) a production is chosen and its right-hand side is pushed into the stack.
3. This process repeats until the stack and the input string become null (empty).

**Non-recursive parser model diagram:**



1. So according to the given diagram the non-recursive parsing algorithm.

**Input:** A input string ‘w’ and a parsing table(‘M’) for grammar G.  
**Output:** If w is in L(G), an LMD of w; otherwise an error indication.

**Example:** Consider the subsequent LL(1) grammar:

**S -> A**

**S -> ( S \* A)**

**A -> id**

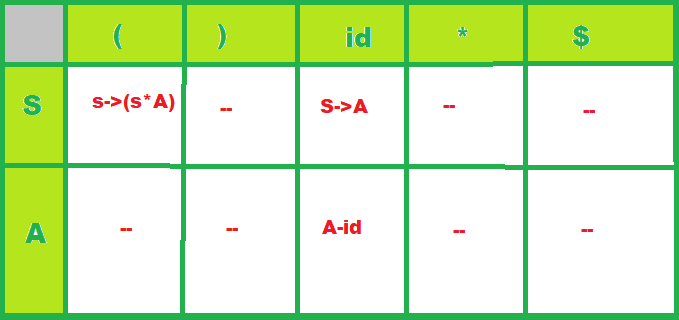
Now let’s parse the given input:

**( id \* id )**

The parsing table:

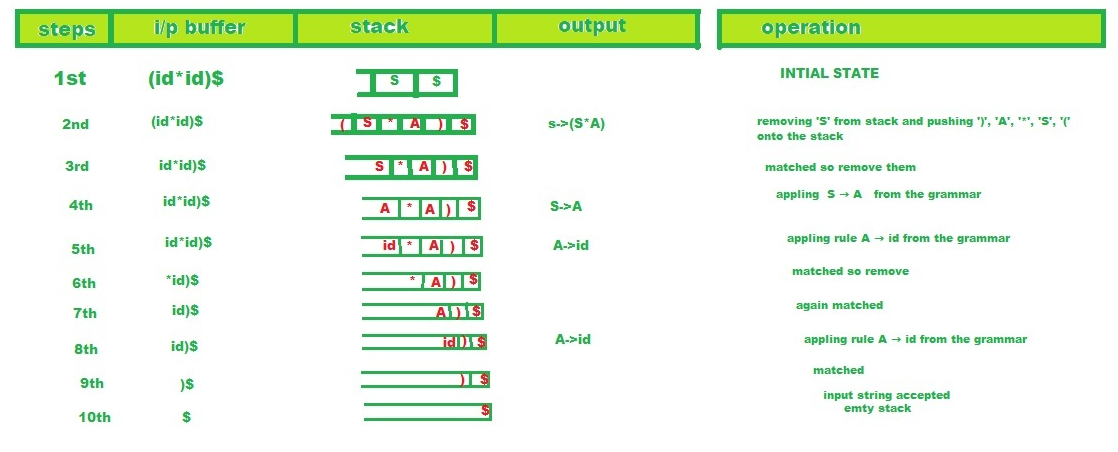
* row-> for each and every non-terminal symbol,
* column-> for each and every terminal (including the special terminal).

Each cell of this table will contain at most one rule of the given grammar:



Now let’s see using the algorithm, how the parser uses this parsing table to process the given input.

**Procedure:**



The parser thus ends because there **remains only ‘$’ on both its stack and its input stream**. In this case, the parser reports that it has accepted the input string and writes the following list of rules to the output stream:

S -> ( S \* A),

S -> A,

A -> id,

A -> id

This is indeed a list of rules for an LMD of the input string, which is:

S -> ( S \* A )

-> ( A \* A )

-> ( id \* A )

-> ( id \* id )

## **\***[**First and Follow**](https://www.gatevidyalay.com/first-and-follow-compiler-design/)

## **First and Follow-**

|  |
| --- |
| **First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position.** |

we will learn how to calculate first and follow functions.

## **First Function-**

|  |
| --- |
| First(α) is a set of terminal symbols that begin in strings derived from α. |

### **Example-**

Consider the production rule-

A → abc / def / ghi

Then, we have-

First(A) = { a , d , g }

## **Rules For Calculating First Function-**

### **Rule-01:**

For a production rule X → ∈,

First(X) = { ∈ }

**Rule-02:**

For any terminal symbol ‘a’,

First(a) = { a }

**Rule-03:**

For a production rule X → Y1Y2Y3,

#### **Calculating First(X)**

* If ∈ ∉ First(Y1), then First(X) = First(Y1)
* If ∈ ∈ First(Y1), then First(X) = {First(Y1) – ∈} ∪ First(Y2Y3)

#### **Calculating First(Y2Y3)**

* If ∈ ∉ First(Y2), then First(Y2Y3) = First(Y2)
* If ∈ ∈ First(Y2), then First(Y2Y3) = { First(Y2) – ∈ } ∪ First(Y3)

Similarly, we can make expansion for any production rule X → Y1Y2Y3…..Yn.

**Follow Function-**

|  |
| --- |
| Follow(α) is a set of terminal symbols that appear immediately to the right of α. |

## **Rules For Calculating Follow Function-**

## 

### **Rule-01:**

For the start symbol S, place $ in Follow(S).

**Rule-02:**

For any production rule A → αB,

Follow(B) = Follow(A)

**Rule-03:**

For any production rule A → αBβ,

* If ∈ ∉ First(β), then Follow(B) = First(β)
* If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

## **Important Notes-**

**Note-01:**

* ∈ may appear in the first function of a non-terminal.
* ∈ will never appear in the follow function of a non-terminal.

**Note-02:**

* Before calculating the first and follow functions, eliminate [**Left Recursion**](https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/) from the grammar, if present.

### **Note-03:**

* We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

## **PRACTICE PROBLEMS BASED ON CALCULATING FIRST AND FOLLOW-**

## **Problem-01:**

Calculate the first and follow functions for the given grammar-

**S → aBDh**

**B → cC**

**C → bC / ∈**

**D → EF**

**E → g / ∈**

**F → f / ∈**

## **Solution-**

The first and follow functions are as follows-

### **First Functions-**

**S → aBDh**

**B → cC**

**C → bC / ∈**

**D → EF**

**E → g / ∈**

**F → f / ∈**

* First(S) = { a }
* First(B) = { c }
* First(C) = { b , ∈ }
* First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
* First(E) = { g , ∈ }
* First(F) = { f , ∈ }

### **Follow Functions-**

* Follow(S) = { $ }
* Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
* Follow(C) = Follow(B) = { g , f , h }
* Follow(D) = First(h) = { h }
* Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
* Follow(F) = Follow(D) = { h }

## **Problem-02:**

Calculate the first and follow functions for the given grammar-

**S → A**

**A → aB / Ad**

**B → b**

**C → g**

## **Solution-**

 We have-

* The given grammar is left recursive.
* So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

**S → A**

**A → aBA’**

**A’ → dA’ / ∈**

**B → b**

**C → g**

Now, the first and follow functions are as follows-

### **First Functions-**

 First(S) = First(A) = { a }

* First(A) = { a }
* First(A’) = { d , ∈ }
* First(B) = { b }
* First(C) = { g }

### **Follow Functions-**

* Follow(S) = { $ }
* Follow(A) = Follow(S) = { $ }
* Follow(A’) = Follow(A) = { $ }
* Follow(B) = { First(A’) – ∈ } ∪ Follow(A) = { d , $ }
* Follow(C) = NA

## **Problem-03:**

Calculate the first and follow functions for the given grammar-

S → (L) / a

L → SL’

L’ → ,SL’ / ∈

**Solution-**

The first and follow functions are as follows-

### **First Functions-**

* First(S) = { ( , a }
* First(L) = First(S) = { ( , a }
* First(L’) = { , , ∈ }

### **Follow Functions-**

* Follow(S) = { $ } ∪ { First(L’) – ∈ } ∪ Follow(L) ∪ Follow(L’) = { $ , , , ) }
* Follow(L) = { ) }
* Follow(L’) = Follow(L) = { ) }

## **Problem-04:**

 Calculate the first and follow functions for the given grammar-

S → AaAb / BbBa

A → ∈

B → ∈

## **Solution-**

The first and follow functions are as follows-

### **First Functions-**

 First(S) = { First(A) – ∈ } ∪ First(a) ∪ { First(B) – ∈ } ∪ First(b) = { a , b }

* First(A) = { ∈ }
* First(B) = { ∈ }

### **Follow Functions-**

 Follow(S) = { $ }

* Follow(A) = First(a) ∪ First(b) = { a , b }
* Follow(B) = First(b) ∪ First(a) = { a , b }

## **Problem-05:**

 Calculate the first and follow functions for the given grammar-

E → E + T / T

T → T x F / F

F → (E) / id

## **Solution-**

We have-

* The given grammar is left recursive.
* So, we first remove left recursion from the given grammar.

After eliminating left recursion, we get the following grammar-

E → TE’

E’ → + TE’ / ∈

T → FT’

T’ → x FT’ / ∈

F → (E) / id

Now, the first and follow functions are as follows-

### **First Functions-**

* First(E) = First(T) = First(F) = { ( , id }
* First(E’) = { + , ∈ }
* First(T) = First(F) = { ( , id }
* First(T’) = { x , ∈ }
* First(F) = { ( , id }

### **Follow Functions-**

* Follow(E) = { $ , ) }
* Follow(E’) = Follow(E) = { $ , ) }
* Follow(T) = { First(E’) – ∈ } ∪ Follow(E) ∪ Follow(E’) = { + , $ , ) }
* Follow(T’) = Follow(T) = { + , $ , ) }
* Follow(F) = { First(T’) – ∈ } ∪ Follow(T) ∪ Follow(T’) = { x , + , $ , ) }

## **Problem-06:**

Calculate the first and follow functions for the given grammar-

S → ACB / CbB / Ba

A → da / BC

B → g / ∈

C → h / ∈

## **Solution-**

The first and follow functions are as follows-

### **First Functions-**

* First(S) = { First(A) – ∈ }  ∪ { First(C) – ∈ } ∪ First(B) ∪ First(b) ∪ { First(B) – ∈ } ∪ First(a) = { d , g , h , ∈ , b , a }
* First(A) = First(d) ∪ { First(B) – ∈ } ∪ First(C) = { d , g , h , ∈ }
* First(B) = { g , ∈ }
* First(C) = { h , ∈ }

### **Follow Functions-**

* Follow(S) = { $ }
* Follow(A) = { First(C) – ∈ } ∪ { First(B) – ∈ } ∪ Follow(S) = { h , g , $ }
* Follow(B) = Follow(S) ∪ First(a) ∪ { First(C) – ∈ } ∪ Follow(A) = { $ , a , h , g }
* Follow(C) = { First(B) – ∈ } ∪ Follow(S) ∪ First(b) ∪ Follow(A) = { g , $ , b , h }

# **Construction of LL(1) Parsing Table**

All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

Now, let’s understand with an example.

**Example-1:**  
Consider the Grammar:

E --> TE'

E' --> +TE' | e

T --> FT'

T' --> \*FT' | e

F --> id | (E)

\*\*e denotes epsilon

Find their first and follow sets:

|  | **FIRST** | **FOLLOW** |
| --- | --- | --- |
| **E –> TE’** | { id, ( } | { $, ) } |
| **E’ –> +TE’/e** | { +, e } | { $, ) } |
| **T –> FT’** | { id, ( } | { +, $, ) } |
| **T’ –> \*FT’/e** | { \*, e } | { +, $, ) } |
| **F –> id/(E)** | { id, ( } | { \*, +, $, ) } |

Now, the LL(1) Parsing Table is:

|  | **ID** | **+** | **\*** | **(** | **)** | **$** |
| --- | --- | --- | --- | --- | --- | --- |
| **E** | E –> TE’ |  |  | E –> TE’ |  |  |
| **E’** |  | E’ –> +TE’ |  |  | E’ –> e | E’ –> e |
| **T** | T –> FT’ |  |  | T –> FT’ |  |  |
| **T’** |  | T’ –> e | T’ –> \*FT |  | T’ –> e | T’ –> e |
| **F** | F –> id |  |  | F –> (E) |  |  |

As you can see that all the null productions are put under the follow set of that symbol and all the remaining productions are lie under the first of that symbol.

**Note:** Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

Let’s see with an example.

**Example-2:**  
Consider the Grammar

S --> A | a

A --> a

Find their first and follow sets:

|  | **FIRST** | | **FOLLOW** |
| --- | --- | --- | --- |
| **S –> A/a** | | { a } | { $ } |
| **A –>a** | | { a } | { $ } |

Parsing Table:

|  | a | **$** | |
| --- | --- | --- | --- |
| **S** | S –> A, S –> a |  | |
| **A** | A –> a |  |

Here, we can see that there are two productions into the same cell. Hence, this grammar is not feasible for LL(1) Parser.

# **\*Error detection and Recovery in Compiler**

In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting it to user is called **Error Handling process**.

**Functions of Error handler**

* Detection
* Reporting
* Recovery

### Classification of Errors



Compile time errors are of three types:-

**Lexical phase errors**

These errors are detected during the lexical analysis phase. Typical lexical errors are

* Exceeding length of identifier or numeric constants.
* Appearance of illegal characters
* Unmatched string

Example 1 : **printf("students");$**

This is a lexical error since an illegal character $ appears at the end of statement.

Example 2 : **This is a comment \*/**

This is an lexical error since end of comment is present but beginning is not present.

**Syntactic phase errors**

These errors are detected during syntax analysis phase. Typical syntax errors are

* Errors in structure
* Missing operator
* Misspelled keywords
* Unbalanced parenthesis

**Example :** swicth(ch)

{

.......

.......

}

The keyword **switch** is incorrectly written as swicth. Hence, **“Unidentified keyword/identifier”** error occurs.

**Error recovery:**

1. **Panic Mode Recovery**
   * In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found.
   * Synchronizing tokens are deli-meters such as ; or }
   * Advantage is that it’s easy to implement and guarantees not to go to infinite loop
   * Disadvantage is that a considerable amount of input is skipped without checking it for additional errors
2. **Statement Mode recovery**
   * In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.
   * The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
   * While performing correction, atmost care should be taken for not going in infinite loop.
   * Disadvantage is that it finds difficult to handle situations where actual error occurred before point of detection.
3. **Error production**
   * If user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
   * If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
   * Disadvantage is that it’s difficult to maintain.
4. **Global Correction**
   * The parser examines the whole program and tries to find out the closest match for it which is error free.
   * The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.
   * Due to high time and space complexity, this method is not implemented practically.

**Semantic errors**

These errors are detected during semantic analysis phase. Typical semantic errors are

* Incompatible type of operands
* Undeclared variables
* Not matching of actual arguments with formal one

**Example :** int a[10], b;

.......

.......

a = b;

It generates a semantic error because of an incompatible type of a and b.

**Error recovery**

* If error **“Undeclared Identifier”** is encountered then, to recover from this a symbol table entry for corresponding identifier is made.
* If data types of two operands are incompatible then, automatic type conversion is done by the compiler.