MODULE 1

Introduction to compilers – Analysis of the source program, Phases of a compiler, Grouping of phases, compiler writing tools – bootstrapping

Lexical Analysis:

The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens.

1.1 INTRODUCTION TO COMPILERS

➕ A compiler is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language).

➕ An important role of the compiler is to report any errors in the source program that it detects during the translation process.
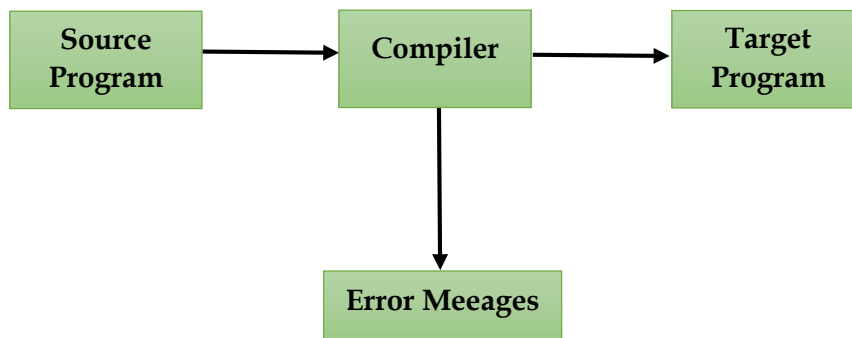
```
┌──────────┐      ┌──────────┐      ┌──────────┐
│  Source  │ ───▶ │ Compiler │ ───▶ │  Target  │
│ Program  │      │          │      │ Program  │
└──────────┘      └────┬─────┘      └──────────┘
                       │
                       ▼
                 ┌──────────────┐
                 │ Error Meeages│
                 └──────────────┘
```

Fig: Compiler

➕ Compilers are sometimes classified as single pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

1.1.1 ANALYSIS OF THE SOURCE PROGRAM

➕ In compiling, analysis consists of three phases:

  ➢ **Lexical Analysis**

  ➢ **Syntax Analysis**

  ➢ **Semantic Analysis**

Lexical Analysis (Scanning)

➕ In a compiler linear analysis is called lexical analysis or scanning. The lexical analysis phase reads the characters in the source program and grouped into tokens that are sequence of characters having a collective meaning.

<u>EXAMPLE</u>

  position : = initial + rate * 60

This can be grouped into the following tokens;

> **1. The identifier position.**
>
> **2. The assignment symbol : =**
>
> **3. The identifier initial**
>
> **4. The plus sign**
>
> **5. The identifier rate**
>
> **6. The multiplication sign**
>
> **7. The number 60**

- Blanks separating characters of these tokens are normally eliminated during lexical analysis.

Syntax Analysis (Parsing)

- Hierarchical Analysis is called parsing or syntax analysis.

- It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output. They are represented using a syntax tree.

- A syntax tree is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands. This analysis shows an error when the syntax is incorrect.
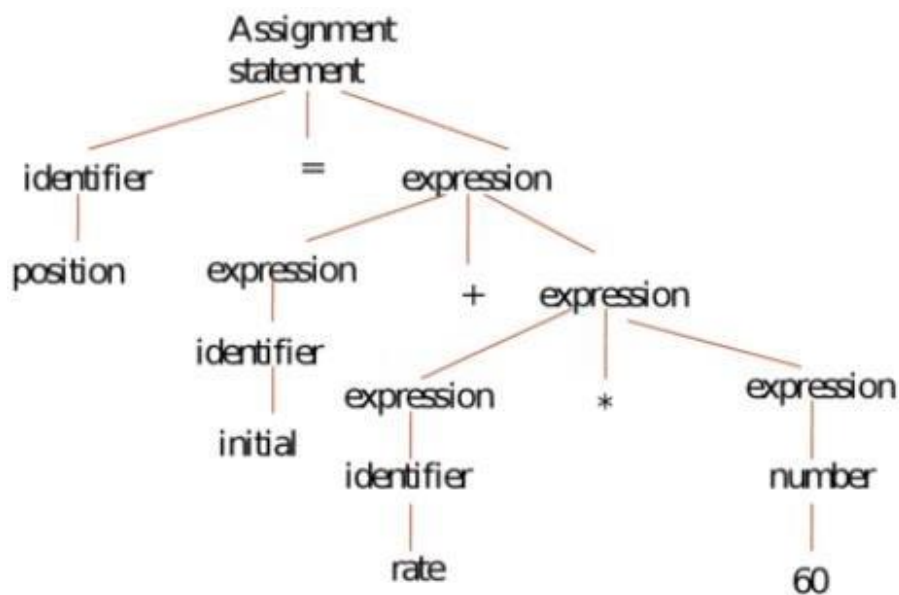
Semantic Analysis

- This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase.

- An important component of semantic analysis is type checking.

- Here the compiler checks that each operator has operands that are permitted by the source language specification.

1.1.2 PHASES OF A COMPILER

The phases include:

- ➢ **Lexical Analysis**

- ➢ **Syntax Analysis**

- ➢ **Semantic Analysis**

- ➢ **Intermediate Code Generation**

- ➢ **Code Optimization**

- ➢ **Target Code Generation**
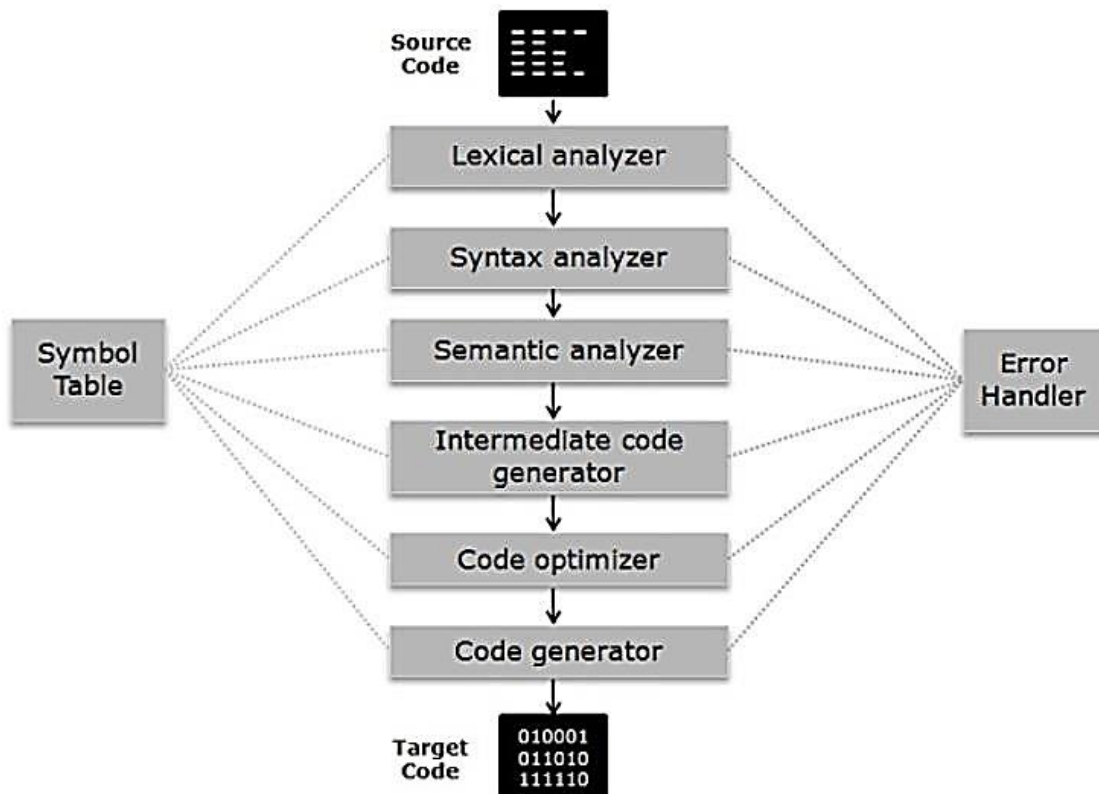
Lexical Analysis

+ The first phase of a compiler is called lexical analysis or scanning.

+ The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

+ For each lexeme, the lexical analyzer produces as output a token of the form

< token- name, attribute-value >

that it passes on to the subsequent phase, syntax analysis.

+ In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.

+ Information from the symbol-table entry 'is needed for semantic analysis and code generation.

+ For example, suppose a source program contains the assignment statement

position = initial + rate * 60

+ The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. position is a lexeme that would be mapped into a token **<id, 1>**, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol- table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token **< = >.** Since this token needs no attribute-value, we have omitted the second component.

3. initial is a lexeme that is mapped into the token **< id, 2>** , where 2 points to the symbol-table entry for initial .

4. + is a lexeme that is mapped into the token **<+>.**

5. rate is a lexeme that is mapped into the token **< id, 3 >,** where 3 points to the symbol-table entry for rate.

6. * is a lexeme that is mapped into the token **<* >** .

7. 60 is a lexeme that is mapped into the token **<60>**

+ Blanks separating the lexemes would be discarded by the lexical analyzer. The representation of the assignment statement position = initial + rate * 60 after lexical analysis as the sequence of tokens as:

< id, l > < = > <id, 2> <+> <id, 3> < * > <60>

Token : Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

> ➢ Identifiers
>
> ➢ keywords
>
> ➢ operators
>
> ➢ special symbols
>
> ➢ constants

Pattern : A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme : A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

| TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, 1, 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"

Syntax Analysis

🞣 The second phase of the compiler is syntax analysis or parsing.

🞣 The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

🞣 A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

🞣 The syntax tree for above token stream is:
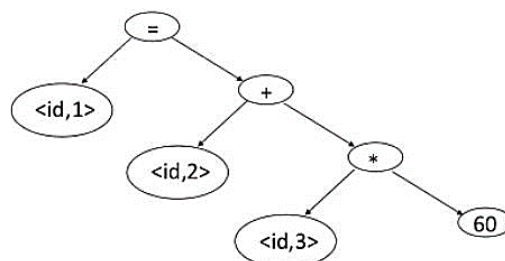
- The tree has an interior node labeled with (id, 3) as its left child and the integer 60 as its right child.

- The node (id, 3) represents the identifier rate.

- The node labeled * makes it explicit that we must first multiply the value of rate by 60.

- The node labeled + indicates that we must add the result of this multiplication to the value of initial.

- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position.

Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.

- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

- Some sort of type conversion is also done by the semantic analyzer.

- For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.

- In our example, suppose that position, initial, and rate have been declared to be floating- point numbers, and that the lexeme 60 by itself forms an integer.

- The semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.

- In this case, the integer may be converted into a floating-point number.

- In the following figure, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.
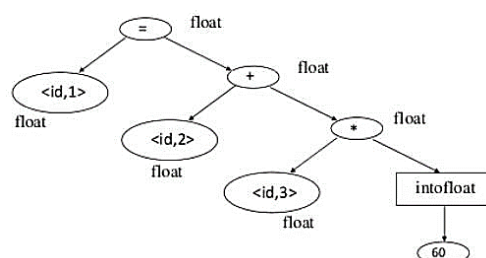
Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.

- This intermediate representation should have two important properties:

  - ➢ It should be simple and easy to produce

  - ➢ It should be easy to translate into the target machine.

- In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

$$t1 = inttofloat(60)$$
$$t2 = id3 * t1$$
$$t3 = id2 + t2$$
$$id1 = t3$$

Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

- The objectives for performing optimization are: faster execution, shorter code, or target code that consumes less power.

- In our example, the optimized code is:

$$t1 = id3 * 60.0$$
$$id1 = id2 + t1$$

Code Generator

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.

- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

- If the target language is assembly language, this phase generates the assembly code as its output.

- In our example, the code generated is:

> **LDF   R2, id3**
> **MULF R2, #60.0**
> **LDF   R1, id2**
> **ADDF R1, R2**
> **STF   id1, R1**

- The first operand of each instruction specifies a destination.

- The F in each instruction tells us that it deals with floating-point numbers.

- The above code loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.

- The # signifies that 60.0 is to be treated as an immediate constant.

- The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.

- Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement **position = initial + rate * 60.**

Symbol Table

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Error Detection And Reporting

- Each phase can encounter errors.

- However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

➕ A compiler that stops when it finds the first error is not a helpful one.
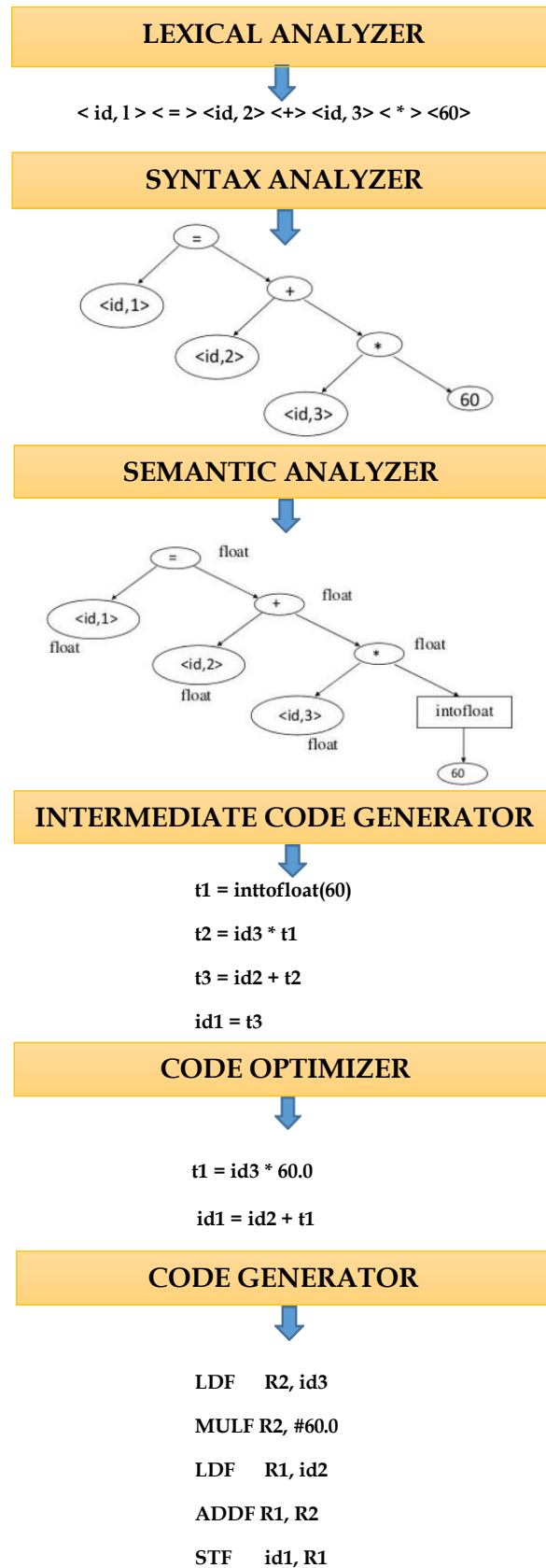
LEXICAL ANALYZER

< id, 1 > < = > <id, 2> <+> <id, 3> < * > <60>

SYNTAX ANALYZER

=
<id,1>
+
<id,2>
*
<id,3>
60

SEMANTIC ANALYZER

=   float
<id,1>
float
+   float
<id,2>
float
*   float
<id,3>
float
intofloat
60

INTERMEDIATE CODE GENERATOR

t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

CODE OPTIMIZER

t1 = id3 * 60.0

id1 = id2 + t1

CODE GENERATOR

LDF   R2, id3

MULF R2, #60.0

LDF   R1, id2

ADDF R1, R2

STF   id1, R1

Figure : *Translation of an assignment statement*

1.1.3 GROUPING OF PHASES

- The process of compilation is split up into following phases:

    ➢ **Analysis Phase**
    ➢ **Synthesis phase**

Analysis Phase

Analysis Phase performs 4 actions namely:

 a. **Lexical analysis**

 b. **Syntax Analysis**

 c. **Semantic analysis**

 d. **Intermediate Code Generation**

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.

- It then uses this structure to create an intermediate representation of the source program.

- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Synthesis Phase

Synthesis Phase performs 2 actions namely:

 a. **Code Optimization**
 b. **Code Generation**

- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

- The analysis part is often called the front end of the compiler; the synthesis part is the back end.

1.1.4 COMPILER WRITING TOOLS

➕ Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler construction tools include the following.

> ➢ **Parser Generators**
> ➢ **Scanner Generators**
> ➢ **Syntax-directed translation engine**
> ➢ **Automatic code generators**
> ➢ **Data-flow analysis Engines**
> ➢ **Compiler Construction toolkits**

Parser Generators.

Input : Grammatical description of a programming language

Output : Syntax analyzers.

➕ These produce syntax analyzers, normally from input that is based on a context-free grammar.

➕ In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

➕ This phase is one of the easiest to implement.

Scanner Generators

Input : Regular expression description of the tokens of a language

Output : Lexical analyzers.

➕ These automatically generate lexical analyzers, normally from a specificalion based on regular expressions.

➕ The basic organization of the resulting lexical analyzer is in effect a finite automaton.

Syntax-directed Translation Engines

Input : Parse tree.

Output : Intermediate code.

➕ These produce collections of routines that walk the parse tree, generating intermediate code.

➕ The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.

Automatic Code Generators

Input : Intermediate language.

Output : Machine language.

- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

- The rules must include sufficient detail that we can handle the different possible access methods for data.

Data-flow Analysis Engines

- Data-flow analysis engine gathers the Information that is, the values transmitted from one part of a program to each of the other parts.

- Data-flow analysis is a key part of code optimization.

1.1.4.1 BOOTSTRAPPING

- Bootstrapping is widely used to design a compiler. **Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.

- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

- A compiler is characterized by three languages:

  - ➤ **Source Language**
  - ➤ **Target Language**
  - ➤ **Implementation Language**

- **Notation:** $^{S}C_{I}^{T}$ represents a compiler for Source **S** , Target **T** , implemented in **I** . The **T**-diagram shown above is also used to depict the same compiler.

  Consider the following T diagram
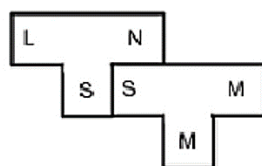
- The first T describes a compiler from L to N written in S
- The second T describes a compiler from S to M written in M (or running on M). This will be your **compiler compiler**.

- Applying the second T to the first T compiles the first T so that it runs on machine M. The result is thus a compiler from L to N running on machine M.

- **To create a new language, L, for machine A:**

  1. Create $^SC_A^A$, a compiler for a subset, S, of the desired language L, using language A, which runs on machine A. (Language A may be assembly language.)
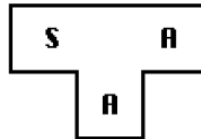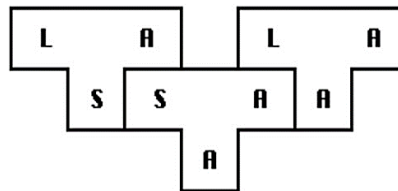

  2. Create $^LC_S^A$, a compiler for language L written in a subset of L.
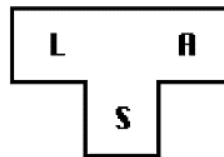
  3. Compile $^LC_S^A$ using $^SC_A^A$ to obtain $^LC_A^A$, a compiler for language L, which runs on machine A and produces code for machine A.

$$^LC_S^A \rightarrow {}^SC_A^A \rightarrow {}^LC_A^A$$

- The process illustrated by the T-diagrams is called bootstrapping and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

Cross Compiler:

- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler

- **Cross Compilers** are compilers that execute on one computer and generate **object code** that can execute on different platform. **for example** a cross compiler that is running on windows pc can produce object code that run on **MAC Os** or **Android Os**.
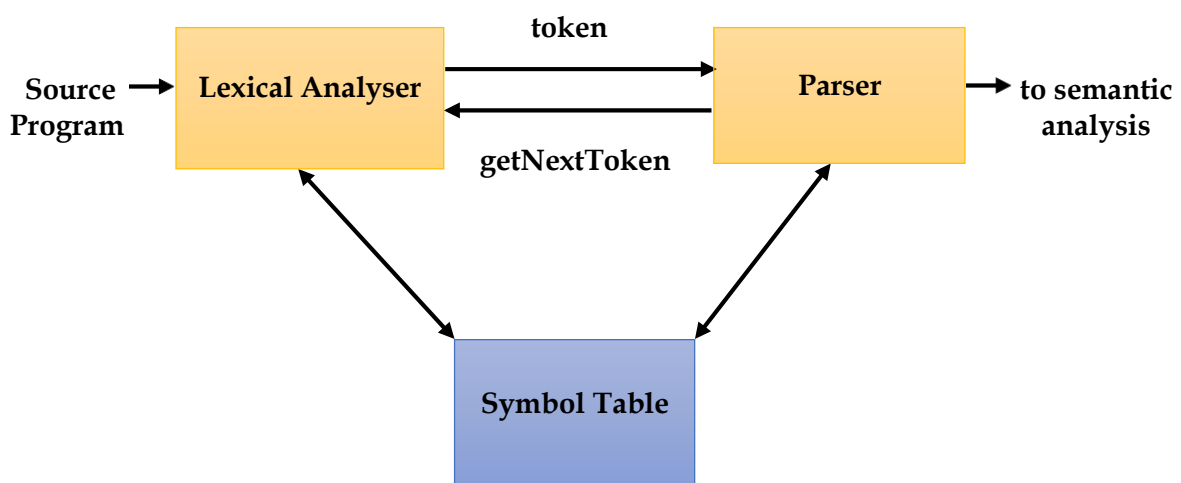
1.2 LEXICAL ANALYSIS

1.2.1 ROLE OF LEXICAL ANALYSIS

- 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 a sequence of tokens for each lexeme in the source program.

- The stream of tokens is sent to the parser for syntax analysis.

Source Program ⟶ **LEXICAL ANALYSER** ⟶ **Sequence of Tokens**

- Lexical Analyzer also interacts with the symbol table.

- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

- These interactions are given in following figure.

- Commonly, the interaction is implemented by having the parser call the lexical analyzer.

- The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Source Program → **Lexical Analyser** — token → **Parser** → **to semantic analysis**

getNextToken

Symbol Table

Interactions between lexical analyser and parser

Other tasks of Lexical Analyzer

1. Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

2. Correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

3. If the source program uses a macro-pre-processor, the expansion of macros may also be performed by the lexical analyzer.

Issues In Lexical Analysis

Following are the reasons why lexical analysis is separated from syntax analysis

Simplicity Of Design

The separation of lexical analysis and syntactic analysis often allows us to simplify at least one of these tasks. The syntax analyzer can be smaller and cleaner by removing the lowlevel details of lexical analysis

Efficiency

Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

Portability

Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

Attributes For Tokens

- Sometimes a token need to be associate with several pieces of information.

- The most important example is the token id, where we need to associate with the token a great deal of information.

- Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.

- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Lexical Errors

- A character sequence that can't be scanned into any valid token is a lexical error.

- Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

- The simplest recovery strategy is **"panic mode"** recovery.

- We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

- Other possible error-recovery actions are:

    1. **Delete one character from the remaining input.**

    2. **Insert a missing character into the remaining input.**

    3. **Replace a character by another character.**

    4. **Transpose two adjacent characters.**

- Transformations like these may be tried in an attempt to repair the input.

- The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation.

- A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

    Connected with Lexical analysis, there are three important terms with similar meanings. They are **Lexeme**, **Token** and **Pattern**.

    Lexeme: These are the smallest logical units (words) of the program, such as A, B, 1.2, true, if, else, <, = …….

    Tokens: They are classes of similar lexemes, such as identifier, constants, operators etc. Hence the tokens are the category to which a lexeme belongs to.

    Pattern: It gives an informal or formal description of a token. For example, an identifier is a string in which the first character is an alphabet and the successive characters are either digits or alphabet. A pattern serves two purposes: It gives a precise description or specification of tokens. It can also be used to automatically generate a lexical analyzer.

1.2.2 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 pointer (**fp**) to keep track of the pointer of the input scanned.

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

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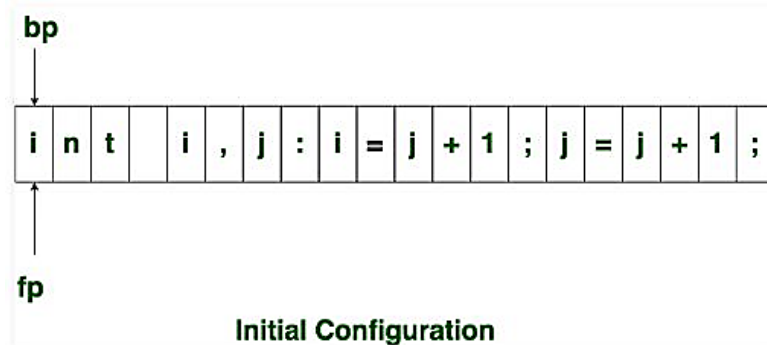
- The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified.

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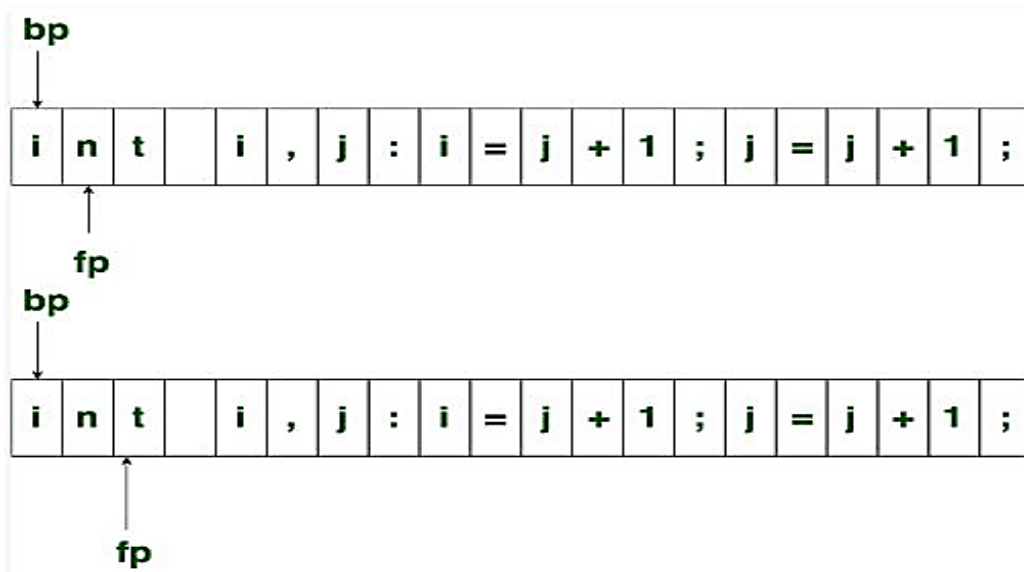
- The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then second by lexical analyzer. There are two methods used in this context: One Buffer Scheme, and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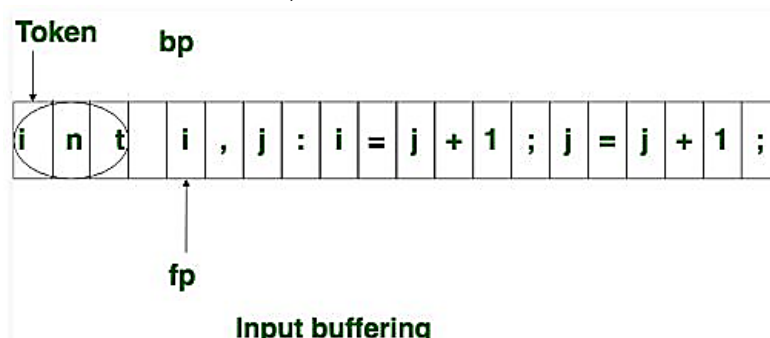

Input buffering

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
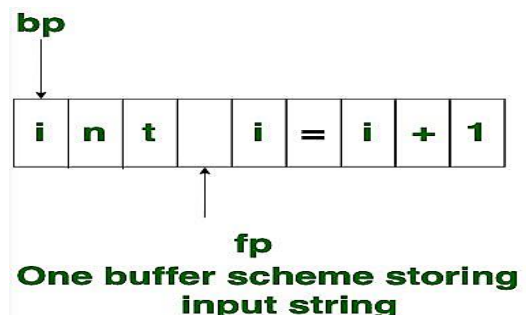
Two Buffer Scheme:

➕ 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.

Two buffer scheme storing input string
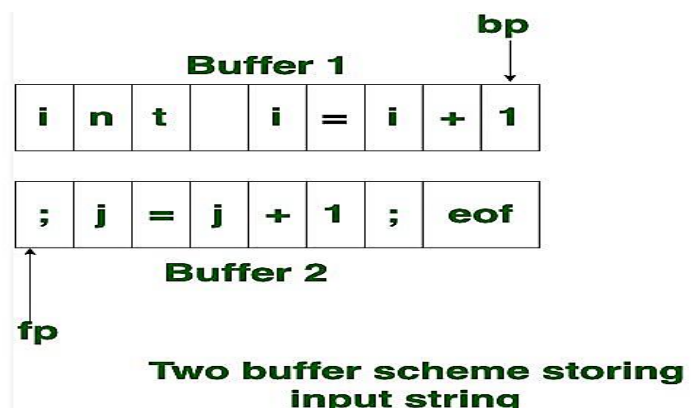
1.2.3 SPECIFICATION OF TOKENS

To specify a token we use **Regular expressions. Regular Expressions** generate Regular Languages.

Alphabet: An alphabet is a finite nonempty set of symbols. Symbols can be letters or other characters includes number or any special character.

Eg: **Σ = {0,1}** is a binary alphabet

Σ = {a, b, c, ..., z} is an alphabet contains lower case letters.

String: Strings are finite set of symbols generated from alphabet

Eg: Let Σ = {a,b}.

Then the strings generated from the given alphabet can be {a,b,aa,ab,ba,bb,............}

The length of the string S is the total number of characters present in the string and can be denoted by **|S|.**

Suppose S=1100 Then |S|=4

The empty sequence of letters is denoted by **ε** is called **empty string.** The length of empty string is Zero. **|ε|= 0**

Language: Set of strings which are generated from alphabet called language. It is a collection of string.

Let Σ={a,b}

    L1 = set of all strings of length two

      = {aa, ab, ba, bb}. So L1 is a finite language.

    L2= set of all strings of length 3

      = {aaa, aab, aba, abb, baa, bab, bba, bbb}. So L2 is a finite language.

    L3= set of all strings where each string starts with a

      = {a, aa, ab, aaa, aab, aba, abb, ……..}. So L3 is infinite language.

There for language may be finite or infinite.

REGULAR EXPRESSION

- Regular expression represents pattern of string of characters.
- Regular expression also has a special characters called meta-characters. Eg: *** , |**

Rules for forming regular expression

a) Let **'a'** be a character in S. The regular expression for this is **'a'**. It matches with the character 'a' by writing **L(a) = {a}**

b) Let an empty string **ε** be a character in **Σ.** The regular expression for this is **ε**. It matches with the character **ε** by writing **L(ε) = { ε }**

c) Let the alphabet set of the language be null. It is represented by Σ = Φ. Now L(Φ)={ }

Operations on Regular Expression

a) **Choice among alternates:** Indicated by the meta character **|** (vertical bar). Let r and s be two regular expressions. Then **r|s** is a regular expression. In terms of langue, **r|s** represents the union of the languages represented by r and s.

Ie, **L(r | s) = L(r) U L(s)**

Eg: Consider L(r) = {a}, L(s) = {b} and L(t) = {ε}. What do the following regular expression represents?

1) **r | s**
   It represents the language containing the symbol 'a' or 'b'.
   Ie, L(**r | s) = L(r) U L(s)**
            = {a} U {b} = {a,b}

2) **r | t**
   It represents the language containing the symbol 'a' or 'ε'.
   Ie, L(**r | t) = L(r) U L(t)**
            = {a} U {t} = {a,ε}

b) **Concatenation:** Concatenation of two regular expression r and s is written as rs. It is represented by **L(rs) = L(r)L(s)**

Eg: Consider L(r) = {a}, L(s) = {b} and L(t) = {ε}, L(v) = {c},   What do the following regular expression represents?

1) **rs**
   It represents the language containing the symbol 'a' followed by 'b'. It is represented as
   L(rs)= L(r)L(s)
       ={a}{b} = {ab}

2) **rt**
   It represents the language containing the symbol 'a' followed by 't'. It is represented as
   L(rt)= L(r)L(t)
       ={a}{ε} = {a}

3) **(r | s)v**
   L((r | s)v)= L(r | s)L(v)
       ={a,b}{c} = {ac, bc}

c) **Repetition:** Repetitive operation of a regular expression is called **Kleene Closure.** It is represented by **r*.** Let s = {a}. Then L(a*) = {ε, a, aa, aaa, aaaa, …….}

Eg: Consider L(r) = {a} , L(s)={b}. What do the following regular expressions represents?

1) **r***
   It represents the language containing the zero or more occurrences of the symbols from L(r)' ie, **L(r*) = {ε, a,aa, aaa,…..}**

2) **(rs)***
   It represents the language containing the zero or more occurrences of the symbols from L(rs)' ie, **L((rs)*) = {ε, ab, abab,ababab,……}**

3) **(r | ss)***
   L((r | ss)) = {ε,a,bb,abb,aabb,aaabbbb……..}

Regular expressions over an alphabet Σ are constructed using following rules

*Prepared by **EBIN P.M** (AP, CSE)*
IES College of Engineering

1. ε is a regular expression.
2. If a is a symbol in Σ, then a is a regular expression.
3. if r and s are regular expressions, then r | s is a regular expression.
4. if r and s are regular expressions, then rs is a regular expression.
5. if r is a regular expression, then r* is a regular expression.
6. if r is a regular expression, then (r) is a regular expression.

Notations used in Regular expressions:

S.No.	Operation	Notation
1	Zero or more instances	r* - Kleene Closure
2	One or more instances	r+ - Postive Closure
3	Zero or one instance	r ?
4	Any one or choice	r | s
5	Concatenation	r s (or) r . s
6 →→	Character classes a|b|c|...|z	[a-z]

1.2.4 REVIEW OF FINITE AUTOMATA

- Refer from TOC

- Finite Automata

  1. Deterministic Finite automata

  2. Non Deterministic Finite automata

- NFA to DFA Conversion

22

1.2.5 RECOGNITION OF TOKENS

➕ Tokens obtained during lexical analysis are recognized using a Finite Automaton.

➕ Finite Automata or Finite state machine is a mathematical way of describing the Regular expression.

➕ It produces a transition diagram for regular expression.

EXAMPLE

Assume the following grammar fragment to generate a specific language

stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | ε
expr → *term* **relop** *term* | *term*
term → **id** | **number**

where the terminals if, then, else, relop, id and num generates sets of strings given by following regular definitions.

if → **if**
then → **then**
else → **else**
rebop → **< | <= | < > | > | > =**
id → **letter (letter | digit)***
num → **digits optional-fraction optional-exponent**

➕ where letter and digits are defined previously

➕ For this language, the lexical analyzer will recognize the keywords i f , then,and else, as well as lexemes that match the patterns for *relop, id,* and *number.*

➕ To simplify matters, we make the common assumption that keywords are also *reserved words:* that is they cannot be used as identifiers.

➕ The num represents the unsigned integer and real numbers of Pascal.

➕ In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.

➕ Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws,** below.

Delim → **blank | tab | newline**
ws → **delim**

➕ If a match for **ws** is found, the lexical analyzer does not return a token to the parser.

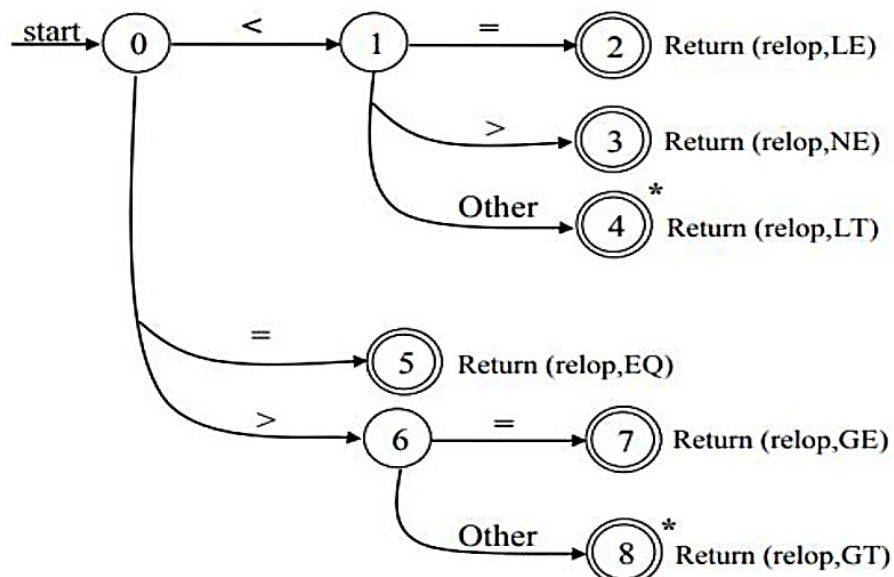## Transition Diagram

- As an intermediate step in the construction of a lexical analyzer, we first produce a flowchart, called a r diagram. Transition diagrams.

- Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

- The TD uses to keep track of information about characters that are seen as the forward pointer scans the input.

- It does that by moving from position in the diagram as characters are read.

COMPONENTS OF TRANSITION DIAGRAM

1. One state is labelled the **Start State**   start◯   It is the initial state of transition diagram where control resides when we begin to recognize a token.

2. Position is a transition diagram are drawn as circles ◯ and are called states.

3. The states are connected by **Arrows** ⟶ called edges. Labels on edges are indicating the input characters.

4. The **Accepting** states in which the tokens has been found ◎

5. **Retract** one character use * to indicate states on which this input retraction.
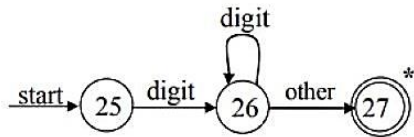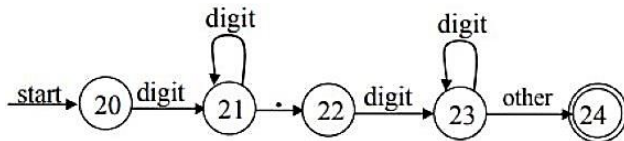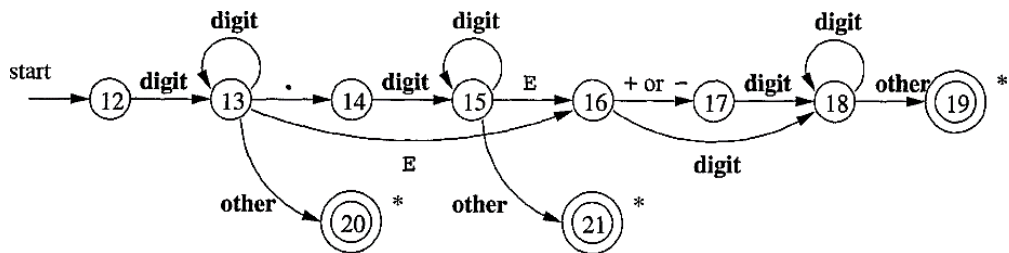
Example: A Transition Diagram for the token relation operators "relop" is shown in Figure below:

Example: A Transition Diagram for the **identifiers** and **keywords**
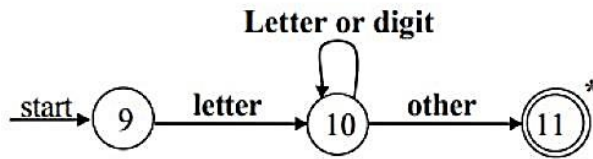
Example: A Transition Diagram for **Unsigned Numbers** in Pascal

num $\longrightarrow$ digit$^+$(. digit$^+$| $\in$)(E(+|-|$\in$)digit$^+$|$\in$)