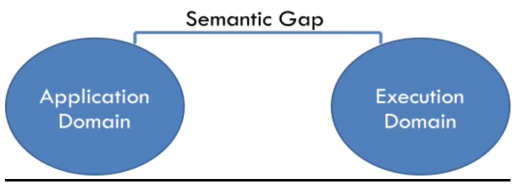
COMP 7.1 COMPILER CONSTRUCTIONS

UNIT – 1
1. Language Processor: Language Processor concepts, Data Structures for Language Processors.
2. Introduction to Compiler: Phases of compilation, Bootstrapping and Porting, Compiler writing tools.
3. Lexical Analyser: The role of Lexical Analyser, Design & Implementation of Lexical Analyser.
4. LEX tool: A language for specifying LA. Study of the features and applications of LEX / FLEX tool.

1. LANGUAGE PROCESSOR CONCEPTS

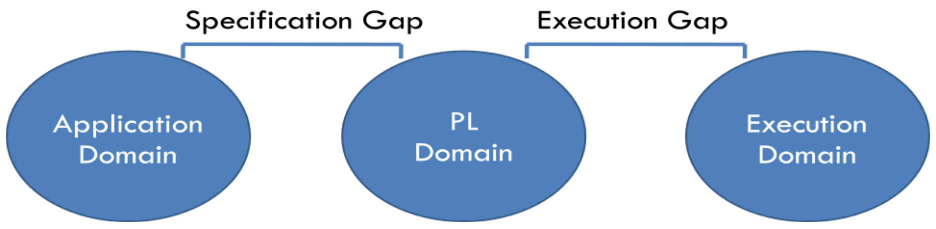
- Language processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behaviour of software and the manner in which these ideas are implemented in a computer system.
- The designer expresses the ideas in terms related to the application domain of the software.
- To implement these ideas, their description has to be implemented in terms related to the execution domain of the computer system.
- The term Semantic gap used to represent the differences between the semantics of application domain and the execution domain.



- The semantic gap has many consequences as,
    - Large development time
    - Large development efforts
    - Poor quality of the software.

- Language Processors has mainly three purposes,
    - bridge gap between Application Domain and Execution Domain
    - translation from one language to another
    - to detect errors in source code during translation

- These issues are tackled by software engineering methodologies and programming languages.
- The software implementation using Programming languages introduces a new domain, the PL domain. The semantic gap between the application and the execution domain is bridged by the software engineering steps.
- **Specification Gap**: It is the semantic gap between two specifications of the same task.
- **Execution Gap:** It is the gap between the semantics of programs (that perform the same task) written in different programming languages.



Language Processor – Definition

- A language processor is software which bridges a specification or execution gap.
- A language processor typically aborts generation of the target program if it detects errors in the source program.

Types of Language Processors

- A spectrum of language processors to meet the requirements are,
  1. **Language Translators** - A Language translator bridges an execution gap to the machine language of a computer system.
  2. **Detranslator** – A detranslator bridges the same execution gap as the language translator, but in the reverse direction.
  3. **Preprocessor** – A preprocessor is a language processor which bridges an execution gap but is not a language translator.
  4. **Language Migrator** – A Migrator bridges the specification gap between two PLs.

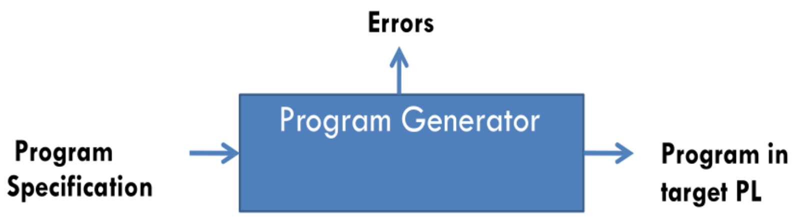
1.1 Language Processing Activities

The fundamental language processing activities to bridge the specification gap and the execution gap are,

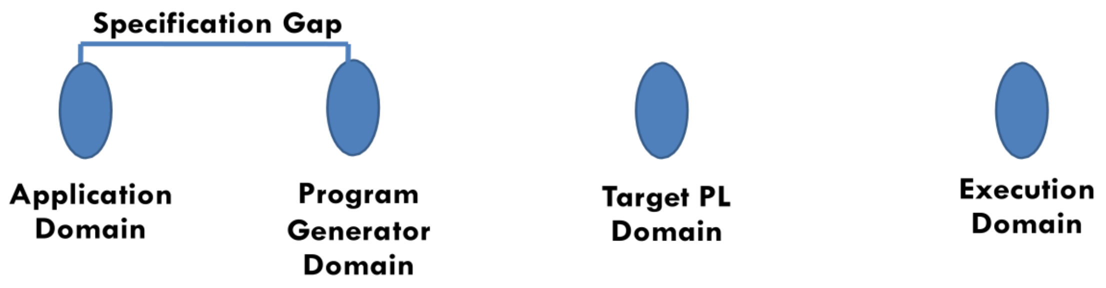
1. Program generation activities
2. Program execution activities

1.1.1. Program Generation activities

- The program generation activity aims at automatic generation of a program.
- The program generator is a software system which accepts the specification of a program to be generated and generates a program in the target PL.



- In effect, the program generator introduces a new domain between the application and PL domains, called as Program generator domain.
- The specification gap is now the gap between the application domain and the program generator domain.
- Reduction in the specification gap increases the reliability of the generated program.

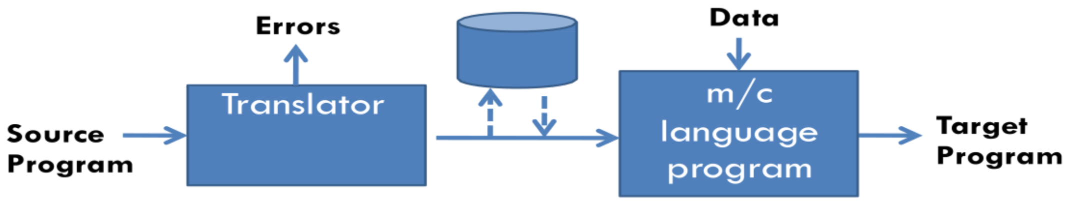


1.1.2. Program Execution activities

- The program execution activity organizes the execution of a program written in a PL on a computer system.
- Two popular models for program execution are
  1. **Translation**
  2. **Interpretation**.

1. Program Translation

- The program translation model bridges the execution gap by translating a program written in a PL, called the source program into an equivalent program in the machine or assembly language of the computer system, called the target program.

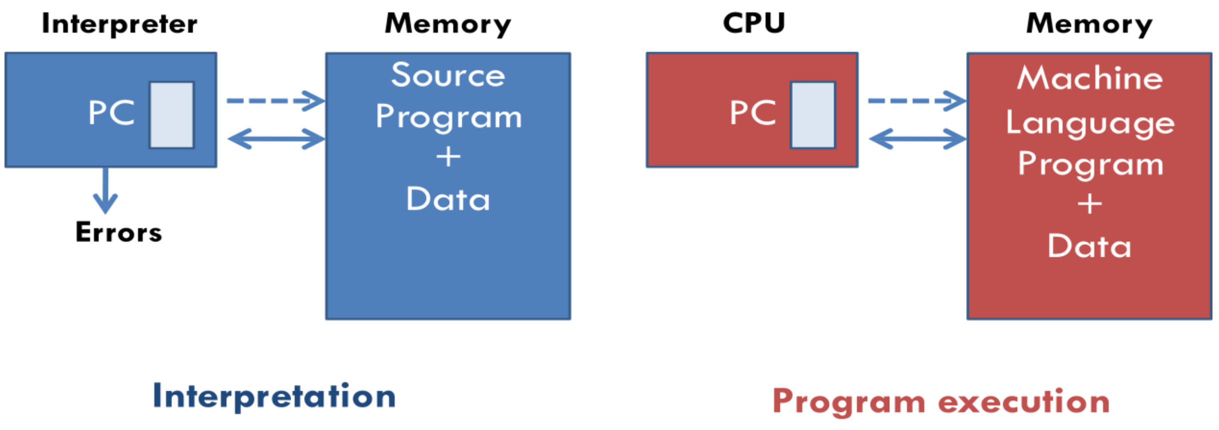


Characteristics of the program translation model:

- A program must be translated before it can be executed.
- The translated program may be saved in a file. The saved program may be executed repeatedly.
- A program must be retranslated following modifications.

2. Program Interpretation

- The interpreter reads the source program statement, determines its meaning and performs actions which implement it.
- The program interpretation cycle consists of the following steps,
  1. Fetch the statement
  2. Analyze the statement and determine its meaning –the computation to be performed and its operands.
  3. Execute the meaning of the statement.



1.2 PHASE AND PASS OF A LANGUAGE PROCESSOR

1.2.1. Phases

There are mainly two phases of language processor which are

1. **Analysis phase**
2. **Synthesis phase**

Analysis Phase

- The objective of the analysis phase is to break up the source code and imposes into a grammatical structure to create an Intermediate Representation (IR).
- In addition to it, it also collects information like labels from source code and stores in a symbol table.

Synthesis Phase

- The synthesis phase, construct the **object code** from the intermediate representation and symbol table.
- Some objectives of this phase includes:
    1. Obtain machine code from mnemonics table
    2. Check the address of an operand from symbol table
    3. Synthesize a machine instruction.

1.2.2. Pass of a Language Processor

- Pass of a language processor describes how many times it processes the source program.
- A pass of a language processor indicates the processing of every statement once in a source program.
- There are two passes of language processors
    - **Pass 1:** Perform the analysis of the source program and collect the important information from it.
    - **Pass 2:** Perform the synthesis of the target programmers.

1.3. DATA STRUCTURES USED FOR LANGUAGE PROCESSING

- The design of Data structures is a crucial issue in language processing activities.
- The data structures used in language processing can be classified on the basis of the following criteria,
    1. **Nature of the Data Structure**
        - Whether Linear or Nonlinear Data Structure
    2. **Purpose of a data structure**
        - Whether search Data structure or an allocation data structure
    3. **Lifetime of a data structure**
        - Whether used during language processing or during target program execution.

1.3.1. LINEAR AND NON-LINEAR DATA STRUCTURES

- A **Linear Data structure** consists of a linear arrangement of elements in the memory. The physical proximity of its elements is used to facilitate efficient search operations.
- The Linear DS requires a contiguous area of memory for its elements. This poses a problem in situations where the size of a DS is difficult to predict.
- The elements of a **Non-Linear DS** are need not occupy contiguous areas of memory and can be accessed using pointers. This avoids the memory allocation problem in the context of Linear DS. However the Non-Linear arrangement leads to lower search efficiency.

1.3.2. SEARCH DATA STRUCTURES

- The Search DS are characterized by the fact that the entry for an entity is created only once but may be searched for a large number of times, hence the search efficiency is therefore very important.
- The Search DS are used during language processing to maintain attribute information concerning different entities in the source program.

1.3.3. ALLOCATION DATA STRUCTURES

- Allocation data structures are characterized by the fact that the address of memory area allocated to an entity is known to the user(s) of that entity and there is no search operations are conducted on them.
- Speed of allocation or de-allocation and efficiency of memory utilization are the important criteria for the allocation data structures.

1.3.4. USE OF SEARCH & ALLOCATION DATA STRUCTURES

- Language Processor uses both search DS and allocation DS during its operation.
- Search DS used to constitute various tables of information.
- Allocation DS used to handle programs with nested structures of some kind.
- A Target program rarely uses search DS and may use allocation data structures.

SEARCH DATA STRUCTURES

- A Search DS is a set of entities, each entry accommodating the information concerning one entity. Each entry is assumed to contain a key field which forms the basis for a search.

(a). ENTRY FORMATS

- Each entry in search structure is a set of fields i.e a record, a row.
- Each entry is divided into two parts:
    1. Fixed Part
    2. Variant Part
- The value in fixed (tag) part determines the information to be stored in the variant part of the entry.

Tag Value (fixed part)	Variant part fields
Variable	type, length, dimension info
Procedure name / function name	Address of parameter list, Number of parameters, type of returned value, length of returned value.
Label	Statement number

- An entry may be declared as a record or a structure of the language in which the language processor is being implemented.

a.1 Fixed length Entry format

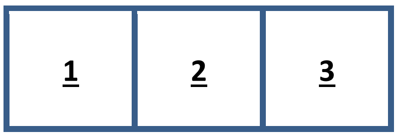
- In the fixed length entry format, each record is defined to consist of the following fields
- For each value V_i in the tag field, the variant part of the entry consists of the set of fields SF_{Vi}.
- **Format**: In a fixed length entry format a record consists of the following fields,
    - Fields in the fixed part of the entry.
    - U_{vi} SF_{vi}, i.e the set of fields in all variant parts of the entry.

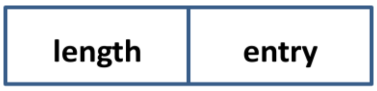
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

1. Symbol
2. Class
3. Type
4. Length
5. Dimension Information
6. Parameter List Address
7. No. of Parameters
8. Type of returned value
9. Length of returned value
10. Statement number.

- In fixed length entries, all the records in search structure have an identical format.
- This enables the use of homogeneous linear data structures like arrays.
- **Drawback**: Inefficient use of memory: Many records may contain redundant fields.

a.2 Variable length Entry format





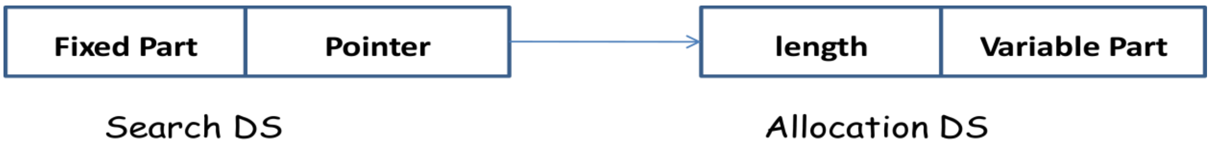
1. Name
2. Class
3. Statement Number

When class = label, all fields excepting name, class and statement number are redundant. Here, Search method may require knowledge of length of entry.

- Variable Length Entry Format, a record consists of the following fields,
  - Fields in the fixed part of entry, including the tag field
  - $\{ f_j \mid f_j \in SF_{Vj} \text{ if tag} = V_j \}$
- This entry format leads to compact organization in which no memory wastage occurs.

a.3 Hybrid Entry format

- A Hybrid entry format is a compromise between Fixed and Variable entry formats
- It combines the access efficiency of Fixed Entry Format with memory efficiency of Variable Entry Format.
- Each entry is divided into two halves (i.e) Fixed Part and Variable Part. A pointer is added to the fixed part which points to the variable part of the entry.
- **Data Structure**: The fixed part and variable parts are accommodated in two different Data structures.
  - Fixed Part : Search DS - Linear DS. Require Efficient Searching.
  - Variable Part : Allocation DS which can be linear or Non Linear in nature..
  - Since the Fixed part has pointer field which do not need searching in variable part.



Operations on Search Structures: The following operations are performed on search DS.

- **ADD:** Add the entry of a symbol. Entry of symbol is created only once.
- **SEARCH:** Search and locate the entry of a symbol. Searching may be performed for more than once.
- **DELETE:** Delete the entry of a symbol. Uncommon operation**.**

(b) GENERIC SEARCH PROCEDURE

- The **generic procedure** to search and locate the entry of a symbol ‘s’ in a search DS,
  1. Make a ***prediction*** concerning the entry of search data structure which symbol s may be occupying. We call this entry e.
  2. Let s_e be the symbol occupying eth entry. *Compare* s with s_e. Exit with success if the two match.
  3. Repeat step 1 and 2 till it can be concluded that the symbol *does not exist* in the search data structure.
- The nature of the prediction varied with the organization of the search DS. Each comparison of step 2 is called a **probe**.
- Efficiency of a search procedure is determined by the No. of probes performed by search procedure.
- **Probe Notations**:
  - Ps : Number of probes in successful search
  - Pu : Number of probes in an unsuccessful search.

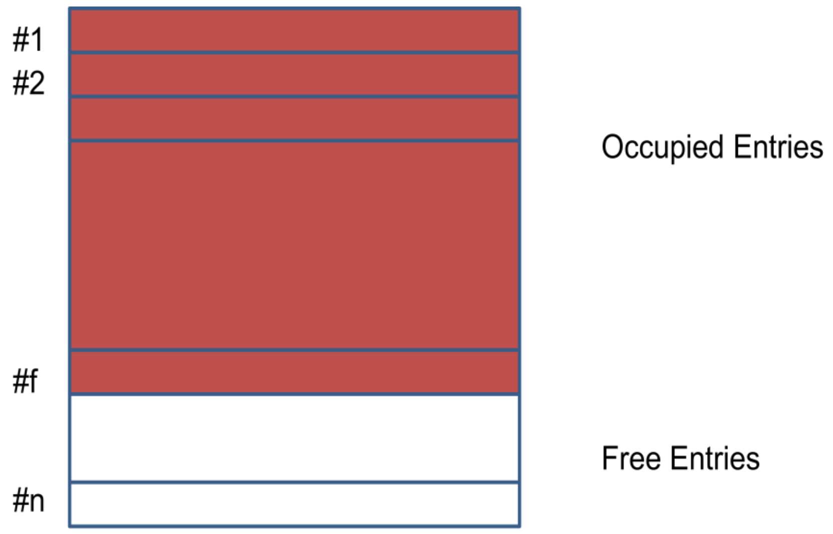
b.1 Table Organization

- The entries of a table occupy adjoining areas of memory.
- **Positional Determinacy:** Tables using fixed length entry organization possess this property. This property states that the address of an entry in a table can be determined from its entry number.

  Example: Address of the eth entry is

  $$\mathbf{a + (e - 1).}\ \boldsymbol{L}$$

  a : address of first entry.
  L : length of an entry.
  e : entry number.



n: Number of entries in the table **f:** Number of occupied entries

- Use of Positional Determinacy:
  - Representation of symbols by e
  - Entry number in the search structure
  - Intermediate code generated by LP

b.2 Sequential Search Organization

- **Search for a symbol**:

  All active entries in the table have the same probability of being accessed.

  Ps = f/2 for a successful search

  Pu = f   for an unsuccessful search

  Following an unsuccessful search, a symbol may be entered in the table using an add operation.

- **Add a symbol**: The symbol is added to the first free entry in the table. The value of f is updated accordingly.

- **Delete a symbol**: Deletion of an entry can be implemented in two ways:
  - **Physical Deletion**: an entry is deleted by erasing or by overwriting. If the dth entry is to be deleted, entries d+1 to f can be shifted up by one entry each. This would require (f-d) shift operations in symbol table. Efficient alternate would be to move fth entry into dth position, requiring only one shift operation.
  - **Logical Deletion**: is performed by adding some information to the entry to indicate its deletion by introducing a field to indicate whether an entry is active or deleted.

Active/ Deleted	Symbol	Other Info

b.3 Binary Search Organization

- All entries in a table are assumed to satisfy an ordering relation.
- The < relation implies that the symbol occupying an entry is smaller than the symbol occupying the next entry.
- At any stage the search prediction is that symbol s occupies the middle entry of that part of the table which is expected to contain its entry.
- **Algorithm:**

```
1. start:=1; end:=f;
2. while start <= end
      (a) e:= [(start+end)/2]; where [] implies a rounded quotient. Exit
          with success if $s=s_e$.
      (b) if $s<s_e$ then end:=e-1; else start:=e+1;
3. Exit with failure.
```

- For a table containing f entries we have $p_s <= [\log_2 f]$ and $p_u=[\log_2 f]$. Thus the search performance is logarithmic in the size of the table.
- **Drawback:** The requirement that the entry number of a symbol in the table should not change after an add operation. (Due to its use in the IC). Thus, this forbids both addition and deletion during language processing.
- Hence, binary search organization is suitable only for a table containing a fixed set of symbols.

b.4 Hash Table Organization

- In the Hash table organization the search prediction depends on the value of s,i and e. Where e is a function of s.
- There are three possibilities exist concerning the predicted entry
  1. Entry may be occupied by s
  2. Entry may be occupied by some other symbol
  3. Entry may be empty.
- Collision – collision occurs when the entry is occupied by some other symbol.

Algorithm (Hash Table Management)

```
1. e:=h(s);
2. Exit with success if s = $s_e$ and with failure if entry e is unoccupied.
3. Repeat steps 1 and 2 with different functions h', h'', etc.
```

h is the hashing function using the following notation and properties.

Hashing Function Notations:

- n : number of entries in the table
- f : number of occupied entries in the table
- P : Occupation density in table, i.e f/n
- k : number of distinct symbols in source language
- k_p: number of symbols used in some source program
- S_p: set of symbols used in some source program
- N : Address space of the table.
- K : Key space of the system
- K_p: Key space of a program

Hashing Function – Properties

- Hashing function has the property: 1 <= h(symb) <= n

Direct Entry Organization

- – If k<=n, select a One0to-one function as hashing function h.
  – This will eliminate collision.
  – Will require large symbol table.
  – Better solution K_p => N which is nearly one to one for set of symbols S_p.
- Effectiveness of a hashing organization depends on average value of p_s.
- If k_p increases, p_s should also increase.
- Assignment Question: What is folding?
- Assignment Question: Write a Short Note on Hashing function.

Hash Collision Handling Mechanism

1. Rehashing Technique: To accommodate a colliding entry elsewhere in the hash table. Disadvantage: Clustering.
2. Overflow Chaining Technique: To accommodate the colliding entry in a separate table. Disadvantage: Extra memory requirement by overflow table.
3. Scatter Table Organization: Overcomes drawback of overflow chaining technique, i.e large memory requirement.

b.5 Linked List and Tree Organization

- Linked List and tree structured organization are non-linear in nature that is elements of search data structure are not located in adjoining memory locations.
- Each entry in linked list organization contains a single pointer field.
- List has to be searched sequentially.
- Hence its performance is identical with that of sequential search tables. i.e $p_s = l/2$ and $p_u = l$.

b.6 Binary Tree Structure

- Each node in the tree is a symbol entry with two pointer fields i.e Left Pointer and Right Pointer.

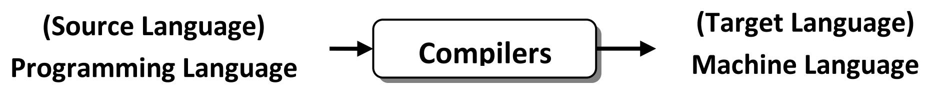
Algorithm b.4 (Binary Tree Search)

```
1. Current_node_pointer := address of root
2. if s = (current_node_pointer)*.symbol then exit with success;
3. if s<(current_node_pointer)*.symbol then
        current_node_pointer:= (current_node_pointer) *. left_pointer;
   else
        current_node_pointer:= (current_node_pointer) *. right_pointer;
4. if current_node_pointer := nill then exit with failure.
     else
        goto step 2.
```

2. INTRODUCTION TO COMPILERS

2.1 COMPILER

- The Compilers act as translators, transforming human-oriented programming language into Computer-oriented machine language.



- 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.

2.1.2 Compiler – target codes

- Compilers may be **distinguished in two ways**
  1. By the kind of machine code they generate
  2. By the format of target code they generate

- **Kind of Machine Codes**

  Compilers may generate **any of three types of code** by which they can be differentiated:

  1. Pure machine code
  2. Augmented machine code
  3. Virtual machine code

1. **Pure Machine Code**:
   - Compilers may generate code for a particular machine, not assuming any operating system or library routines.
   - This is "pure code" because it includes nothing beyond the instruction set.
   - Pure code can execute on bare hardware without dependence on any other software.

2. **Augmented Machine Code**:
   - Commonly, compilers generate code for a machine architecture augmented with operating system routines and run-time language support routines.
   - To use such a program, a particular operating system must be used and a collection of run-time support routines (I/O, storage allocation, mathematical functions, etc.) must be available.

3. **Virtual Machine Code**:
   - Generated code can consist entirely of virtual instructions (no native code).
   - This supports transportable code that can run on a variety of computers.
   - **Java**, with its JVM (Java Virtual Machine) is a great example of this approach.

- **Formats of Translated Programs**

  Compilers differ in the format of the target code they generate. Target formats may be categorized as

  1. Assembly language,
  2. Relocatable binary,
  3. Memory-image (or) Absolute binary

1. **Assembly Language (Symbolic) Format**:
   - A text file containing assembler source code is produced.
   - A number of code generation decisions (jump targets, long vs. short address forms, and so on) can be left for the assembler.
   - Generating assembler code supports cross compilation and also simplifies debugging and understanding a compiler

2. **Memory-Image (Absolute Binary) Form**:
   - Compiled code may be loaded into memory and immediately executed.
   - This is faster than going through the intermediate step of link/editing.
   - The ability to access library and precompiled routines may be limited.
   - The program must be recompiled for each execution.

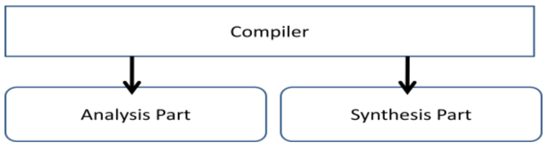
3. Relocatable Binary Format:

- Target code may be generated in a binary format with external references and local instruction and data addresses are not yet bound.
- Instead, addresses are assigned relative to the beginning of the module or relative to symbolically named locations.
- A linkage step adds support libraries and other separately compiled routines and produces an absolute binary program format that is executable.

2.1.3 Structure of a Compiler

A Compiler is a program that maps a source program into a semantically equivalent target program. There are two parts to that mapping:

1. Analysis part
2. Synthesis part



Analysis Part - Analysis of the Source Program

Synthesis Part - Synthesis of the Machine language Program

A. Analysis Part

The Analysis part collects information about the source program and performs the following,

- Breaks up the source program into constituent pieces called TOKENs.
- Imposes a grammatical structure on the pieces
- Makes the source program syntactically well semantically sound.
- Create the intermediate representation of the source program.

Analysis on the Source Program

The analyses helps to collect information about the source program are,

1. Linear or Lexical Analysis
2. Hierarchical or Syntax Analysis
3. Semantic Analysis

1. Linear (or) Lexical Analysis:

- The stream of characters making up the source program is read from left to right and grouped into Tokens.
- Linear analysis is also called lexical analysis or scanning.
- The blanks separating the character of tokens are eliminated.

2. Hierarchical (or) Syntax Analysis:

- The tokens of the source program are grouped into grammatical phrases that are used by the compiler to synthesize the output.
- Hierarchical analysis is also called Parsing or Syntax Analysis.
- The hierarchical structure of a program is usually expressed by recursive rules.

3. Semantic Analysis:

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

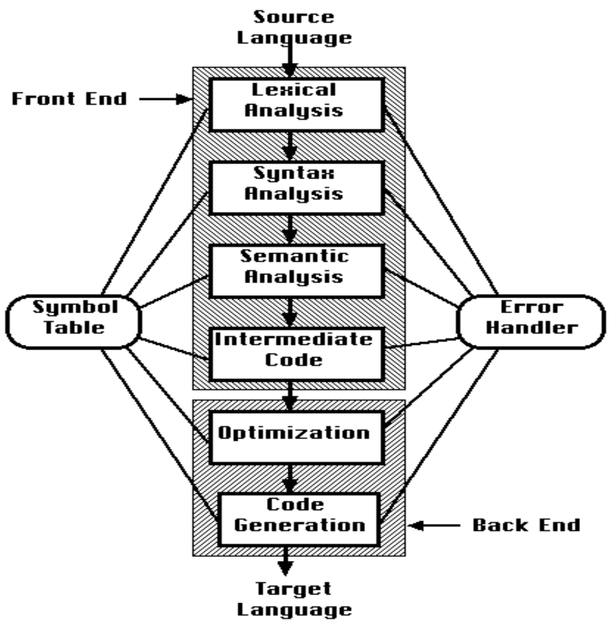
B. Synthesis Part:

The Synthesis part reads intermediate representation from analysis part and produces optional assembly language or constructs desired target language.

2.1.4 Phases of Compilers

- The compilation process operates in phases, each of which transforms one representation of the source program into another.
- The typical decomposition of phases of compilers are
    1. Lexical Analysis
    2. Syntax Analysis
    3. Semantic Analysis
    4. Intermediate Code Generation
    5. Code optimization
    6. Code Generation

  and two supporting phases includes
    1. Symbol Table Management
    2. Error Handling.



1. Lexical Analysis

- The Lexical Analyzer SCAN the source program one character at a time from left-to-right. It groups the stream of characters into meaningful sequences called TOKENs.
- It keeps track of line numbers to indicate errors.
- The blanks separating the tokens and comment line statements would be discarded.
- It makes an entry into the symbol table on recognizing identifiers used in the source program.

2. Syntax Analysis

- The Syntax analyzer or parser groups the token stream produced by the Lexical analyzer into grammatical phrases with collective meaning.
- It checks them against the Grammar of the Source language and produces the valid syntactical construct in the form of tree, called parse tree.

3. Semantic Analysis

- The Semantic analyzer uses the parse tree generated by the syntax analyzer and the information in the Symbol table to,
  - Check the source program for semantic consistency with the language definition,
  - Update the type information into the symbol table.

4. Intermediate Code Generation

- The ICG generates an explicit intermediate representation of the source program.
- The Intermediate representation can have a variety of forms as
  1. Syntax tree
  2. Postfix form
  3. Three Address Code (TAC) Statements
- The Intermediate representation should have two properties
  - It should be easy to produce from the source program
  - It should be easy to translate into equivalent target program.

5. Code Optimization

- The code optimizer attempts to improve the quality of Intermediate code, so that faster running target code will result.
- The Code optimizer considers the factors like algorithm, memory, running time of the target program by applying code improving transformations.
- The Code optimization is optional and can be performed,
  - **Before Code generation** – transformation applied over the intermediate representation of the source program, called Pre-Optimization or Machine Independent Optimization.
  - **After Code Generation** – transformation applied on the target code, called post optimization or machine dependent optimization.

6. Code Generation

- The code generator takes as input an optimized Intermediate representation of the source program and maps it into the target language.
- Each intermediate instruction is translated into a sequence of machine instructions that perform the same task by using some pattern matching algorithm.

7. Symbol Table Management

- The Symbol Table is a data structure containing a record for each identifier with fields for attributes for the same.
- It is an essential function of compiler to record the symbols used in the source program and collects the information about various attributes (like storage, type, scope) of each symbol.

8. Error Handling

- One of the most important functions of a compiler is the detection and reporting of errors in the source program.
- The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all the phases of a compiler.
- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.
- Both of the symbol table-management and error-Handling routines interact with all phases of the compiler.

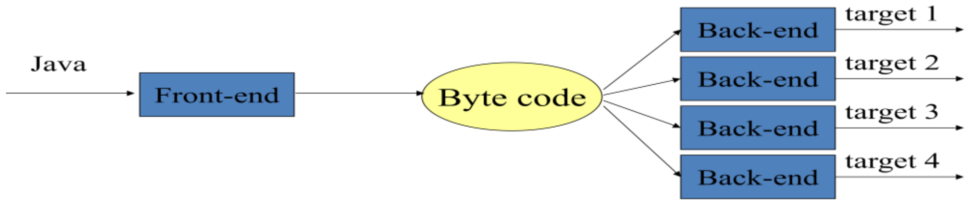
2.1.5 Grouping of Phases

- In an implementation, activities of several phases may be grouped into a pass that reads an input and writes an output file.
- The Front-end phases of Lexical analysis, Syntax analysis, Semantic analysis and Intermediate code generation might be grouped into one pass.
- There could be a back-end pass consisting of optional code optimization and code generation for a particular target machine.

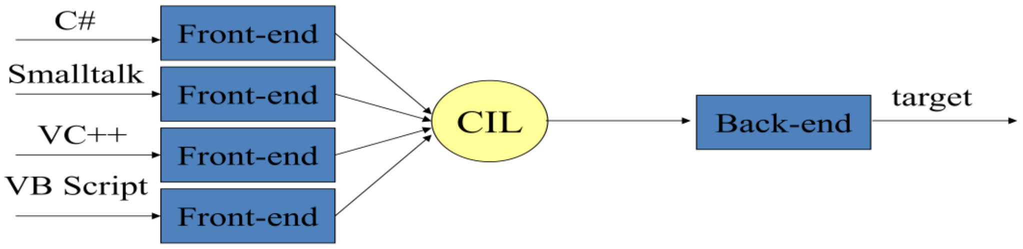
Benefits - Grouping of Compiler Phases

A carefully designed Intermediate representation allows the front-end for a particular language to interface with the back-end for different target machine and also different front-ends to interface with a single back-end.

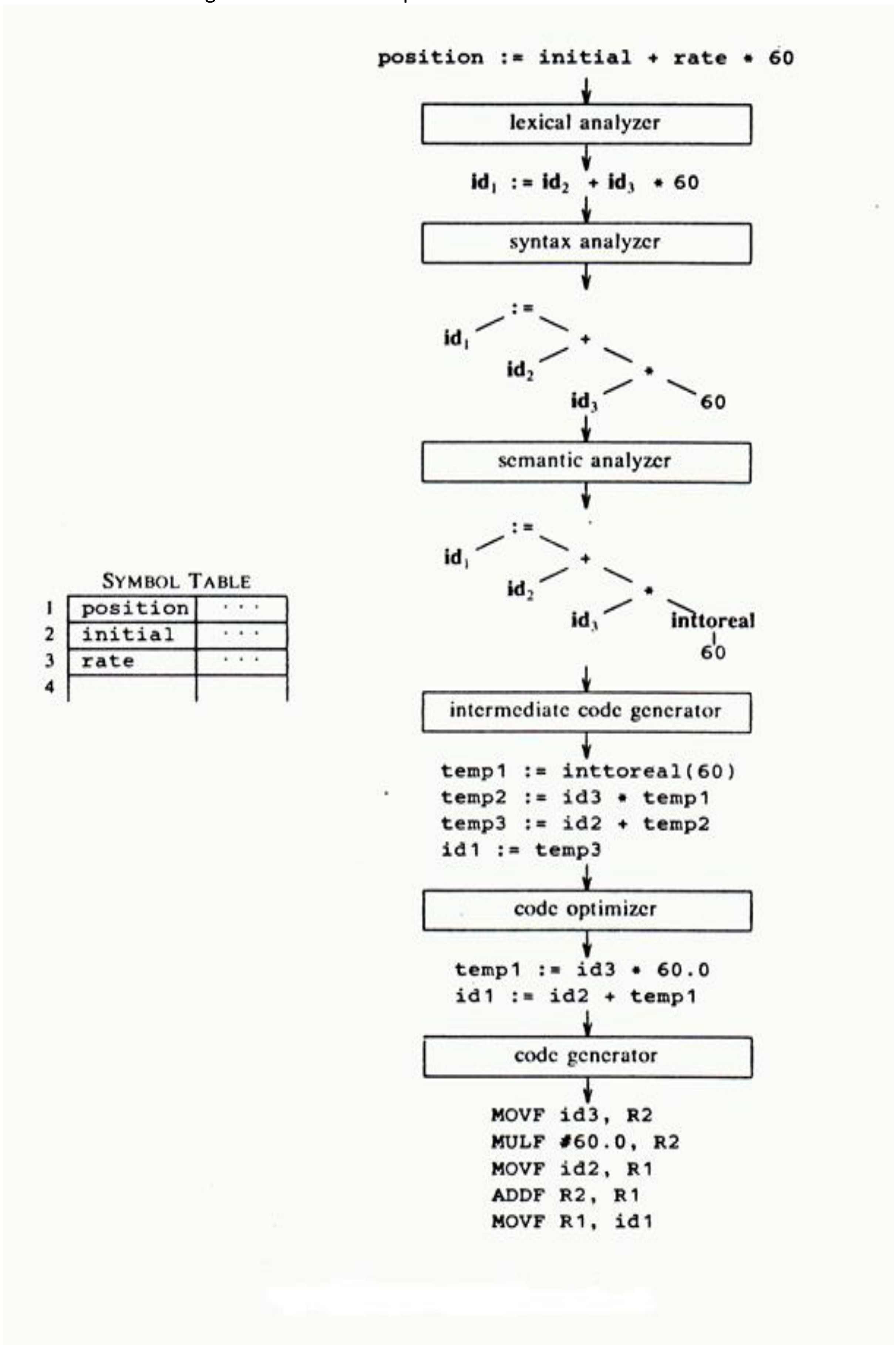
- **Platform Integration** - Producing compilers for different target machines, by combining a front end with back-ends for different target machines.
  Example: JAVA technology.



- **Language Integration**: Producing compilers for different source languages for one target machine by combining different front-ends with the back-end for that target machine. **Example: .NET technology.**



Translation of an Assignment Statement: position = initial + rate * 60



1	position	. . .
2	initial	. . .
3	rate	. . .
4		

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

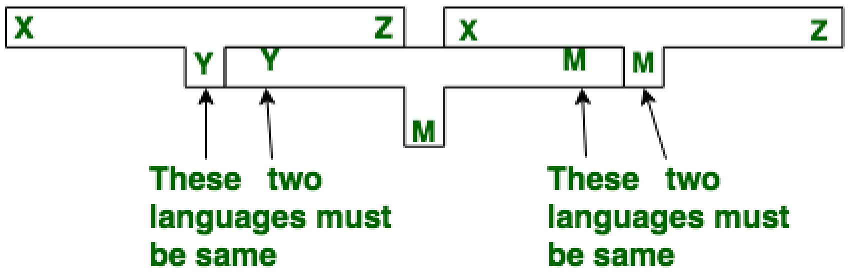
```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

2.2 BOOTSTRAPPING AND PORTING

2.2.1 Bootstrapping Compilers

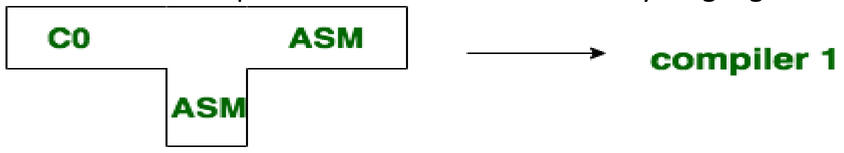
- **Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.
- **Bootstrapping** is the process of writing a compiler (or assembler) in the source programming language which it is intended to compile. Applying this technique leads to a self-hosting compiler.
- Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.

- **Example:** To clearly understand the **Bootstrapping**, consider a following scenario.
  - Suppose we want to write a cross compiler for new language X.
  - The implementation language of this compiler is say Y and the target code being generated is in language Z.
  - That is, we create X_YZ.
  - Now if existing compiler Y runs on machine M and generates code for M then it is denoted as Y_MM. Now if we run X_YZ using Y_MM then we get a compiler X_MZ.
  - That means a compiler for source language X that generates a target code in language Z and which runs on machine M.
- Many compilers for many programming languages are bootstrapped, including compilers for BASIC, ALGOL, C, Pascal, PL/I, Factor, Haskell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Java, Python, Scala, Nim, Eiffel, and more
- **Example:** We can create compiler of many different forms. Now we will generate.


X
Z
Y
Y
M
X
M
M
Z
These two languages must be same
These two languages must be same

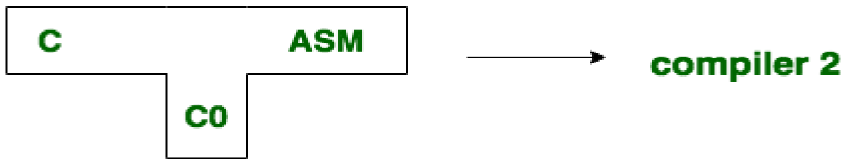
Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

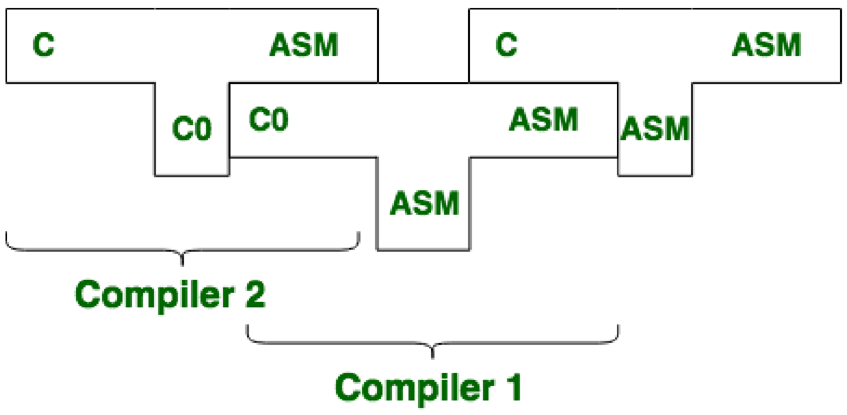
Step-1: First we write a compiler for a small of C in assembly language.



Step-2: Then using with small subset of C i.e. C0, for the source language c the compiler is written.

Step-3: Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.





Step-4: Thus we get a compiler written in ASM which compiles C and generates code in ASM.

<u>To obtain a compiler for language X (which is written in language X), then</u>

1. First build a compiler (or interpreter) for language X in some other language Y.
2. Once that is done, Write a new version of the compiler in language X. You use the first bootstrap compiler to compile the compiler, and then use this compiled compiler to compile everything else (including future versions of itself).

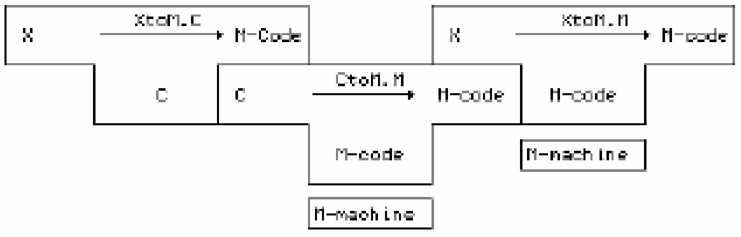
<u>Bootstrapping a compiler has the following advantages:</u>

1. It is a non-trivial test of the language being compiled.
2. Compiler developers only need to know the language being compiled.
3. Compiler development can be done in the higher level language being compiled.
4. Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself.
5. It is a comprehensive consistency check as it should be able to reproduce its own object code.

<u>2.2.2 CROSS COMPILERS</u>

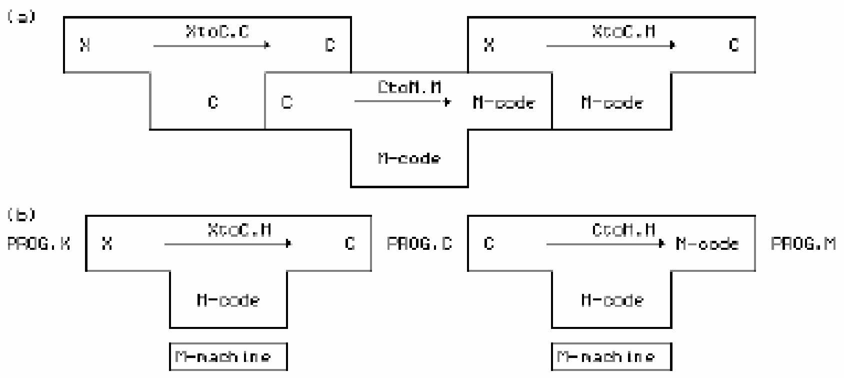
- A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. Example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.
- A cross compiler is necessary to compile for multiple platforms from one machine.

- A platform could be infeasible for a compiler to run on, such as for the microcontroller of an embedded system because those systems contain no operating system.
- Cross compilers are not to be confused with source-to-source compilers. A cross compiler is for cross-platform software development of binary code, while a source-to-source "compiler" just translates from one programming language to another in text code.
- ***<u>When a translator is required for an old language on a new machine or a new language on an old machine (or even a new language on a new machine), is to make use of existing compilers on either machine, and to do the development in a high level language.</u>***
- **<u>Example</u>**: One's dream machine *M* is supplied with the machine coded version of a compiler for a well-established language like C, then the production of a compiler for one's dream language *X* is achievable by writing the new compiler, say *XtoM*, in C and compiling the source (*XtoM.C*) with the C compiler (*CtoM.M*) running directly on *M*. This produces the object version (*XtoM.M*) which can then be executed on *M*.



2.2.3 PORTING

- The process of modifying an existing compiler to work on a new machine is often known as **porting** the compiler.
- When a compiler *XtoC* for a popular language *X* has been implemented in C on machine *A* by writing a high-level translator to convert programs written in *X*to C and where it is desired to use language *X* on a machine *M* that, like *A*, has already been blessed with a C compiler of its own.
- To construct a two-stage compiler for use on either machine, all one needs to do, in principle, is to install the source code for *XtoC* on machine *M* and recompile it.



2.3. COMPILER CONSTRUCTION / WRITING TOOLS

- In addition to general-software development tools, more specialized tools have been created to help implement various phases of a compiler.
- They also have known as Compiler Compilers or Compiler Writing Systems.
  The commonly used compiler-construction tools include,

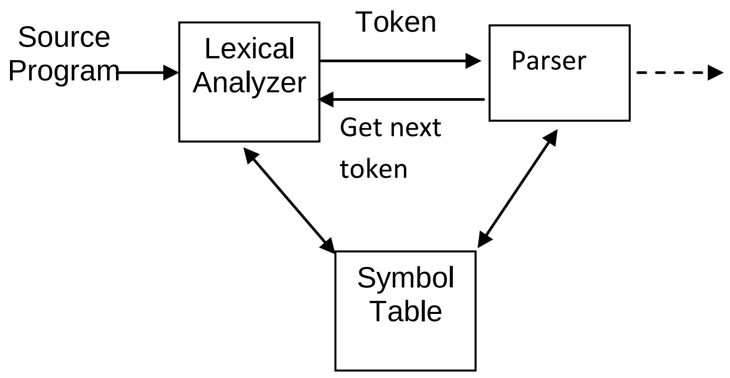
1. Scanner Generator
2. Parser Generator
3. SDT Engine
4. Data flow analysis Engine
5. Code Generator Generators
6. Compiler-Construction Toolkit

1. **Scanner Generators** – produce lexical analysers from a regular expression description of the tokens of a language. Example: LEX tool.
2. **Parser Generators** – automatically produce syntax analyzer from a grammatical description of a programming language.
3. **Syntax-Directed translation Engines** – produce collection of routines for traversing a parse tree and generating intermediate code.
4. **Code Generator Generators** – produce a code generation from a collection of rules for translating each operation of the intermediate language into machine language for target machine.
5. **Data Flow Analysis Engine** – facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data flow analysis is a key part of code optimization.
6. **Compiler Construction Toolkit** – provide an integrated set of routines for constructing various phases of a compiler.

3. THE LEXICAL ANALYZER

3.1 The Role of the Lexical Analyser

- Lexical analyzer is the first phase of compiler.
- The lexical analyzer reads the input source program from left to right one character at a time and generates sequence of tokens.
- As the lexical analyzer scans the source program to recognize tokens, it is also called as scanner.



Functions of the lexical analyzer

- The Lexical analyzer is the only phase of the compiler that reads the source program, it may perform certain tasks besides identification of lexemes which includes,
  - Stripping out comments and whitespaces (blank, newline, tab and other characters that are used to separate tokens in the input.)
  - Correlating error messages generated by the compiler with the source program.
  - It may keep track of newline characters seen, so it can associate a line number with error message.
  - On recognizing any identifier, the information about an identifier – its lexeme and the location at which it is first found is entered in the Symbol table.

The lexical analyzers are divided into a cascade of two processes,

1. Scanning

- Scanning is the process of reading the source text from left to right one character at a time.
- Tasks include deletion of comments and compaction of consecutive white space characters into one. It do not require tokenization of the input

2. Lexical Analysis

- Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner

3.1.2 Lexical Analyzer - Design Issues

- There are a number of reasons why the analysis part of a compiler is normally separated into lexical analysis and parsing phases,

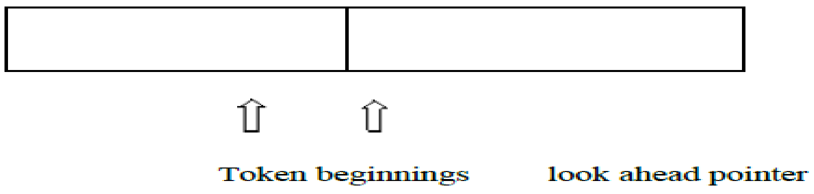
    - **Simplicity of design** – The separation of lexical and syntactic analyses allows the compiler designer to simplify at least one of these tasks.
    - **Compiler efficiency is improved** – a separate lexical analyzer allows the designer to apply specialized techniques that serve only the lexical task. Specialized Input buffering technique for reading input character can speed up the compiler significantly.
    - **Compiler portability is enhanced** – Input device specific peculiarities can be restricted to the lexical analyzer.

3.1.3 Lexical Errors & Recovery actions

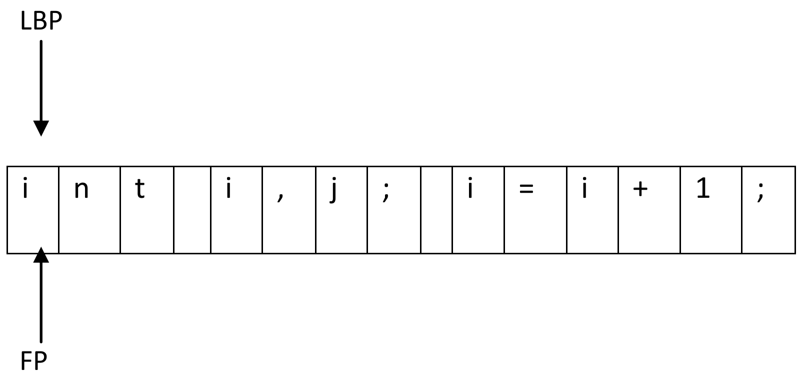
- The lexical errors are very simple in nature like misspelling of Identifiers, keywords of operators, etc.
- Possible Error recovery actions are,
    - ✓ Delete one character from the remaining input
    - ✓ Insert a missing character into the remaining input
    - ✓ Replace a character by another character
    - ✓ Transpose two adjacent characters

3.1.4 Input Buffering – Scanning

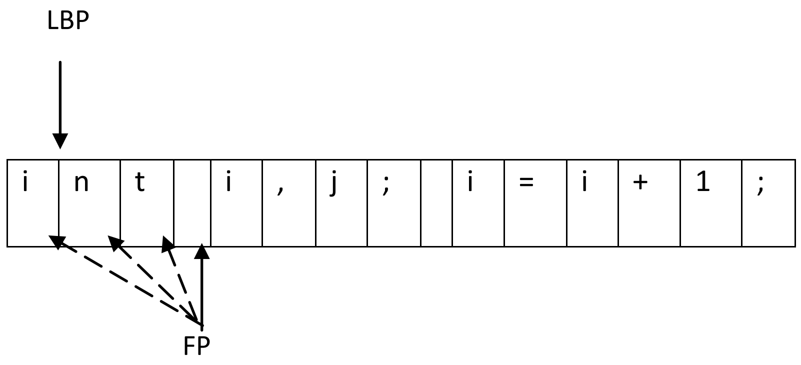
- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- Using a pair of buffers (instead of single buffer) cyclically and ending each buffer's contents with a sentinel character that warns of its ends are the two techniques that accelerate the process of buffering.
- The lexical analyzer scans the input string from left to right one character at a time. It uses two pointers Lexeme_begin_pointer (LBP) and Forward_ptr (FP) to keep track of the portion of the input scanned. Initially both the pointers point to the first character of the input string.



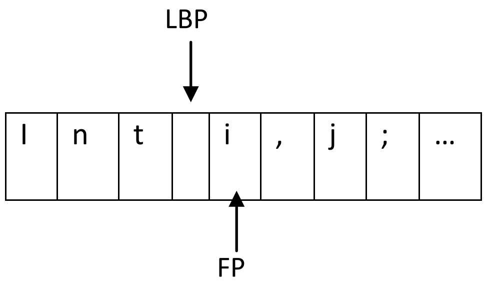
- **Example:** Input scanned from the source program is → int i,j; i=i+1;



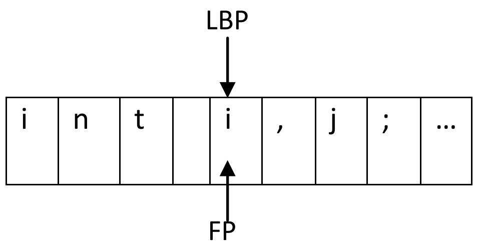
- The forward_ptr moves ahead to search for end of lexeme. As soon as the white space is encountered it indicates end of lexeme. In the above example as soon as the forward_ptr (FP) encounters a blank space the lexeme "int" is identified.



- At the moment the forward_ptr is located in the blank space. So the forward_ptr will ignore the blank space and it will move ahead. Then both the Lexeme_begin_ptr (LBP) and the forward_ptr (FP) will be set for the next string.



Forwarding the LBP ahead.



Setting of the two pointers

Organization of Dual Buffering Scheme:

- This scheme involves two buffers that are alternatively reloaded. Each buffer is of the same size N and N is usually the size of a disk.

Two pointers to the input are maintained

Pointer *Lexeme Begin*, marks the beginning of the current lexeme, whose extent we are attempting to determine

Pointer *Forward,* scans ahead until a pattern match is found.

Once the next lexeme is determined, *forward* is set to character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

Sentinels

If we use the scheme of Buffer pairs 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.**

Note that **EOF** retains its use as a marker for the end of the entire input. Any **EOF** that appears other than at the end of a buffer means that the input is at an end.

3.1.5 LEXICAL ANALYSIS

TOKEN, LEXEME, PATTERN

TOKEN

- Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) Keywords 3) Operators 4) Special symbols 5) Constants.

LEXEME

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A lexeme is identified by the lexical analyzer as an instance of the token.

PATTERN

- A pattern is a description of the form the lexemes of a token may take.
- Example: Keywords – the pattern is just the sequence of characters that form the token.

Examples of Tokens

Token	lexeme	pattern
const	c,co, con, cons, const	const
if	i, if	if
Relational Operator	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or
Identifier	pi , name, a, n	Letter followed by letter or digits
Number	3.14	any numeric constant
Literal	"core"	pattern

Attributes for Tokens

- When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent phases additional information about the particular lexeme that matched.
- The lexical analyzer collects information about tokens into their associated attributes.
- The token influence parsing decision; the attribute influence the translation of tokens.
- An ID token has usually only a single attribute- a *pointer* (*index*) to the symbol-table entry in which the information about the token is kept.
- The Lexical analyzer produces as output a token of the form,

< Token name, Attribute value >

Where

Token name – is an abstract symbol representing a kind of lexical unit.

Attribute value – an optional attribute value providing additional information to the parser.

Lexeme	**Token Name**	**Attribute Value**
Any white space	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to symbol table entry
Any number	number	pointer to symbol table entry
<	relop	LT

Let C = A + B; is any statement in the source program, the tokens are

<ID, Pointer to ST Entry for C>
<Assign_OP>
<ID, Pointer to ST Entry for A>
<Arith_OP>
<ID, Pointer to ST Entry for B>

3.1.6 Specification of Tokens

- Regular Expressions are an important notation for specifying lexeme patterns.
- Regular expression (R.E.) is a mathematical symbolism which is used to describe the representation of strings by using defining rules.
- The REs are built recursively out of smaller regular expressions using the rules and properties.
- Some of the algebraic properties that hold for arbitrary regular expression r, s and t

$r \mid s = s \mid r$	$\rightarrow$ | is Commutative
$r \mid (s\|t) = (r\|s)t$	$\rightarrow$ | is Associative
$r(st) = (rs)t$	$\rightarrow$ Concatenation is associative
$r(st) = rs\|rt;\ (s\|t)r = sr\|st$	$\rightarrow$ Concatenation is distributive over |
$\epsilon r = r\epsilon = r$	$\rightarrow$ ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	$\rightarrow$ ϵ is guaranteed in a closure
$r^{**} = r^*$	$\rightarrow$ * is idempotent.

Strings and Languages

- An alphabet denotes any finite set of symbols,
  - {0,1}: binary alphabet
  - ASCII code: computer alphabet
- A string over some alphabet is a finite sequence of symbols drawn from the alphabet.
- A language denotes a set of strings over some fixed alphabet.
- The string exponentiation operation is defined as $s^0 = \varepsilon$ (empty string);
  $s^i = s^{i-1}s$, for $i>0$ (string concatenation)

3.1.7 Specification of tokens using RE

(1) Pattern for Identifier Token

letter $\rightarrow$ [A-Za-z]
digit $\rightarrow$ [0-9]
id$\rightarrow$ letter (letter|digit)*

(2) Pattern for Integer Constant (signed/unsigned) Token

digit $\rightarrow$ [0-9]
num $\rightarrow$ $(+|-|\epsilon)$ digit$^+$

(3) Pattern for Real constant Token

digit $\rightarrow$ [0-9]
num$\rightarrow$ digit$^+$.digit$^+$ (E[+-]digit$^+$)

(4) Pattern for keywords Token

if → if
main →main
int → int
else → else
void → void

(5) Pattern for Operators Token

rel_op → < | <= |> |>= | == | <>
arith_op → + | - | * | / |%

(6) Pattern for delimiters (whitespace) Token

ws → (blank | tab | newline)$^+$

- R.E. is only one statement rule.
- If the used statement is used to define the basic symbols of the R.E. then this type of statement is called Regular Definition such as the case in Letter and the case in Digit.

3.1.8 Recognition of Token

- The Token recognition is the process of takes the patterns for all the needed tokens and examines the input string and finds a lexeme matching one of the patterns.
- As an intermediate step in construction of lexical analyzer, first convert patterns into stylized flow charts, called "**transition diagrams**".
- Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- Edges are directed from one state to another and are labeled by a symbol or set of symbols.

(1) Recognizer for Identifier Token

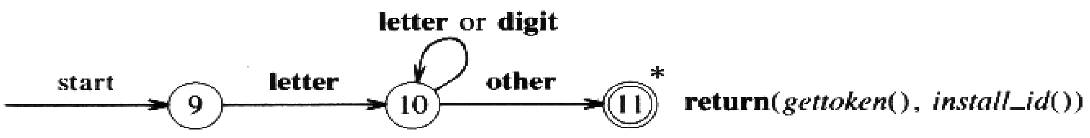
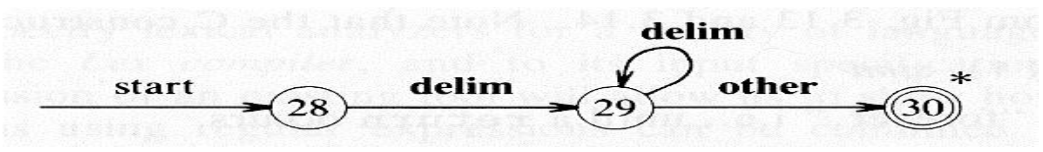


Fig. 3.13. Transition diagram for identifiers and keywords.

(2) Recognizer for delimiters Token



(3) Recognizer for Operators Token (relational operator)

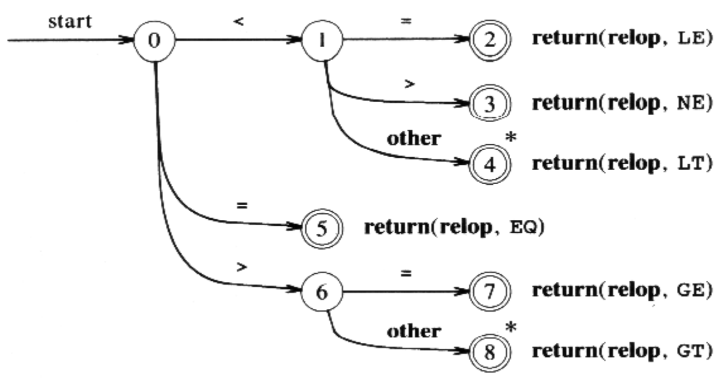
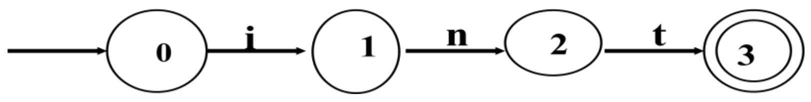


Fig. 3.12. Transition diagram for relational operators.

(4) Recognizer for keyword token - “int”



(5) Recognizer for Literals Token

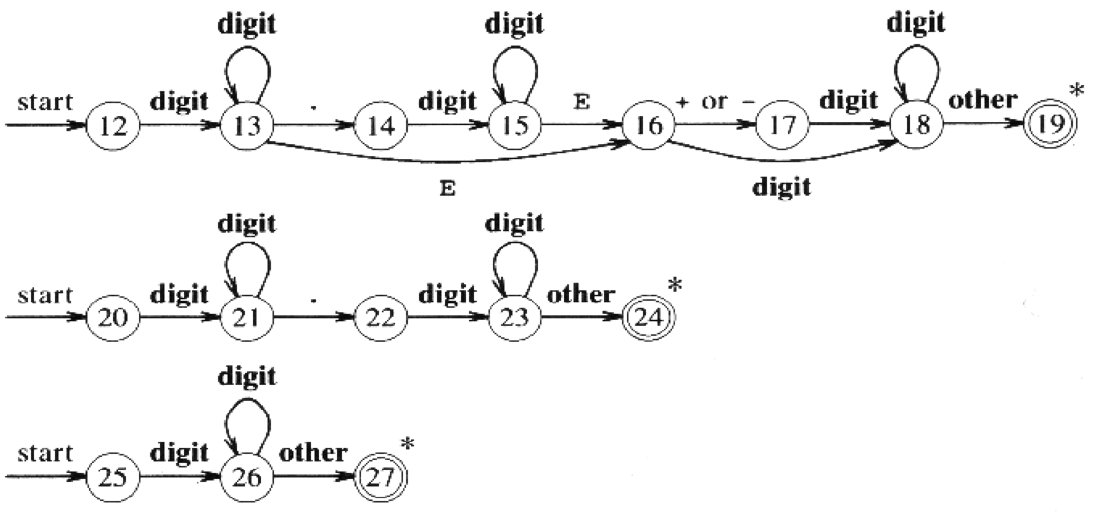
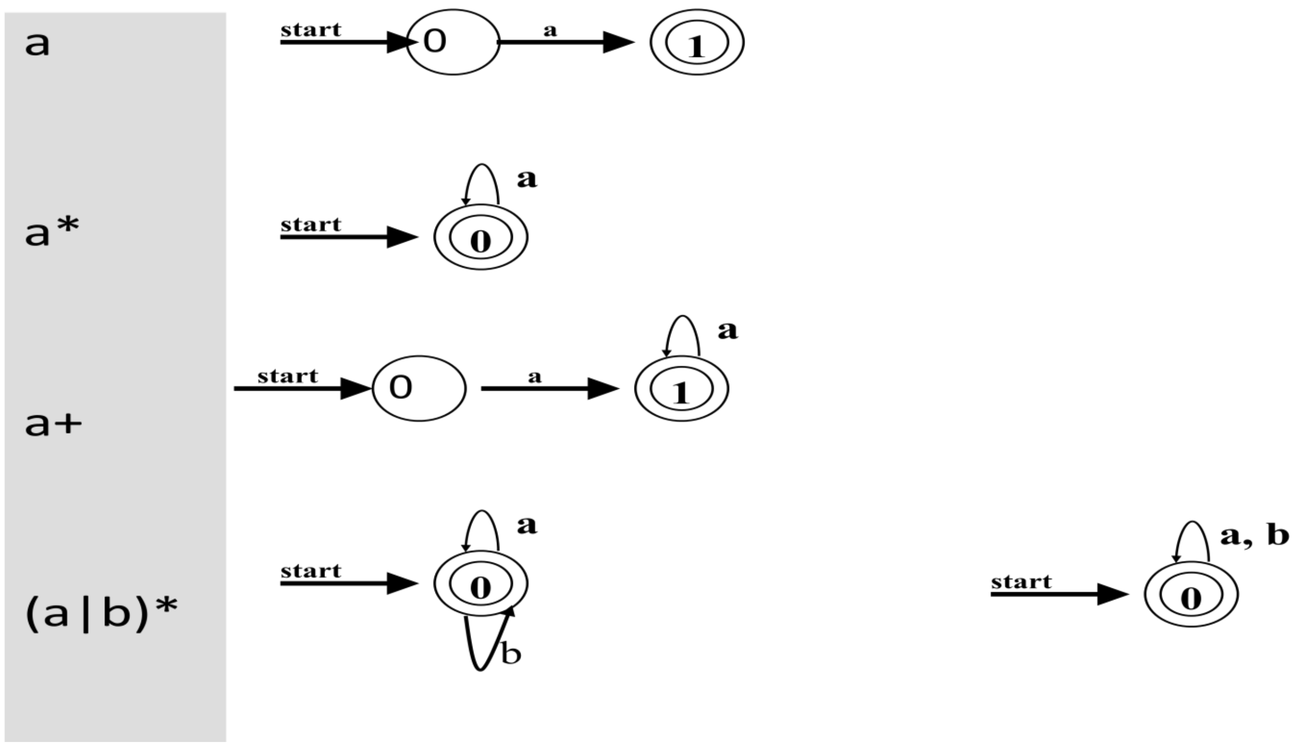


Fig. 3.14. Transition diagrams for unsigned numbers in Pascal.

3.1.9 Finite Automata (FA)

- FA also called Finite State Machine (FSM) ia an abstract model of a computing entity.
- FA can decide whether to accept or reject a string. Every regular expression can be represented as a FA and vice versa
- There are two types of FAs:
  - **Non-deterministic (NFA):** Has more than one alternative action for the same input symbol.
  - **Deterministic (DFA):** Has at most one action for a given input symbol.

- Regular expression is a declarative way to describe the token. It describes what a token is, but not how to recognize the token, whereas the FA is used to describe how the token is recognized.



NFA (Non-deterministic Finite Automaton)

An NFA is a 5-tuple (S, Σ, δ, S0, F)

S: a set of states;

Σ: the symbols of the input alphabet;

δ : a set of transition functions; move(state, symbol) → a set of states

S0: s0 ∈S, the start state;

F: F ⊆ S, a set of final or accepting states.

DFA (Deterministic Finite Automaton)

- A special case of NFA where the transition function maps the pair (state, symbol) to one state.
  - When represented by transition diagram, for each state *S* and symbol *a,* there is at most one edge labeled *a* leaving *S;*
  - When represented transition table, each entry in the table is a single state.
  - There are no ε-transition

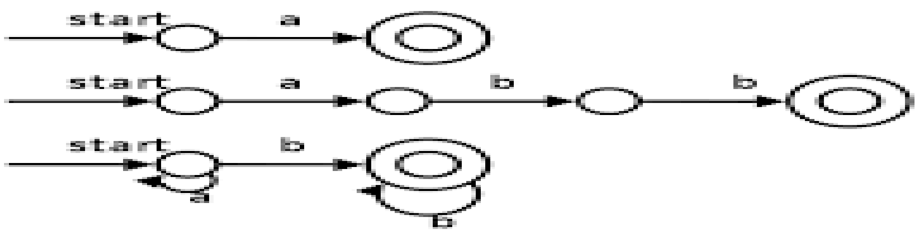
Conversion from NFA to DFA: Algorithm & Example: Please refer Class-work Notebook.

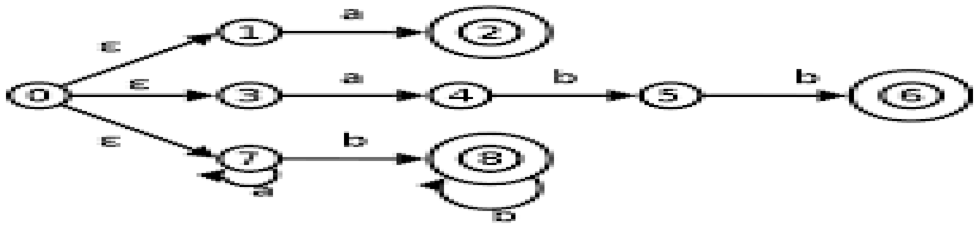
3.2 DESIGN OF LEXICAL ANALYZER GENERATOR

For the LEX program

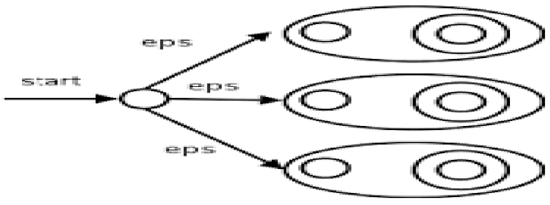
a	**{action A1 for pattern p1}**
abb	**{action A2 for pattern p2}**
a*b+	**{action A3 for pattern p3}**

Step 1: Construct the automaton by taking each regular –expression pattern in the LEX program and converting it to an NFA.

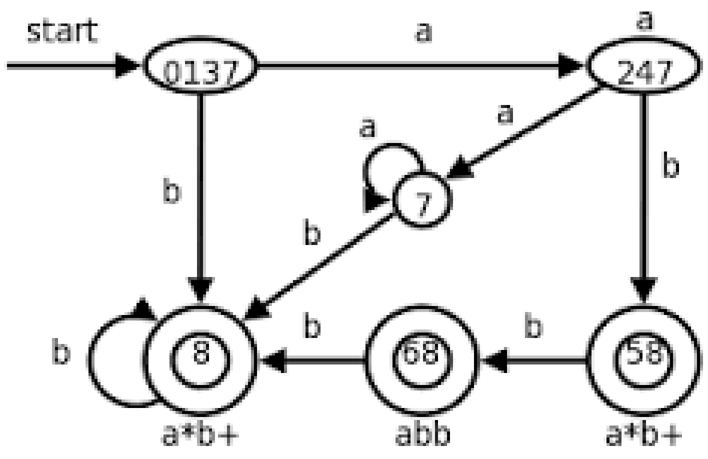




Step 2: Combine all the NFA's into one by introducing a new start state with ϵ – transitions to each start states of the NFA$_i$ for pattern P$_i$.

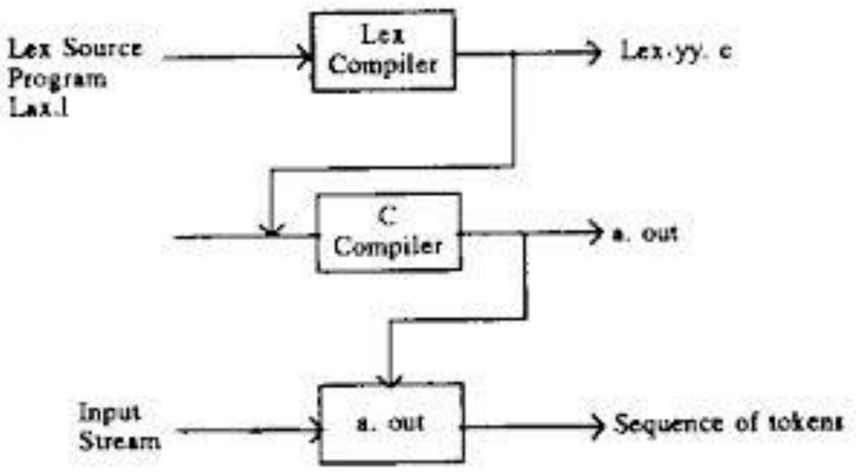


Step 3: Convert the NFA for all the patterns into an equivalent DFA, using Subset Construction method.



3.4 THE LEX TOOL

- LEX is a tool that allows one to specify a lexical analyzer by specifying regular expression to describe patterns for tokens.
- The notation for LEX tool is referred to as the Lex Language and the tool itself is the LEX compiler.
- The LEX Compiler transforms the input patterns into a transition diagram and generates code in a file called lex.yy.c that simulates this transition diagram.



3.4.1 LEX Program – Execution procedure

1. First, a specification of a lexical analyzer is prepared by creating a program lexp.l in the LEX language.
2. The Lexp.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c
3. The program lex.yy.c consists of a table constructed from the Regular Expressions of Lexp.l, together with standard routines that uses the table to recognize lexemes.
4. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

3.4.2 LEX specifications / Program Structure

A Lex program (the .l file) consists of three parts:

declarations

%%
translation rules
%%

auxiliary procedures

1. The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.

2. The translation rules of a Lex program are statements of the form

pattern 1 ***{action 1}***
pattern 2 ***{action 2}***
…….

where each pattern is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In LEX the actions are written in C.
The *rules* portion of the input contains a sequence of rules. Each rule has the form
pattern ***{ action }***
where:

pattern describes a pattern to be matched on the input
pattern must be un-indented
action must begin on the same line. For multi lined action: use { }

3. The third section holds whatever auxiliary procedures are needed by the actions. Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

3.4.2. Pattern Matching Primitives used in LEX

Meta character	Matches
.	any character except newline
\n	newline
*****	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a or b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal "**a+b**" (C escapes still work)
[ ]	character class

Examples:

Pattern	Matches the input
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

3.4.3 LEX – Predefined Variables

Name	Function
int yylex(void)	Function call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

3.4.4 Conflicts Resolution in LEX

When several prefixes of the input match one or more patterns then
1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the LEX program.

Sample LEX programs: Please refer your class-work note-book & CC Practical - Journal.