UNIT 1 : INTRODUCTION AND LEXICAL ANALYSIS

❖ LANGUAGE PROCESSING:

The different steps involved in converting instructions in high level language to machine level code is called **language processing**.

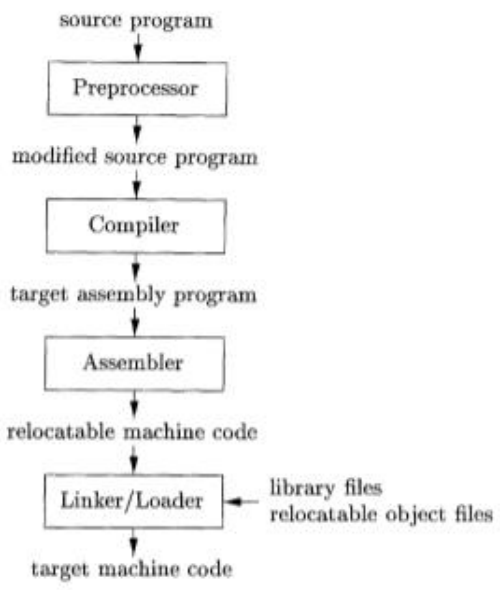


Figure 1.5: A language-processing system

There are different components involved in language processing.
They are:

1. **Pre-processor:**
   - First step in language processing.
   - Input to this phase is source program. Different parts of the source program may be stored in different files.
   - Pre-processor **collects all these files** and creates a single file. It also performs **macro expansion**.
   - In C #define and #include are expanded during pre-processing. Some pre-processors also **delete comments** from the source program.

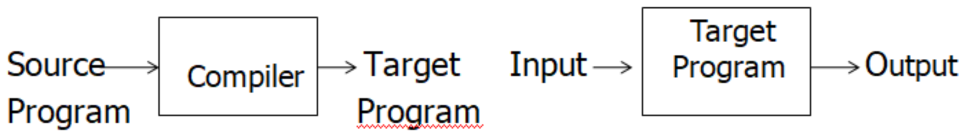
2. **Compiler:**
   - Compiler takes pre-processed file and generates assembly level code.
   - It also generates symbol table and literal table. Compiler has error handler which displays error messages and performs **some error recovery if necessary.**

3. **Assembler:**
   - Assembler takes assembly code as input and converts it into **re-locatable object code.**
   - The instruction in assembly code will have three parts label, opcode part and an operand part.
     Ex. : mov r1,r2

❖ Examples of Language processor:

1. Compiler:

- It is a translator program that can read program written in one language (Source language) and **translate** it into an equivalent program in another language (target language).
- If there is **any error** in the source program that it **detects** in the translation process.
- The target program is executable machine language, called by the user to process the inputs and produce the required outputs.



There are 2 types of compilers:

1. **Hybrid Compiler :**

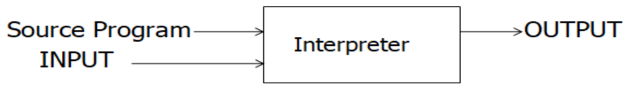
- This language processor combines the compilation and interpretation.
- That is first source program may be compiled into an intermediate form called **BYTECODES** and these byte codes are interpreted by virtual machine.
- Example: JAVA virtual machine
- The advantage is that the byte codes are compiled on one machine can be interpreted by another machine.

2. **Just in Compiler :**

- In order to achieve faster processing of inputs to outputs, some Java compilers translate the BYTECODE into machine language immediately before they run the intermediate program to process the input. These are called Just-In-Compiler.

2. Interpreter:

- An interpreter is another common kind of language processor.
- Instead of producing a target program as a translation, an interpreter appears to directly **execute the operations** specified in the source program on inputs supplied by the user.



❖ Compiler v/s Interpreter:

Compiler	Interpreter
1. Compiler generates intermediate code.	1. It doesn't generates intermediate code
2. It has to generate the intermediate code. Hence there is more scope for code optimization.	3. It will not generate Intermediate code. Hence there is a less scope for code optimization.
3. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs	4. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.
4. Examples : Pascal, C, C++ etc..	4. Examples BASIC, LISP and POST SCRIPT

❖ The Structure of a Compiler :

The process of converting source program to target code has teo parts.

1. ***Analysis part*** :
   - This part breaks up the source program into consistent pieces and creates an intermediate representation of the source program.
   - If it detects that the source program is either syntactically ill formed or semanti- cally unsound, then it provides informative messages.
   - It is often called the front end of the compiler.

2. ***Synthesis part*** :
   - This part constructs the designed target program from the intermediate representation.
   - This part requires the most specilised techniques.
   - It is often called the back end of the compiler.

❖ Phases of the compiler:

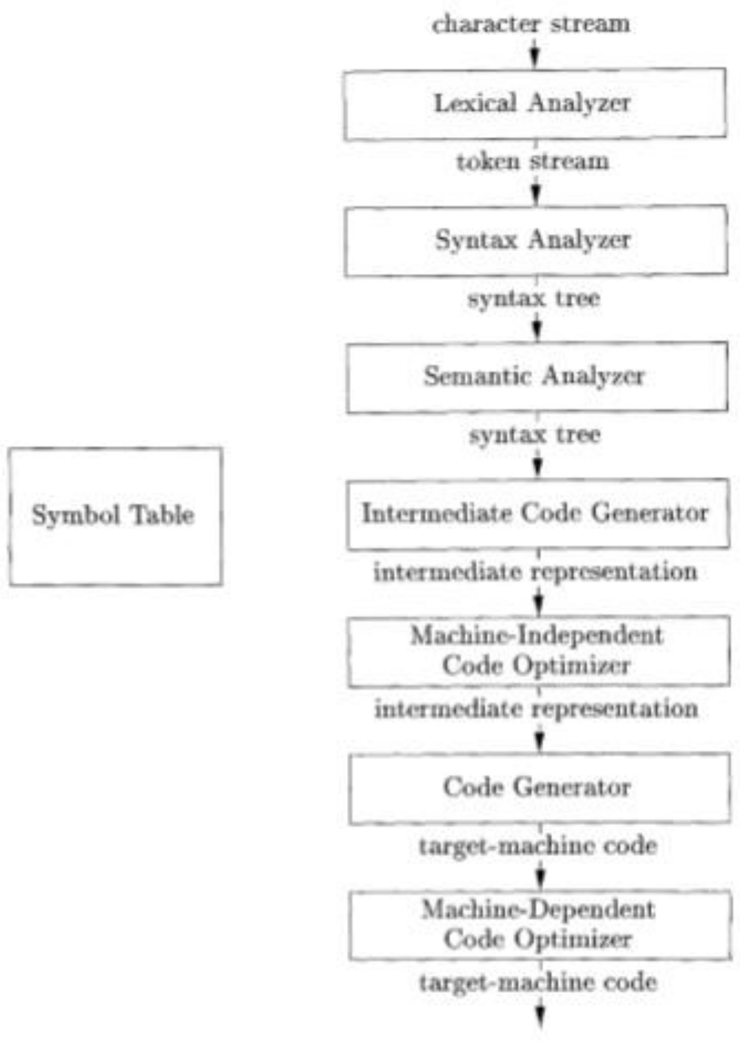


Figure 1.6: Phases of a compiler

1. **Lexical Analysis:**

- The main function of Lexical Analyser is **to break the source program into tokens**.
- Tokens are classified as key words, identifiers, operators, etc.
- Lexical analyser then **creates symbol table** and **literal table** to store the identifiers and constants respectively.
- This phase takes care of **detecting few lexical errors** and applies some strategy and tries to recover from errors.

2. **Syntax Analysis:**

- Syntax Analyser **determines the structure of the program**.
- The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
- Syntax Analyser **uses context free grammar** to define and validate rules for language construct.
- Output of Syntax Analyser is **parse tree or syntax** tree which is hierarchical / tree structure of the input.

3. **Semantic Analysis:**

- Input to semantic analysis phase is parse tree or syntax tree.
- Important function of semantic analysis phase is:

  1.Type checking

  2.Type conversion.

  3.Checks for some semantic errors like array size specified as float.
- Output of semantic analyser is an **annotated parse tree**.

4. **Intermediate Code Generation:**

- It generates intermediate code for annotated parse tree.
- This code is dependent on source program and **independent of machine architecture**.
- The intermediate code uses three address codes and uses temporary variables to store the intermediate results.

5. **Code optimization:**

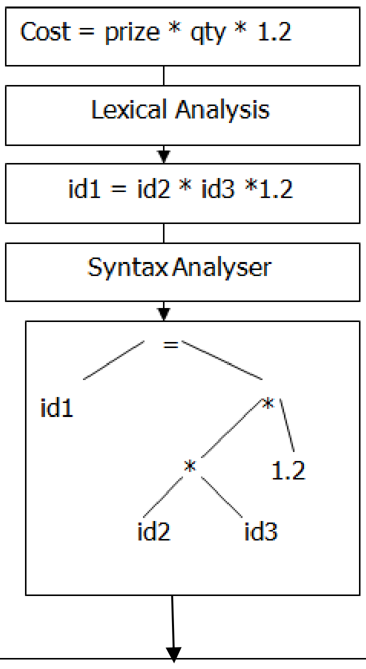
- This is aimed at **reducing the number of operations** and thus time taken for execution.
- It also takes care that it **uses minimum temporaries** to store intermediate values
- Code optimizer concentrates on that part of the code which will be executed many times and tries to optimise that code.

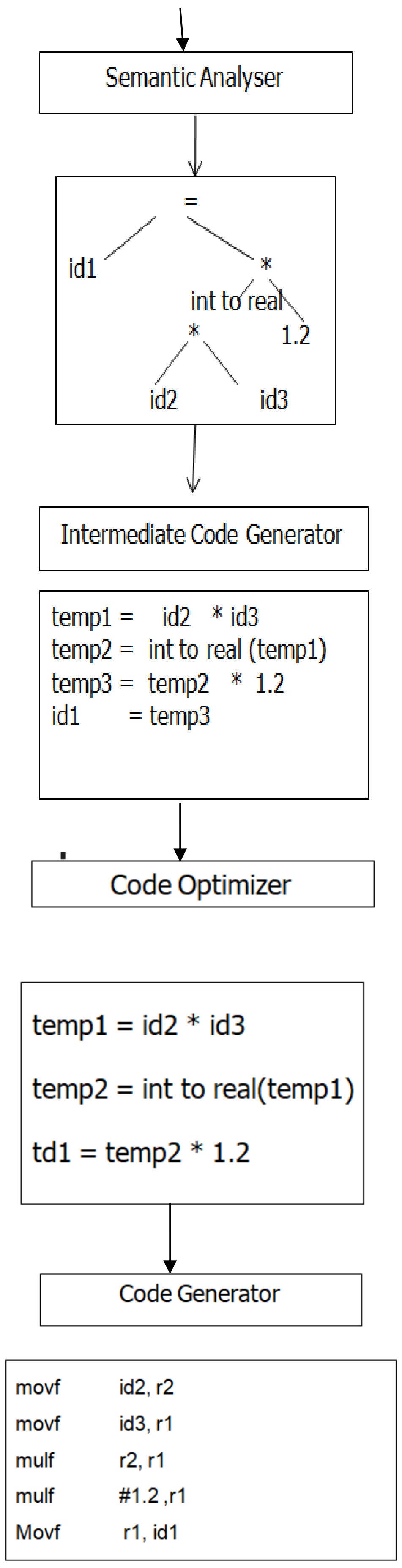
6. **Code Generation:**

- Input to code generation phase is the intermediate optimised code generated from code optimization phase or intermediate code with no optimization.
- It **generates a code** that runs on target machine.
- It is the most difficult phase as it is **an NP problem**.

❖ **Examples:**

1. **Illustration of phases of compiler through an example Cost = prize * qty * 1.2:**





```
temp1 =    id2   * id3
temp2 =  int to real (temp1)
temp3 =  temp2   *  1.2
id1       = temp3
```

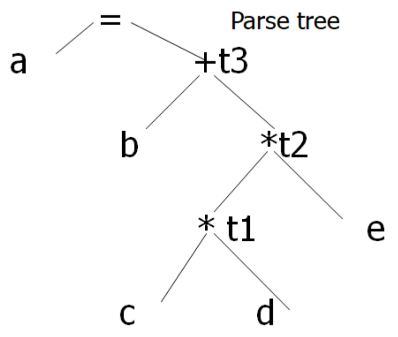
```
temp1 = id2 * id3

temp2 = int to real(temp1)

td1 = temp2 * 1.2
```

```
movf        id2, r2
movf        id3, r1
mulf        r2, r1
mulf        #1.2 ,r1
Movf         r1, id1
```

2. **An arithmetic expression a = b+c*d*e**

Intermediate code

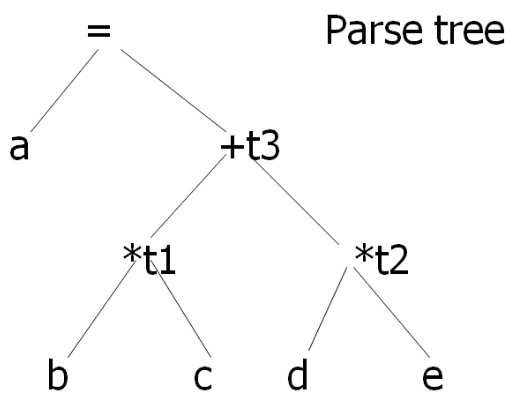
t1= c*d
t2 = t1*e
t3 = b+t2
a = t3



3. **An arithmetic expression a = b*c + d*e**

Intermediate code

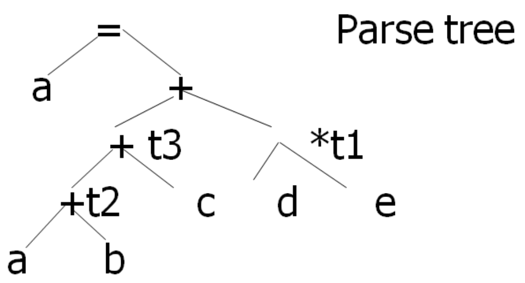
t1 = b*c
t2 = d*e
t3 = t1+t2
a = t3



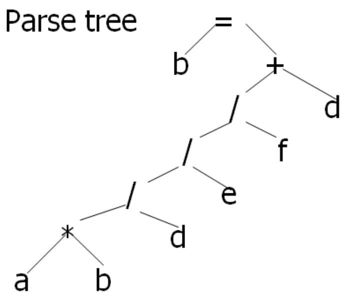
4. **An arithmetic expression a = a+b+c+d*e**

Intermediate code

t1 = d*e
t2 = a + b
t3 = t2 +c
a = t3 + t1



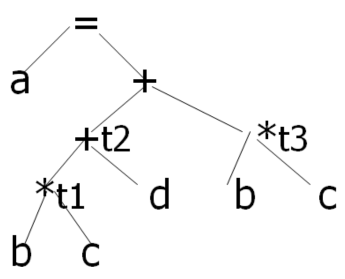
5. **Ex 7: b = a*b/d/e/f+d**



Intermediate code

t1 = a*b
t2 =t1/d
t3 = t2/e
t4 = f+d
b = t3/t4

6. **An example for Code Optimization for exp a = b*c + d + b*c**



Intermediate code

t1 = b*c
t2 = t1+d
t3 =b*c
t4 = t2 +t3
a =t4

Optimized Code

t1= b*c
t2 = t1 + d
t3 = t2 +t1

❖ Lexical Analysis :

❖ Implementation of a lexical analyzer :

There are 2 ways of doing this:

1. By hand:

- It starts with a **diagram** or other **description for the lexemes** of each token.
- We can then **write code** to identify each occurrence of each lexeme on the input and to return information about the token identified.

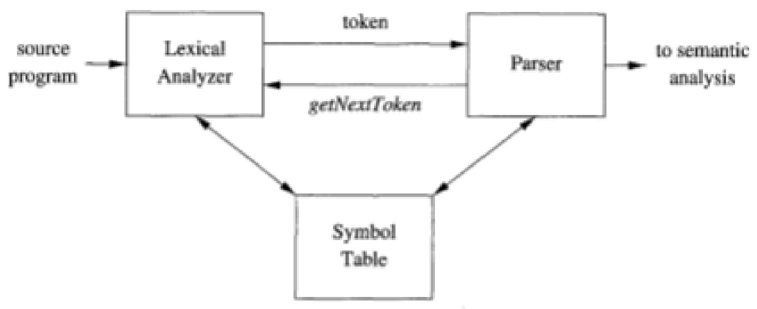
2. Automatic:

- It starts with **specifying the lexeme patterns** to a lexical analyzer generator and compiling those patterns into code that functions as a lexical analyzer.
- This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program.
- It **also speeds up** the process of implementing the lexical analyzer, since the programmer specifies the software at the very high level of patterns and relies on the generator to produce the detailed code.

❖ The Role of Lexical Analyser:

- The main function of Lexical Analyser is to **break the source program into tokens**.
- Tokens are classified as key words, identifiers, operators.
- Generates token with the help of regular expression or finite automata.
- Eliminates blank spaces, tab and new line characters.
- Detecting lexical errors if any, correlating error message with position of error, like line number, function where the error is detected or file where the error is found.
- Macro expansion

❖ Interactions between the lexical analyzer and the parser:



- 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.

❖ Lexical analyzers are divided into a cascade of two processes:

a) **Scanning:**

It consists of the simple processes that do **not require tokenization** of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) **Lexical analysis:**

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

❖ Tokens, Patterns, and Lexemes :

Token:

- It 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 and are the input symbols that the parser processes.

Pattern:

- It is a **description** of the form that the lexemes of a token may take.
- 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.

Lexeme:

- It 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.

Examples:

1. In the C statement ,

   printf ("Total = %d\nI1, score) ;

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

2. **float limitedSquare(x) float x ;**

```
{
        /* returns x-squared, but never more than 100 */
       return ( x<=-10.0  || x>=10.0 ) ? 100: x*x;
 }
```

Here,

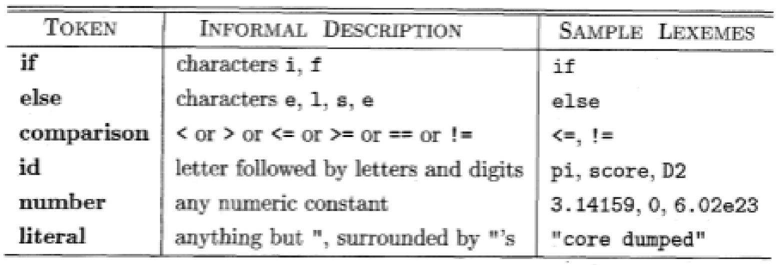
float, **limitedSquare**, **x** and **return** are lexemes matching the pattern for token **id**.

-10.0, **10.0**, **100** are lexemes matching the pattern for token **number**.

<=, >= are lexemes matching the pattern for token **relop** (relational operator).

|| is lexeme matching the pattern for token **logop** (logical operator).
* is lexeme matching the pattern for token **arithop** (arithmetic operator).

3.



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 3.2: Examples of tokens

❖ Lexical errors:

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.

- For instance, if the string **fi** is encountered for the first time in a C program in the context: fi(a == f(x))...

- A lexical analyzer cannot tell whether **fi** is a misspelling of the keyword if or an undeclared function identifier.

- Since **fi** is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - probably the parser in this case However, 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.

- 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.

❖ Input buffering:

Here are the some ways that **speed up the process of reading** the source program.This is needed because we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme as this consumes much time.

1. **Buffer Pairs:**

An important scheme involves two buffers that are alternately reloaded, as suggested in figure.

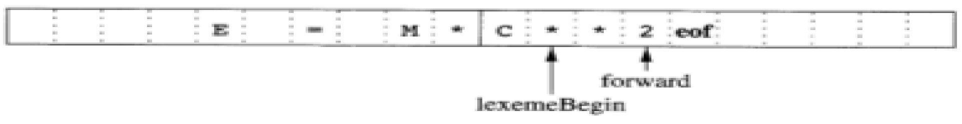


Figure 3.3: Using a pair of input buffers

- Each buffer is of the same size **N**, and N is usually the size of a disk block, e.g., 4096 bytes.
- Using one system **read** command we can read N characters into a buffer, rather than using one system call per character.
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.
- Two pointers to the input are maintained:
  I. **Pointer lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. **Pointer forward** scans ahead until a pattern match is found.

2. **Sentinels :**

- If we use the above scheme, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read.
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold **a sentinel character** at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.

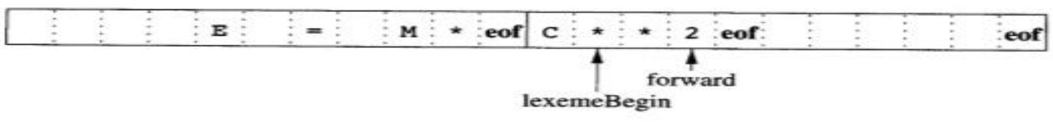


Figure 3.4: Sentinels at the end of each buffer

```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

❖ Specification of tokens:

- Alphabet is a finite set of all characters, digits, operators and punctuation marks that can be used in the source language.
- Example: in C language ∑= {0,1,…9, a, b,c, ….z, A,B,….Z,+,- , *, / , (,), &, <, > , =,…..}
- String is a finite sequence of symbols drawn from the alphabet.
- Example: w = abc or s = abc234
- We can also use regular expressions.
- **Regular Definitions:** If C is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:
  d1->r1
  d2->r2
  ...
  dn->rn
  where:
  1. Each di is a new symbol, not in C and not the same as any other of the d's, and
  2. Each ri is a regular expression over the alphabet C U {dl, d2,. . . , di-l).

❖ Recognition of Tokens:

- Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token.
- Using the token, code has been generated that examines the input string and finds a prefix that is lexeme matching one of the patterns.

❖ Recognition of Tokens and Implementation:

- **Grammar fragment for Branching statement:**

  Stmt → **if** *expr* **then** *stmt*

  | **if** *expr* **then** *stmt* **else** *stmt*

  expr → *term* **relop** *term*

  |*term*

  term→ **id** |**number**

- **Patterns for description of tokens using regular definitions :**

digit	→ [0-9]
digits	→ *digit+*
number	→ *digits* (•*digits*) ? (E[+–]? *digits*) ?
letter	→ [A-Za-z]
id	→ *letter* (*letter* | *digit*) *
if	→ if
then	→ then
else	→ else
relop	→ <|> |<=|>= |= |<>

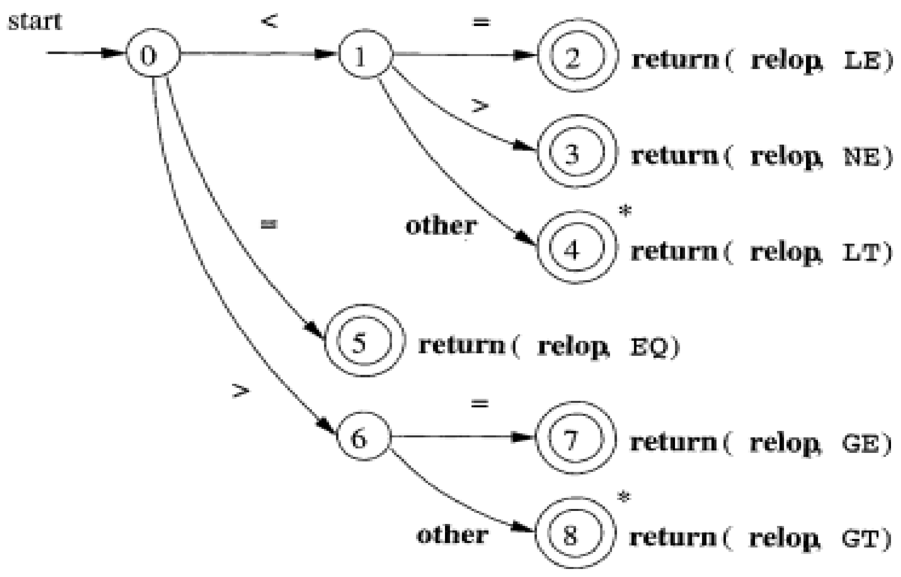


Figure 3.13: Transition diagram for **relop**

❖ Method for Translating the transition diagram into a program using the following steps:

- Every **STATE** gets piece of code that reads a character from the input buffer.
- The character read is compared with the labels of the outgoing edges for the current **STATE**. If any match then control is transferred to the respective **Next-STATE** by calling the code associated with it, otherwise there is a failure to recognize the token and next transition diagram may be activated for another token.
- If state is a **FINAL STATE** then no character is read and the **token definition** along with **NAME** and **ATTRIBUTE** is returned.
- If a **FINAL STATE** is marked with '*' then one or more previously read characters, are not part of the lexeme hence retract is performed accordingly to move the **FORWARD** pointer one or more positions back.

❖ Some functions that are being used here:

1. **getRelop()** → It is C++ function whose job is to simulate the transition diagram for **relational operators** and returns an object of type **TOKEN,** that is pair consisting of the **token name** and an **attribute value.**
   It also first creates a new object **retToken** and initializes its first component to **RELOP**, the symbolic code for **token relop .**
2. **nextChar()** → It obtains the next character from the input buffer and advances the forward pointer to next character.
3. **retract()** → It moves the forward pointer one or more position back to reconsider the processed input character which are not part of lexeme
4. **Fail()** → It resets the forward pointer to lexmebegin and activates the next transition diagram to be applied to the true beginning of the unprocessed input. It might also change the **STATE** value to be the **START STATE** for next transition diagram. If there are no transition diagram to be applied, it can initiate the error recovery routine to repair the input to find a lexeme.

❖ Implementation of lexical analyzer for relational operator:

```
// it is assumed that the current state is stored in variable state
TOKEN getRelop ()
{
   // initialization of Token's first component to be RELOP
   TOKEN retToken = new(RELOP)
   while(1)          // repeat character processing until a return or failure occurs
   {
```

```
switch (State){
case 0:    C = nextchar( );
           if (C ==‘<’) state = 1;
           else if (C== ‘=’) state = 5;
           else if (C== ‘>’) state = 6;
           else fail();
           break;
case 1:    C = nextchar( );
           if( C==‘=’ ) state = 2;
           else if (C==‘>’) state = 3;
           else  state =4 ;
           break;
case 2:    retToken.attrtibute = LE;
           return(retToken)
           break;

case 3:    retToken.attrtibute = NE;
           return(retToken)
           break;
case 4:    retract();
           retToken.attribute = LT;
           return(retToken)
case 5:    retToken.attribute = EQ;
           return(retToken);
case 6: ----------------
---------
case 8:    retract();
           retToken.attribute = GT;
           return(retToken)
   } // end of switch

}// end of while

 }// end of function
```

❖ Recognition of Identifiers and keywords:

- Usually all the keywords like **if, then else** etc… are not identifiers even though they look like identifiers and the transition diagrams used for identifying the identifiers will also recognize all the keywords .
- Hence recognition of keywords and identifiers creates a problem. This can be handled in two different ways.

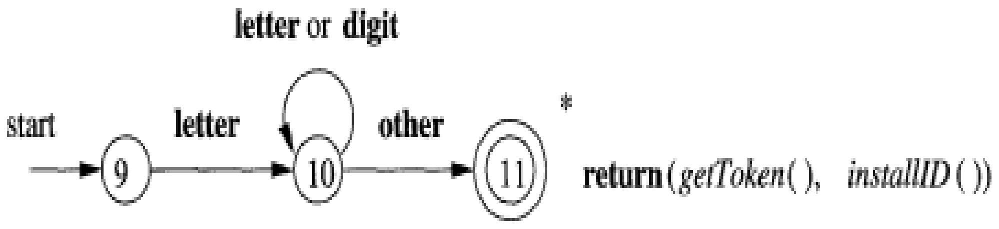


Figure 3.14: A transition diagram for **id**'s and keywords

1. Installation of all the keywords in the symbol table.

- Here the field of symbol table entry indicates that these strings are keywords and never ordinary identifiers and also tells which token they represent. When the lexme is recognized we can use proper mechanism to differentiate the keywords and identifier tokens.
- To do that we execute the following two functions
- InstallId():When the identifier lexme is found this function is invoked that checks the symbol table for the lexme for the following conditions.
- If found and marked as keyword then it returns **ZERO.**
- If found and is a program variable then it returns a **pointer to the symbol table entry.**
- If not found then it is installed as a variable and returns **a pointer to the newly created entry**
- getToken() : When this function is invoked it examines the symbol table and returns the token type that is either ID or keyword itself.

2. Separate transition diagram for each keywords:

- Here every keywords gets a transition diagram that increases the number of states and is tedious to write a lexical analyzer.
- Also we prioritize the tokens so that the keywords are recognized in precedence to Identifier.

❖ Implementation lexical analyzer for identifier:

```
TOKEN getID ()
{
    TOKEN retToken
    while(1)                    // repeat character processing until a return or failure occurs
    {
           switch (State){
           case 9:    C = nextchar( );
                           if isletter (C) state = 10;
                      else fail();
                      break;
           case 10:   C = nextchar( );
                           if  isletter ( C ) || isdigit ( C ) state = 10;
                      else  state =11 ;
                      break;
           case 11:   retract ();
                      retToken = getToken();
                      retToken.attrtibute = install_ID ();
                      return(retToken)
                      break;
            } // End of Switch
      } // End of while
}
```

```
int start =0, state=0;
int fail()
{
    forward = lexme_beginning;
     switch( start )
     {
           case 0   :  start =9; break;
           case 9   :  start =12; break;
           case 12  :  start =22; break;
           case 22 :  recover();
           default  :   // compiler error
     }
}
```

❖ Implementation lexical analyzer for unsigned numbers:

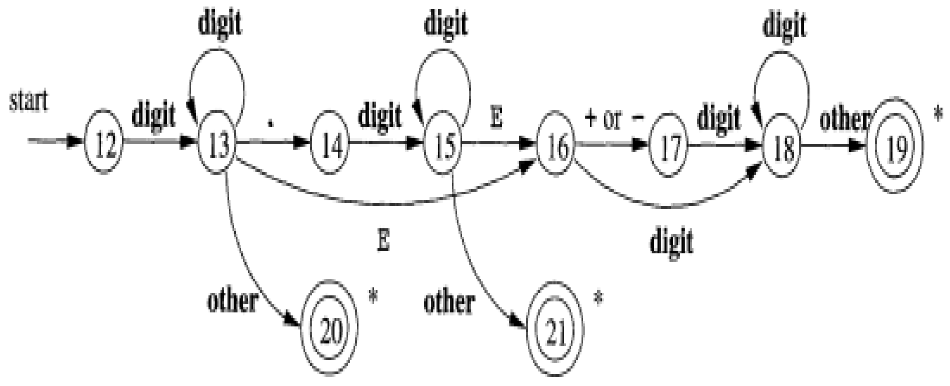


Figure 3.16: A transition diagram for unsigned numbers

```
//lexical analyzer
 // it is assumed that the current state is stored in variable state
 TOKEN getNumber ()
 {
    TOKEN retToken
     while(1)           // repeat character processing until a return or failure occurs
     {
          switch (State){
          case 12:  C = nextchar( );
                    if ( isdigit( C ) ) state = 13;
                    else fail();
                    break;
          case 13:    C = nextchar( );
                    if( isdigit( C ) ) state = 13;
                    else if (C=='.') state = 14;
                      else if (C=='E') state = 16;
                    else  state =20 ;
                    break;
          case 14:    ...
               ...
               ...
         case 20:    retract();
                    retToken.attribute = install_num();
                    return(retToken)
         case 21:   retract();
```

```
                    retToken.attribute = install_num();
                    return(retToken)


            } // end of switch

        }// end of while

         }// end of function
```

*****