Programming Languages

Hierarchy of Programming Languages based on increasing machine independence includes the following:

1- Machine – level languages.

2- Assembly languages.

3- High – level or user oriented languages.

4- Problem - oriented language.

1- Machine level language: is the lowest form of computer. Each instruction in program is represented by numeric code, and numerical addresses are used throughout the program to refer to memory location in the computer's memory.

2- *Assembly language:* is essentially symbolic version of machine level language, each operation code is given a symbolic code such ADD for addition and MULT for multiplication.

3- *A high level language* such as Pascal, C.

4- *A problem oriented language* provides for the expression of problems in specific application or problem area. Examples of such as languages are SQL for database retrieval application problem oriented language.

Using a high-level language for programming has a large impact on how fast

programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought run on many different machines.

Language processing system

Actually we are trying to convert the high-level language (the source-code we written) to Low-level language (Machine Language). This process involves four stages and utilizes following 'tools':

1. Pre-processor
2. Compiler
3. Assembler
4. Loader/Linker

Compiler

Is a program that reads a program written in one language, (the source language) and translates into an equivalent program in another language (the target language) as shown in figure (1).

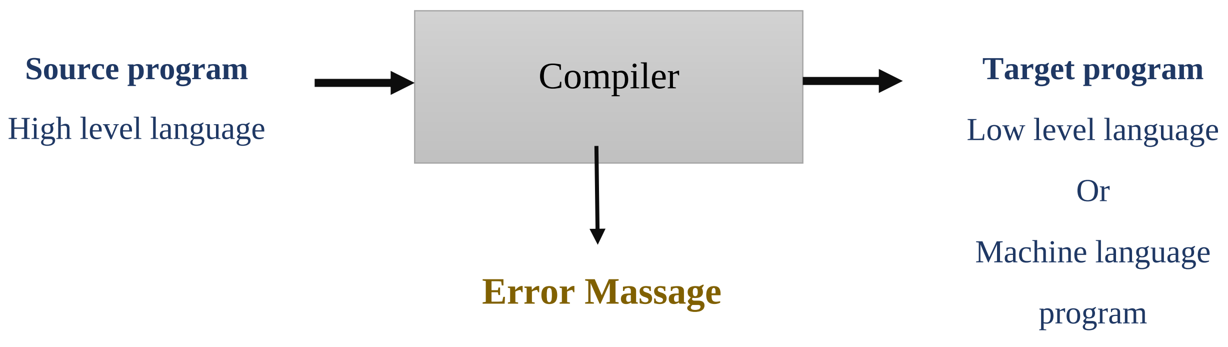


Figure (1): General structure of compiler program

Translator

A translator is program that takes as input a program written in a given programming language (the source program) and produce as output program in another language (the object or target program). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program as shown in figure (2).
If the source language being translated is assembly language, and the object program is machine language, the translator is called **Assembler.**

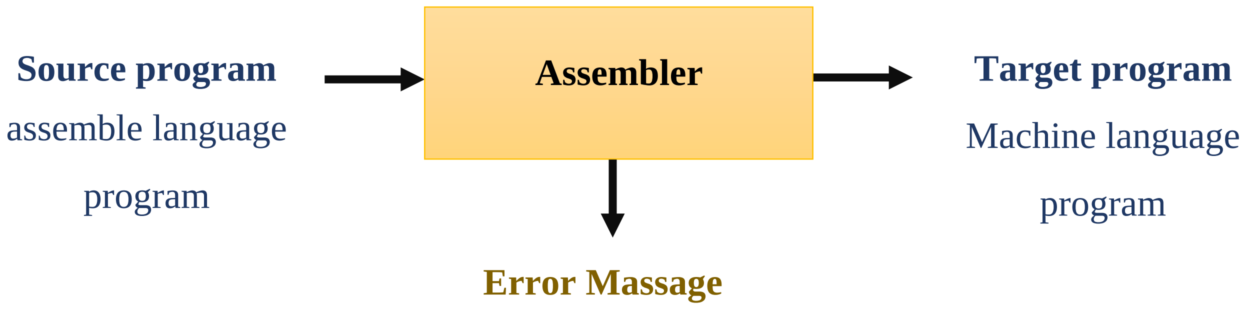


Figure (2) : General structure of Assembler program

A translator, which transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler**.

Another kind of translator called an **Interpreter** . An interpreter converts high level language into low level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. Compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower with respect to compiled ones. Figure (3) illustrate the interpretation process.

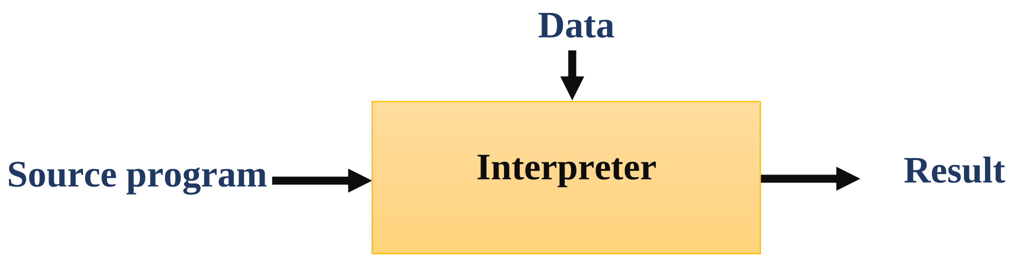


Figure (3) : Interpretation process

The Analysis - Synthesis model of compilation

There are two parts to compilation: analysis and synthesis.

1- Analysis phase (Front- end):- an intermediate representation is created from the give source code

1. Lexical Analyzer (scanner)
2. Syntax Analyzer (parser)
3. Semantic Analyzer
4. Intermediate Code generator

2- Synthesis Phase (Back-end) :- – equivalent target program is created from the intermediate representation. It has two components:

1. Code Optimizer
2. Code Generator

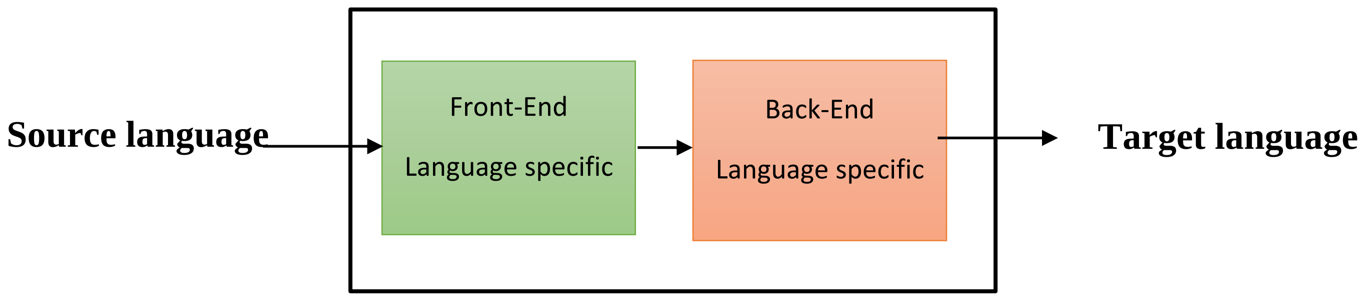


Figure (4) : The Analysis - Synthesis model of compilation

Phases of a Compiler:

A Compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is divided into a series of sub process called ***Phases.*** Figure (5) illustrated the compiler phases

- The different phases of a compiler are as follows

  Analysis Phases:

  1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Intermediate Code generator

  Synthesis Phases:

  5. Code Optimization
  6. Code generation

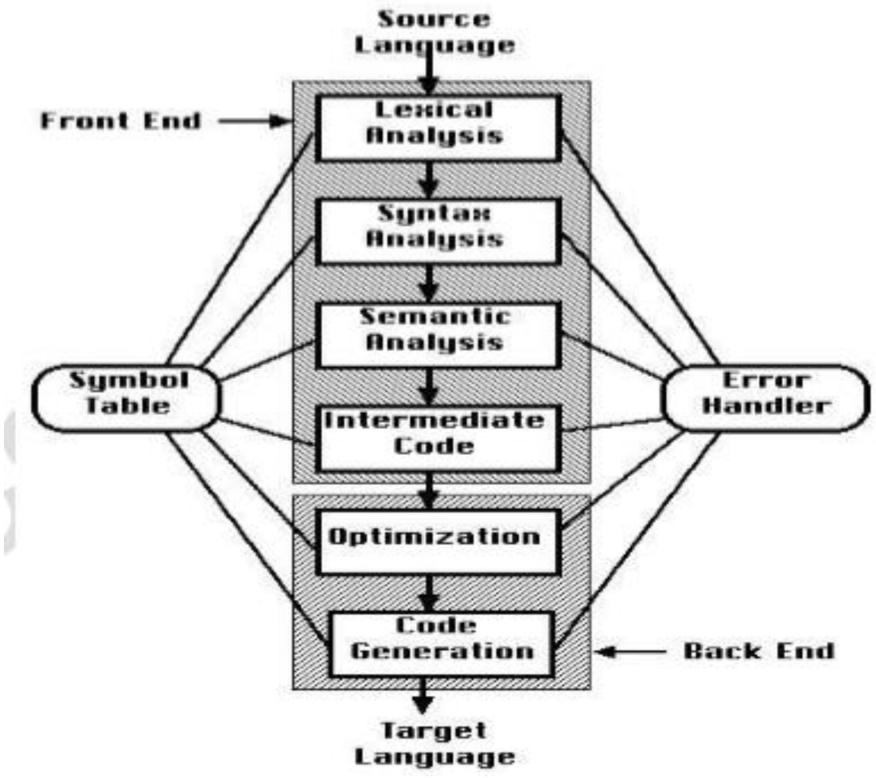


Figure (5) : Phases of Compiler

Compiler structure:

1- lexical analysis

1.1 The Role of lexical analysis

The lexical analyzer is the first stage of a compiler. The main task of 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.

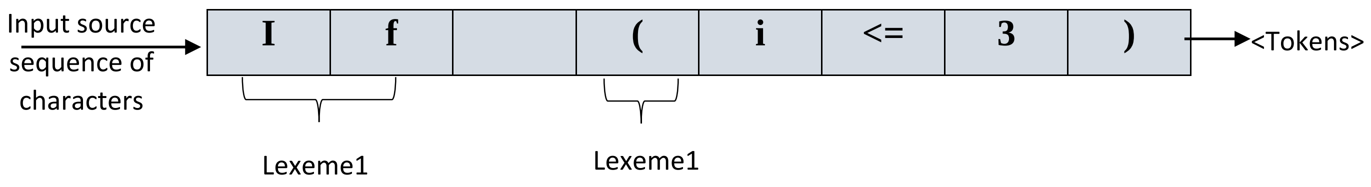


Figure (6): Example of the lexeme

As shown in figure (6) the lexical analyzer scanning the input source characters one by one whenever formatted lexeme then results to this lexeme token that the parser uses for syntax analysis as shown in figure (7).

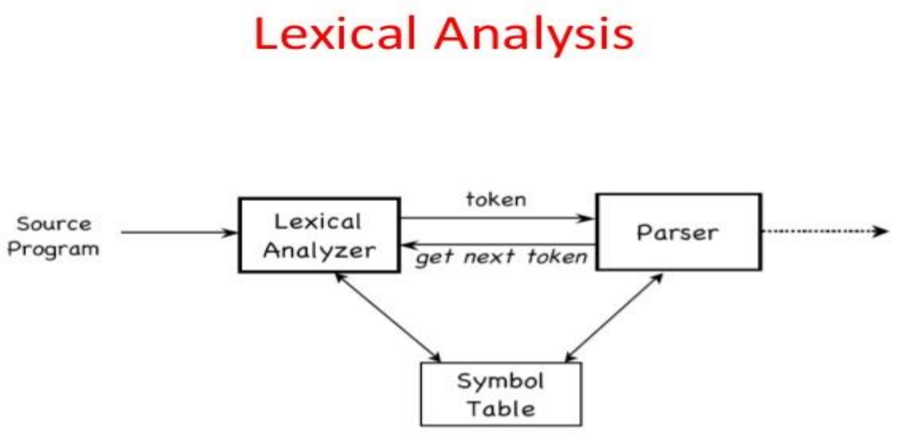


Figure (7): Interaction of lexical analyzer with parser

1.2 lexical analyzer tasks

lexical analyzer tasks are divided into following process:

a) Scanning: consists of the simple processes that don't require tokenization of the input, such as deletion of comments compaction of consecutive whitespace characters into one.

b) Lexical analysis proper: is more complex portion, where the scanner produces the sequence of tokens as output.

1.3 Tokens, Patterns, Lexemes

When discussing lexical analysis ,we use three related but distinct terms :

Tokens is pair consisting a token name and optional attributed value. Token name is abstract symbol representing kind of token unit which are

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants.

The token names are the input symbols that the parser processes.

Note: The optional attributed means it can be and may not exist.

For example <If > or < id , pointer symbol-table entry E>

Lexeme is a sequence of characters in the source program that is matched by the pattern for a token. In general, the lexeme is stored in symbol table specially if the lexeme is identifier.

Pattern: is a description of the form that the lexeme of token my take. In case of keyword as a token the pattern is the sequence of characters that form the keyword.

Table (1): Example of the token

Tokens	***Pattern***	***Example of Lexeme***
if	Characters I,f	if
else	Characters e,l,s,e	else
Comparison	<, >, <= , >= , == , !=	<, >, <= , >= , == , !=
id	Letter followed by letter or digit	X ,y3,count
Number	Any numeric constant	3.14159, 0, 6, 02e23
literal	fixed value in source code	String s = "cat" int a=1

In many programming languages, most or all of the tokens are:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the1 operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

1.***4 Attributed of token***

The attribute of token is a structure that combines several pieces of information the most important example is the token of identifier. The properties of attributed of identifier is a pointer to symbol table entry for that identifier as shown in figure (8).

Example

The token name and associated attribute values of the Fortran statement: E = M * C ** 2

Lexical analyzer is writing Fortran statement as sequence of pair

<id, pointer to symbol –table entry E>

< assign-op>

<id, pointer to symbol –table entry M>

< Multi-op>

<id, pointer to symbol –table entry C>

< Exp -op>

<number, integer value 2>

Figure (8) : Example of the attribute of token

1.5 Input buffer

Lexical analyzer scans the characters of the source program one at a time to discover tokens. It is desirable for the lexical analyzer to input from buffer.

1.5.1 Buffer pairs

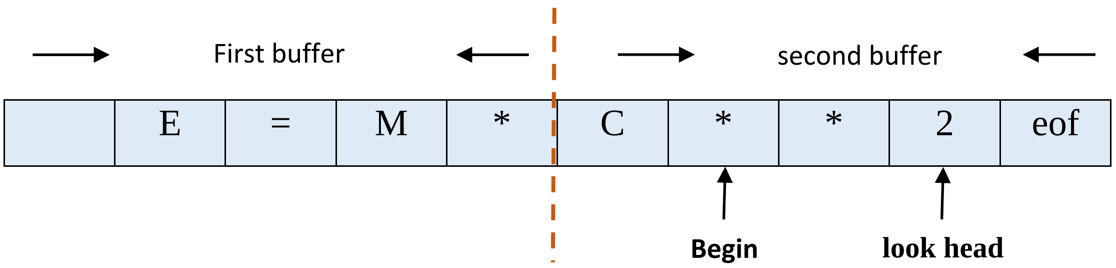


Figure (9) Using a pair of input buffer.

1- Pointer lexeme Begin, marks the beginning of the token being discovered.

2- look head pointer scans ahead of the beginning pointer, until a token is discovered.

Symbol Table

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various source language constructs. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code. We required several capabilities of the symbol table we need to be able to:

1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.

insert(s,t) : this function is to add a new name to the table

Lookup(s) : returns index of the entry for string s, or 0 if s is not found.

2- Access the information associated with a given name, and add new information for a given name.

3- Delete a name or group of names from the tables.
For example, consider tokens **begin,** we can initialize the symbol-table using the function**: insert("begin",1).**

Symbol table management refers to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

1) A symbol table is a data structure, where information about program objects is gathered.

2) Is used in all phases of compiler.

3) The symbol table is built up during the lexical and syntax analysis.

4) Help for other phases during compilation:

1.6 Specification of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

1.6.1 Strings and Languages

The term of ***alphabet*** or ***character class*** denotes any finite set of symbols. Typical examples of symbol are letter and characters. The set {0, 1} is the ***binary alphabet*** ASCII is the examples of ***computer alphabets***.

String: is a finite sequence of symbols taken from that alphabet. The terms ***sentence*** and ***word*** *are* often used as synonyms for term "string".

|S|: is the **Length** of the string S.
Example: |banana| =6

Empty String (∈): special string of length zero.

Exponentiation of Strings

$\mathbf{S^2}$ = SS $\mathbf{S^3}$ = SSS $\mathbf{S^4}$ = SSSS

$\mathbf{S^i}$ is the string **S** repeated **i** times.

By definition S^0 is an empty string.

Languages: A language is any set of string formed some fixed alphabet.

Operations on Languages

There are several important operations that can be applied to languages.

For lexical Analysis the operations are:

1- Union.

2- Concatenation.

3- Closure.

Operation	Definition
Union ***L*** and ***M*** written ***L***∪***M***	***L***∪***M***={s | s is in ***L*** or s in ***M***}
Concatenation of ***L*** and ***M*** written ***LM***	***LM***={st | s is in ***L*** and t is in ***M***}
Kleene closure of ***L*** written ***L****	$L^{*}=\bigcup_{i=0}^{\infty} L^{i}$ ***L^* denotes "zero or more Concatenation of "L***
Positive closure of L written L^{+}	$L^{+}=\bigcup_{i=1}^{\infty} L^{i}$ ***L+ denotes "One or more Concatenation of "L***

Example: Let ***L*** and ***M*** be two languages where ***L*** = {a, b, c} and ***D***= {0, 1} then

- Union: ***LUD*** = {a, b, c, 0,1}
- Concatenation: ***LD*** = {a0,a1, b0, b1, c0,c1}
- Expontentiation : L^2 = LL

By definition: L^0= {∈}

1.6.2 Regular Definitions

A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

Example1: The set of C identifiers is the set of strings of letters and digits beginning with a letter.Here is a regular definition for this set:

letter → A | B | . . . | Z | a | b | . . . | z

digit → 0 | 1 | 2 | . . . | 9

id → letter (letter | digit)*

The regular expression **id** is the pattern for the C identifier token and defines **letter** and **digit.** Where **letter** is a regular expression for the set of all upper-case and lower case letters in the alphabet and **digit** is the regular for the set of all decimal digits.

Example 2: Unsigned numbers in Pascal are strings such as 5280, 39.37, 6.336E4, or 1.894E-4. The following regular definition provides a precise specification for this class of strings:

digit → 0 | 1 | 2 | . . . | 9

digits → digit $^{+}$

optional-fraction → . digits | ∈

optional-exponent → **(E** (+ | - | ∈) **digits)** |∈

Num → digits optional-fraction optional-exponent

This regular definition says that

- An optional-fraction is either a decimal point followed by one or more digits or it is missing (i.e., an empty string).
- An optional-exponent is either an empty string or it is the letter E followed by an optional + or - sign, followed by one or more digits.

Running Example:

Stmt $\rightarrow$ *if Expr then Stmt*

| *if Expr then Stmt else Stmt*

| *e*

Expr $\rightarrow$ *Term relop term*|*Term*

Term $\rightarrow$ *id*| *number*

In the above example, the production of the regular definition has two type of tokens (terminal and non -terminal), the lexical analysis interest about terminal tokens such as (if, then, else, and relop (relational operation).

Terminal tokens

{if, else, then}	$\rightarrow$ keywords
relop	$\rightarrow$ operation
Number	$\rightarrow$ number
id	$\rightarrow$ identifier

Now we need regular definition for each terminal tokens which is representing patterns for each token as following.

$digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

$digits \rightarrow digit^{+}$

$Number \rightarrow digit\ (.\ digits \mid \in)(E\ (+ \mid - \mid \in)\ digits) \mid \in$

$letter \rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z$

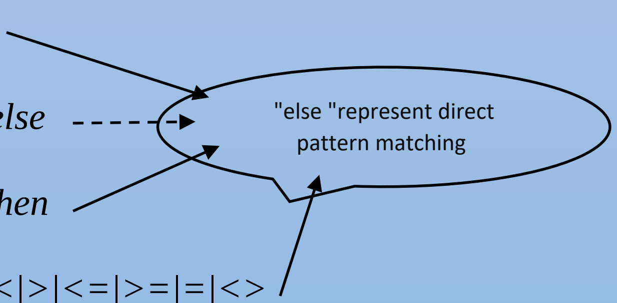
$id \rightarrow letter\ (letter \mid digit\ )^{*}$

$if \rightarrow if$

$else \rightarrow else$

$then \rightarrow then$

$relop \rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$



In addition , we assign the lexical analyzer the job of stripping out white space ,by recognizing the "token" *ws* defined by

$ws \rightarrow (blank \mid tab \mid newline)^{+}$

Since the {blank ,tab , newline } are terminal tokens then the lexical analyzer can be recognition by single ASCII code for each them.as noted the assign ($^{+}$) means there is a possibility of one or more spaces.

Table (2): Lexeme , Tokens , and attributes for Running example

Lexeme	***Token***	***Attribute value***
Any ws	-	-
If	*If*	-
Then	*Then*	-
Else	*Else*	-
Any id	*id*	*Pointer to table entry*
Any number	*number*	*Pointer to table entry*
<	*Relop*	*LT*
<=	*Relop*	*LE*
=	*Relop*	*EQ*
<>	*Relop*	*NE*
>	*Relop*	*GT*
>=	*Relop*	*GE*

Transition diagrams

A transition diagram is similar to a flowchart for (a part of) the lexical. We draw one for each possible token. It shows the decisions that must be made based on the input seen. The two main components are circles representing states and arrows representing edges.

Example : transition diagram for the relational operation (relop) shown in figure (10).

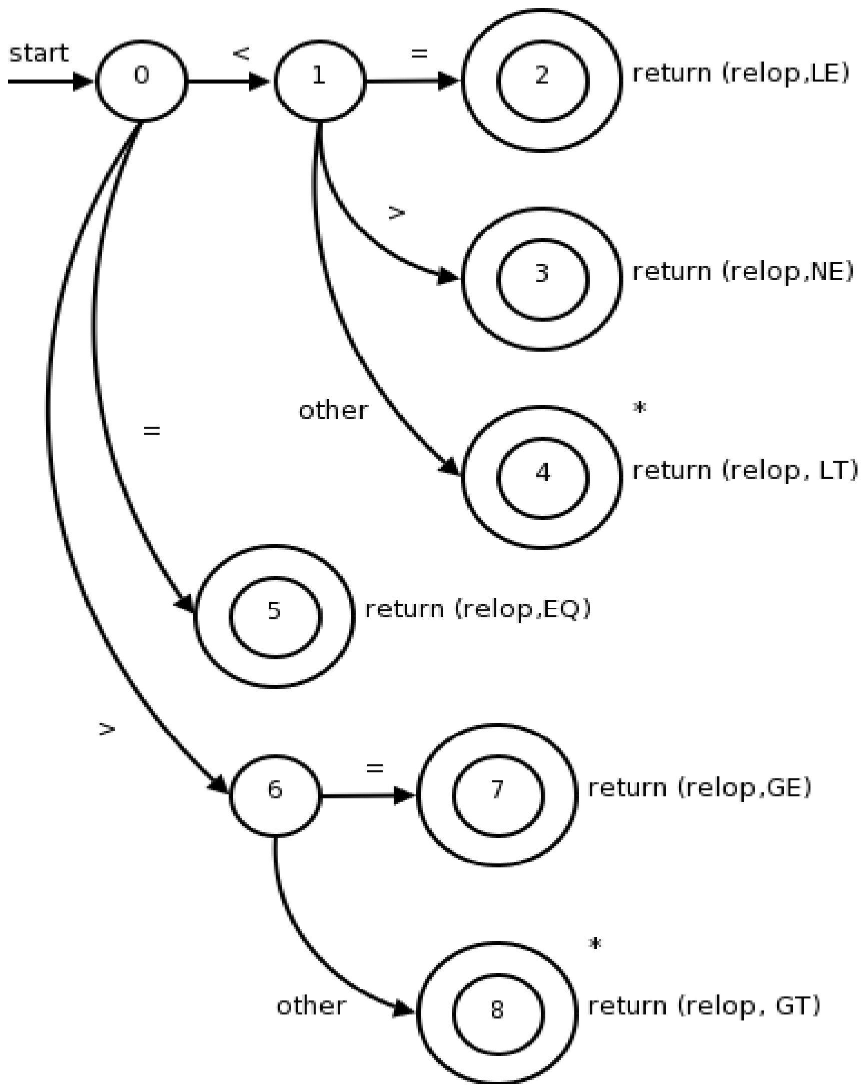


Figure (10): Transition diagram of the relop

1. The double circles represent *accepting* or *final* states at which point a lexeme has been found. There is often an action to be done (e.g., returning the token), which is written to the right of the double circle.
2. Each edge is labeling by symbol or set of symbols.

Recognition of tokens

1. ***Recognition of Reserved Words and Identifiers***

To turn a collection of transition diagrams into a program, we construct a segment of code for each state. The first step to be done in the code for any state is to obtain the next character from the input buffer. For this purpose, we use a function ***GETCHAR***, which returns the next character, advancing the look ahead pointer at each call. The next step is to determine which edge, if any out of the state is labeled by a character, or class of characters that includes the character just read. If no such edge is found, and the state is not one which indicates that a token has been found (indicated by a double circle), we have failed to find this token. The look ahead pointer must be retracted to where the beginning pointer is, and another token must be search for using another token diagram. If all transition diagrams have been tried without success, a lexical error has been detected and an error correction routine must be called as shown in figure (11) which is illustrated transition diagram for identifier.

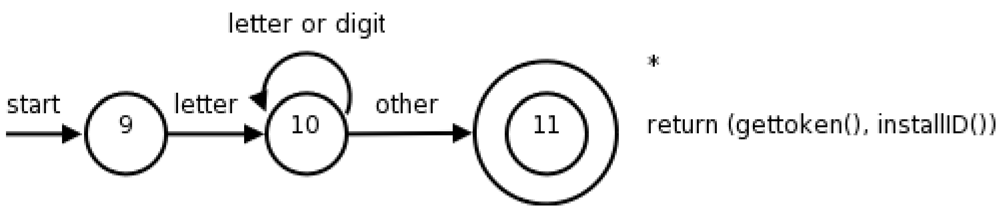


Figure (11): Transition diagrams for identifier.

```
State 0 : C = GETCHAR ( )
if LETTER(C) then goto state1
else FAIL( )
State1 : C= GETCHAR ( )
if LETTER(C) or DIGIT(C) then goto state1
```

```
else if DELIMTER(C) then goto state2
else FAIL ( )
State2: RETRACT ( )
return( id, INSTALL( ) )
```

LETTER(C) is a procedure which return true if and only if C is a letter.

DIGIT(C) is a procedure which return true if and only if C is one of the digit 0,1,…9.

DELIMITER(C) is a procedure which return true whenever C is character that could follow an identifier. The delimiter may be: blank, arithmetic or logical operator, left parenthesis, equals sign, comma,…

Install ( ) checks if the lexeme is already in the symbol table

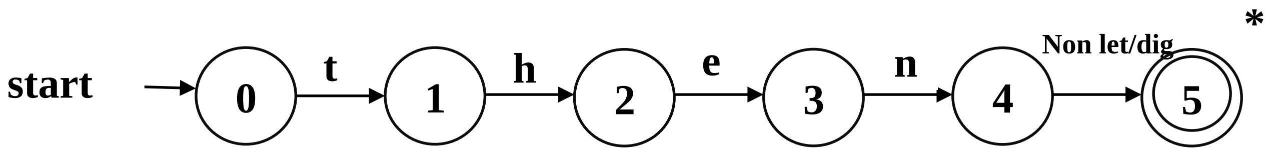


Figure (12) : transition diagram of the keyword then.

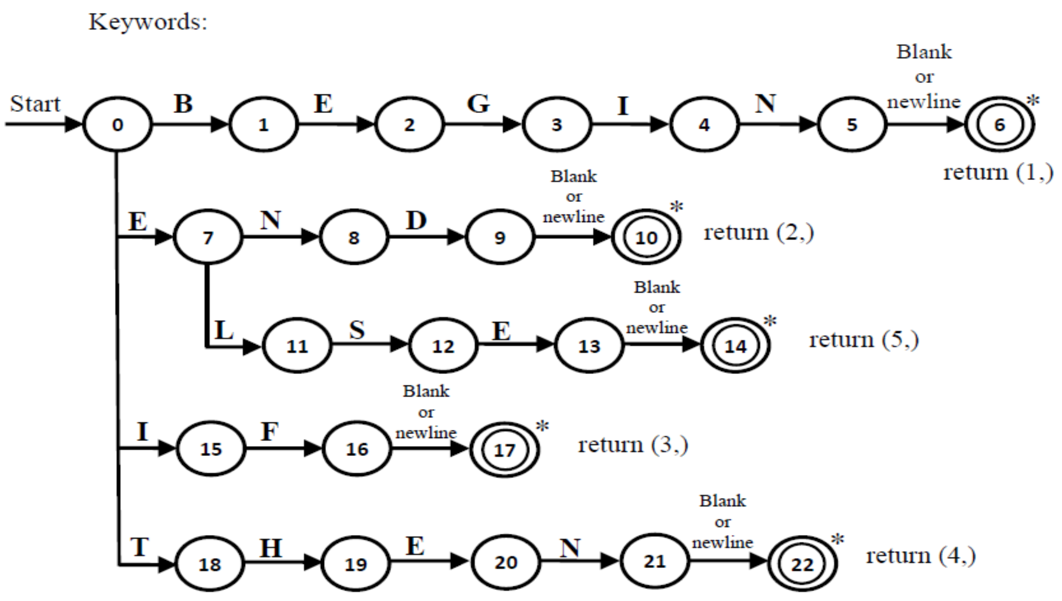


Figure (13) : transition diagram of the keyword then.

Recognizing Numbers

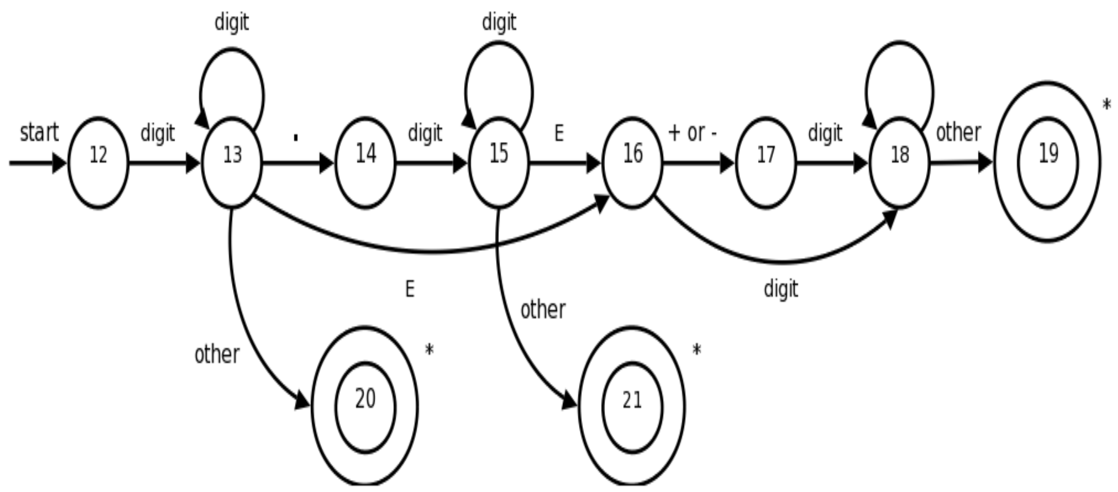


Figure (14) : transition diagram of the number.

The lexical analyzer returns to parser a representation for the token it has found. This representation is:

• An **integer code** if there is a simple construct such as a left parenthesis, comma or colon.

• Or a **pair** consisting of an **integer code** and a **pointer to a table** if the token is more complex element such as an **identifier or constant.**