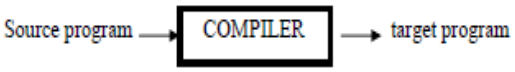
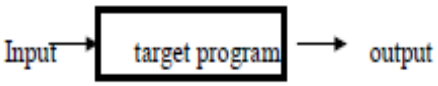
INTRODUCTION, LEXICAL ANALYSIS

1.1 Language Processors:

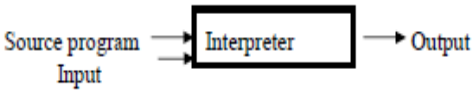
- ***Compilers:***
  - A compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language
  - An important role of the compiler is to report any errors in the source program that it detects during the translation process.



  - If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs



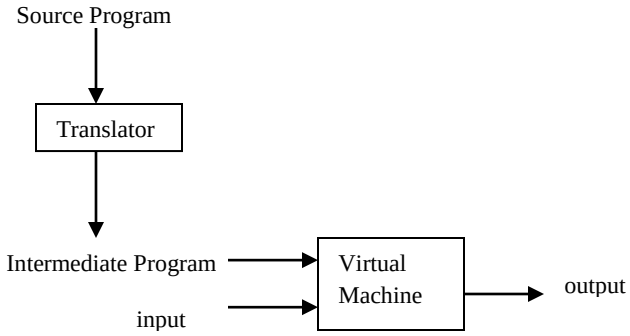
- ***Interpreters:***
  - An *interpreter* is another kind of language processor which directly execute the operations specified in the source program on inputs supplied by the user



Note:

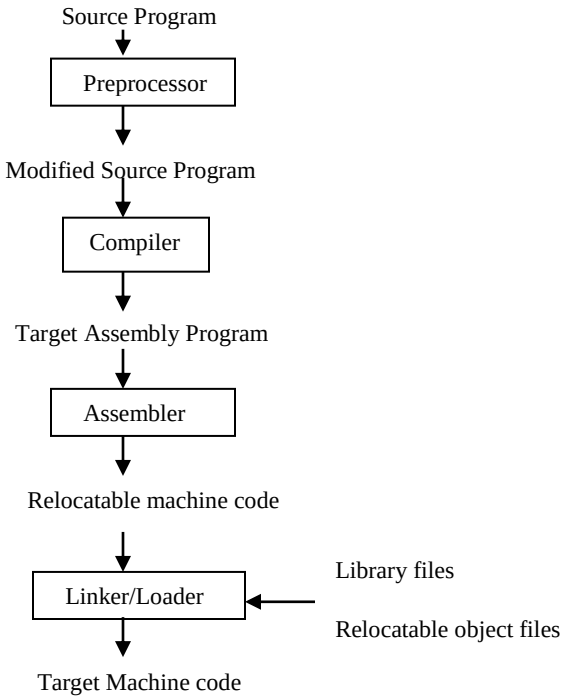
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An interpreter, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

- ***Hybrid Compilers:***
  - Ex: Java Language processors: It combines compilation and interpretation,
    - ❖ A Java source program first compiled into an intermediate form called *bytecodes.*
    - ❖ The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.



  - In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

- Along with compiler, other language processors are required to create an executable target program. These are: ***Preprcessors, assembler, linker*** and ***loader***. The language processing system is shown below.



A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor.*

- The preprocessor may also expand shorthands, called macros, into source language statements.
- The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.
- The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.
- The relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file.
- The *loader* then puts together all of the executable object files into memory for execution.

1.2 The Structure of a Compiler:

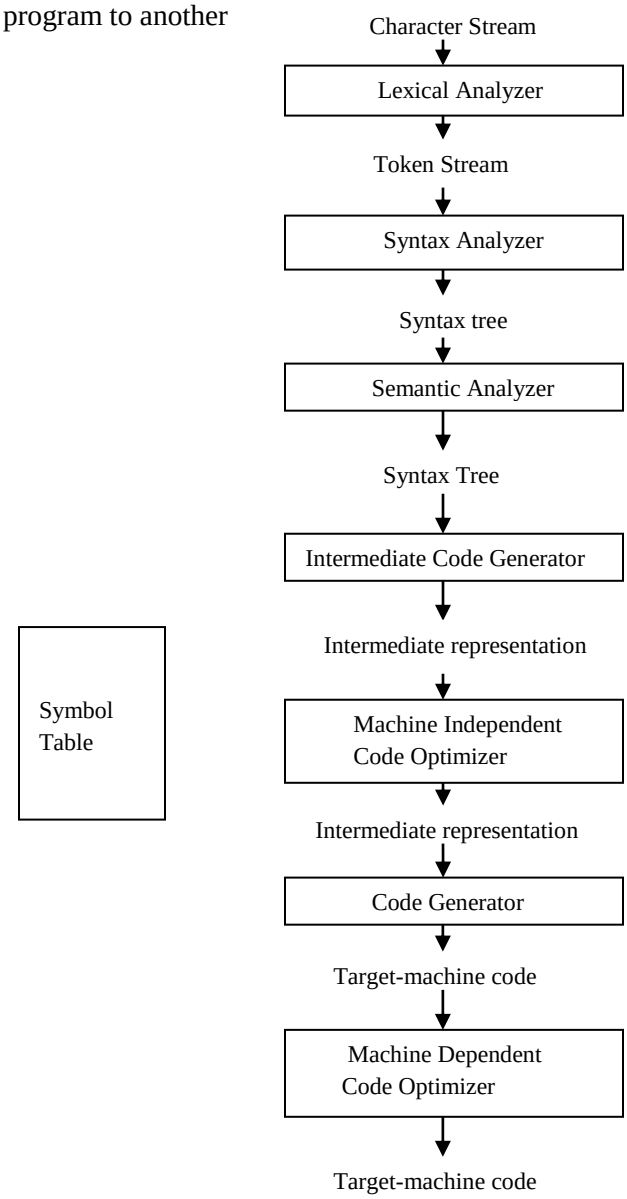
- There are two major parts of a compiler: **Analysis(or front end)** and **Synthesis(or back end)**

- 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

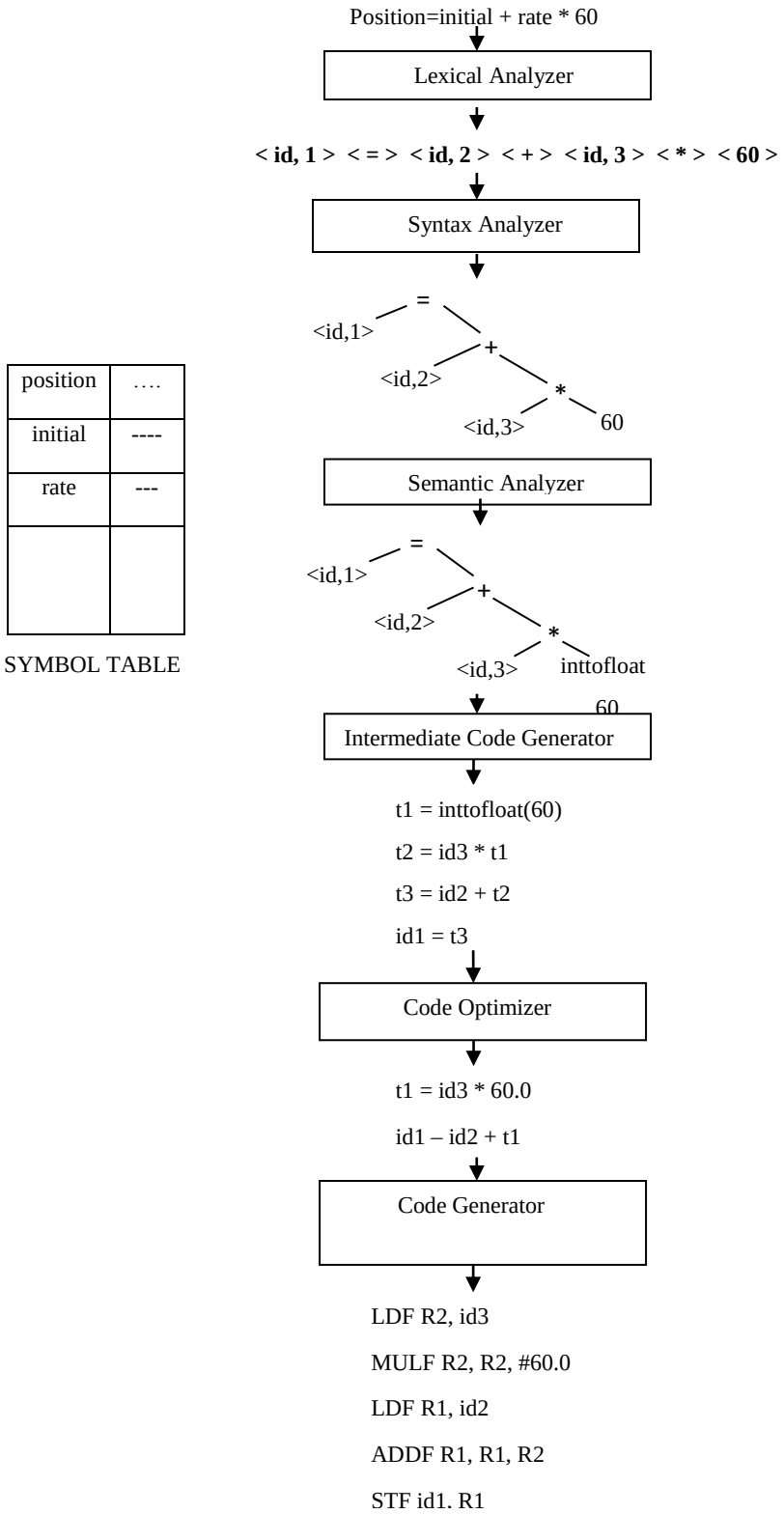
- The analysis part also collects information about the source program and stores it in a data structure called a *symbol table,* which is passed along with the intermediate representation to the synthesis part.
- The *synthesis* part constructs the desired target program from the intermediate representation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

➢ ***Compilation process:***

- It operates as a sequence of *phases,* each of which transforms one representation of the source program to another



Ex: Consider an assignment statement, **position = initial + rate * 60 ,** where all the variables are real, Then the translation of this statement is shown in the following diagram:



Lexical Analysis (Also called as Scanning):

- This is the first phase of compiler.
- The lexical analyzer reads the stream of characters of source program and groups the characters into meaningful sequences called ***lexemes***.
- It produces o/p as ***token*** of the form: **< *token-name, attribute- value* >**
  - ***token-name*** is an abstract symbol that is used during syntax analysis.
  - ***attribute-value*** points to an entry in the symbol table for this token.
- For the above example, the following table shows lexemes and their corresponding tokens.

Lexeme	token-name	attribute-value
position	id	1(refers to symbol table entry)
=	=	---
initial	id	2(refers to symbol table entry)
+	+	---
rate	id	3(refers to symbol table entry)
*	*	---
60	60	---

- Thus, after lexical analysis, the sequence of token generated is:
  < id, 1 > < = > < id, 2 > < + > < id, 3 > < * > < 60 >

Syntax Analysis (Also called as Parsing):

- This is the second phase of the compiler.
- It uses token stream generated 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 generated after parsing for the example is shown in the diagram.

Semantic Analysis:

- This is the third phase of the compiler.
- 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 generatuion.
- An important part of semantic analysis is ***type checking***, where the compiler checks that each operator has matching operands.
  In some cases, it should also support **coercions**.(i.e, type casting int to float etc.).
- The output of semantic analysis for the example is shown in the diagram.

Intermediate Code Generation:

- A compiler may produce an explicit intermediate code representing the source program.

- There exist different forms representing intermediate code. One of the form is **three-address code.**
- A three address code consists of a sequence of assembly-like instructions with three operands per instruction.
  It has at the max, only one operator on the right side.
  Temporary names have to be generated by the compiler in order to hold the value computed by a three-address instruction.
- For the assignment statement example, intermediate code generated in three-address code form is shown in the diagram.

Machine independent code optimization:

- This phase is an intermediate phase between front end and back end.
- The purpose of this phase is to perform transformations on the intermediate representation so that the back end can produce a better target program than it would have otherwise produced from an un-optimized intermediate representation.
- i.e, this phase attempts to improve the intermediate code so that better target code will result.
- For the assignment statement example, the code generated after this phase is shown in the diagram., where **intofloat** operation is eliminated by replacing the integer 60 by the floating point number 60.0.

Code Generation:

- 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.
- For the assignment statement example, the target code generated is shown in the diagram.

Symbol Table Management:

- The symbol table is a data structure containing a record for each variable name used in the source program.
- This stores attributes of each name
  o Eg: name, its type, its scope
  o Method of passing each argument(by value or by reference)
  o Return type

Grouping of phases into passes:

- In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- For example,

- ❖ front-end phases of lexical analysis, syntax analysis, semantic analysis and from back end intermediate code generation might be grouped together into one pass.
- ❖ Code optimization might be an optional pass.
- ❖ There could be a back-end pass consisting of code generation for a particular target machine.

Compiler construction Tools:

Some commonly used compiler construction tools are:

1. **Parser generators**: These automatically produce syntax analyzers from a grammatical description of a programming language.

2. **Scanner generators**: These produce lexical analyzers from a regular-expression description of the tokens of a language.

3. **Syntax-directed translation engines**: These produce collections of routines for walking a parse tree and generating intermediate code.

4. **Code-generator generators**: These produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. **Data-flow analysis engines**: These facilitate the gathering of information about how values are transmitted from one part of a program to each other.

6. **Compiler-construction toolkits**: these provide an integrated set of routines for constructing various phases of a compiler.

1.3 The Evolution of Programming Languages:

The Move to Higher-Level Languages:

Classification of Languages:

1. Based on generation:
   a) First generation language-machine languages.
   b) Second generation languages-assembly languages.
   c) Third generation languages-higher level languages, ex: Fortran, Cobol, Lisp, C, C++, C#, Java
   d) Fourth generation languages-designed for specific applications like SQL for database applications, Postscript for text formatting
   e) Fifth generation languages-applied to logic and constraint based languages like prolog and OPS5.

2. Imperative Vs declarative languages:
   Imperative languages : these are the languages in which a program specifies how a computation is to be done.

eg: C, C++,C#,JAVA

Declarative languages: These are the languages in which a program specifies what computation is to be done.
Eg: ML, Haskel,Prolog

3. Von Neumann language: These are the languages whose computational model is based on the Von Neumann architecture
   Eg: Fortran, C etc

4. Object Oriented Language: Languages which support object-oriented programming style.
   Eg: C++, JAVA, C#, Ruby

5. Scripting Languages: Languages that makes use of high level operators to perform computations.
   Eg: JavaScript, Perl, PHP, Python, Ruby, Tcl

Impacts on Compilers:

- As new architectures and programming languages were evolving, the compiler writers had to track new language features and had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.
- A compiler by itself is a large program that must translate correctly the potentially infinite set of programs that could be written in the source language.
- A compiler writer must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

1.4 The science of Building a Compiler:

- The science behind the compiler:
  1) Take a problem
  2) Formulate a mathematical abstraction that captures the key characteristics- requires solid understanding of the characteristics of computer programs.
  3) Solve it using mathematical techniques.

- A compiler must accept all source programs that conform to the specification of the language.
- Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled.

Modeling in compiler design and Implementation:

- Some of the models used are:
  1) Finite state machines and regular expressions: useful
     - ❖ For describing the lexical units of programs
     - ❖ For describing the algorithms used by the compiler to recognize those units.
  2) Context-Free-Grammars:
     Used to describe the syntactic structure of programming languages
  3) Trees:
     Used for representing the structure of programs and their translation into object code.

The Science of code optimization:

- It refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.
- The optimization of code that a compiler performs has become both more important and more complex.
  - ❖ More complex: Because processor architectures have become more complex.
  - ❖ More important: Because most of the parallel computers require optimization, o.w., performance degrades.
- Compiler optimizations must meet the following design objectives:
  1. The optimization must be correct, i.e., preserve the meaning of the compiled program.
  2. It must improve the performance of many programs.-refers to shorter code, faster execution of the program, minimum power consumption.
  3. The compilation time must be kept reasonable- refers to short compilation time in order to support a rapid development and debugging cycle.
  4. The engineering effort required must be manageable.- keep the system simple to assure that the engineering and maintenance cost of the compiler are manageable.

1.5 Applications of Compiler Technology:

Implementation of High Level Programming languages:

- A high-level programming language defines a programming abstraction:
  The programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.
- Generally High level Programming languages are easier to program in, but are less efficient, i.e., the target program runs more slowly.
- Programmers using Low level programming Language have more control over a computation and can produce more efficient code.
- Unfortunately, Low level programs are harder to write and still worse less portable, more prone to errors and harder to maintain.
- Optimizing compilers include techniques to improve the performance of general code, thus offsetting the inefficiency introduced by HL abstractions.
- Ex: usage of **register** keyword in the earlier C Programming language:
  As effective register-allocation techniques were developed, this keyword lost its efficiency.
- Programming languages: C,Fortran
  These support user defined aggregate data types(eg:arrays, structures) and high level control flow(loops, procedures).
  A component 'data-flow optimizations' analyzes the flow of data through the program and removes redundancies across these constructs.
- Object oriented programming languages: C++,C#,JAVA
  These supports:
  Data abstraction and Inheritance properties.
  Optimizations to speed up virtual method dispatches have also been developed.

Optimizations for Computer Architectures

- The rapid evolution of computer architecture has also led to an insatiable demand for a new compiler technology.
- Almost all high performance systems take advantage of the same basic 2 techniques:
  parallelism and ***memory hierarchies***.\

- Parallelism can be found at several levels :
  at the instruction level- where multiple operations are executed simultaneously
  at the processor level- where different threads of same application are run on different processors.
- Memory hierarchies are a response to the basic limitation that we can built very fast storage or very large storage, but not storage that is both fast and large.

- **Parallelism:**

  - ❖ All modern microprocessors exploit instruction-level parallelism. this can be hidden from the programmer.
  - ❖ The hardware scheduler dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible.
  - ❖ Whether the hardware reorders the instruction or not, compilers can rearrange the instruction to make instruction-level parallelism more effective.

- **Memory Hierarchies:**

  - ❖ Memory hierarchy consists of several levels of storage with different speeds and sizes.
  - ❖ A processor usually has a small number of ***registers*** consisting of hundred of bytes, several levels of ***caches*** containing kilobytes to megabytes, and finally ***secondary storage*** that contains gigabytes and beyond.
  - ❖ Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude.
  - ❖ The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem.
  - ❖ While compliers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Design of New Computer Architectures.

- In modern computer architecture development, compilers are developed in the processor design stage, and compiled code running on simulators, is used to evaluate the proposed architectural design.
- One of the best known example of how compilers influenced the design of computer architecture was the invention of **RISC (reduced instruction set computer)** architecture.
- Over the last 3 decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW(very long instruction word) machines, multiprocessors with shared memory, and with distributed memory.
- The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.
- Compiler technology is not only needed to support programming of these architectures but also to evaluate the proposed architectural designs.

Program Translations:

- Normally we think of compiling as a translation of a high level Lang to machine level Language, the same technology can be applied to translate between different kinds of languages.
- The following are some of the important applications of program translation techniques:
- **Binary Translation:**
  - ❖ Compiler technology can be used to translate the binary code for one machine to another.
  - ❖ Eg:Binary translators have been developed to convert X86 code into both Alpha and Spare Code.

- ❖ Binary translation can also be used to provide backward compatibility
  Eg: PowerPC processors were allowed to run legacy MC 68040 code.

- **Hardware Synthesis:**
  - ❖ Compiler technology is also used in high level hardware description languages like **VHDL**.
- **Database Query Interpreters:**
  - ❖ Compiler technology is also used in query languages such as **SQL** which are used to search databases**.**
- **Compiled Simulation:**
  - ❖ It can run orders of magnitude faster than an interpreter based approach, which are used mainly in scientific and engineering disciplines to understand the phenomenon or to validate a design.

Software Productivity Tools:

- Several ways in which program analysis, building techniques originally developed to optimize code in compilers, have improved software productivity.

- **Type Checking**:
  - ❖ It is an effective technique to catch inconsistencies in programs.
  - ❖ It can be used to catch errors
    Ex: Type mismatch of the object, parameter type mismatches with the procedure signature etc.
  - ❖ Through Program analysis i.e., by analyzing the flow of data through program, errors can be detected.
    Ex: usage of null pointer
  - ❖ This technique can also be used to catch a variety of security holes.
    Ex: If the attacker supplies "dangerous" string, and if this string is not checked properly and there are chances that this string would influence the control flow of the code at some point in the program.

- **Bounds Checking:**
  - ❖ This technique mainly used when buffer overflows occur in the program.
  - ❖ Ex: C programs doesn't support array bound checks. It is upto the user to ensure that arrays are not accessed out of bounds. Failing to check this, the program may store the data outside this buffer.
  - ❖ There are several other techniques that would perform the similar job.
    Ex: Data flow analysis

- **Memory Management Tools**:
  - ❖ There are several tools that deal with checking memory management issues.
  - ❖ "Garbage collection" is the main issue to be seen in memory management.
  - ❖ Ex: "Purify" is a tool that dynamically catches memory management errors as they occur.

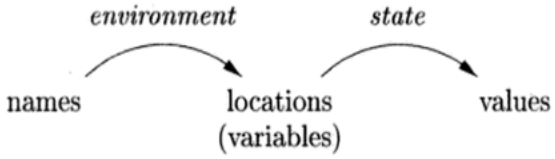
1.6 Programming Language Basics

- **Static/Dynamic Distinction**
  - ❖ If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a ***static policy*** or that the issue can be decided at ***compile time***.

- If a language uses a policy that allows a decision to be made when we execute the program, then we say that the language uses a ***dynamic policy*** or that the issue can be decided at ***run time***.
- Ex: public static int x; // static policy only one location is held for the usage by all the objects.

  Public int y; // dynamic policy. Cannot determine as multiple objects having different copies of 'y'

- **Environments and States**
  - The issue is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data.
  - The association of names with locations in memory(the store) and then with values can be described by two mappings that change as the program runs:



  1) The *environment* is a mapping from names to locations in the store.
  2) The *state* is a mapping from locations in store to their values.

  - The environment and state mapping are dynamic with few exceptions:
    1) Static Vs dynamic binding of names to locations:
       Most of these bindings are dynamic except for the global declaration of variables.

    2) Static Vs dynamic binding of locations to values:
       Most of these bindings are dynamic except for the declared constants.

  - Environment change according to the scope rules of a language.

  - Ex:

```
……..
int i;                          // global i
……..
void f(…..)
{
        int i;                  // local i;
        ……
        i = 3;                  // use of local i
        ……
}
……
        x = i + 1;              // use of global i
```

- **Static Scope and Block Structure**

  - The C static-scope policy is as follows:
    1) A C program consists of a sequence of top-level declarations of variables and functions.

2) Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.
3) The scope of a top-level declaration of a name *x* consists of the entire program that follows with the exception of those statements that lie within a function that also has a declaration of *x*

❖ The syntax of blocks in C is given by
1) One type of statement is a block. Blocks can appear anywhere that other types of statements such as assignment statements can appear.
2) A block is a sequence of declarations followed by a sequence of statements all surrounded by braces.

This syntax allows blocks to be nested inside each other. This nesting property is referred to as ***block structure.***

❖ We say that a declaration D "belongs" to a block B if B is the most closely nested block containing D; i.e., D is located within B, but not within any block that is nested within B.

❖ The static scope rule for variable declarations in a block-structured languages is as follows. : If declarations *D* of name *x* belongs to block *B*, then the scope of *D* is all of *B*, except for any blocks *B*` nested to any depth within *B*, in which is *x* is redeclared.

❖ Ex: consider the following C++program :

```
main() {
int a = 1;                          B1
int b = 1;
{
    int b = 2;                      B2
    {
        int a = 3;                  B3
        cout << a << b;
    }
    {
        int b = 4;                  B4
        cout << a << b;
    }
    cout << a << b;

}
cout << a << b;

}
```

Scope of declarations:

Declaration	Scope
int a = 1;	B1-B3
int b = 1;	B1-B2
int b = 2;	B2-B4
int a = 3;	B3
int b = 4;	B4

- **Explicit Access Control**
  - ❖ *Classes* and *structures* introduce a new scope for their members.

- ❖ The scope of a member declaration *x* in class C extends to any subclass C' except if C' has a local declaration of the same name *x*.
- ❖ Through the use of keywords like ***public, private*** and ***protected***, C++/Java provide explicit control over access to member names in superclass.

- **Dynamic Scope**
  - ❖ The term dynamic scope refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration.
  - ❖ **Example 1: Macro expansion in the C preprocessor**:

```
#define a (x+1)
int x=2;
void b( ) { int x=1; printf("%d\n",a); } // uses local definition of i
void c( ) { printf("%d\n",a); } //
void main( ) { b( ); c( ); }
```

Output:

2

3

  - ❖ **Example 2: Method resolution in the Object Oriented Programming**
    - In OOP, each object has the ability to invoke the appropriate method in response to a message.
    - In other words, the procedure called when *x.m* is executed depends on the class of the object denoted by *x* at that time.
    - Ex:
      1) There is a class *C* with a method named *m()*.
      2) *D* is a subclass of *C* and *D* has its own method named *m()*.
      3) There is a use of *m* of the form *x.m()* where x is an object of class *C*.

      Here which definition of *m()* needs to be called can be determined at run time only.

- **Parameter Passing Mechanisms:**

  Actual parameters : parameters used in the call of a procedure
  Formal parameters : parameters used in the procedure definition

  - ❖ **Call-by-value:**
    - Here, all computation involving the formal parameters done by the called procedure is local to that procedure.
    - Hence, the actual parameters themselves can not be changed.
    - However, there is an exception when pointer / array are passed which changes the content of the actual parameter.
      Eg: if *a* is an array passed by value to the formal parameter *x*, then any changes made to array *x* will change the corresponding elements of array *a*.
  - ❖ **Call-by-Reference:**
    - Here, address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
    - Formal parameter is implemented as pointer to this location.
    - Hence, any changes made to the formal parameter changes this location.
  - ❖ **Call-by-name:**
    - This mechanism was used in the earlier programming.
    - It was a kind of macro expansion.

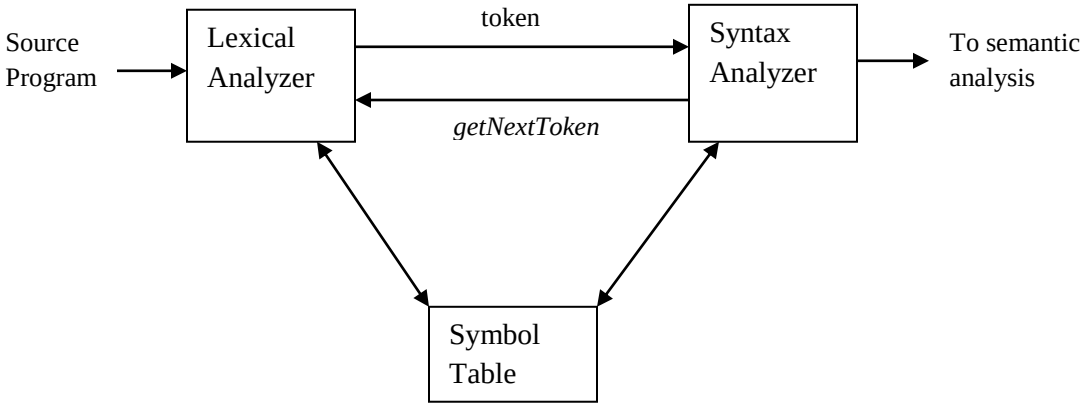
- It became unfavorable as it did not support actual parameters containing expressions.

❖ **Aliasing:**
- It is possible that two formal parameters can refer to the same location; such variables are said to be ***aliases*** of one another.

Lexical Analysis:

2.1 The role of the Lexical Analyzer:

- Lexical Analyzer reads the source program character by character, group them into lexemes and produce sequence of tokens for each lexeme in the source program.
- These tokens are sent to syntax analyzer.
- The interaction between lexical analyzer and the parser is shown below:



- Here parser makes a call using *getNextToken* command, which 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.
- The lexical analyzer also interacts with the symbol table, wherein newly identified identifier lexeme is written also information from the table can be read .
- Some Other tasks performed by the Lexical Analyzer:
  - ***Stripping out comments and whitespace***
    Normally Lexical analyzer doesn't return a comment as a token.
    It skips a comment, and return the next token (which is not a comment) to the parser.
  - ***Correlating error messages***
    It can associate a line number with each error message..
    In some compilers it makes a copy of the source program with the error messages inserted at the appropriate positions.
    If the source program uses **macro-processor**, the expansion of macros may also be performed by the Lexical analyzer
- Sometimes, lexical analyzers are divided into a cascade of two processes:
  - ***Scanning***: This is the simple process that do not require tokenization of the input, such as deletion of comments and stripping out whitespace
  - ***Lexical analysis:*** This is the complex process, wherein the scanner produces a sequence of tokens.

- **Lexical Analysis Versus Parsing:**

The reasons for separating analysis phase of the compilation into lexical analysis and syntax analysis:

1. ***Simplicity in the design :***
   - The tasks to be performed made easier
   - It leads to cleaner overall language design.

2. ***Compiler efficiency improved:***
   - Specialized techniques such as ***input buffering*** can speed up the compiler significantly.

3. ***Compiler portability enhanced:***
   - Input device specific peculiarities can be restricted only to the lexical analyzer, hence the other phases are free from these limitations.

- **Tokens,Patterns,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 units.
- Ex: Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas, semicolons)

Pattern:
- Description of the form that the lexemes of a token may take.

Lexeme:
- It is a sequence of characters in the source program that matches the pattern for a token

Ex 1:

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	3.14, 0.6, 20
Literal	Anything but surrounded by " " 's	"total= %d\n" , "core dumped"

Ex 2:
Consider the following statement:

```
if(a>=b)
    printf("total=%d\",a);
```

Lexemes: if (a >= b) printf ("total=%d" a)
Tokens : **IF LP id relop id RP id LP literal id RP**

- Most of the token belongs to the following classes:
  1) One token for each keyword.
  2) Tokens for the operators either individually or in classes
  3) One token representing all identifiers
  4) One or more tokens representing constants such as numbers and literal strings
  5) Tokens for each punctuation symbol such as left and right parentheses, comma and semicolon.

- **Attributes for tokens:**

  - The lexical analyzer returns to the parser not only a token name but an attribute value that describes the lexeme presented by the token.
  - These attribute information is kept in the symbol table for the future reference.
  - Ex: for a token **id,** the attribute information includes its lexeme, its type, the location at which it is first found etc. Thus, the appropriate attribute value here would be pointer to the symbol table entry.
  - Ex: the token names and associated attribute values for the Fortran statement E=M * C ** 2 are:

    < **id**, pointer to symbol table entry for E >
    <**assign_op**>
    < **id**, pointer to symbol table entry for M >
    <**mult_op**>
    < **id**, pointer to symbol table entry for C >
    <**exp_op**>
    < **number**, integer value 2>

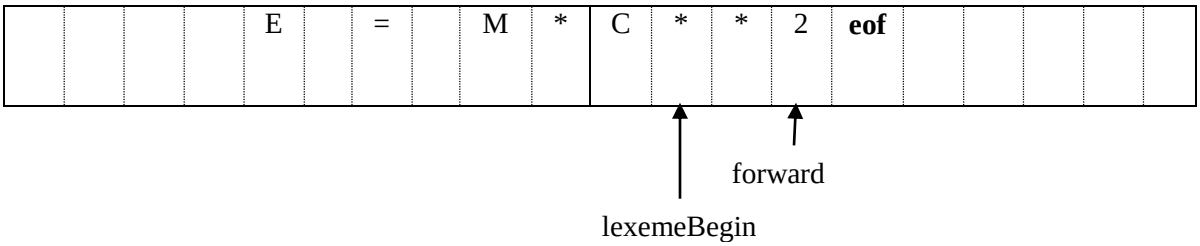
  - ***Tricky problem when recognizing tokens***:
    Ex: In FORTRAN,
    DO 5 I = 1.25 // DO5I is a lexeme
    DO 5 I = 1, 25 // do- statement

- **Lexical Errors:**

  - Instead of *if(a==b)* statement if we mistype it as *fi(a==b)* then lexical analyzer will not rectify this mistake as it treats *fi* as a valid lexeme.
  - In certain situations lexical analyzer cannot proceed further as there are no matching patterns for the remaining input. In this case it can adopt ***panic mode error recovery strategy***.
    The possible actions here would be:
    - 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.
    - Delete one character from the remaining input.
    - Insert a missing character into the remaining input.
    - Replace a character by another character.
    - Transpose two adjacent characters.

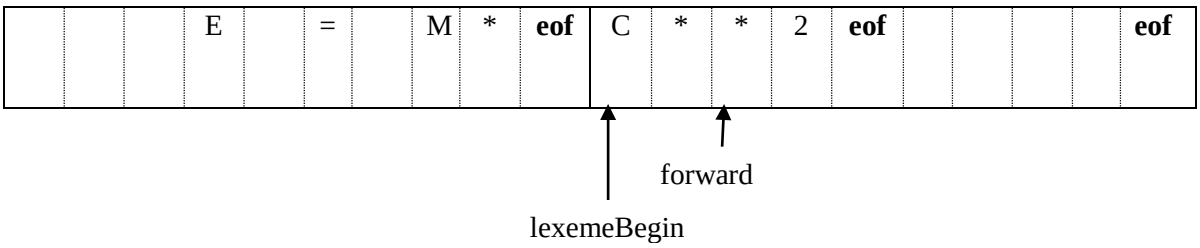
2.2 Input Buffering:

- To recognize tokens, atleast one extra character has to be read which incurs certain overhead. To minimize this overhead and thereby to speed up the reading process special buffer technique have been developed .
- One such technique is **two-buffer** scheme in which two buffers are alternately reloaded.:



- **Buffer Pairs:**
  - Size of each buffer is *N*(size of disk block) Ex:4096 bytes.
  - System read command is used to read N characters into a buffer.
  - If fewer than N characters remain in the input file , then a special character, represented by **eof**, marks the end of source file.
  - Two pointers to the input are maintained.
    1) 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 if found.
  - Once the next lexeme is determined, forward is set to the character at its right end.
  - Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *lexemeBegin* is set to the character immediately after the lexeme just found.
  - Advancing forward requires that we first test whether we have reached the end of one of the buffers and if so we must reload the other buffer from the input and move forward to the beginning of the newly loaded buffer.

- **Sentinel:**
  - For each character read, we make two tests:
    1) For the end of the buffer
    2) To determine what character is read.
  - Here, buffer end test can be done by using ***sentinel*** character at the end of the buffer.
  - Sentinel is a special character that cannot be a part of source program. **eof** is used as sentinel.
  - The arrangement is shown below:



  - The following algorithm summarizes advancing *forward* pointer:
    (lookahead code with sentinels)

```
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
}
```

2.3 Specification of tokens:

- Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.
- **Strings and Languages:**

  - An **alphabet** is any finite set of symbols such as letters, digits, and punctuation.
    - The set {0,1) is the binary alphabet
    - ACII character set is an alphabet

  - A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
    - |s| represents the length of a string s, Ex: **college** is a string of length 7
    - The empty string **ε**, is the string of length zero.
    - The empty string is the identity under concatenation; that is, for any string s, εS = Sε = s.
    - If x and y are strings, then the concatenation of x and y is also string, denoted xy, For example, if x = hello and y = world, then xy = helloworld.
  -

    The following string-related terms are commonly used:

    1. A *prefix* of string *s* is any string obtained by removing zero or more symbols from the end of *s*. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.
    2. A *suffix* of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.
    3. A *substring* of *s* is obtained by deleting any prefix and any suffix from *s*. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.
    4. The *proper* prefixes, suffixes, and substrings of a string *s* are those, prefixes, suffixes, and substrings, respectively, of *s* that are not ϵ or not equal to *s* itself.
    5. A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*. For example, **baan** is a subsequence of **banana**.

  - A **language** is any countable set of strings over some fixed alphabet.

- Abstract languages like Φ, the empty set, or {ε},the set containing only the empty string, are languages under this definition.

- **Operations on Languages:**
  - operations performed on languages is shown below:

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

  - Example:
    Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z } and let D be the set of digits {0,1,.. .9}.
    Other languages can be constructed from L and D, using the operators illustrated above :

    1. L U D is the set of letters and digits - strictly speaking the language with 62 (52+10) strings of length one, each of which strings is either one letter or one digit.
    2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.(10×52).
       Ex: A1, a1,B0,etc
    3. L^4 is the set of all 4-letter strings. (ex: aaba, bcef)
    4. L^* is the set of all strings of letters, including ε, the empty string.
    5. $L(L \text{ U } D)^*$ is the set of all strings of letters and digits beginning with a letter.
    6. D^+ is the set of all strings of one or more digits.

- **Regular Expressions**

  - Regular Expressions can be defined as follows:

    Basis:
    1) If **ε** is a R.E., then L(**ε**)={ **ε**}
    2) If L(a)={a}, then *a* is a R.E.

    Induction:
    Larger regular expressions are built from smaller ones. Let **r** and **s** are regular expressions denoting languages L(r) and L(s), respectively.

    1. (r) | (s) is a regular expression denoting the language L(r) U L(s).
    2. (r) (s) is a regular expression denoting the language L(r) L(s) .
    3. (r) * is a regular expression denoting (L (r)) * .
    4. (r) is a regular expression denoting L(r). This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.
       For example, we may replace the regular expression (a) | ((b) * (c)) by a| b*c.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression **a|b** denotes the language $\{a, b\}$.
2. **(a|b)(a|b)** denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is **aa|ab|ba|bb**.
3. **a*** denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. **(a|b)*** denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is **(a*b*)***.
5. **a|a*b** denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

- **Regular Definition:**

  - To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
  - We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.
  - A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

  - Ex. 1: C identifiers are strings of letters, digits, and underscores. The regular definition for the language of C identifiers:

Letter → A | B | *C*|...| Z | a | b | … |z| -
digit → *0|1|2* |... | *9*
id → *letter*(*letter* | *digit*)*

Ex. 2: Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition is

digit → *0|1|2* |... | *9*
digits → *digit digit**
optionalFraction → *.digits* | ε
optionalExponent → (E(+ |- | ε) *digits*) | ε
number → *digits optionalFraction optionalExponent*

2.4 Recognition of Tokens:

- Our current goal is to perform the lexical analysis needed for the following grammar.

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | ε
expr → term relop term          // relop is relational operator =, >, etc
     | term
```

term → id
| number

- The **terminals** of the grammar, which are **if, then**, **else**, **relop**, **id**, and **number**, are the names of tokens as used by the lexical analyzer.
- The pattern description using regular definition for the terminals is

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

- The lexical analyzer also has the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

ws → (blank | tab | newline) +

- Tokens, and attribute values for the above grammar is:

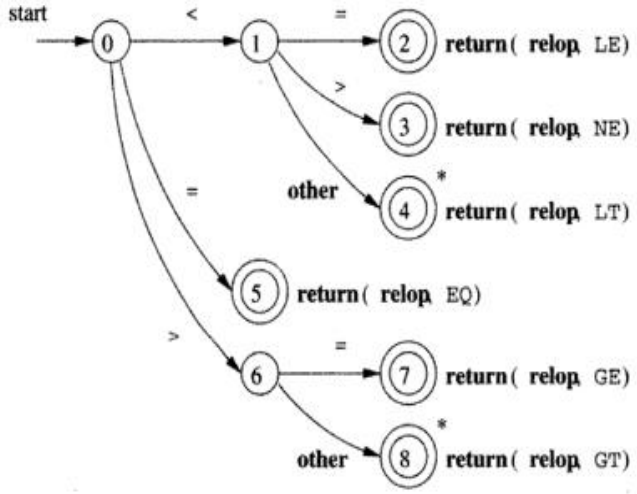
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

- Note:
  - The lexer will be called by the parser when the latter needs a new token. If the lexer then recognizes the token ***ws***, it does *not* return it to the parser but instead goes on to recognize the next token, which is then returned.
  - We can't have two consecutive *ws* tokens in the input because, for a given token, the lexer will match the **longest** lexeme starting at the current position that yields this token.
  - For the parser, all the relational operators are to be treated the same so they are all the same token, **relop**.
  - Other parts of the compiler, for example the code generator, will need to distinguish between the various relational operators so that appropriate code is generated. Hence, they have distinct attribute values.

- To recognize tokens there are 2 steps
  1. Design of Transition Diagram
  2. Implementation of Transition Diagram

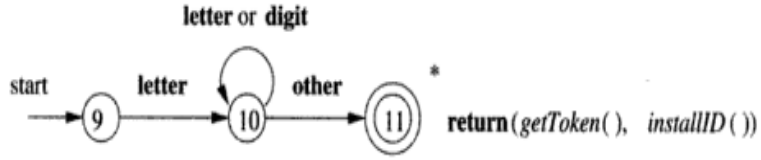
- **Transition diagram**:
  - A transition diagram is similar to a flowchart for (a part of) the lexer. We draw one for each possible token. It shows the decisions that must be made based on the input seen.
  - The two main components are circles representing *states* (think of them as decision points of the lexer) and arrows representing *edges* (think of them as the decisions made).
  - The transition diagram always begins in the start state before any input symbols have been read.
  - The accepting states indicate that a lexeme has been found.
  - Sometimes it is necessary to retract the *forward* pointer one position(i.e., the lexeme does not include the symbol that got us the to the accepting state), then we shall additionally place a * near that accepting state.

- **Transition diagram for the token relop**:



  - It is fairly clear how to write code corresponding to this diagram. look at the first character, if it is <, look at the next character. If that character is =, return (relop,LE) to the parser.
  - If instead that character is >, return (relop,NE).
  - If it is another character, return (relop,LT) and adjust the input buffer so that we will read this character again since this is not used it for the current lexeme.
  - If the first character is =, return (relop,EQ).

  - **Recognition of reserved words and identifiers:**

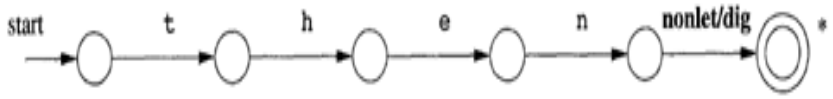


    Note again the star affixed to the final state.
  - Two questions remain.
    1. How do we distinguish between identifiers and keywords such as *then*, which also match the pattern in the transition diagram?
    2. What is (*gettoken(), installID()*)?

  - There are two ways that we can handle reserved words that look like identifiers.
    1. We will continue to assume that the keywords are *reserved*, i.e., may not be used as identifiers.
       - These reserved words should be installed into the symbol table prior to any invocation of the lexer. The separate entry in the table will indicate that the entry is a keyword.

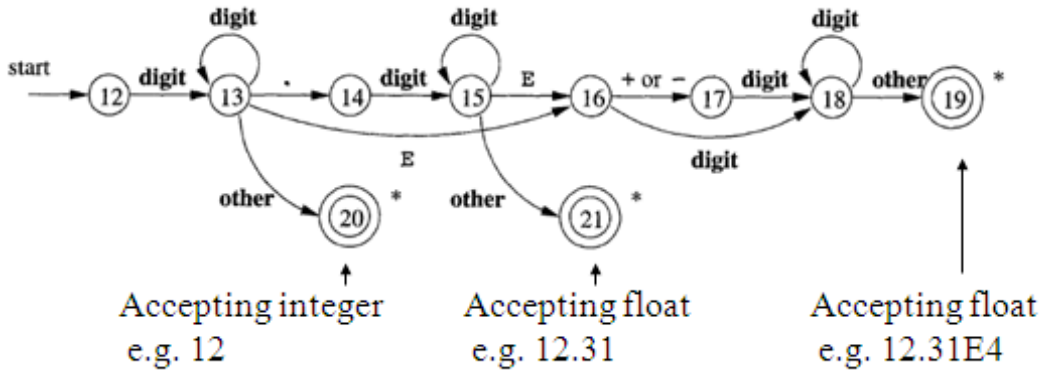
- When we find an identifier, ***installID()*** checks if the lexeme is already in the table. If it is not present, the lexeme is installed as an **id** token. In either case a pointer to the entry is returned.
- ***gettoken()*** examines the lexeme and returns the token name, either id or a name corresponding to a reserved keyword.

2. Create separate transition diagrams for each keyword
   Ex: transition diagram for the keyword **then** is:

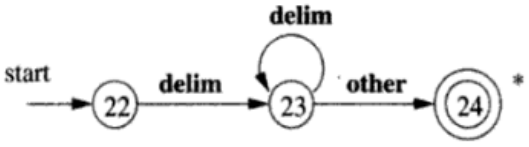


Note: If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to **id**, when lexeme matches both patterns.

- **Recognizing Numbers:**
  - When an accepting state is reached, the identified number is stored in the table in which numbers are stored. Also a pointer to the corresponding lexeme is returned.
  - These numbers are needed when code is generated.
  - Depending on the source language, we may wish to indicate in the table whether this is a real or integer. A similar, but more complicated, transition diagram could be produced if the language permitted complex numbers as well.



- **Recognizing Whitespace:**



  - The **delim** in the diagram represents any of the whitespace characters, say space, tab, and newline.
  - The final star is there because we needed to find a non-whitespace character in order to know when the whitespace ends and this character begins the next token.
  - There is no action performed at the accepting state. Indeed the lexer does *not* return to the parser, but starts again from its beginning as it still must find the next token.

- **Architecture of a Transition-Diagram-Based Lexical Analyzer:**

  - The idea is that we write a piece of code for each transition diagram collectively to build a lexical analyzer.
  - Here, we may imagine a variable ***state*** holding the number of the current state for a transition diagram.
  - A switch based on the value of state takes us to code for each of the possible states, where we find the action of the state.
  - Ex: Implementation of **relop** transition diagram using C++ function is described below:

```
TOKEN getRelop()                    // TOKEN has two components
{
        TOKEN retToken = new(RELOP);        // First component set here
        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();                       // lexeme is not a relop
                                 break;
                        case 1: ...
                        ...
                        case 8:  retract();                         // an accepting state with a star
                                 retToken.attribute = GT;           // second component
                                 return(retToken);
                }
        }
}
```

    - This piece of code contains a case for each state, which typically reads a character and then goes to the next case depending on the character read.
    - The numbers in the circles are the names of the cases.
    - Accepting states often need to take some action and return to the parser. Many of these accepting states (the ones with stars) need to restore one character of input. This is called ***retract()*** in the code.
    - What should the code for a particular diagram do if at one state the character read is not one of those for which a next state has been defined? That is, what if the character read is not the label of any of the outgoing arcs? This means that we have failed to find the token corresponding to this diagram. The code in this case ,calls **fail()**.
        - This is **not** an error case. It simply means that the current input does not match this particular token. So we need to go to the code section for another diagram after restoring the input pointer so that we start the next diagram at the point where this failing diagram **started**.
        - If we have tried all the diagram, then we have a real failure and need to print an error message and perhaps try to repair the input.

- **Alternate Methods to fit the code into the lexical analyzer:**
  1. The order in which the diagrams are tried is important.

One of the ordering could be sequential, If the input matches more than one token, the first one tried will be chosen.
For Example, code must be written first for the keywords followed by the identifiers.

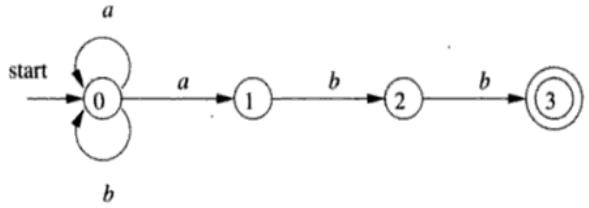
2. Run the various transition diagrams in parallel. That is, each character read is passed to each diagram (that hasn't already failed). Care is needed when one diagram has accepted the input, but others still haven't failed . One strategy would be to accept a longer prefix of the input.
Ex: prefer the identifier *thenext* to keyword *then.*
3. The preferred approach is to combine all the diagrams into one. Because all the diagrams begin with different characters being matched. Hence we just have one large start with multiple outgoing edges. However, in general the problem of combining transition diagrams for several tokens is more complex.

2.5 The Lexical-Analyzer Generator (*Given as assignment topic, Q.No.4*)

2.6 Finite Automata:

- Finite automata are *recognizers;* they simply say "yes" or "no" about each possible input string.
- Finite automata come in two flavors:
(a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ε, the empty string, is a possible label.
(b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.
- Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. The languages accepted by these automata are *regular languages.*

- **Nondeterministic Finite Automata**
  - A *nondeterministic finite automaton* (NFA) consists of:
    1. A finite set of states *S.*
    2. A set of input symbols *Σ*, the *input alphabet.* We assume that *ε*, which stands for the empty string, is never a member of *Σ.*
    3. A *transition function* that gives, for each state, and for each symbol in Σ U{*ε*} a set of *next states.*
    4. A state s_0 from *S* that is distinguished as the *start state* (or *initial state).*
    5. A set of states *F*, a subset of *S*, that is distinguished as the *accepting states* (or *final states).*
  - We can represent either an NFA or DFA by a *transition graph,* where the nodes are states and the labeled edges represent the transition function. There is an edge labeled *a* from state *s* to state *t* if and only if *t* is one of the next states for state *s* and input *a.*
  - Example:The transition graph for an NFA recognizing the language of regular expression **(a|b)*abb** is :



- **Transition Tables**
  - We can also represent an NFA by a *transition table,* whose rows correspond to states, and whose columns correspond to the input symbols and ε.
  - The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put *ϕ* in the table for the pair.
  - Example: The transition table for the above NFA is shown below:

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

- The transition table has the advantage that we can easily find the transitions on a given state and input.
- Its disadvantage is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

- **Acceptance of Input Strings by Automata**

  - An NFA *accepts* input string *x* if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out *x*. Note that ε labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.
  - **Example :** The string *aabb* is accepted by the NFA of above Fig. The path labeled by *aabb* from state 0 to state 3 demonstrate this fact

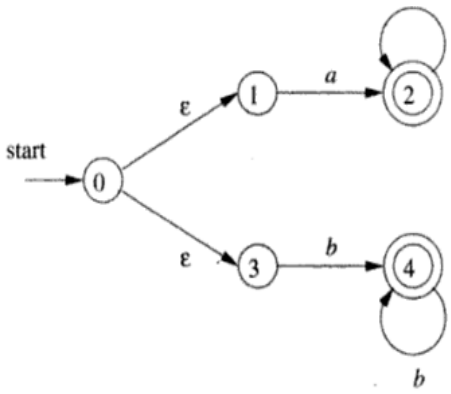
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

  - Note that several paths labeled by the same string may lead to different states. For instance, for the same string *aabb* the path may end at state 0 also.

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

  - The *language defined* (or *accepted)* by an NFA is the set of strings labeling some path from the start to an accepting state.

  - **Example:** The following Figure is an NFA accepting *L*(**aa*|bb*).** String *aaa* is accepted because of the path $0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$



- **Deterministic Finite Automata**
  - A *deterministic finite automaton* (DFA) is a special case of an NFA where:
    1. There are no moves on input ε, and
    2. For each state *s* and input symbol *a*, there is exactly one edge out of *s* labeled *a*

  - While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings.

- It is indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.

- The following algorithm shows how to apply a DFA to a string.

  Algorithm: Simulating a DFA.

  INPUT: An input string *x* terminated by an end-of-file character **eof.** A DFA *D* with start state s_0, accepting states *F,* and transition function *move.*

  OUTPUT: Answer "yes" if *D* accepts *x;* "no" otherwise.

  METHOD: Apply the following algorithm to the input string *x.* The function *move(s,c)* gives the state to which there is an edge from state *s* on input *c*. The function *nextChar* returns the next character of the input string *x.*

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

- **Example:** In the following Fig, we see the transition graph of a DFA accepting the language **(a|b)*abb**. Given the input string *ababb,* this DFA enters the sequence of states 0 , 1 , 2 , 1 , 2 , 3 and returns "yes".