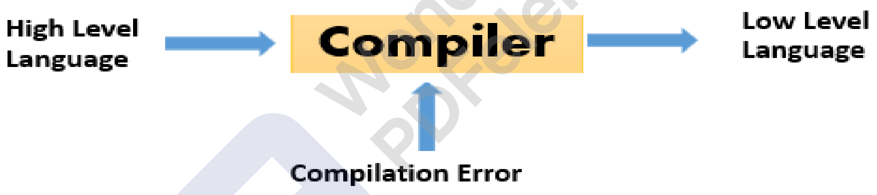
UNIT - I

Introduction: The structure of a compiler, the science of building a compiler, programming language basics.

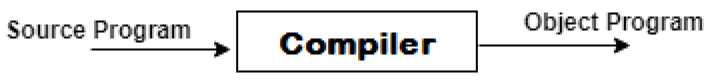
Lexical Analysis: The Role of the Lexical Analyzer, Input Buffering, Recognition of Tokens, Design of a Lexical-Analyzer Generator using Lex, Finite Automata, From Regular Expressions to Automata, , Optimization of DFA-Based Pattern Matchers.

1.1 INTRODUCTION

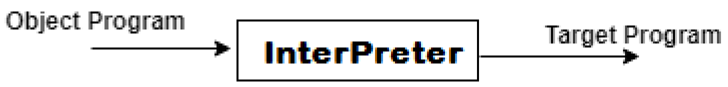
A compiler is a translator that converts the high-level language into the machine language. High-level language is written by a developer and machine language can be understood by the processor. The compiler also makes the end code efficient which is optimized for execution time and memory space. The main purpose of compiler is to change the code written in one language without changing the meaning of the program and also used to show errors to the programmer.



When you execute a program which is written in HLL programming language then it executes into two parts. In the first part, the source program compiled and translated into the target object program (low level language).



In the second part, object program translated into the target program through the Interpreter.



Features of Compilers

- Correctness

- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

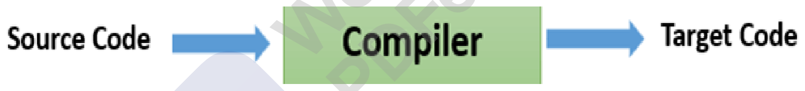
Types of Compiler

Following are the different types of Compiler:

- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

Single Pass Compiler

In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

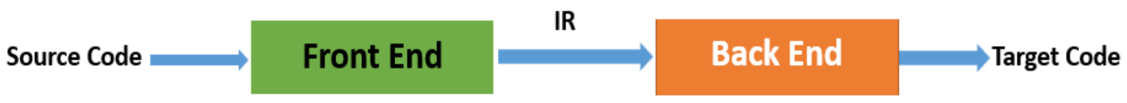


Single Pass Compiler

Two Pass Compiler

Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

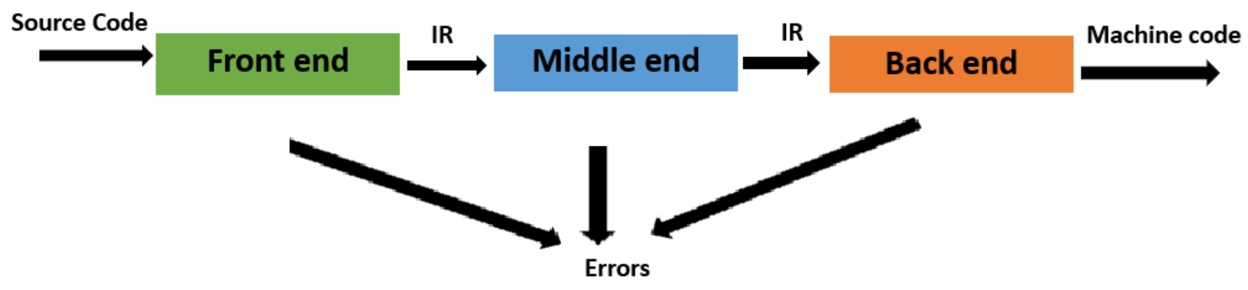


Two Pass Compiler

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Multipass Compilers

The multipass compiler processes the source code it divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory.



Multi Pass Compiler

Tasks of Compiler

Main tasks performed by the Compiler are:

- Breaks up the up the source program into pieces and impose grammatical structure on them
- Allows to construct the desired target program from the intermediate representation and also create the symbol table.
- Compiles source code and detects errors in it
- Manage storage of all variables and codes.
- Support for separate compilation
- Read, analyze the entire program, and translate to semantically equivalent
- Translating the source code into object code depending upon the type of machine

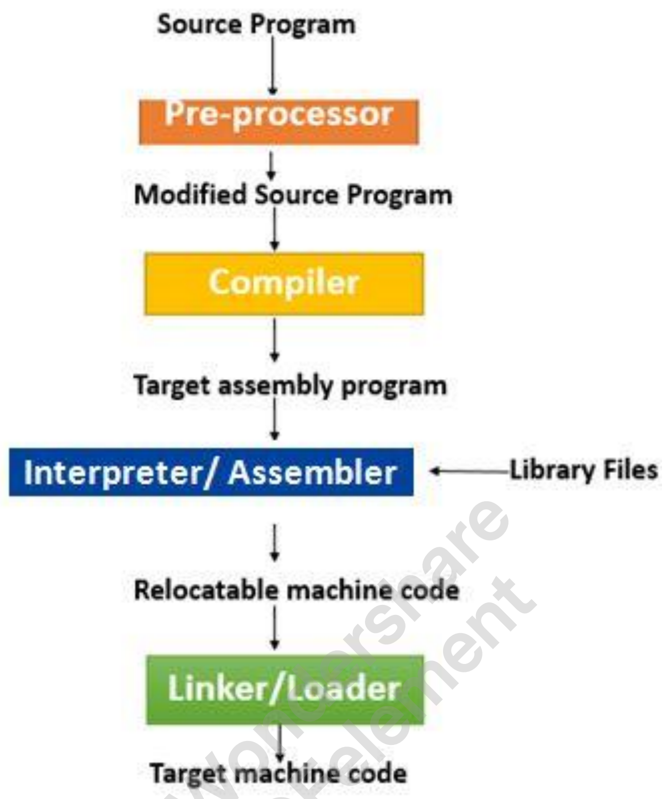
History of Compiler

Important Landmark of Compiler's history are as follows:

- The "compiler" word was first used in the early 1950s by Grace Murray Hopper
- The first compiler was build by John Backum and his group between 1954 and 1957 at IBM
- COBOL was the first programming Languages which was compiled on multiple platforms in 1960
- The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution.

Steps for Language processing systems

Before knowing about the concept of compilers, we first need to understand a few other tools which work with compilers.



- **Preprocessor**: The preprocessor a tool which produces input for Compiler. It deals with macro processing, language extension, etc.
- **Interpreter**: An interpreter is like Compiler which translates object program into target program. The main difference between both is that interpreter reads and transforms code line by line. Compiler reads the entire code at once and creates the machine code.
- **Assembler**: It translates assembly language code into machine understandable language. The output result of assembler is a combination of machine instructions as well as the data required to store these instructions in memory.
- **Linker**: The linker helps to link and merge various object files to create an executable file. All these files might have been compiled separately. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.

- **Loader**: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.

Compiler Construction Tools

Compiler construction tools were introduced as computer-related technologies spread all over the world. They are also known as a compiler- compilers, compiler- generators or translator. These tools use specific language or algorithm for specifying and implementing the component of the compiler. Following are the example of compiler construction tools.

- **Scanner generators**: This tool takes regular expressions as input. For example LEX for Unix Operating System.
- **Syntax-directed translation engines**: These software tools offer an intermediate code by using the parse tree. It has a goal of associating one or more translations with each node of the parse tree.
- **Parser generators:** A parser generator takes a grammar as input and automatically generates source code which can parse streams of characters with the help of a grammar.
- **Automatic code generators**: Takes intermediate code and converts them into Machine Language
- **Data-flow engines**: This tool is helpful for code optimization. Here, information is supplied by user and intermediate code is compared to analyze any relation. It is also known as data-flow analysis. It helps you to find out how values are transmitted from one part of the program to another part.

Uses of Compiler

- Compiler verifies entire program, so there are no syntax or semantic errors
- The executable file is optimized by the compiler, so it is executes faster
- There is no need to execute the program on the same machine it was built
- Translate entire program in other language
- Link the files into an executable format
- Check for syntax errors and data types
- Helps to handle language performance issues
- The techniques used for constructing a compiler can be useful for other purposes as well

Applications of Compiler

- Compiler design helps full implementation of High-Level Programming Languages
- Support optimization for Computer Architecture Parallelism
- Design of New Memory Hierarchies of Machines
- Widely used for Translating Programs
- Used with other Software Productivity Tools

1.2 STRUCTURE OF A COMPILER

Any large software is easier to understand and implement if it is divided into well-defined modules. The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

The different phases of compiler are as follows:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

All of the aforementioned phases involve the following tasks:

- Symbol table management.
- Error handling.

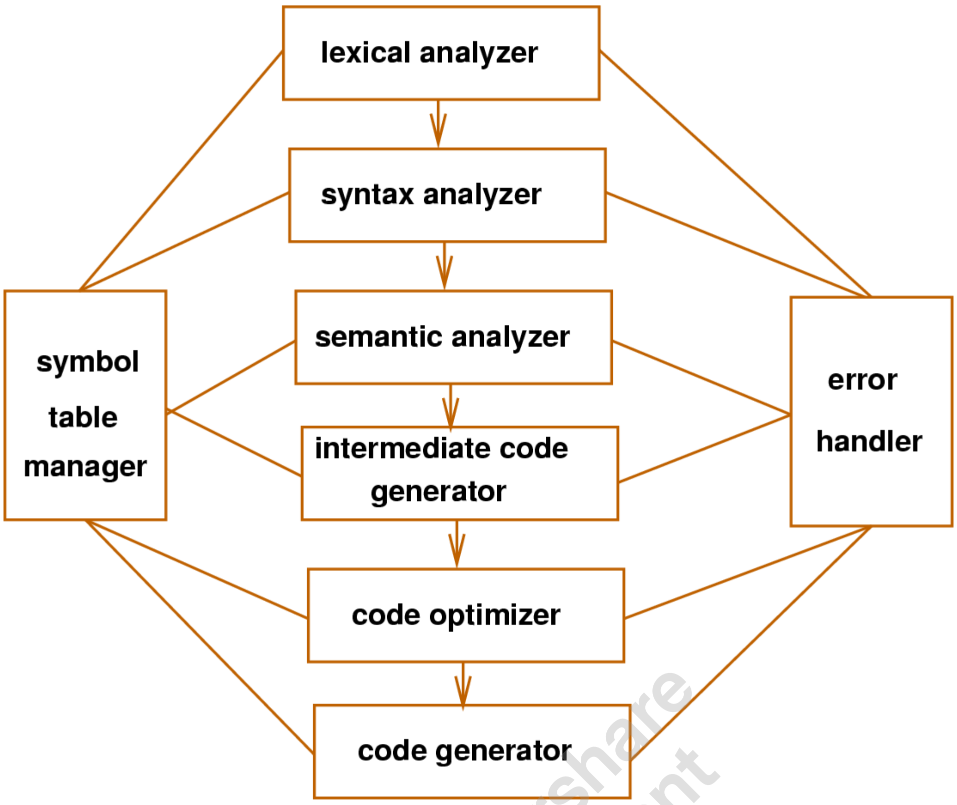


Figure: The structure of a compiler.

Lexical Analysis

- Lexical analysis is the first phase of compiler which is also termed as scanning.
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.
- **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.
- **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,
- **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

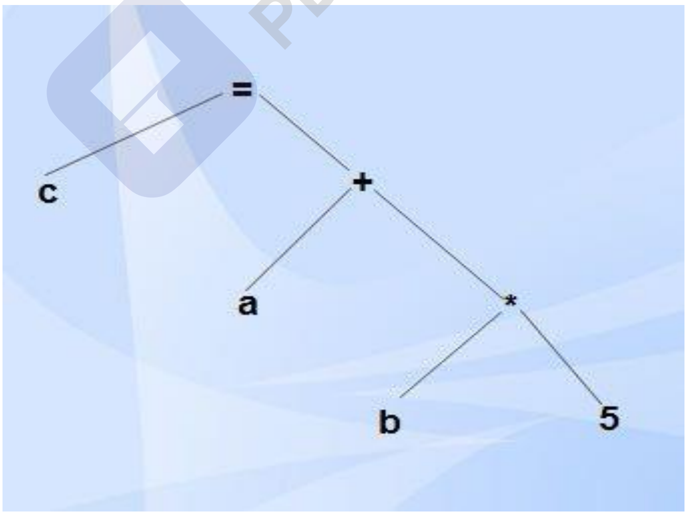
Example: c=a+b*5;

Lexemes	Tokens
C	identifier
=	assignment symbol
A	identifier
+	+ (addition symbol)

B	identifier
*	* (multiplication symbol)
5	5 (number)
;	Semicolon

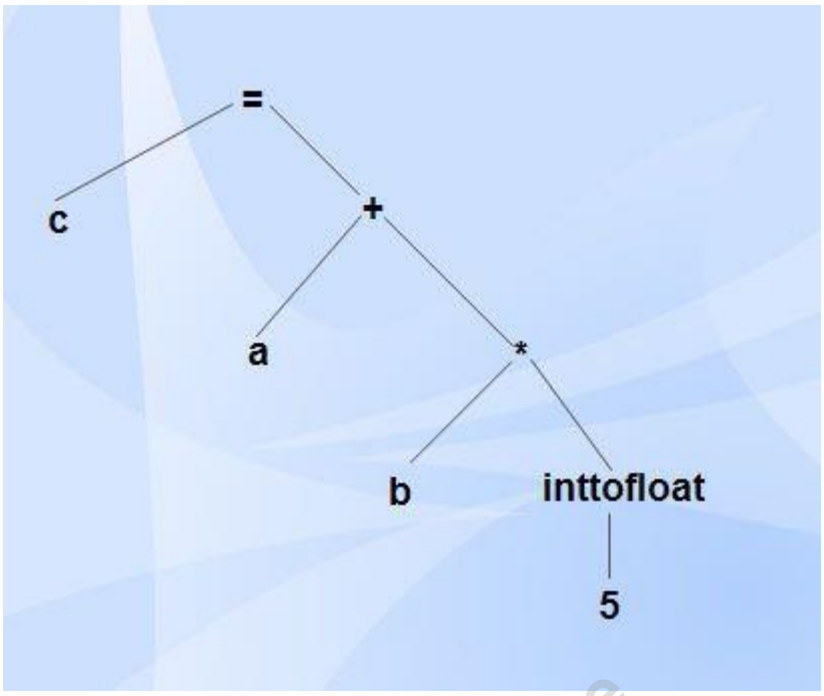
Syntax Analysis

- Syntax analysis is the second phase of compiler which is also called as parsing.
- Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.
- Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.



Semantic Analysis

- Semantic analysis is the third phase of compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.



Intermediate Code Generation

- Intermediate code generation produces intermediate representations for the source program which are of the following forms:
  - ➢ Postfix notation
  - ➢ Three address code
  - ➢ Syntax tree

Most commonly used form is the three address code.

t_1 = inttofloat (5)

t_2 = id_3* tl

t_3 = id_2 + t_2

id_1 = t_3

Code Optimization

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It results in faster running machine code.
- It can be done by reducing the number of lines of code for a program.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.

- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
    - ➢ Deduction and removal of dead code (unreachable code).
    - ➢ Calculation of constants in expressions and terms.
    - ➢ Collapsing of repeated expression into temporary string.
    - ➢ Loop unrolling.
    - ➢ Moving code outside the loop.
    - ➢ Removal of unwanted temporary variables.

$t_1 = id_3 * 5.0$

$id_1 = id_2 + t_1$

Code Generation

- Code generation is the final phase of a compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
    - ➢ Generation of correct references.
    - ➢ Generation of correct data types.
    - ➢ Generation of missing code.

LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1

Symbol Table Management

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

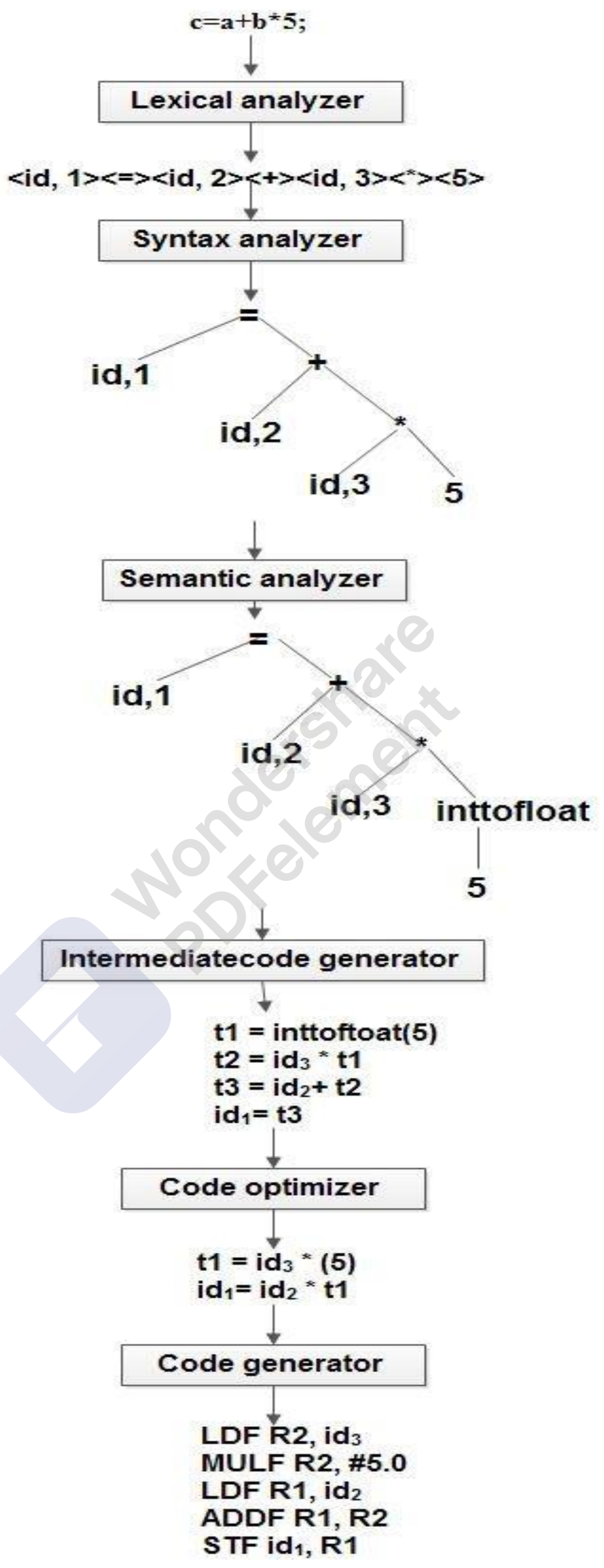
Example

char a,b,c; const:5

Symbol name	Type	Address
A	char	1000
b	char	1002
C	char	1004
5	const	1008

Error Handling

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors may occur at the following cases:
    - ➢ When the compiler detects constructs that have right syntactic structure but no meaning
    - ➢ During type conversion.
- In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.



1.3 THE SCIENCE OF BUILDING A COMPILER

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modeling in compiler design and implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

The science of code optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

1.4 PROGRAMMING LANGUAGE BASICS

To design an efficient compiler, we should know some language basics.

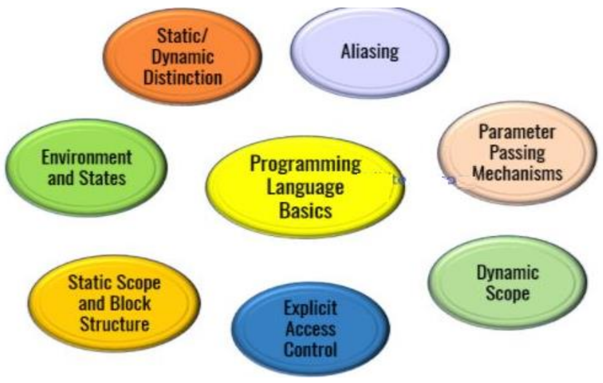

Static/ Dynamic Distinction
Aliasing
Environment and States
Programming Language Basics
Parameter Passing Mechanisms
Static Scope and Block Structure
Explicit Access Control
Dynamic Scope

Figure: Programming Language Basics

Static and Dynamic Distinction

- **Static** - Events occur at compile time.
- **Dynamic** - Events occur at run time.

Example

- The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.
- A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program.
- Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

Environment and States

- The environment is mapping from names to locations in the store. Since variables refer to locations, we could alternatively define an environment as a mapping from names to variables.
- The state is a mapping from locations in store to their values.

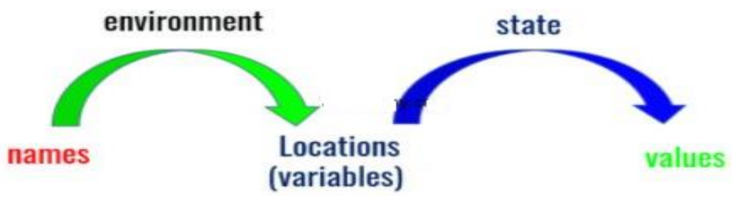


Figure: Environment and States

Static Scope and Block Structure

- The scope rules for C are based on program structure. The scope of a declaration is determined implicitly by where the declaration appears in the program.

- Programming languages such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like public, private, and protected.
- A block is a grouping of declarations and statements. C uses braces { and } to delimit a block, the alternative use of begin and end in some languages.

Explicit Access Control

- Classes and Structures introduce a new scope for their members.
- If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition.
- Through the use of keywords like public, private, and protected, object oriented languages such as C++ or Java provide explicit control over access to member names in a super class. These keywords support encapsulation by restricting access.
    - **Public** - Public names are accessible from outside the class
    - **Private** - Private names include method declarations and definitions associated with that class and any "friend" classes.
    - **Protected** - Protected names are accessible to subclasses.

Dynamic Scope

- The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.
- Dynamic scoping of this type appears only in special situations. The two dynamic policies are:
    - Macro expansion in the C
    - Method resolution in OOPs

Parameter Passing Mechanisms

- Every language has some method for passing parameters to functions and procedures.
- **Formal Parameters:** The identifier used in a method to stand for the value that is passed into the method by a caller.
- **Actual Parameters:** The actual value that is passed into the method by a caller.
    - **Call by Value** - The actual parameter is evaluated (if it is an expression) or copied (if it is a variable) in a formal parameter.

- **Call by Reference** - The address of the actual parameter is passed as value of the corresponding formal parameter.
- **Call by Name** - The Called object execute as if the actual parameter were substituted literally for the formal parameter.

Aliasing

- When two names refer to the same location in memory.
- There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value.
- It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another.
- As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other.

1.4 THE ROLE OF THE LEXICAL ANALYZER

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.

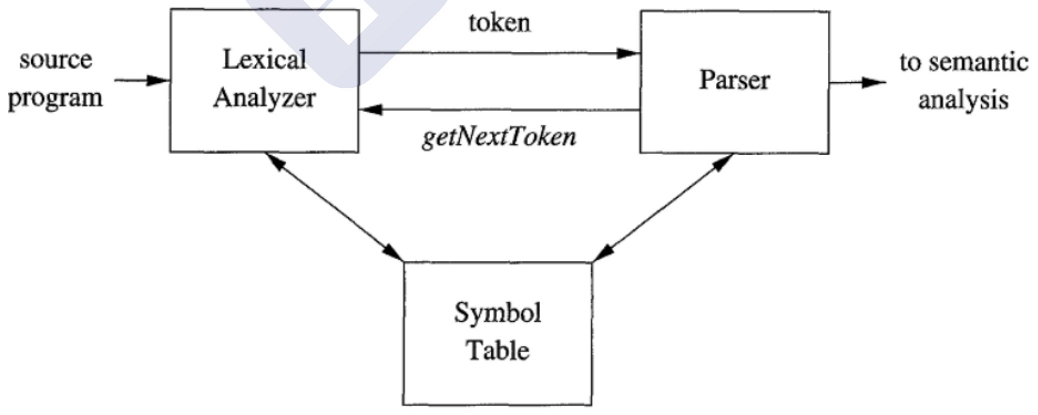


Figure : Lexical Analyzer

When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()**

command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

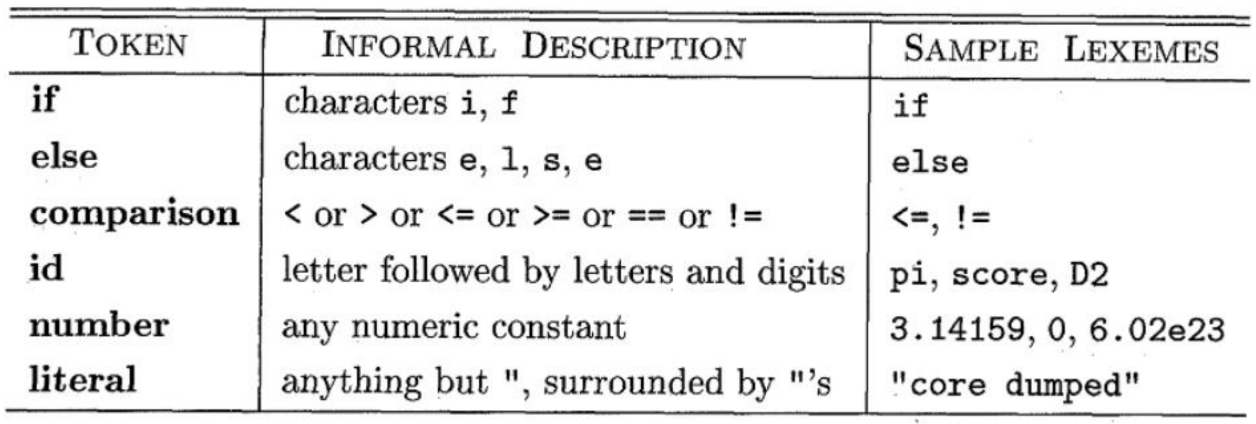
A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token. Example: In the following C language statement ,

printf ("Total = %d\n‖, score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖** is a lexeme matching **literal [or string]**.

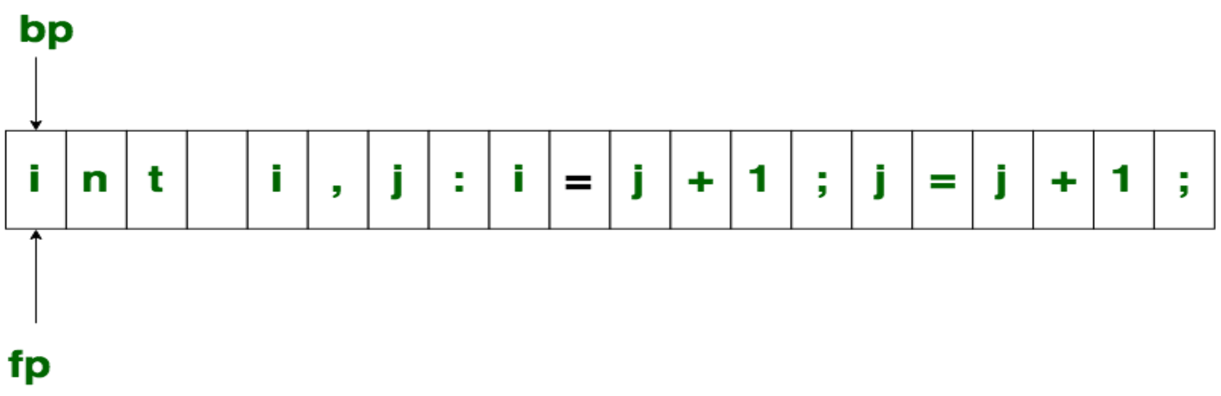


TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure: Examples of Tokens

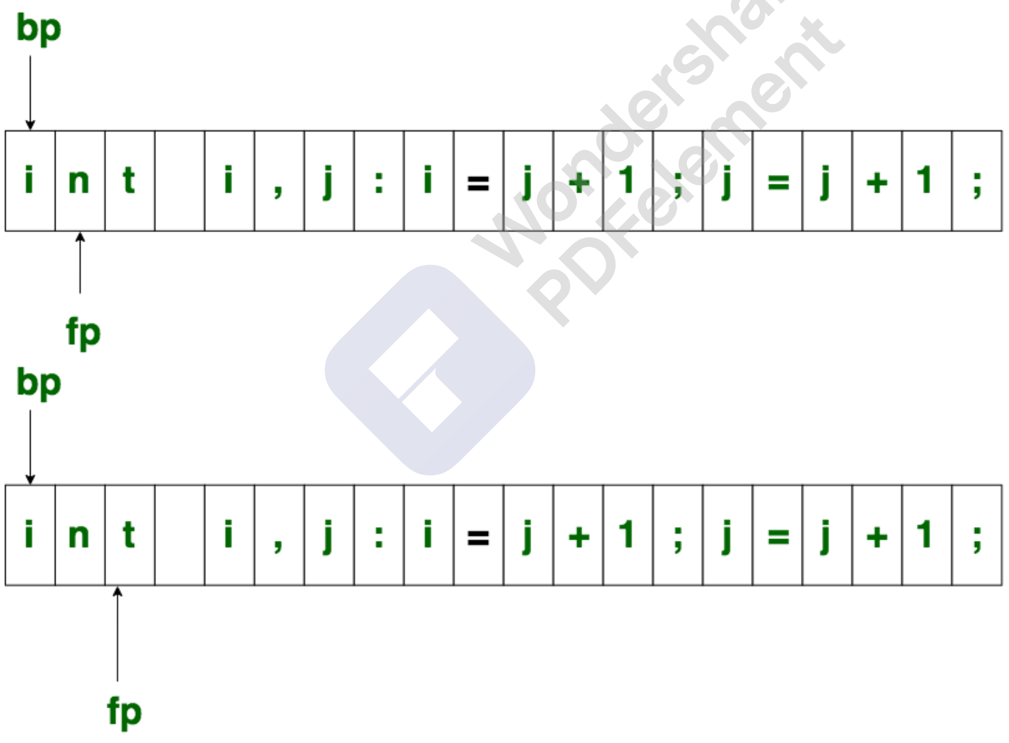
1.5 INPUT BUFFERING

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward to keep track of the pointer of the input scanned.



Initial Configuration

Initially both the pointers point to the first character of the input string as shown below

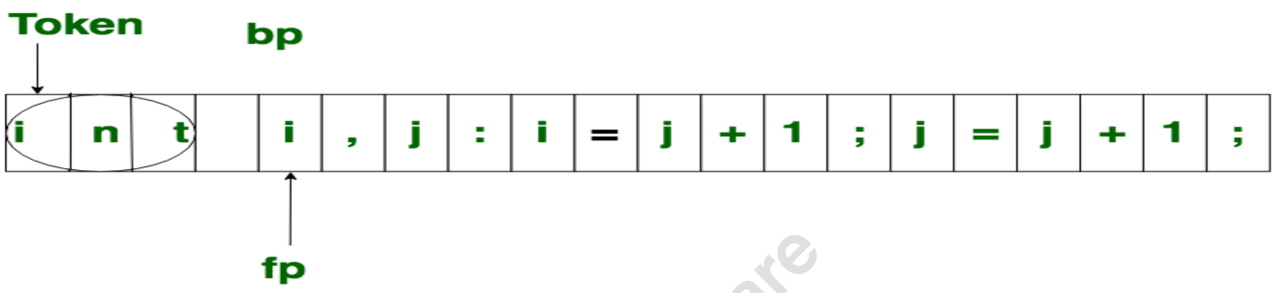

bp
i n t i , j : i = j + 1 ; j = j + 1 ;
fp
bp
i n t i , j : i = j + 1 ; j = j + 1 ;
fp

Input Buffering

The forward **ptr** moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as **ptr (fp)** encounters a blank space the lexeme "int" is identified.

The **fp** will be moved ahead at white space, when **fp** encounters white space, it ignore and moves ahead. then both the begin **ptr(bp)** and forward **ptr(fp)** are set at next token.

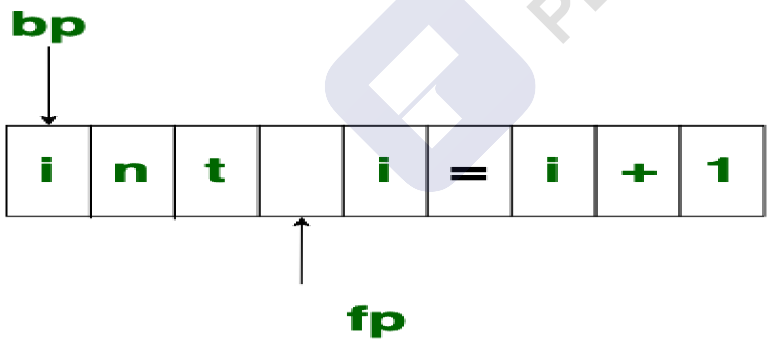
The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used. A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Input buffering

1. **One Buffer Scheme:**

In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

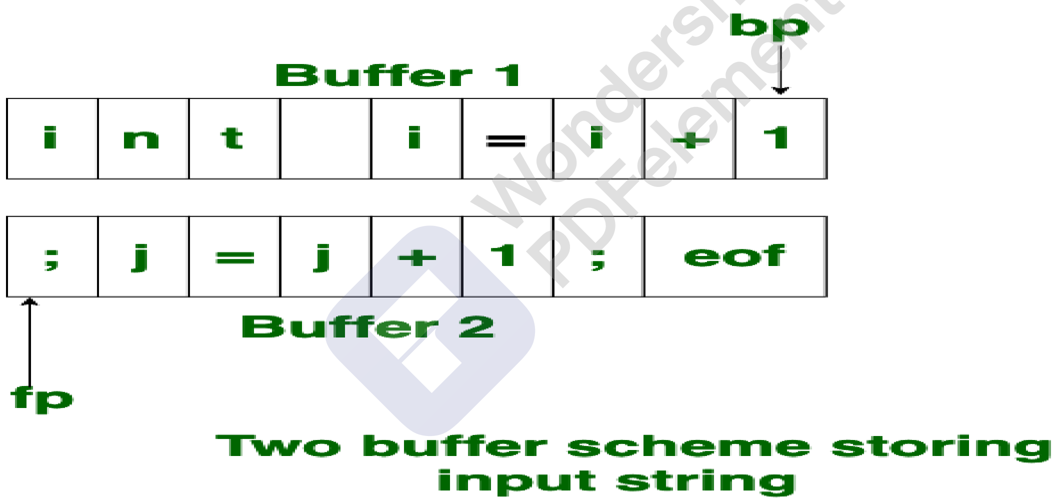
2. **Two Buffer Scheme:**

To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

Initially both the **bp** and **fp** are pointing to the first character of first buffer. Then the **fp** moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between **bp** and **fp** is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer.

Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer.

Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.

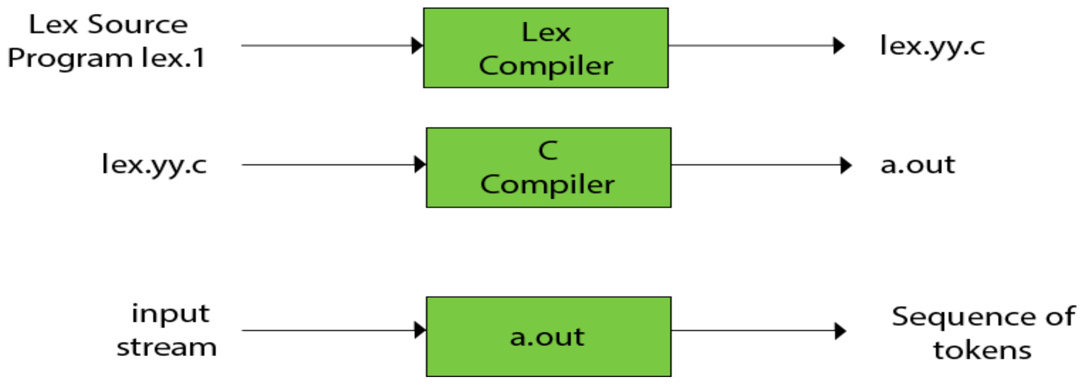


1.6 THE LEXICAL-ANALYZER GENERATOR USING - LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

```
{ definitions }
%%
 { rules }
%%
{ user subroutines }
```

- **Definitions** include declarations of constant, variable and regular definitions.
- **Rules** define the statement of form p1 {action1} p2 {action2}....pn {action}.
- Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
- **User subroutines** are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

1.7 FINITE AUTOMATA

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ),

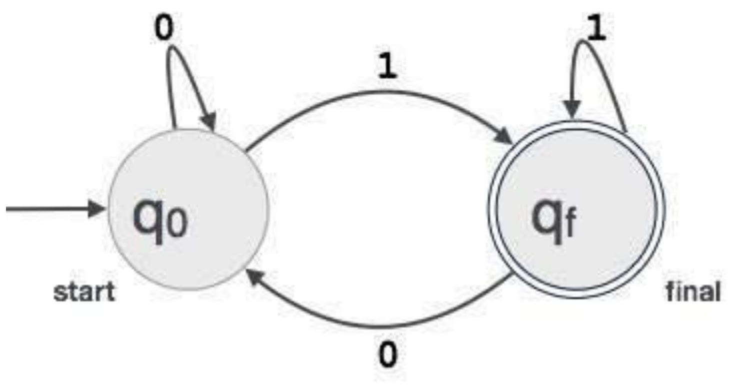
$Q \times \Sigma \rightarrow Q$

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

- **States** : States of FA are represented by circles. State names are written inside circles.
- **Start state** : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
- **Intermediate states** : All intermediate states have at least two arrows; one pointing to and another pointing out from them.
- **Final state** : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.
- **Transition** : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example : We assume FA accepts any three digit binary value ending in digit 1. FA = $\{Q(q_0, q_f), \Sigma(0,1), q_0, q_f, \delta\}$



1.8 REGULAR EXPRESSIONS

Regular expression

- Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
- It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.
- In regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx,.....}
- In regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,.....}

Operations on Regular Language

The various operations on regular language are:

1. **Union:** If L and M are two regular languages then their union L U M is also a union.
   L U M = {s | s is in L or s is in M}

2. **Intersection:** If L and M are two regular languages then their intersection is also an intersection.
   L ∩ M = {st | s is in L and t is in M}

3. **Kleene closure:** If L is a regular language then its kleene closure L1* will also be a regular language.
   L* = Zero or more occurrence of language L.

Example

Write the regular expression for the language:

L = {ab^n w:n ≥ 3, w ∈ (a,b)$^+$}

Solution:

The string of language L starts with "a" followed by atleast three b's. Itcontains atleast one "a" or one "b" that is string are like abbba, abbbbbba, abbbbbbbb, abbbb.....a

So regular expression is:

$r = ab^3b^* (a+b)^+$

Here + is a positive closure i.e. $(a+b)^+ = (a+b)^* - \in$

1.9 OPTIMIZATION OF DFA

To optimize the DFA you have to follow the various steps. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states and T2 contains non-final states.

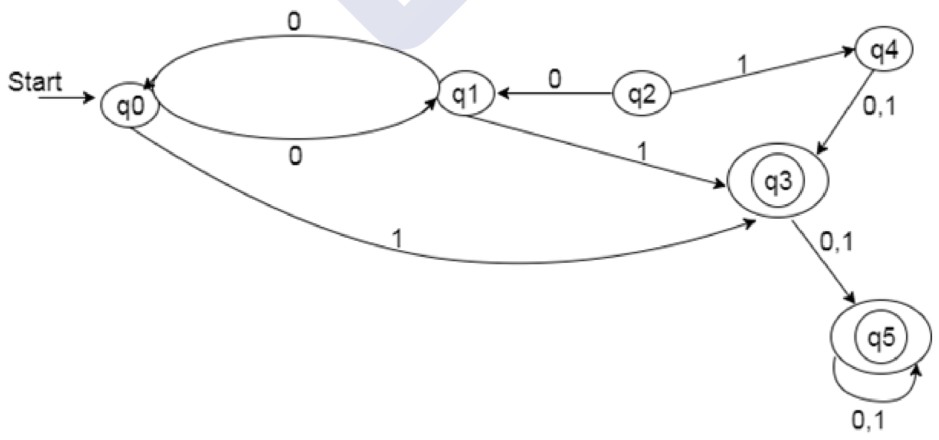
Step 4: Find the similar rows from T1 such that:

Step 5: Repeat step 3 until there is no similar rows are available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3:

Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final sates:

State	0	1
q0	q1	q3
q1	q0	q3

2. Other set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.

Transition diagram of minimized DFA:

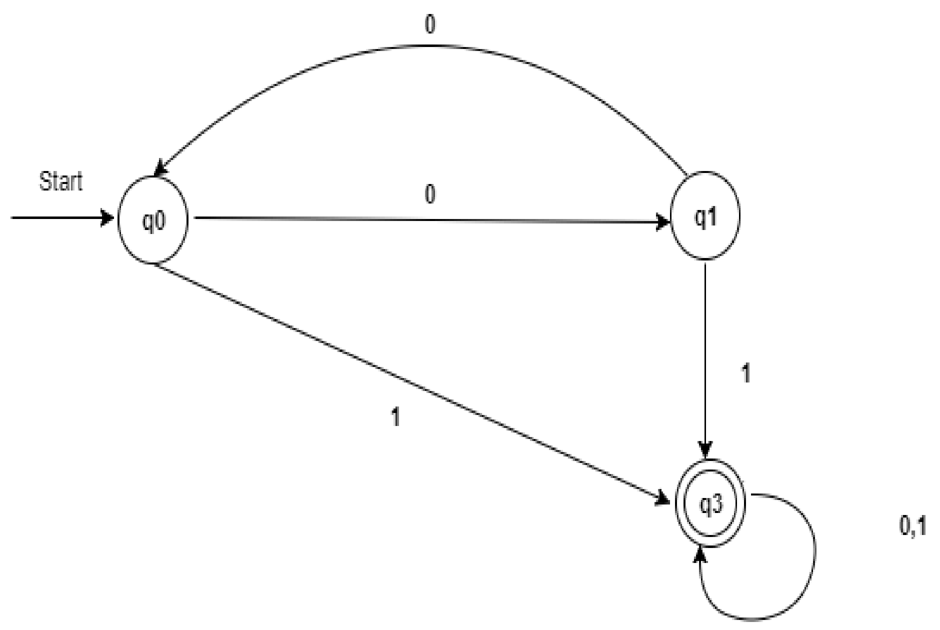


Fig: Minimized DFA