University of Baghdad
College of Education for Pure Science
Ibn-AL-Haithem/ Dep. Of Computer Science

M.Sc. Shaimaa Al-Obaidy
2021-2022
Third Stage

Chapter One

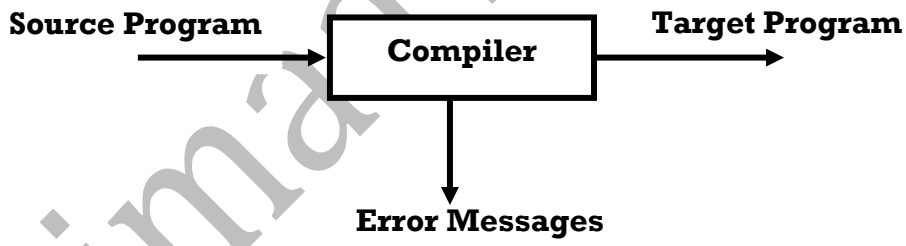
REFRENCES:-

- **Compilers Principles, Techniques and Tools by Alferd V.Aho.**
- **Compiler Construction for Digital Computers by David Gries.**
- **Introduction Theory of Computer Science by E.R. Krishuamurthy.**
- الأسس النظرية و التطبيقية لتصميم مترجمات لغات برمجة الحاسبة د.صباح محمد أمين
  د.جنان عبد الوهاب

A Compiler

Is a program that reads a program written in one language -the Source Language- and translates it into an equivalent program in another language - the Target Language -.

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.



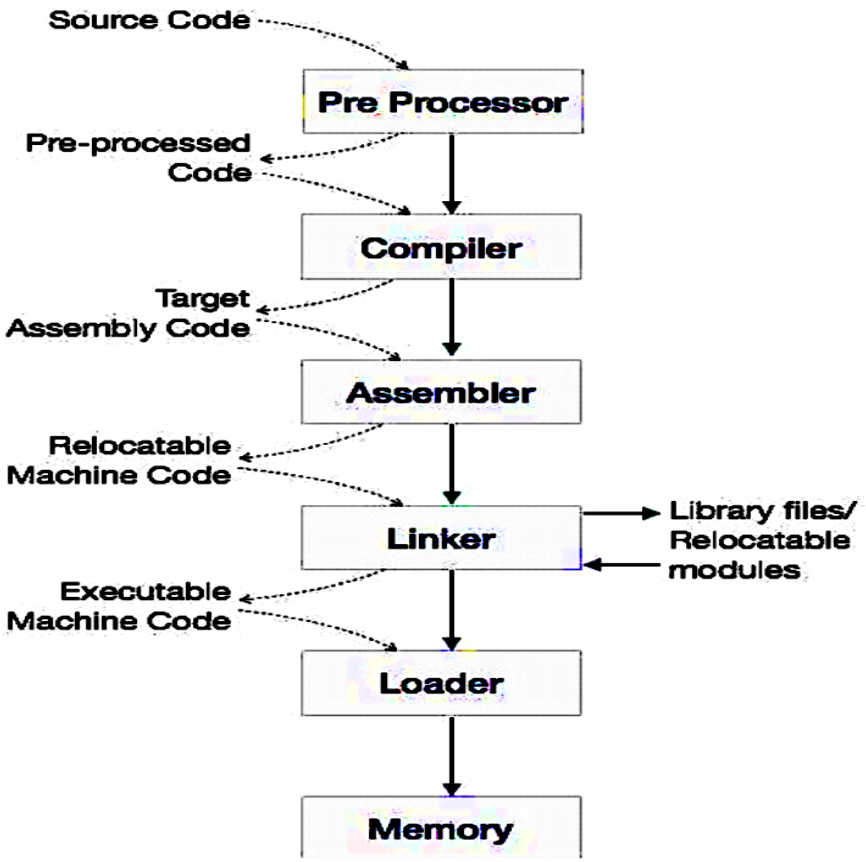
Compiler Design

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in

binary format, which is simply a series of 1s and 0s. It would be a difficult task for computer programmers to write such codes, which is why we have compilers to write such codes.

Language Processing System

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to

assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

1. **User writes a program in C language (High-Level Language).**
2. **The C *compiler* compiles the program and translates it to assembly program (Low-Level Language).**
3. ***An Assembler* then translates the assembly program into machine code (object).**
4. ***A Linker* tool is used to link all the parts of the program together for execution (Executable Machine Code).**
5. ***A Loader* loads all of them into memory and then the program is executed.**

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers.

Interpreter

An *interpreter*, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A *compiler* reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an *interpreter* reads a statement from the input converts it to an intermediate code, executes it, then takes the next statement in

sequence. If an error occurs, an interpreter stops execution and reports it. Whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. The major task of a linker is to determine the memory location where these files will be loaded.

Loader

Loader is a part of operating system and is responsible for loading executable files into memory and executes them. It calculates the size of a program (instructions and data) and creates memory space for it.

Compiler Architecture:-

A compiler can broadly be divided into two phases based on the way they compile.

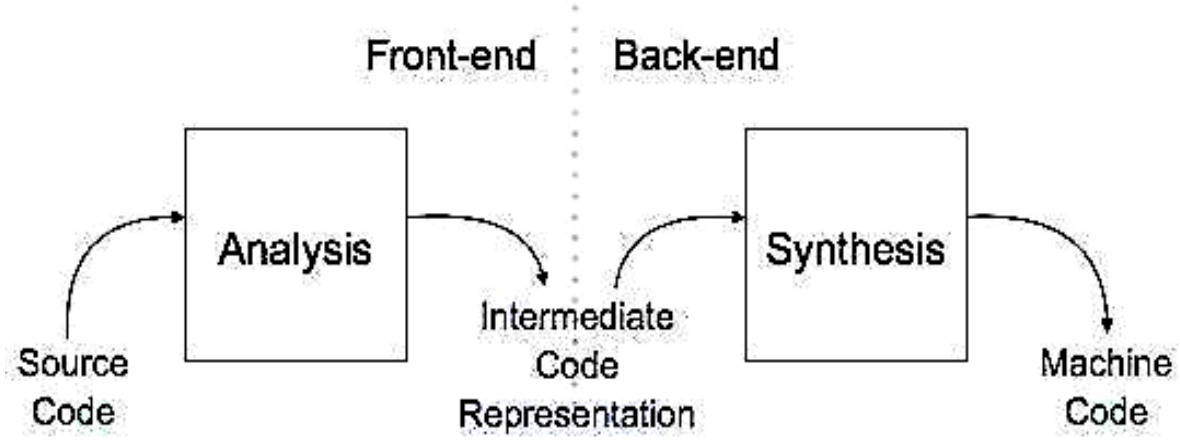
1. *Analysis Phase*

Known as the Front-End of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase

generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

2. *Synthesis Phase*

Known as the Back-End of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.



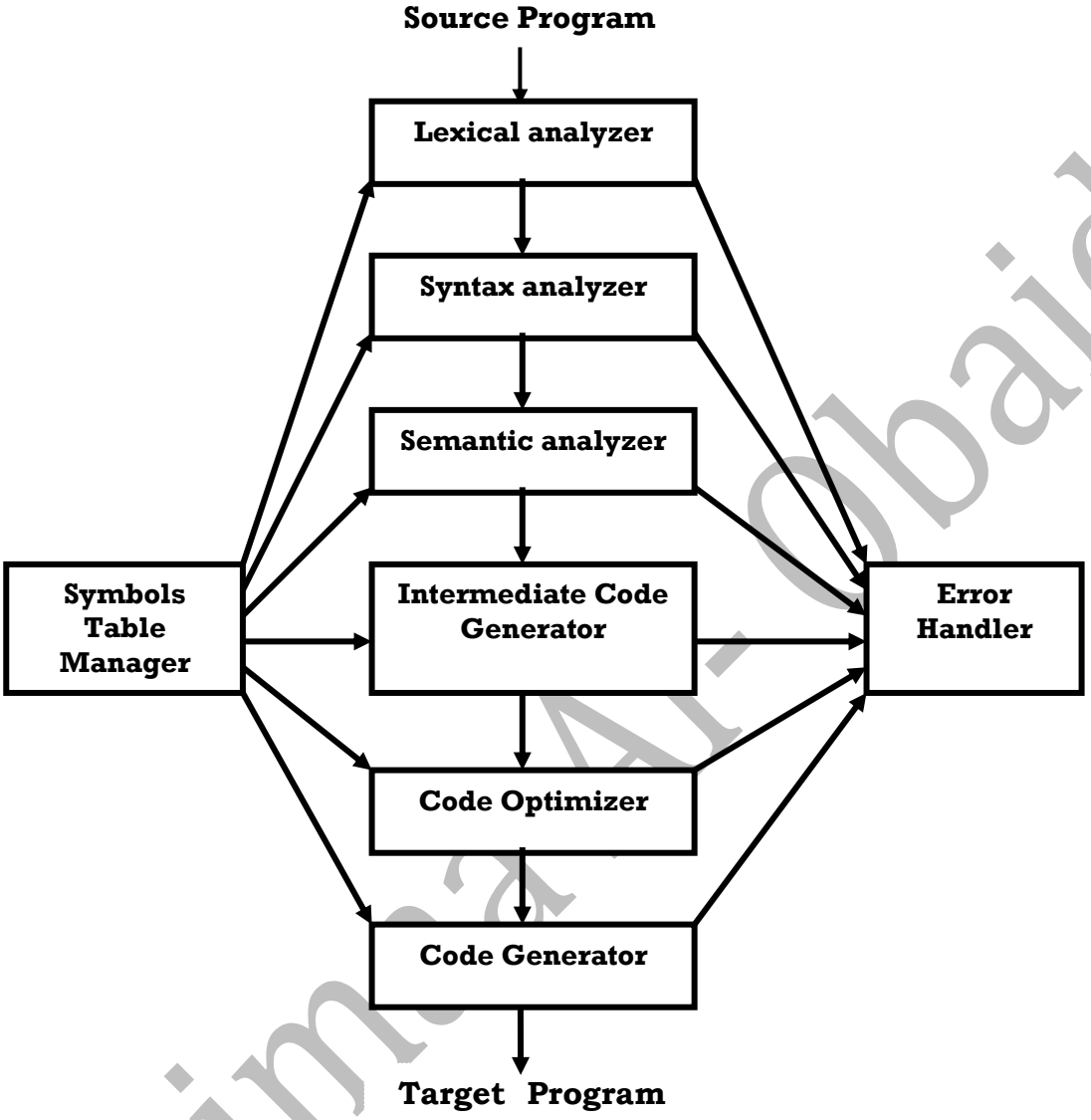
The Phases of a Compiler :-

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

1. **Lexical Analyzer.** مرحلة التحليل اللفظي
2. **Syntax Analyzer.** مرحلة التحليل القواعدي
3. **Semantic Analyzer.** مرحلة التحليل المعنوي
4. **Intermediate Code Generator.** مرحلة توليد الشفرات الوسطية
5. **Code Optimizer.** مرحلة تحسين الشفرات
6. **Code Generator.** مولد الشفرات

In each phase we need variables that can be obtained from a table called *Symbol Table manager*, and in each phase some errors may be

generated so we must have a program used to handle these errors , this program called *Error Handler*.



- ***Lexical Analyzer*** **:- Is the initial part of reading and analyzing the program text (source program); The text is read (character by character) and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.**
- ***Syntax analyzer*** **:- The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input**

and generates a parse tree (or syntax tree) that reflects the structure of the program. This phase is often called parsing.

- ***Semantic Analysis*:- Semantic analysis checks whether the parse tree constructed follows the rules of language. Also is known as Type checking which main function is to analyze the syntax tree to determine if the program violates certain consistency requirements, such as, if a variable is used but not declared, assignment of values is between compatible data types, and adding string to an integer.**
- ***Intermediate Code Generator* :- After syntax and semantic analysis, It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. This phase bridges the analysis and synthesis phases of translation.**
- ***Code Optimization phase* :- The code optimization phase attempts to improve the intermediate code which results that the output runs faster and takes less space. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).**
- ***Code Generator* :- The final phase of complier is the generation of target code, which represents the output of the code generator in the machine language.**

- ***Symbol Table*** **:- It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.**
- ***Error Handler*** **:- Each phase can produce errors. However, after detecting an error, a phase must deal with that error, so that the compilation can proceed. So dealing with that error is done by a program known as Error Handler which is software used to handle any error that may be produced from any phase and it is needed in all phases of the compliers.**

Note **:- Each phase of the complier has two inputs and two outputs; for example:- for the first phase (*Lexical Analyzer*) the first input to it is the source program while the second input is some variables that may be needed in that phase; while the first output is the errors that may be generated in it and will be manipulated by the Error Handler program, and the second output from it will represent the input for the next compiler phase (Syntax).**

Lexical Analysis:- A Review

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

***For example*, in C language, the variable declaration line**

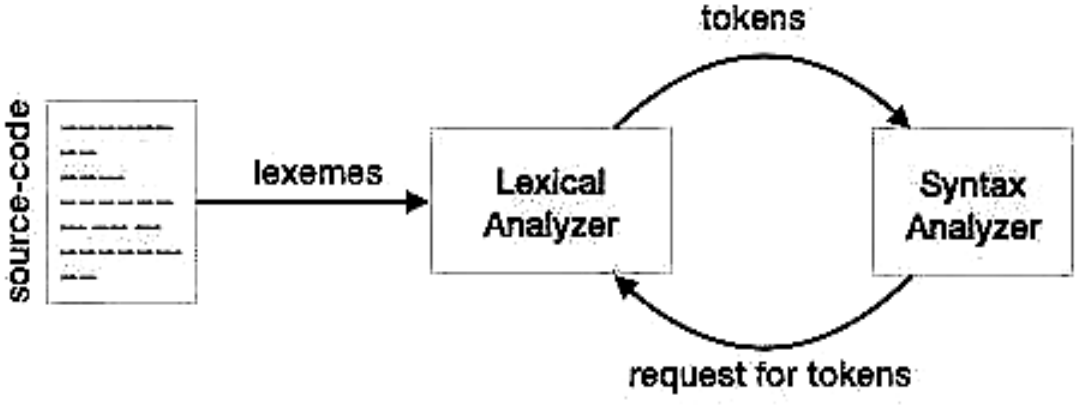
```
int value = 100;
```

Contains the tokens:-

int	keyword
value	identifier
=	operator
100	constant
;	symbol

The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorials point is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by Ɛ (epsilon).

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

The various operations on languages are:

- **Union of two languages L and M is written as**

  L U M = {s | s is in L or s is in M}

- **Concatenation of two languages L and M is written as**

  LM = {st | s is in L and t is in M}

- **The Kleene Closure of a language L is written as**

  L* = Zero or more occurrence of language L.

Grammars

A grammar is a set of formal rules for constructing correct sentences in any language; such sentences are called *Grammatical Sentences.*

Concatenation

We define the *Concatenation* of two symbols U and V by:-

UV= { X | X= uv, u is in U and v is in V }

Note that:- UV ≠ VU

U (VW) = (UV) W

***Example* ①:-**

Let ∑ = {0,1} and U= {000,111} and V= {101,010}

⇨ UV= {000101, 000010, 111101, 111010}

⇨ VU= {101000, 101111, 010000, 010111}

∴UV ≠ VU

***Example* ②:-**

Let ∑ = {a,b,c,d} ; U= {abd , bcd} ; V= {bcd , cab} and W= {da , bd}

To prove the following:- U (VW) = (UV) W

First, take the left side;

U (VW) ={abd , bcd} {bcdda, bcdbd, cabda, cabbd}

= { abdbcdda, abdbcdbd, abdcabda, abdcabbd, bcdbcdda, bcdbcdbd, bcdcabda, bcdcabbd }

Second, take the right side;

(UV) W = { abdbcd, abdcab, bcdbcd, bcdcab} {da , bd}

= { abdbcdda, abdcabda, bcdbcdda, bcdcabda, abdbcdbd, abdcabbd, bcdbcdbd, bcdcabbd }

∴ U (VW) = (UV) W

Closure or Star Operation :-

This operation defines on a set S, a derived set S^*, having as members the empty word and all words formed by concatenating a finite number of words in S, as shown below:-

$$S^* = S^0 \cup S^1 \cup S^2 \cup \ldots\ldots$$

Where :-

$S^0 = \varepsilon$ **and** $S^i = S^{i-1}S$ **for** $i > 0$

Example **:-**

Let S = {01, 11}, then

S^*={ε , 01,11 , 0101,0111,1101,1111 , 010101 , 010111, … }

S^0 S^1 S^2 S^3

Formalization:-

A phrase structure grammar is of the form G= (N, T, S, P); where:-

N = A finite set of non-terminal symbols denoted by A, B, C,…

T = A finite set of terminal symbols denoted by a, b, c,…

With N ∪ T = V and N ∩ T= φ (null set).

P = A finite set of ordered pairs (α, β) called the Production Rules, α and β being the string over V^* and α involving at least one symbol from N.

S = is a special symbol called the Starting Symbol.

Example **:-**

Let G= (N, T, S, P); N= {S, B, C}, T= {a, b}

P= {(S → aba), (SB →b), (b→bB), (b→λ)}

This grammar is not a structure grammar because of the production rule b →bB because the left side of this rule containing only a terminal symbol (b) and in any production rule the left side must involve at least one non-terminal symbol.

Example **:-**

Let G= (N, T, S, P) where N= {S, A}, T= {a, b}

P= {(S→aAa), (A→bAb), (A→a)}

S → aAa → abAba → abbAbba → abbabba

Note **:-**

1. **The production rules can be written in another form, for the above example, the production rule is written as follows:-**
   P= {(S, aAa), (A, bAb), (A, a)}
2. **Some times it may be that two different grammars G and Ğ generated the same language L (G)=L(Ğ) ∴ the grammars are said to be *equivalent*.**

Example **:-**

G= (N,T,S,P)

N= {number, integer, fraction, digit}

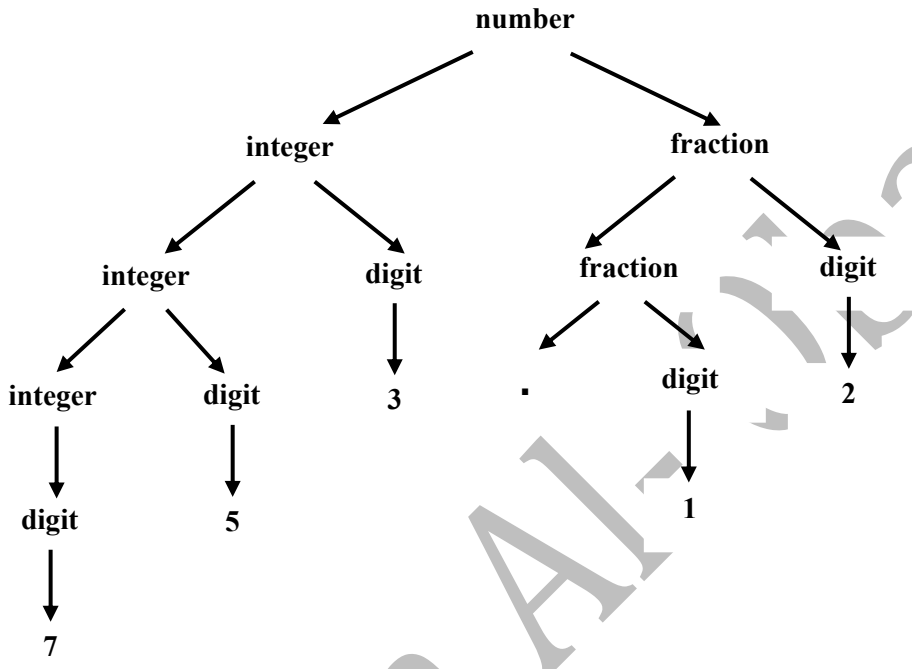
T= {., 0, 1, 2, 3, …, 9}

S=number

P={(number→integer fraction), (integer→digit), (integer→ integer digit), (fraction→.digit), (fraction→fraction digit),(digit→0), (digit→1),

(digit→2), (digit→3), (digit→4), (digit→5), (digit→6), (digit→7), (digit→8), (digit→9)}

Now we want to prove if the following number is accepted or not 753.12?



Kinds of Grammar Description **:-**

1. **Transition Diagram.**
2. **BNF (Backus_ Naur form).**
3. **EBNF.**
4. **Cobol_Meta Language.**
5. **Syntax Equations.**
6. **Regular Expression (R.E.).**

By using BNF the grammar can be represented as follows:-

(For the previous example)

G= (N, T, S, P)

N= {<number>, <integer> , <fraction> , <digit>}

T= {., 0, 1, 2, 3, …, 9}

S= <number>

Production rules P will be represented as follows:

<number> ::= <integer> <fraction>

<integer> ::= <digit>|<integer> <digit>

<fraction> ::= .<digit>|<fraction> <digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

Regular Expression (R.E.)

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language.

The various operations on languages are:

- **Union of two languages L and M is written as**

  L U M = {s | s is in L or s is in M}

- **Concatenation of two languages L and M is written as**

  LM = {st | s is in L and t is in M}

- **The Kleene Closure of a language L is written as**

  L* = Zero or more occurrence of language L.

For example, R^* is R.E. denoting $\{\varepsilon\} \cup L_R \cup L_R^2 \cup \ldots \cup L_R^n$

The main components of RE are

1. **ε or λ is R.E. denoting by L^0={ε}=L**
2. **Any terminal symbol like a is R.E. denoting L={a}**
3. **[a-z] is all lower-case alphabets of English language.**
4. **[A-Z] is all upper-case alphabets of English language.**
5. **[0-9] is all natural digits used in mathematics.**

Transformation of R.E. to Transition Diagram (Formal Method)

1. **For each non terminal NT draw a circle.**
2. **Connect with arrows between any two circles with respect to the following rules:-**
   - **If NT→NT connect the two circles with arrow labeled λ or ε.**
   - **If NT→T NT connect the two circles with arrow labeled T.**
   - **If NT→T creates a new circle with a new NT (final) then connect the left-hand side NT of the rule and the new NT with arrow labeled T.**
   - **If NT→T′s NT create circles (as the length of T′s-1).**

Example **:-**

Let G= {{S, R, U},{a, b}, S, P}

P=

S → a

R → abaU

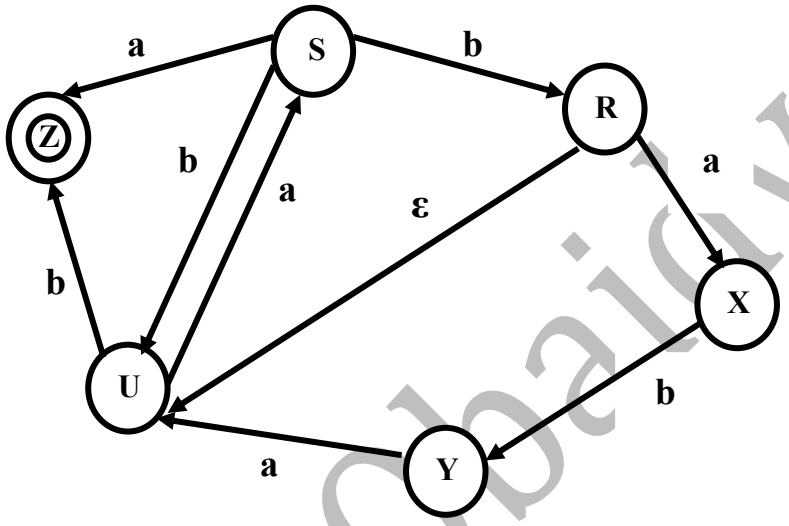
U → b

S → bU

R → U

U → aS

S → bR



Transformation of BNF to Transition Diagram (Informal Method)

1. **Draw a separate transition diagram for each production rule.**
2. **Substitute each non-terminal symbol by its corresponding transition diagrams.**

Example **:-**

G= (N, T, S, P)

N= {<number>, <integer>, <fraction>, <digit>}

T= {., 0, 1, 2, 3, …, 9}

S= <number>

P=

<number> ::= <integer> <fraction>

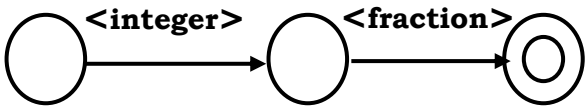
<integer> ::= <digit>|<integer> <digit>

<fraction> ::= .<digit>|<fraction> <digit>

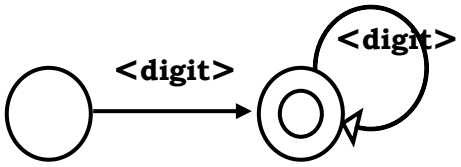
<digit> ::= 0|1|2|3|4|5|6|7|8|9

Now we take each production rule and draw to it a separate transition diagram:-

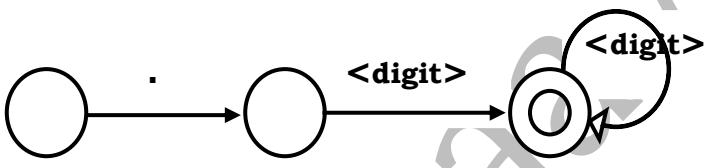
<number> ::= <integer> <fraction>



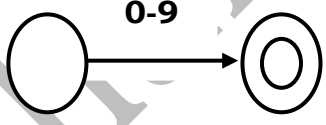
<integer> ::= <digit>|<integer> <digit>



<fraction> ::= .<digit>|<fraction> <digit>



<digit> ::= 0|1|2|3|4|5|6|7|8|9



Now we must substitute each non-terminal symbol by its corresponding transition diagram.