



ST. ANNE'S COLLEGE OF ENGINEERING AND TECHNOLOGY

Approved by AICTE, New Delhi. Affiliated to Anna University, Chennai

Accredited by NAAC

ANGUCHETTPALAYAM, PANRUTI – 607 106.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3501 COMPILER DESIGN

UNIT I INTRODUCTION TO COMPILERS & LEXICAL ANALYSIS

Introduction- Translators- Compilation and Interpretation- Language processors -The Phases of Compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Finite Automata – Regular Expressions to Automata NFA, DFA – Minimizing DFA - Language for Specifying Lexical Analyzers – Lex tool.

1. What is a compiler? Explain about the different types of software tools available for analyzing. (May/June-14)

TRANSLATOR

A translator is a program that takes as input a program **written in one language** and produces as output a **program in another language**. Beside program translation, the translator performs another very important role, the error-detection. Any violation of the HLL specification would be detected and reported to the programmers. Important roles of a translator are:

1. Translating the **HLL program input into an equivalent ML program**.
2. Providing diagnostic messages wherever the programmer violates specification of the HLL.

TYPE OF TRANSLATORS:-

- INTERPRETOR
- COMPILER
- PREPROCESSOR

COMPILER (5 MARKS) :-

Compiler is a translator program that **translates a program written in (HLL)** the source program and **translates it into an equivalent program in (MLL)** the target program. As an important part of a compiler is error showing to the programmer.

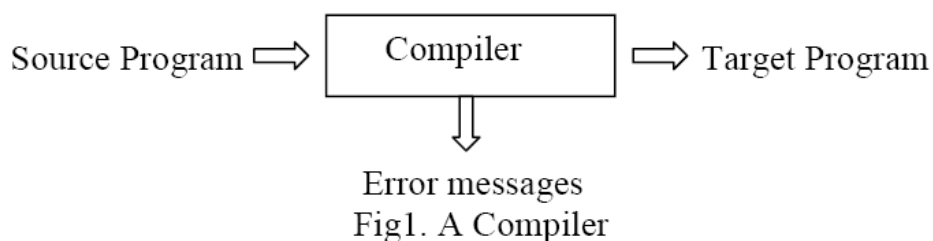


Fig1. A Compiler

Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into **object program**. Then the results object program is **loaded into a memory** executed.

Analysis – Synthesis Model of Compilation:

The process of compilation has two parts namely:

- ☐ Analysis
- ☐ Synthesis

Analysis: The analysis part breaks up the source **program into constituent pieces** and creates an intermediate representation of the source program. The analysis part is often called the **front end** of the compiler

Synthesis:

The synthesis part **constructs the desired target** program from the intermediate representation. The synthesis part is the **back end** of the compiler.

Software Tools:

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

- Structure Editors
- Pretty printers
- Static Checkers
- Interpreters

- **Structure Editors**

- A structure editor takes as input a sequence of commands **to build a source program**.
- The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- Example – while do and begin..... end.

- **Pretty printers**

- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- For example, comments may appear in a special font.

- **Static Checkers**

- A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- For example, a static checker may detect that parts of the source program can never be executed.

- **Interpreters**

- Translate from high level language (BASIC, FORTRAN, etc..) into assembly or machine language. Interpreters are frequently used to execute command language.
- An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
- Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.

Examples of Compiler:

The following examples are similar to that of a conventional compiler.

- Text formatters
- Silicon Compiler
- Query interpreters

Text formatters:

It takes input as a stream of characters includes commands to indicate paragraphs, figures.

Silicon Compiler:

Variables represent logical signal 0 & 1 and output is a circuit design.

Query interpreters: It translates a predicate containing relational & Boolean operators into commands to search a database.

2. What advantages are there to a language processing system in which the compiler produces assembly language rather than machine languages. OR

Explain language processing system with neat diagram. (8) May/June 2016

LANGUAGE PROCESSORS

COMPILER :- Compiler is a translator program that **translates** a program written in **(HLL)** the source program and translates it into an equivalent program in **(MLL)** the target program. As an important part of a compiler is error showing to the programmer.

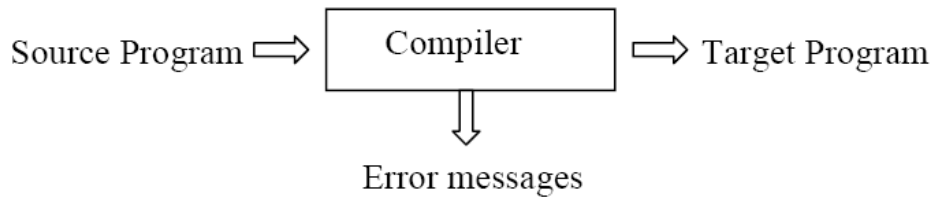
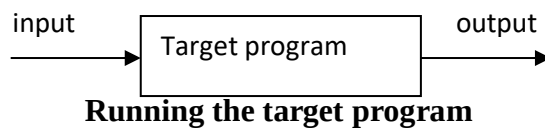


Fig1. A Compiler

Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into a **object program**. Then the results object program is **loaded into a memory executed**.

If the target program is an executable machine- language program. It can then be called by the user to process inputs and produce output see fig. below



Running the target program

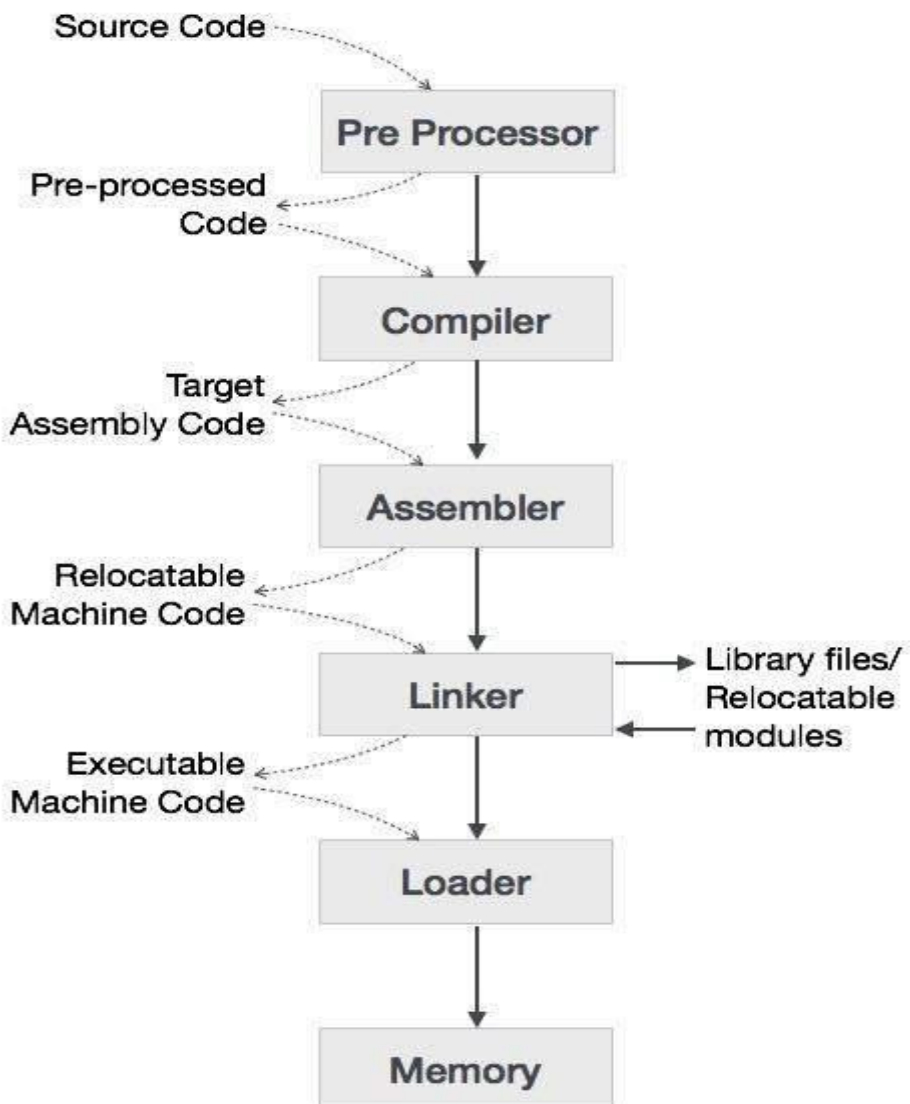
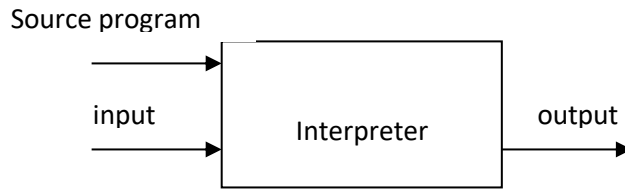


Fig. A language- processing System

INTERPRETOR

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



An Interpreter

PREPROCESSOR:

A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions

1. Macro processing
2. File Inclusion
- 3."Rational Preprocessors
4. Language extension

ASSEMBLER

Assembler translates assembly language program, which uses mnemonic names for operation codes and data addresses into the machine language.

LINKERS AND LOADERS:

- The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.
- A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

3.What is a compiler? Explain the various phases of compiler with a neat sketch.(Nov/Dec 2016)
(OR)Explain the structure of the compiler.(OR)Discuss about the analysis-synthesis model of the compiler with neat diagram.(OR)Describe the various phases of complier and trace the program

segment 4 : * + = c b a for all phases.(OR)Explain the need for dividing the compilation process into various phases and explain its functions.

[May/June 2013] [May/June 2012] [Apr/May 2011] [Nov/Dec 2011] [May/June 2009] [Nov/Dec 2007] [May/June-14] Or Describe the various phases of compiler and trace it with the program segment (position := initial + rate*60). (16)

STRUCTURE OF THE COMPILER DESIGN (10 MARKS):-

Compiler Definition- Compiler is a translator program that **translates** a program written in **(HLL)** the source program and translates it into an equivalent program in **(MLL)** the target program. As an important part of a compiler is error showing to the programmer

Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:

- 1) Lexical analysis
- 2) Syntax analysis

- 3) Semantic analysis
- 4) Intermediate code generation
- 5) Code optimization
- 6) Code generation

Sub-Phases:

- 1) Symbol table management
- 2) Error handling

A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by a subsequent pass. In an implementation of a compiler, portions of one or more phases are combined into a module called pass.

ANALYSIS PHASE

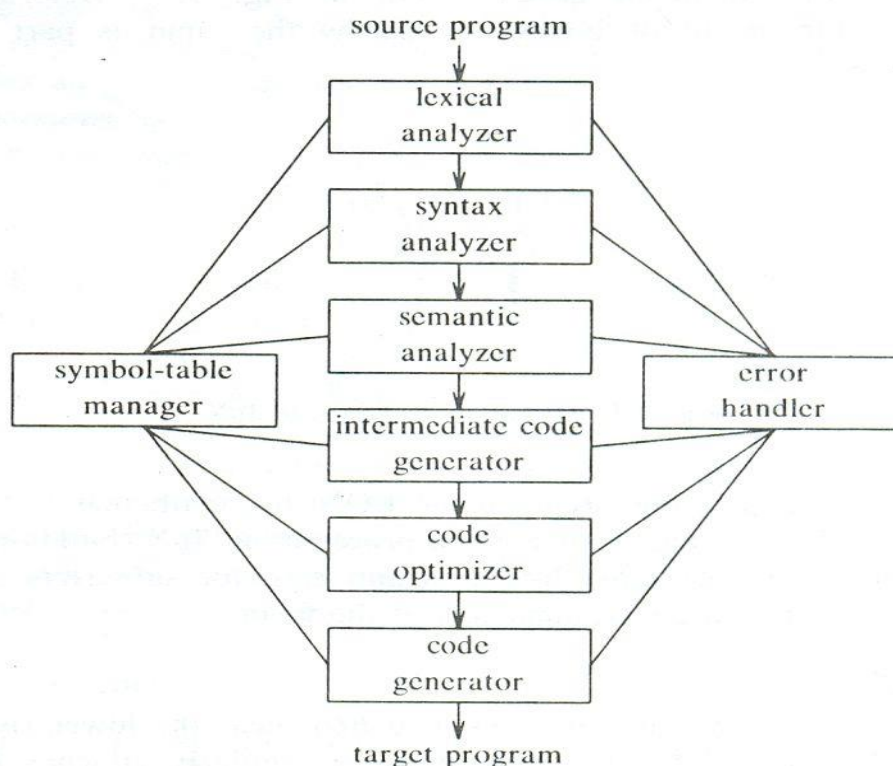


Fig. 1.9. Phases of a compiler.

LEXICAL ANALYSIS:

- It is the **first phase** of the compiler. It gets input from the source program and **produces tokens as output**.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Consider the statement,

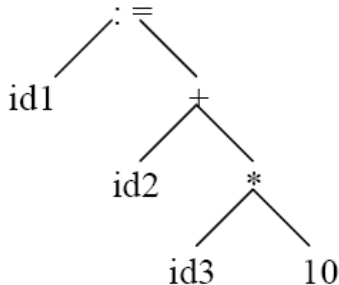
position := initial + rate * 10

- Group of characters forming a token is called the **Lexeme**.
- The representation of the statement given above after the lexical analysis would be:
- **id1, :=, id2, +, id3, *, 10**

- The lexical analyzer not only generates a token but also enters the lexeme into the symbol table if it is not already there.

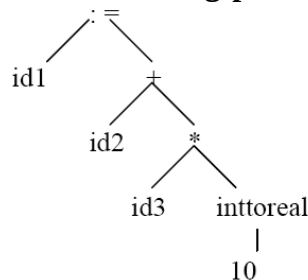
SYNTAX ANALYSIS:

- It is the **Second phase** of the compiler.
- The tokens from the lexical analyzer are grouped hierarchically into nested collections with collective meaning called “**Parse Tree**” followed by syntax tree as output.
- A Syntax Tree is a compressed representation of the parse tree in which the operators appears as interior nodes & the operands as child nodes.



SEMANTIC ANALYSIS:

- It is the **third phase** of the compiler.
- An important part of semantic analysis is **type checking**, where the compiler checks that each operator has matching operands.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may **convert the integer into a floating-point number**.



INTERMEDIATE CODE GENERATION:

- It is the **fourth phase** of the compiler.
- It gets input from the semantic analysis and converts the input into **output as intermediate code** such as three-address code, postfix notation etc.
- The three-address code consists of a sequence of instructions, each of which has atmost three operands.
- **Example: t1=t2+t3**
- The intermediate representation should have two important properties:
 - it should be easy to produce and
 - it should be easy to translate into the target machine
- We consider an intermediate form called three-address code, which consists of a sequence of assembly like instructions with three operands per instruction.

Properties of three-address instructions:

1. Each three-address assignment instruction has at most one operator on the right side.
2. The compiler must generate a temporary name to hold the value computed by a three-address instruction.
3. Some "three-address instructions may have fewer than three operands

In three-address code, the source program might look like this,

```
temp1: = inttoreal (10)
temp2: = id3 * temp1
temp3: = id2 + temp2
id1: = temp3
```

THE SYNTHESIS PHASE

CODE OPTIMIZATION:

- It is the **fifth phase** of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
 - deduction and removal of dead code (unreachable code).
 - calculation of constants in expressions and terms.
 - collapsing of repeated expression into temporary string.
 - loop unrolling.
 - moving code outside the loop.
 - removal of unwanted temporary variables.

The output will look like this:

```
temp1: = id3 * 10.0
id1: = id2 + temp1
```

CODE GENERATION:

- It is the **final phase** of the compiler.
- The code generator takes as input an intermediate representation of the source program and maps it into the target language
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
 - allocation of register and memory
 - generation of correct references
 - generation of correct data types
 - generation of missing code

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.

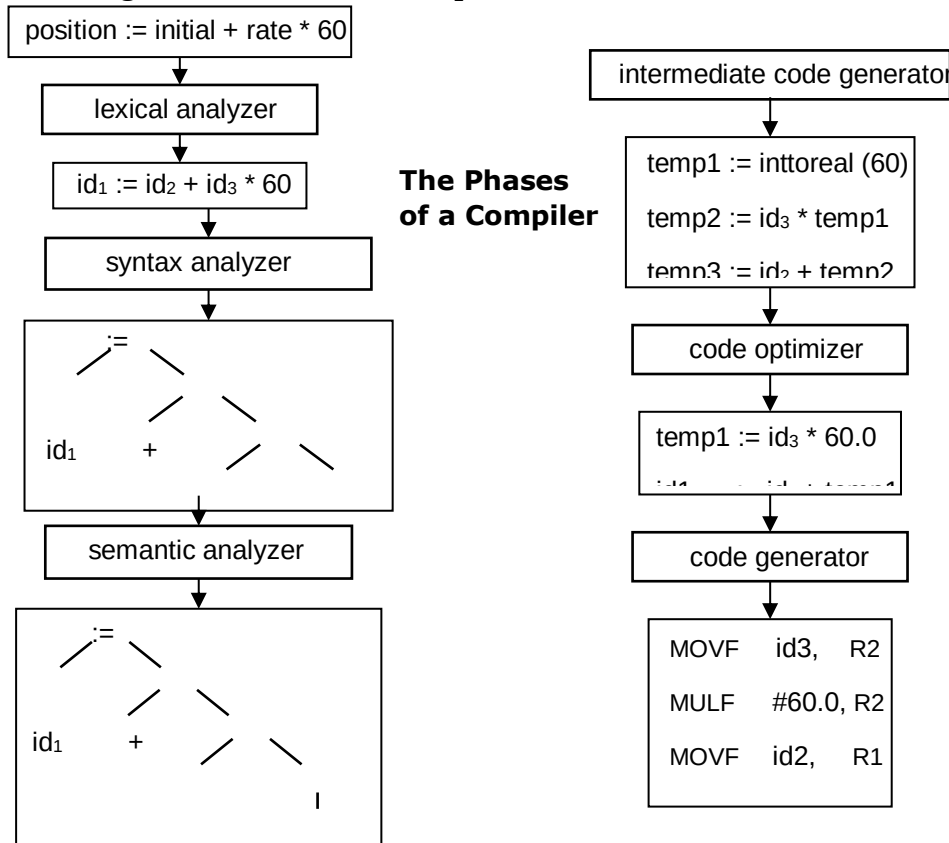
```
MOVF id3, R2
MULF #10.0, R2
MOVF id2 , R1
ADDF R2, R1
MOVF R1, id1
```

SYMBOL TABLE MANAGEMENT:

- Symbol table is used **to store all the information about identifiers** used in the program.
- It is a data structure containing a **record for each identifier**, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.

- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

The figure shows the representation of this statement after each phase:



ERROR HANDLING:

- Each phase can encounter errors. After **detecting an error**, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.
- To illustrate the translation of source code through each phase, consider the statement `Position := initial + rate * 10`.

1. Explain briefly about Errors Encountered in Different Phases.

Or explain various Errors encountered in different phases of compiler. (4) (8) May/June2016 (Nov/Dec 2016)

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- Lexical** : name of some identifier typed incorrectly
- Syntactical** : missing semicolon or unbalanced parenthesis
- Semantical** : incompatible value assignment

- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

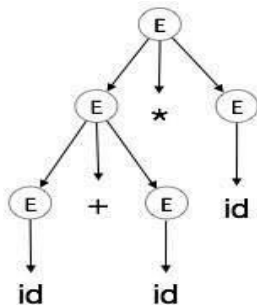
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

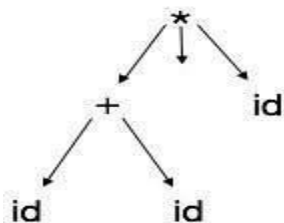
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

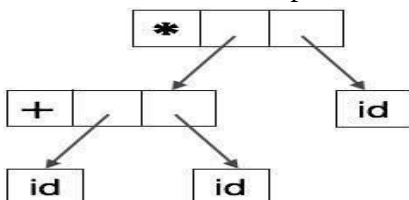
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

1. **Write note on front and back end of compiler. Or**

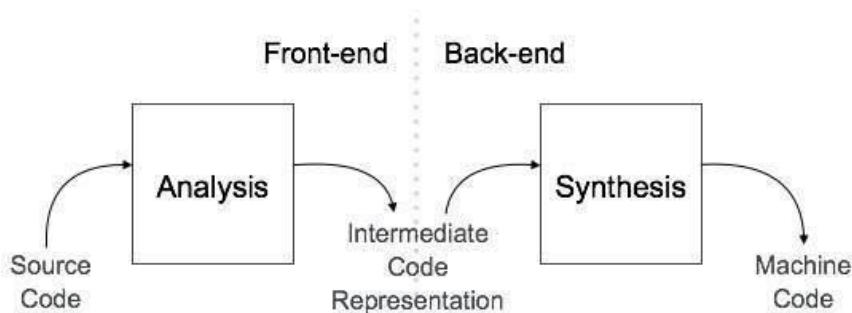
Explain the need for grouping of phases. (4) (8) (May/June 2016) (Nov/Dec 2016)

GROUPING OF PHASES

Definition:

Activities from more than one phase are often grouped together. The phases are collected into a front end and a back end

- **Front End**
- **Back End**



Front End: analysis (machine independent)

- The Front End consists of those phases or parts of phases that depends primarily on the source language and is largely **independent of target machine**.
- Lexical and syntactic analysis, symbol table, semantic analysis and the generation of intermediate code is included.
- Certain amount of code optimization can be done by the front end.
- It also includes error handling that goes along with each of these phases.

Back End: synthesis (machine dependent)

- The Back End includes those portions of the compiler that depend on the target machine and these portions **do not depend on the source language**.
- Find the aspects of code optimization phase, code generation along with necessary error handling and symbol table operations.

Compiler passes

- **Passes:** Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.
- ❖ **Single pass:** usually requires everything to be defined before being used in source program.
- ❖ **Multi pass:** compiler may have to keep entire program representation in memory.
- It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass.
- Eg: Lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.
- **Reducing the number of passes:**
 - It is desirable to have relatively few passes, since it takes time to read and write intermediate files.
 - If we group several phases into one pass, we may forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it.

6. Explain construction tools in compiler. [May/June 2012] [Apr/May 2011] [Nov/Dec 2011][Nov/Dec2016]

COMPILER CONSTRUCTION TOOLS

Definition:

These tools have been developed for helping implement various phases of a compiler. These systems have often been referred to as compiler-compilers, compiler- generators or translator writing systems.

Some commonly used compiler-construction tools include

1. Parser generator
2. Scanner generator
3. Syntax-directed translation engine
4. Data flow engine
5. Automatic code generator

I) Scanner generators:

Generates

Input(Regular Expression)----- Output(Lexical Analyzer)

- Automatically generates lexical analyzers from a specification based on regular expression.
- The basic organization of the resulting lexical analyzers is finite automation.

II) Parser generators:

Generates

Input(Context Free Grammar)----- Output(Syntax Analyzer)

- Produce syntax analyzers from input that is based on context-free grammar.
- Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand.
- It consumes a large fraction of the running time of a compiler.
Example-YACC (Yet Another Compiler-Compiler).

III) Syntax-directed translation engines:

Generates

Input(Parse or Syntax Tree)----- Output(Intermediate code)

- Produce collections of routines that walk a parse tree and generating intermediate code.
- The basic idea is that one or more “translations” are associated with each node of the parse tree.
- Each translation is defined in terms of translations at its neighbor nodes in the tree.

IV) Data-flow analysis engines:

Generates

Input(Intermediate code)----- Output(Optimized code)

- 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 from Intermediate codes.

V) Automatic Coder generators:

Generates

Input(Optimized code)----- Output(Object Code)

- A tool takes a collection of rules that define the translation of each operation of the optimized language into the machine language for a target machine.
- The rules must include sufficient details that we can handle the different possible access methods for data.
- Eg. Variables may be in registers, in a fixed location in memory or may be allocated a position on a stack.
- The basic technique is “template matching”. The intermediate code statements are replaced by templates.
- That template represents sequences of machine instructions.
- The assumptions about storage of variables match from template to template.

1. Explain briefly about Programming Language Basics

1. The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. 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. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.

One issue on which we shall concentrate is the scope of declarations. The scope of a declaration of x is the region of the program in which uses of x refer to this declaration. A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x .

Most languages, such as C and Java, use static scope.

Example 1: As another example of the static/dynamic distinction, consider the use of the term "static" as it applies to data in a Java class declaration. In Java, a variable is a name for a location in memory used to hold a data value. Here, "static" refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory where the declared variable can be found. A declaration like

public static int x ;

makes x a class variable and says that there is only one copy of x , no matter how many objects of this class are created. Moreover, the compiler can determine a location in memory where this integer x will be held. In contrast, had "static" been omitted from this declaration, then each object of the class would have its own location where x would be held, and the compiler could not determine all these places in advance of running the program

2. Environments and States

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as $x = y + 1$ changes the value denoted by the name x . More specifically, the assignment changes the value in whatever location is denoted by x .

It may be less clear that the location denoted by x can change at run time. For instance, as we discussed in Example 1.3, if x is not a static (or "class") variable, then every object of the class has its own location for an instance of variable x . In that case, the assignment to x can change any of

those "instance" variables, depending on the object to which a method containing that assignment is applied.

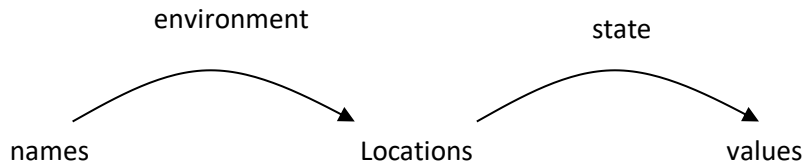


Figure 1 .8: Two-stage mapping from names to values

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 (see Fig. 1 .8) :

1. The environment is a mapping from names to locations in the store. Since variables refer to locations ("l-values" in the terminology of C) , we could alternatively define an environment as a mapping from names to variables.
2. The state is a mapping from locations in store to their values. That is, the state maps l-values to their corresponding r-values, in the terminology of C.

Environments change according to the scope rules of a language.

3. Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like **public**, **private**, and **protected**.

In this section we consider static-scope rules for a language with blocks, where a block is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of begin and end for the same purpose dates back to Algol.

4. Explicit Access Control

Classes and structures introduce a new scope for their members. If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition. In analogy with block structure, the scope of a member declaration x in a 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, object oriented languages such as C++ or Java provide explicit control over access to member names in a superclass. These keywords support encapsulation by

restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name x associated with the class C may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

5. Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term dynamic scope, however, usually 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. Dynamic scoping of this type appears only in special situations. We shall consider two

examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

6. Parameter Passing Mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest .

Call- by-Reference

In call-by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference IS used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call-by-Name

A **third mechanism** - call-by-name - was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (**with renaming of local names in the called procedure**, to keep them distinct) . When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

7. Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that **two formal parameters can refer to the same location**; such variables are said to be aliases of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

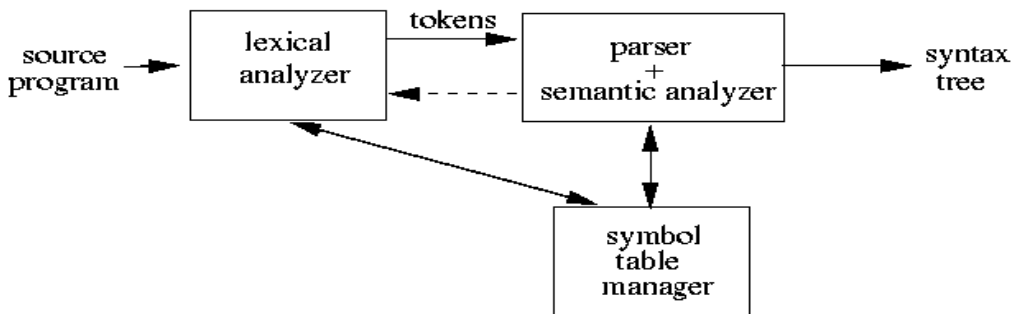
PART-B

1. What is a compiler? Explain the various phases of compiler in detail, with a neat sketch. [Q.No. 3]
1. Explain in detail about the role of the lexical analyzer with the possible error recovery actions. [May/June 2013] [May/June 2012] [Apr/May 2011] [Nov/Dec 2007] [May/June-14] Or
Discuss the role of lexical analyzer in detail with necessary examples. (Nov/Dec 2016) (8)

LEXICAL ANALYSIS

THE ROLE OF THE LEXICAL ANALYZER

- The lexical analyzer is the first phase of compiler.
- Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis.
- As in the figure, upon receiving a “get next token” command from the parser the lexical analyzer reads input characters until it can identify the next token.



- Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is **stripping out from the source program comments and white space in the form of blank, tab, and new line character.**
- Another is correlating error messages from the compiler with the source program.
- Sometimes lexical analyzers are divided into a cascade of two phases, the first called “scanning” and the second “lexical analysis”.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

- 1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
- 2) Compiler efficiency is improved.
- 3) Compiler portability is enhanced.

Tokens, Patterns and Lexemes.

Token:

A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except”

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

The following are the error-recovery actions in lexical analysis:

1) Deleting an extraneous character.

2) Inserting a missing character.

3) Replacing an incorrect character by a correct character.

4) Transforming two adjacent characters.

5) **Panic mode recovery:** Deletion of successive characters from the token until error is resolved.

1. Differentiate between lexeme, token and pattern.(6) May/June2016**Tokens**

A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,>,>=	< or <= or = or > or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except”

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

2. What are the issues in lexical analysis?(4) May/June2016

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

- 1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
- 2) Compiler efficiency is improved.
- 3) Compiler portability is enhanced.

3. Explain briefly about specification of token.

(or) Explain briefly about Expressing Tokens by Regular Expressions

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

- 1) Strings
- 2) Language
- 3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

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

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s , usually written $|s|$, is the number of occurrences of symbols in s .

For example, banana is a string of length six. The empty string, denoted ϵ , is the string of length zero.

Operations on strings

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 string s .

For example, ban is a prefix 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 is a suffix of banana.

3. A **substring** of s is obtained by deleting any prefix and any suffix from s .

For example, nan is a substring 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.

Operations on languages:

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The following example shows the operations on strings:

Let $L = \{0,1\}$ and $S = \{a,b,c\}$

1. Union : $L \cup S = \{0,1,a,b,c\}$
2. Concatenation : $L.S = \{0a,1a,0b,1b,0c,1c\}$
3. Kleene closure : $L^* = \{\epsilon, 0, 1, 00, \dots\}$

4. Positive closure : $L^+ = \{0, 1, 00, \dots\}$

Regular Expressions

Each regular expression r denotes a language $L(r)$.

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in Σ , then 'a' is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with 'a' in its one position.
3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,
 - a) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
 - b) $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
 - c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
 - d) (r) is a regular expression denoting $L(r)$.
4. The unary operator $*$ has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. $|$ has lowest precedence and is left associative.

Regular set

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write $r = s$.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, $r|s = s|r$ is commutative; $r|(s|t) = (r|s)|t$ is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.....

$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

$\text{letter} \rightarrow A | B | \dots | Z | a | b | \dots | z |$

$\text{digit} \rightarrow 0 | 1 | \dots | 9$

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

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. One or more instances (+):

- The unary postfix operator $+$ means "one or more instances of".
- If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $+$ has the same precedence and associativity as the operator $*$.

2. Zero or one instance (?):

- The unary postfix operator ? means “zero or one instance of”.
- The notation $r?$ is a shorthand for $r \mid \epsilon$.
- If ‘ r ’ is a regular expression, then $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$.

3. Character Classes:

- The notation $[abc]$ where a , b and c are alphabet symbols denotes the regular expression $a \mid b \mid c$.
- Character class such as $[a-z]$ denotes the regular expression $a \mid b \mid c \mid d \mid \dots \mid z$.
- We can describe identifiers as being strings generated by the regular expression, $[A-Za-z][A-Za-z0-9]^*$

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set.

Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

4. How do you recognize a token? [May/June 2013] [May/June 2012] [Apr/May 2011]

Or Discuss how finite automata is used to represent tokens and perform lexical analysis with examples. (Nov/Dec 2016)

RECOGNITION OF TOKENS

Consider the following grammar fragment:

```

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

```

A grammar for branching statements

Example:

For relop, we use the comparison operators of languages like Pascal or SQL, where $=$ is "equals" and $<>$ is "not equals," because it presents an interesting structure of lexemes. The terminals of the grammar, which are `if`, `then`, `else`, `relop`, `id`, and `number`, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions, as in Fig. 3.11.

The patterns for `id` and `number` are similar

```

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

```

Patterns for tokens

For this language, the lexical analyzer will recognize the keywords `if`, `then`, and `else`, as well as lexemes that match the patterns for `relop`, `id`, and `number`. To simplify matters, we make the common

assumption that keywords are also reserved words : that is , they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" ws defined by:

ws -t (blank | tab | newline)+

Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that , when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

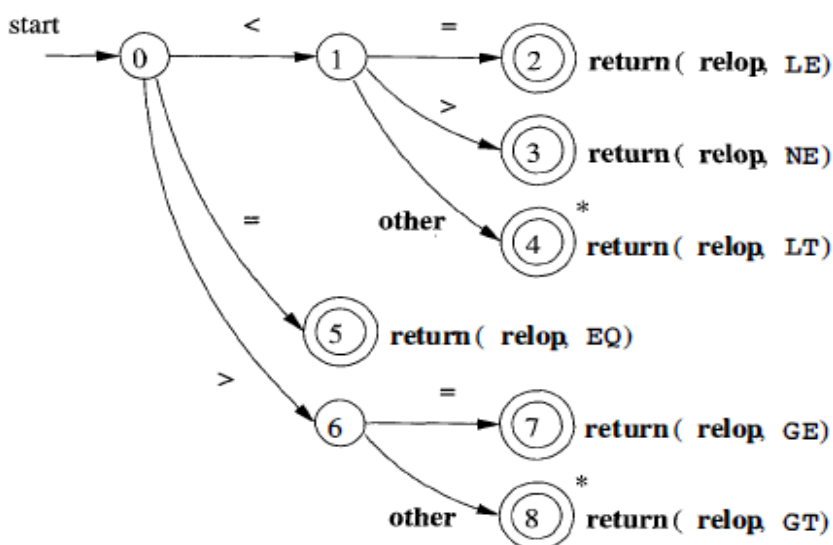
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	-	-
if	if	-
then	then	-
else	else	-
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Tokens, their patterns, and attribute values

1. Transition diagrams

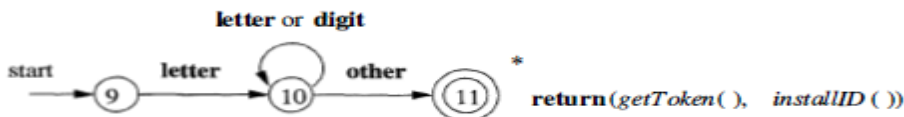
It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

Transition diagram for relop

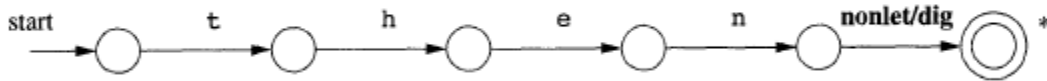


2. Recognition of reserved words and Identifiers

A transition diagram for id's and keywords



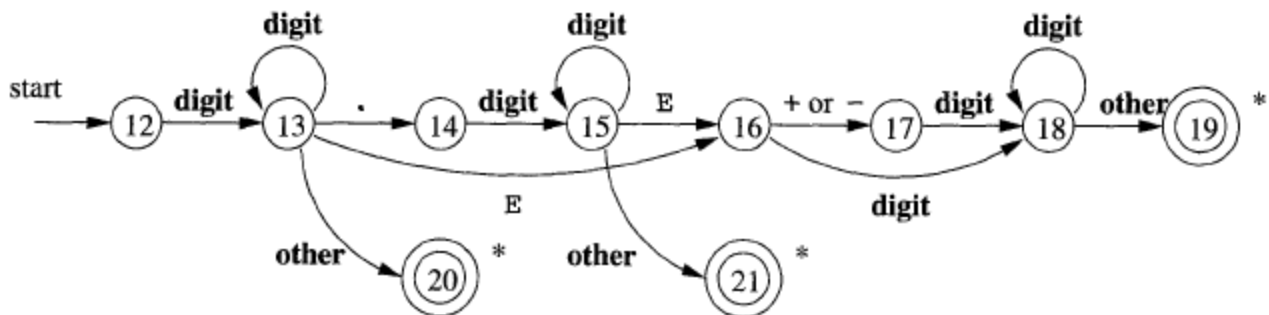
Hypothetical transition diagram for the keyword then



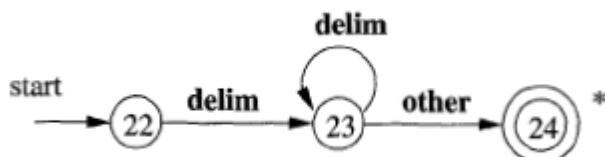
3. Completion of the Running Example

Beginning in state 12, if we see a digit, we go to state 13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in the form of an integer; 123 is an example. That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered. These mechanics are not shown on the diagram but are analogous to the way we handled identifiers

Transition diagram for unsigned numbers in pascal



A transition diagram for whitespace



4. Architecture of a Transition-Diagram-Based Lexical Analyzer

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. 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 that state. Often, the code for a state is itself a switch statement or multi way branch that determines the next state by reading and examining the next input character.

5. Write notes on regular expressions. (6) May/June2016

Regular Expressions

Each regular expression r denotes a language $L(r)$.

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.
2. If 'a' is a symbol in Σ , then 'a' is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with 'a' in its one position.
3. Suppose r and s are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,
 - a) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
 - b) $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
 - c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
 - d) (r) is a regular expression denoting $L(r)$.
4. The unary operator $*$ has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. $|$ has lowest precedence and is left associative.

Regular set

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write $r = s$.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, $r|s = s|r$ is commutative; $r|(s|t) = (r|s)|t$ is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.....

$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

$\text{letter} \rightarrow A | B | \dots | Z | a | b | \dots | z |$

$\text{digit} \rightarrow 0 | 1 | \dots | 9$

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

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. One or more instances (+):

- The unary postfix operator $+$ means "one or more instances of".
- If r is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $+$ has the same precedence and associativity as the operator $*$.

2. Zero or one instance (?):

- The unary postfix operator $?$ means "zero or one instance of".
- The notation $r?$ is a shorthand for $r | \epsilon$.
- If 'r' is a regular expression, then $(r)?$ is a regular expression that denotes the language

$L(r) \cup \{\epsilon\}$.

3. Character Classes:

- The notation $[abc]$ where a , b and c are alphabet symbols denotes the regular expression

$a | b | c$.

- Character class such as $[a-z]$ denotes the regular expression $a | b | c | d | \dots | z$.

- We can describe identifiers as being strings generated by the regular expression,

$[A-Za-z][A-Za-z0-9]^*$

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set.

Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

A regular expression is a string that describes the whole set of strings according to certain syntax rules. Regular expressions **serve as the input language for many systems that process strings**. These expressions are used by many text editors and utilities to search bodies of text for certain patterns etc.

Regular expressions denote languages. For Example - The regular expression 01^*+10^* denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's.

The three operators involved in the Regular Expression are,

- (i) The **Union** of two languages L and M , denoted by $L \cup M$, is the set of strings that are either in L or M , or both.

Example: If $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L \cup M = \{\epsilon, 10, 001, 111\}$

- (ii) The **Concatenation** of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M .

Example: If $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$ then

$L.M$ (Or) $LM = \{001, 10, 111, 001001, 10001, 111001\}$

- (iii) The **closure** of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L , possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them.

Example: If $L = \{0,1\}$, then L^* is all strings of 0's and 1's

If $L = \{0,11\}$, then L^* consists of those strings of 0's and 1's such that the 1's come in pairs, e.g., 011, 11110, and ϵ , but not 01011 or 101.

The regular expression involving the **Closure** operation and its types are,

- Kleene or star closure
- Positive closure

6. Explain Briefly about Finite Automata

Finite Automata

A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.

The generalized transition for regular expression is called finite automaton. It is a labeled directed graph. Hence the nodes are the states and the labeled edges are transitions.

The automata can be grouped into two as given below:

1. Non-deterministic Finite Automata (NFA)
2. Deterministic Finite Automata (DFA)

"Non-deterministic" means that more than one transition out of a state may be possible on the same input symbol.

Both finite automata are capable of recognizing precisely the regular sets.

- Finite Automata are used as a model for:

- Software for designing digital circuits
- Lexical analyzer of a compiler
- Searching for keywords in a file or on the web.
- Software for verifying finite state systems, such as communication protocols.

1. Non-Deterministic Finite Automata

A non-deterministic finite automaton (NFA) is a mathematical model that consists of

1. A set of states S .
2. A set of input symbols Σ (the input symbol alphabet).
3. A transition function δ that maps state-symbol pairs to sets of states.
4. A state s_0 that is distinguished as the start (or initial) state.
5. A set of states F distinguished as accepting (or final) states.

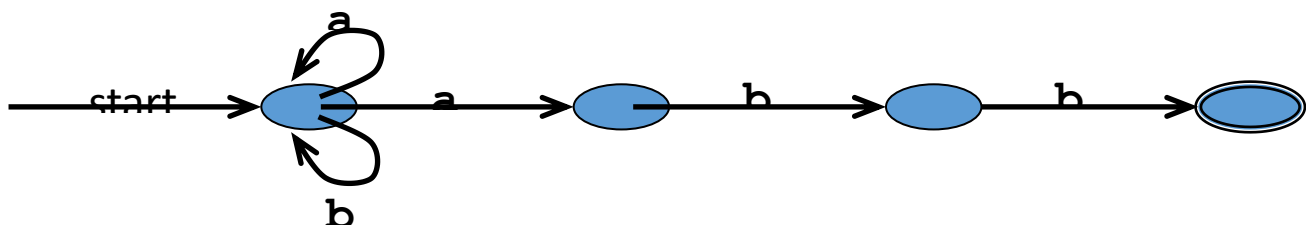
An NFA can be represented diagrammatically by a labeled directed graph called a transition graph, in which the nodes are the states and the labeled edges represent the transition function.

This graph looks like a transition diagram but the same character can label two or more transitions out of one state, and edges can be labeled by the special symbol ϵ as well as input symbols.

(eg) NFA that recognizes the language $(a/b)^*abb$

1.1 Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$$S = \{0,1,2,3\}$$

$$\Sigma = \{a,b\}$$

$$s_0 = 0$$

$$F = \{3\}$$

1.2 Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$$\delta(0,a) = \{0,1\}$$

$$\delta(0,b) = \{0\}$$

$$\delta(1,b) = \{2\}$$

$$\delta(2,b) = \{3\}$$

State	Input a	Input b
0	{0, 1}	{0}
1		{2}
2		{3}

Transition Table for the NFA

1.3 The Language Defined by an NFA

- An NFA *accepts* an input string x if and only if there is some path with edges labeled with symbols from x in sequence from the start state to some accepting state in the transition graph

- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as $(a | b)^*abb$ for the example NFA

Advantage of NFA

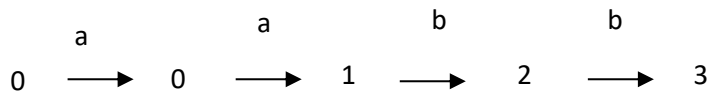
It provides fast access to the transitions of a given character.

Disadvantage of NFA

Disadvantage of using transition table is that it can take up a lot of space when the input alphabet is large and most transitions are to the empty set.

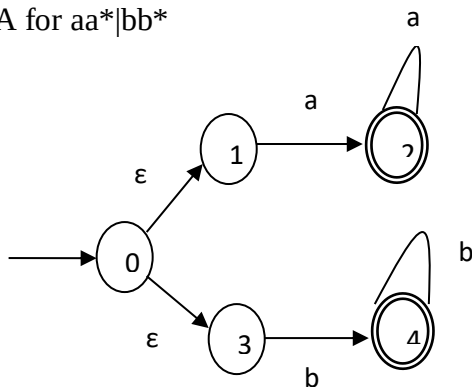
Accepting the string aabb

A path can be represented by sequence of state transitions called moves. The following diagram shows the moves made in accepting the input string aabb.



Example:

Design NFA for $aa^*|bb^*$

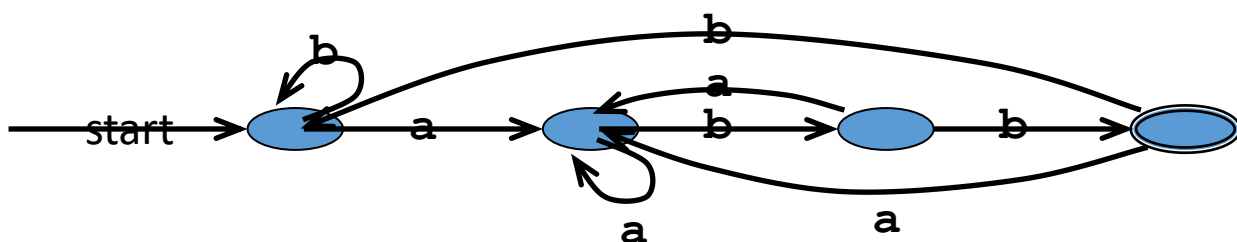


2. Deterministic Finite Automaton

- A *deterministic finite automaton* is a special case of an NFA in which:
 - No state has an ϵ -transition
 - For each state s and input symbol a there is at most one edge labeled a leaving s
- Each entry in the transition table is a single state
 - At most one path exists to accept a string
 - Simulation algorithm is simple

Example DFA

A DFA that accepts $(a | b)^*abb$



8. Explain briefly about converting from a Regular Expressions to An NFA

This is one way to convert a regular expression into a NFA.

- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.

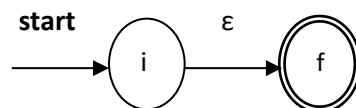
Algorithm: (Thompson's construction) An NFA from a regular expression.

Input: A regular expression r over an alphabet Σ .

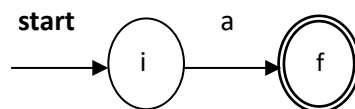
Output: An NFA N accepting $L(r)$.

Method:

1. For ϵ , construct the NFA

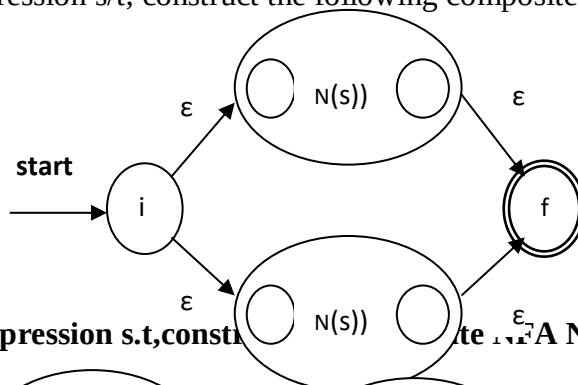


2. For a in Σ , construct the NFA

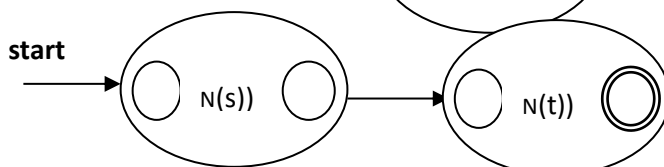


3. Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t .

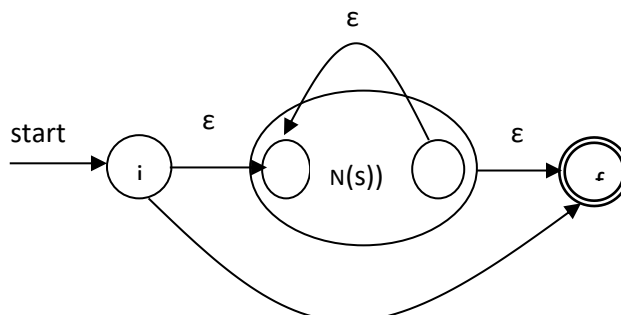
- a) For the regular expression s/t , construct the following composite NFA $N(s/t)$:



- b) For the regular expression $s.t$, construct the composite NFA $N(s.t)$



- c) For the regular expression S^* , construct the composite NFA $N(S^*)$:



- d) For the parenthesized regular expression (S) , construct the composite NFA $N(S)$ itself as the NFA.

The construction produces an NFA $N(r)$ with the following properties.

1. $N(r)$ has the most twice as many states as the number of symbols and operators in r .
2. $N(r)$ has exactly one start and one accepting state. The accepting state has no outgoing transitions.

- Each state of $N(r)$ has either one outgoing transition on a symbol in ϵ or at most two outgoing ϵ -transitions.

9. Explain briefly about converting of an NFA into a DFA.

Conversion of an NFA into a DFA

- The *subset construction algorithm* converts an NFA into a DFA using:

$$\epsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \rightarrow_{\epsilon} \dots \rightarrow_{\epsilon} t\}$$

$$\epsilon\text{-closure}(T) = \bigcup_{s \in T} \epsilon\text{-closure}(s)$$

$$\text{move}(T, a) = \{t \mid s \rightarrow_a t \text{ and } s \in T\}$$

- The algorithm produces:

$Dstates$ is the set of states of the new DFA consisting of sets of states of the NFA

$Dtran$ is the transition table of the new DFA

Algorithm: (subset sonstruction)

Initially, $\epsilon\text{-closure}(s)$ is the only state in $Dstates$ and it is unmarked;

While there ia an unmarked state T in $Dstates$ do begin

mark T ;

For each input symbol a do begin

$U := \epsilon\text{-closure}(\text{move}(T, a))$;

If U is not in $Dstates$ then

add U as an unmarked state to $Dstate$;

$Dtran[T, a] := U$

End

End

The subset sonstruction

Computation of ϵ -closure:

push all states in T onto stack;

initialize $\epsilon\text{-closure}(T)$ to T ;

while stack is not emptydo begin

pop t , the top element, off of stack;

for each state u with an edge from t to u labeled ϵ do

if u is not in $\epsilon\text{-closure}(T)$ do begin

add u to $\epsilon\text{-closure}(T)$;

push u onto stack

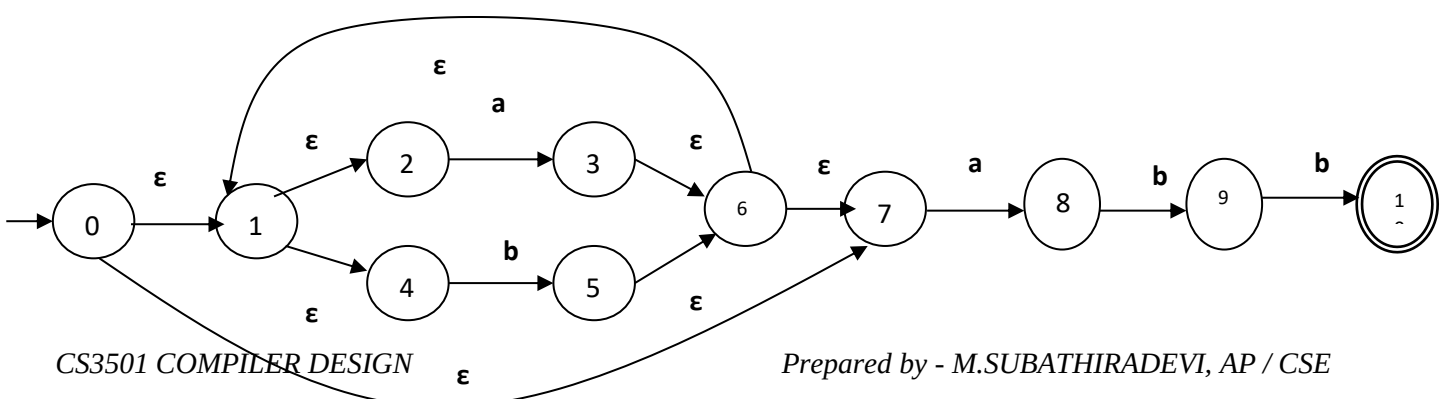
end

end

Subset Construction Example 1

Solution:

The NFA for the RE $(a+b)^* abb$ is shown below,



Step 1:

ϵ – closure (0) = { 0,1,2,4,7} $\rightarrow$ A

Now process the state A with ‘a’ and ‘b’.

Step 2:

Move [A,a] = {3,8} (Similar to $\delta(A,a)$)

ϵ – closure (Move [A,a]) = { 1,2,3,4,6,7,8} $\rightarrow$ B

Move [A,b] = {5}

ϵ – closure (Move[A,b]) = { 1,2,4,5,6,7} $\rightarrow$ C

Step 3:

Move [B,a] = {3,8}

ϵ – closure (Move [B,a]) = {1,2,3,4,6,7,8} $\rightarrow$ B

Move [B,b] = {5}

ϵ – closure (Move [B,b]) = {1,2,4,5,6,7,9} $\rightarrow$ D

Step 4:

Move[C,a] = {3,8}

ϵ – closure (Move[C,a]) = {1,2,3,4,6,7,8} $\rightarrow$ B

Move [C,b] = {5}

ϵ – closure (Move [C,b]) = {1,2,4,5,6,7} $\rightarrow$ C

Step 5:

Move[D,a] = {3,8}

ϵ – closure (Move[D,a]) = {1,2,3,4,6,7,} $\rightarrow$ B

Move[D,b] = {5,10}

ϵ – closure (Move[D,b]) = {1,2,4,5,6,7,10} $\rightarrow$ E

Step 6:

Move [E,a] = {3,8}

ϵ – closure (Move [E,a]) = {1,2,3,4,6,7,8} $\rightarrow$ B

Move [E,b] = {5}

ϵ – closure (Move [E,b]) = { 1,2,4,5,6,7} $\rightarrow$ C

Dstates

A = {0,1,2,4,7}

B = {1,2,3,4,6,7,8}

C = {1,2,4,5,6,7}

D = {1,2,4,5,6,7,9}

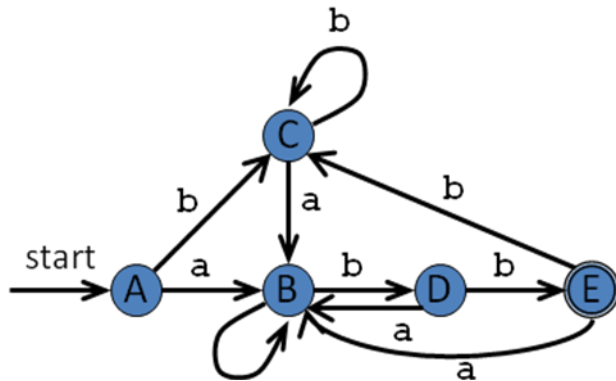
E = {1,2,4,5,6,7,10}

The transition table for DFA is given below,

	a	b
A	B	C
B	B	D
C	B	C

D	B	E
*E	B	C

DFA Daigram



10. Explain about converting From Regular Expression to DFA Directly

From Regular Expression to DFA Directly

- The “important states” of an NFA are those without an ϵ -transition, that is if $move(\{s\}, a) \neq \emptyset$ for some a then s is an important state
- The subset construction algorithm uses only the important states when it determines ϵ -closure($move(T, a)$)

From Regular Expression to DFA Directly (Algorithm)

ALGORITHM : Construction of a DFA from a regular expression r .

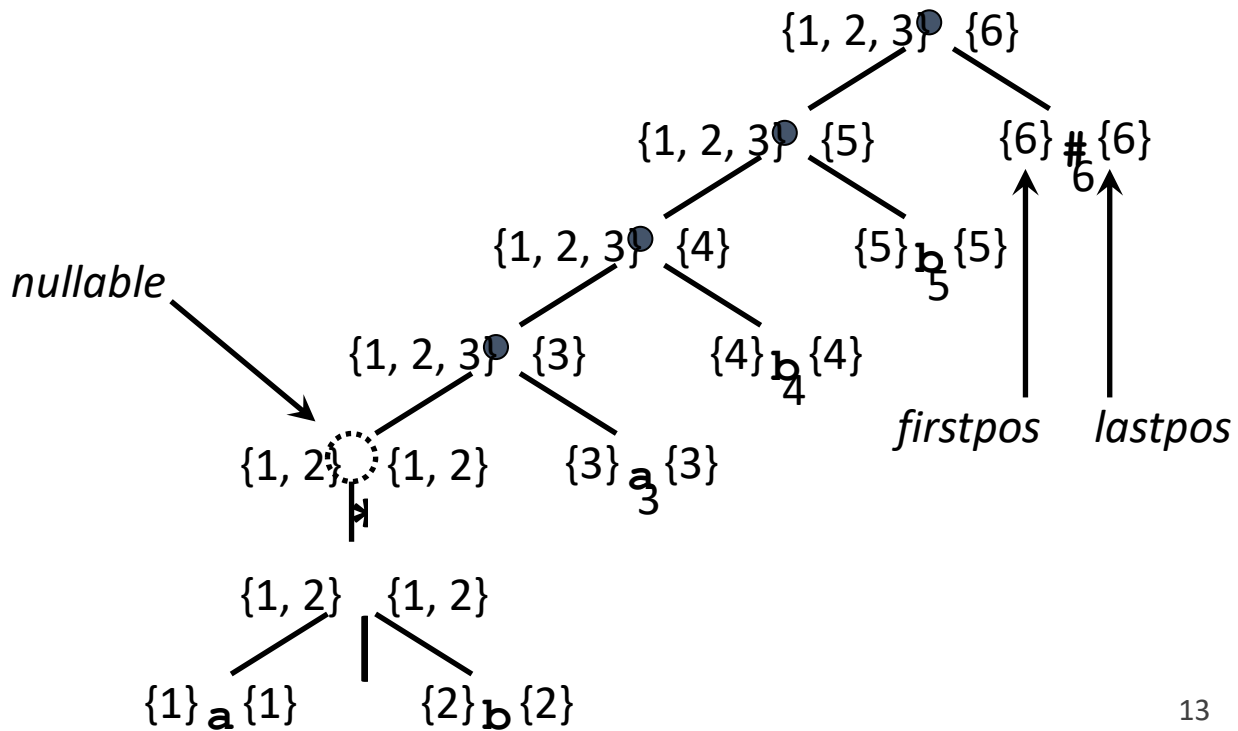
INPUT : A regular expression r .

OUTPUT: A DFA D that recognizes $L(r)$.

METHOD:

1. Construct a syntax tree T from the augmented regular expression $(r)\#$.
2. Compute nullable, firstpos, lastpos, and followpos for T
3. Construct $Dstates$, the set of states of DFA D , and $Dtran$, the transition function for D , by the procedure of Fig. 3.62. The states of D are sets of positions in T . Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is $firstpos(no)$, where node no is the root of T . The accepting states are those containing the position for the endmarker symbol $\#$.

From Regular Expression to DFA Directly: Syntax Tree of $(a|b)^*abb\#$



13

From Regular Expression to DFA Directly: *followpos*

```

for each node  $n$  in the tree do
  if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
    for each  $i$  in  $lastpos(c_1)$  do
       $followpos(i) := followpos(i) \cup firstpos(c_2)$ 
    end do
  else if  $n$  is a star-node
    for each  $i$  in  $lastpos(n)$  do
       $followpos(i) := followpos(i) \cup firstpos(n)$ 
    end do
  end if
end do

```

From Regular Expression to DFA Directly: Algorithm

```

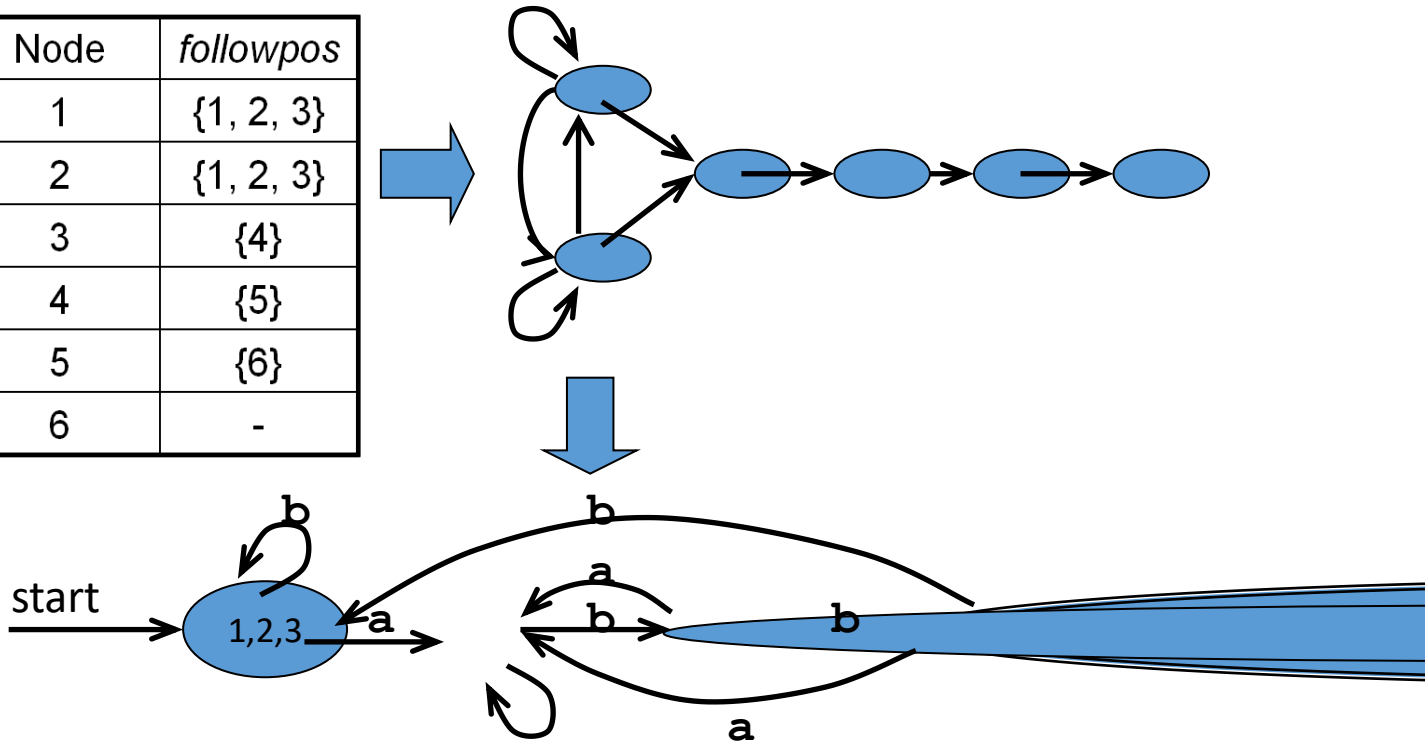
 $s_0 := firstpos(root)$  where  $root$  is the root of the syntax tree
 $Dstates := \{s_0\}$  and is unmarked
while there is an unmarked state  $T$  in  $Dstates$  do
  mark  $T$ 
  for each input symbol  $a \in \Sigma$  do
    let  $U$  be the set of positions that are in  $followpos(p)$ 
      for some position  $p$  in  $T$ ,
      such that the symbol at position  $p$  is  $a$ 
    if  $U$  is not empty and not in  $Dstates$  then
      add  $U$  as an unmarked state to  $Dstates$ 
    end if
     $Dtran[T, a] := U$ 
  end do

```

end do
end do

From Regular Expression to DFA Directly: Example

Node	<i>followpos</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-



16

11. Write an algorithm for minimizing the number of states of a DFA. (8) Nov/Dec 2016

Minimization the number of states of a DFA

We can construct a minimum state DFA by reducing the number of states in a given DFA to the bare minimum without attacking the language that is being recognized.

The following steps are used to minimize the number of states of a DFA.

Partition the set of states into two groups.

- G_1 : Set of accepting states
- G_2 : Set of no-accepting states

For each new group G

- Partition G into subgroups such that S_1 and S_2 are in the same group iff for all input symbols a , states s_1 and s_2 have transitions to state in the same group.

Start state of the minimized DFA is the group containing the start state of the original DFA.

Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Algorithm : Minimizing the number of states of a DFA.

Input: A DFA M with set of states S , set of input Σ , transitions defined for all states and inputs start state S_0 and set of accepting states F .

Output: A DFA M^1 accepting the same languages as M and having as few states as possible.

Method:

1. Construct an initial partition Π of the set of states with two groups: the accepting F and nonaccepting states $S-F$.
2. Apply the procedure to construct a new partition Π_{new} .
initially, let $\Pi_{\text{new}} = \Pi$;
for (each group G of Π) {
partition G into subgroups such that two states s and t
are in the same subgroup if and only if for all
input symbols a , states s and t have transitions on a
to states in the same group of Π ;
/* at worst, a state will be in a subgroup by itself */
replace G in Π_{new} by the set of all subgroups formed;
}
3. If $\Pi_{\text{new}} = \Pi$, let $\Pi_{\text{final}} = \Pi$ and continue with step (4) . Otherwise, repeat step (2) with Π_{new} in place of Π .
4. Choose one state in each group of Π_{final} as the representative for that group. The representatives will be the states of the minimum-state DFA M' .
Let S be a representative state, and suppose on input a there is a transition of M from s to t . Let r be the representative of t 's group (r may be t). Then M' has a transition from s to r on a . Let the start state S_0 of M , and let the accepting states of M' be the representatives that are in F .
Note that each group of Π_{final} either consists only of states in F or has no states in F .
7. If M' has a dead state, that is, a state d that is not accepting and that has transitions to itself on all input symbols, then remove d from M' . Also remove any states not reachable from the start state. Any transitions to d from other states become undefined.

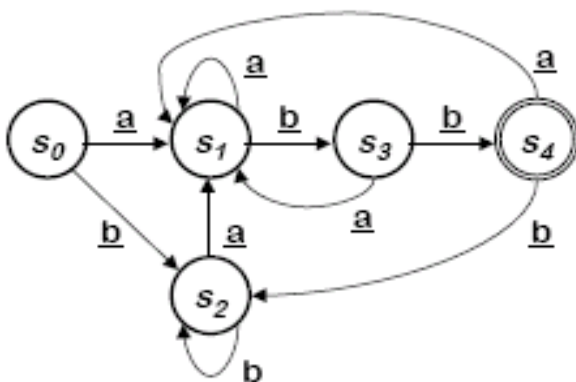
Procedure for Π_{new} construction

```

for ( each group  $G$  of  $\Pi$  ) {
partition  $G$  into subgroups such that two states  $s$  and  $t$ 
are in the same subgroup if and only if for all
input symbols  $a$ , states  $s$  and  $t$  have transitions on  $a$ 
to states in the same group of  $\Pi$ ;
/* at worst, a state will be in a subgroup by itself */
replace  $G$  in  $\Pi_{\text{new}}$  by the set of all subgroups formed;
end

```

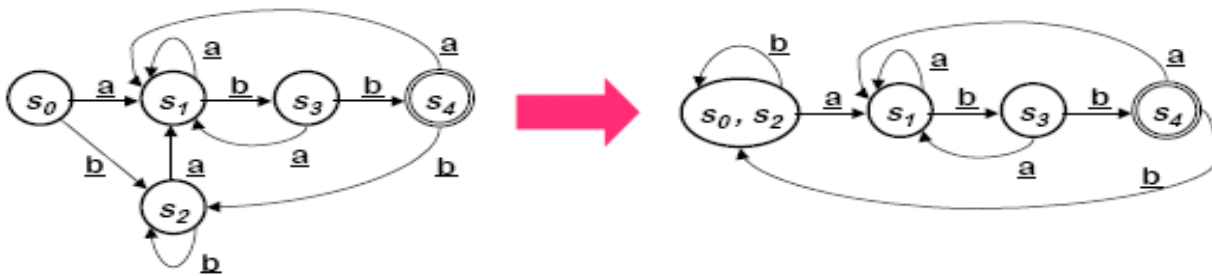
Example # 1



State	Character	
	<u>a</u>	<u>b</u>
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

Minimization of DFA

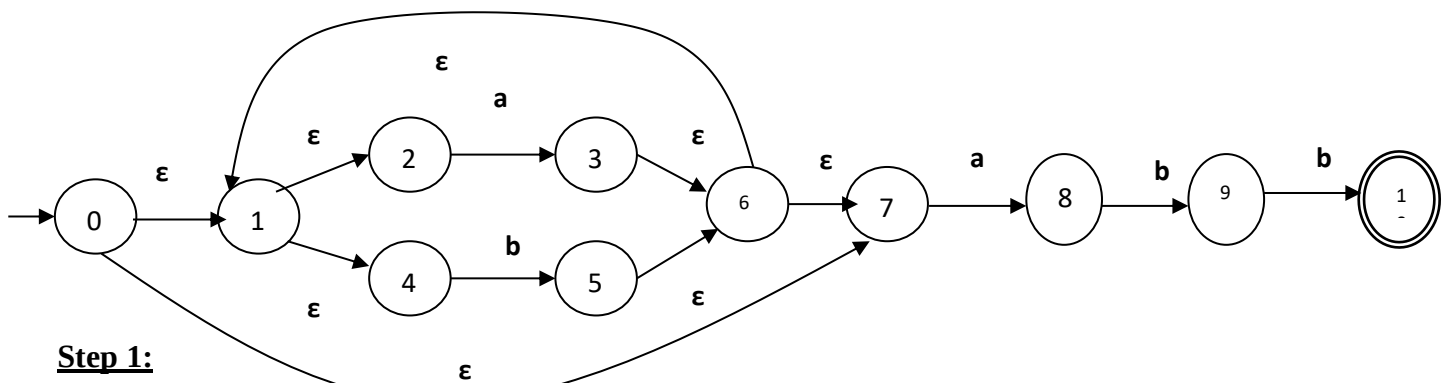
	Current Partition	Split on <u>a</u>	Split on <u>b</u>
P_0	$\{s_4\} \{s_0, s_1, s_2, s_3\}$	none	$\{s_3\} \{s_0, s_1, s_2\}$
P_1	$\{s_4\} \{s_3\} \{s_0, s_1, s_2\}$	none	$\{s_1\} \{s_0, s_2\}$
P_2	$\{s_4\} \{s_3\} \{s_1\} \{s_0, s_2\}$	none	none
P_2	$\{s_4\} \{s_3\} \{s_1\} \{s_0, s_2\}$	none	none



12. Convert the regular expression $(a+b)^* abb$ into NFA – ϵ and find the equivalent minimum state DFA. [NOV/DEC 2008]

Solution:

The NFA for the RE $(a+b)^* abb$ is shown below,



Step 1:

ϵ – closure (0) = { 0, 1, 2, 4, 7} $\rightarrow A$

Now process the state A with 'a' and 'b'.

Step 2:

Move [A, a] = {3, 8} (Similar to $\delta(A, a)$)

ϵ – closure (Move [A, a]) = { 1, 2, 3, 4, 6, 7, 8} $\rightarrow B$

Move [A, b] = {5}

ϵ – closure (Move [A, b]) = { 1, 2, 4, 5, 6, 7} $\rightarrow C$

Step 3:

Move [B, a] = {3, 8}

ϵ – closure (Move [B, a]) = {1, 2, 3, 4, 6, 7, 8} $\rightarrow B$

Move [B, b] = {5}

ϵ – closure (Move [B, b]) = {1, 2, 4, 5, 6, 7, 9} $\rightarrow D$

Step 4:

Move [C, a] = {3, 8}

ϵ – closure (Move[C,a]) = {1,2,3,4,6,7,8} $\rightarrow$ B

Move [C,b] = {5}

ϵ – closure (Move [C,b]) = {1,2,4,5,6,7} $\rightarrow$ C

Step 5:

Move[D,a] = {3,8}

ϵ – closure (Move[D,a]) = {1,2,3,4,6,7,} $\rightarrow$ B

Move[D,b] = {5,10}

ϵ – closure (Move[D,b]) = {1,2,4,5,6,7,10} $\rightarrow$ E

Step 6:

Move [E,a] = {3,8}

ϵ – closure (Move [E,a]) = {1,2,3,4,6,7,8} $\rightarrow$ B

Move [E,b] = {5}

ϵ – closure (Move [E,b]) = { 1,2,4,5,6,7} $\rightarrow$ C

The transition table for DFA is given below,

	a	b
$\rightarrow$ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

To Find Minimized DFA

By using minimized algorithm the minimized DFA is as follows,

[A B C D E]

[A B C D] [E]

{ E is final state }

[A B C] [D] [E]

{ Since $\delta(D, b) = E$ }

[A C] [B] [D] [E]

$A \equiv C$, Eliminate C and replace ‘C’ by ‘A’

The transition table for minimum state DFA is given below,

	a	b
$\rightarrow$ A	B	A
B	B	D
D	B	E
* E	B	A

13. Discuss about LEX tool.

A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

1. Use of Lex

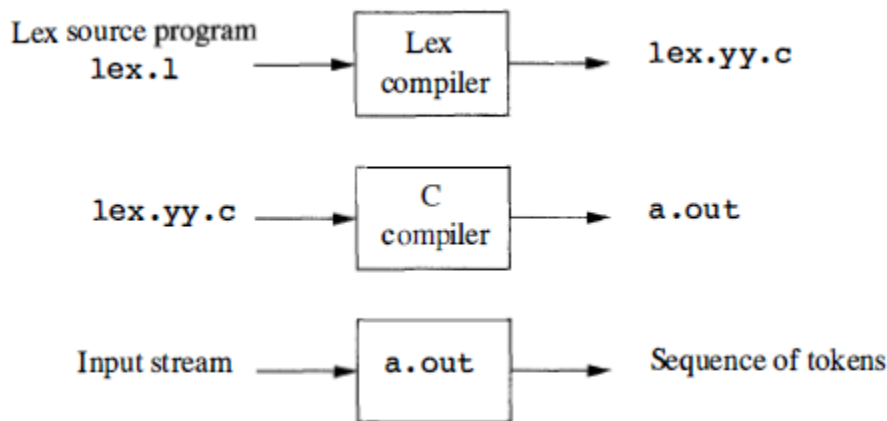
In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, 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.

There is a wide range of tools for constructing lexical analyzers.

□ Lex

LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.



Creating a lexical analyzer with Lex

2. Structure of Lex programs

First, a specification of a lexical analyzer is prepared by creating a program `lex.l` in the Lex language. Then, `lex.l` is run through the Lex compiler to produce a C program `lex.yy.c`.

Finally, `lex.yy.c` 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.

lex.l lex.yy.c

lex.yy.c a.out

Lex Specification

A Lex program consists of three parts:

declarations

%%

translation rules

%%

auxiliary functions

Definitions or declarations include declarations of variables, constants, and regular definitions

Rules are statements of the form

`p1 {action 1}`

`p2 {action 2}`

`...p`

`n {action n}`

where `pi` is regular expression and action `i` describes what action the lexical analyzer should take when pattern `pi` matches a lexeme. Actions are written in C code.

□ **User subroutines or auxiliary functions** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Example:

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

/* regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}     { /* no action and no return */}
if       {return(IF);}
then     {return(THEN);}
else     {return(ELSE);}
{id}     {yylval = (int) installID(); return(ID);}
{number} {yylval = (int) installNum(); return(NUMBER);}
"<"     {yylval = LT; return(RELOP);}
"<="    {yylval = LE; return(RELOP);}
"="      {yylval = EQ; return(RELOP);}
">"     {yylval = NE; return(RELOP);}
">"     {yylval = GT; return(RELOP);}
">="    {yylval = GE; return(RELOP);}

%%

int installID() { /* function to install the lexeme, whose
                  first character is pointed to by yytext,
                  and whose length is yyleng, into the
                  symbol table and return a pointer
                  thereto */
}

int installNum() { /* similar to installID, but puts numer-
                   ical constants into a separate table */
}
```

Lex program for the tokens of Fig.

3. Conflict Resolution in lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

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.

4. The Lookahead Operator

Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

14. Discuss about Lexical errors.

Detection of Errors

- To be useful, a compiler should detect all errors in the source code and report them to the user.
- These errors could be:

Compilation Errors

- Lexical errors: e.g., badly formed identifiers or constants, symbols which are not part of the language, badly formed comments, etc.
- Syntactic errors: chains of syntactic units that do not conform to the syntax of the source language.
- Semantic errors: e.g., operations conducted on incompatible types, undeclared variables, double declaration of variable, reference before assignment, etc.

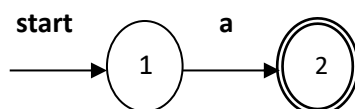
Execution Errors

- Run-time errors: errors detectable solely at run time, pointers with null value or whose value is outside allowed limits, or indexing of vectors with unsuitable indices, etc.

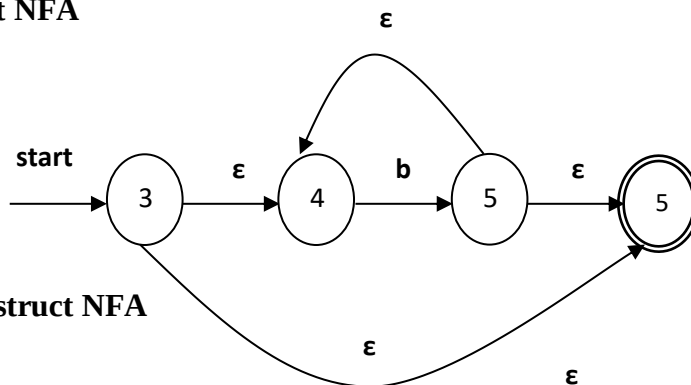
15. Draw NFA for the regular expression ab^*/ab . [May/June-14]

Solution:

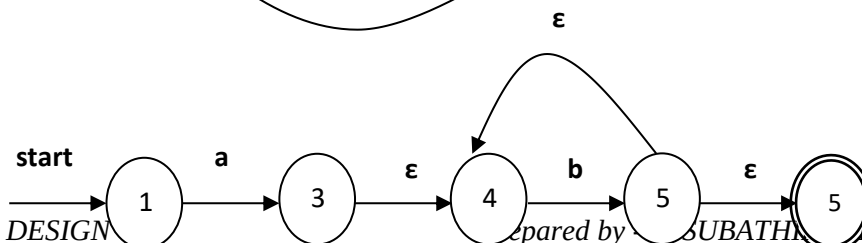
For $r_1 = a$ construct NFA



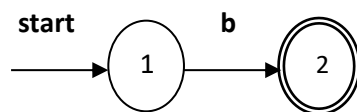
For $r_2 = b^*$ construct NFA



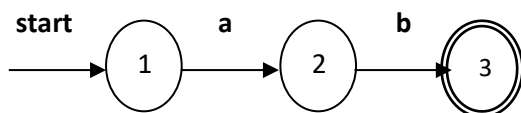
For $r_3 = r_1 \cdot r_2$ construct NFA



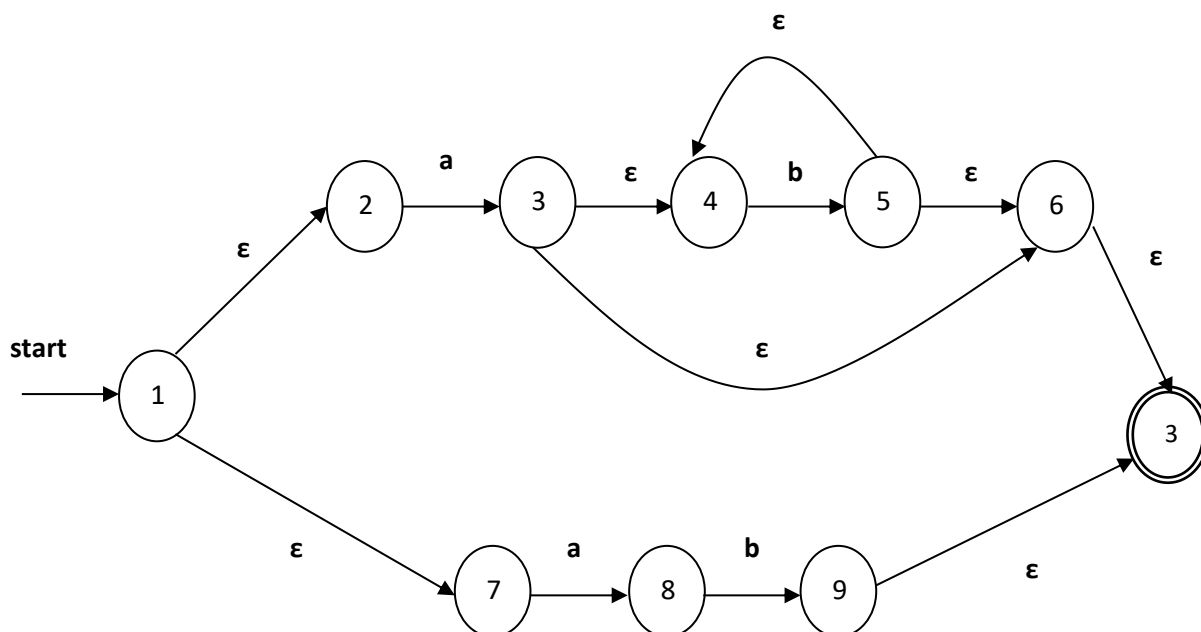
For $r_4 = b$ construct NFA



For $r_5 = r_1. r_4$ construct NFA



For $r_7 = r_3/r_5$ construct NFA



16. Explain briefly about Design of Lexical Analyzer for a sample Language.

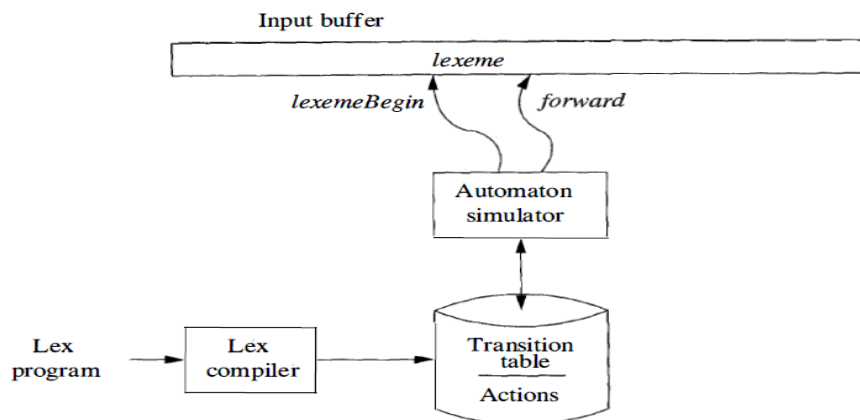
We assume that we have a specification of a lexical analyzer of the form

P1 { action1 }

P2 { action2 }

.....

Pn { actionn }



where, each pattern p_i is a regular expression and each action $action_i$ is a program fragment that is to be executed whenever a lexeme matched by p_i is found in the input.

Our problem is to construct a recognizer that looks for lexemes in the input buffer. If more than one pattern matches, the recognizer is to choose the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the first-listed matching pattern is chosen.

A finite automaton is a natural model around which to build a lexical analyzer, and the one constructed by our Lex compiler has the form shown in Fig. above. There is an input buffer with two pointers to it, a lexeme beginning and a forward pointer. The Lex compiler constructs a transition table for a finite automaton from the regular expression patterns in the Lex specification. The lexical analyzer itself consists of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer.

compiler can be based on either nondeterministic or deterministic automata, At the end of the last section we saw that the transition table of an NFA for a regular expression pattern can be considerably smaller than that of a DFA, but the DFA has the decided advantage of being able to recognize patterns faster than the NFA.

Pattern Matching Based on NFA'S

One method is to construct the transition table of a nondeterministic finite automaton N for the composite pattern $P_1 | P_2 | \dots | P_n$. This can be done by first creating an NFA $N(p_i)$ for each pattern p_i using Algorithm, then adding a new start state s_0 , and finally linking s_0 to the start state of each $N(p_i)$ with an ϵ -transition, as shown in Fig. below.

To simulate this NFA we can use a modification of Algorithm 3.4. The modification ensures that the combined NFA recognizes the longest prefix of the input that is matched by a pattern. In the combined NFA, there is an accepting state for each pattern p_i . When we simulate the NFA using Algorithm 3.4, we construct the sequence of sets of states that the combined NFA can be in after seeing each input character. Even if we find a set of states that contains an accepting state, to find the longest match we must continue to simulate the NFA until it reaches *termination*, that is, a set of states from which there are no transitions on the current input symbol.

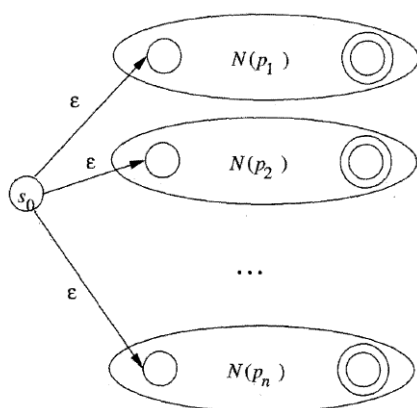


Figure 3.50: An NFA constructed from a **Lex** program

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
a*b⁺	{ action A_3 for pattern p_3 }

We presume that the Lex specification is designed so that a valid source program cannot entirely fill the input buffer without having the NFA reach termination. For example, each compiler puts

restriction on the length of an identifier, and violations of this limit will be detected when the input buffer overflows, if not sooner,

To find the correct match, we make two modifications to Algorithm. First, whenever we add an accepting state to the current set of states, we record the current input position and the pattern p_i corresponding to this accepting state. If the current set of states already contains an accepting state, then only the pattern that appears first in the Lex specification is recorded.

Second, we continue making transitions until we reach termination. Upon termination, we retract the forward pointer to the position at which the last match occurred. The pattern making this match identifies the token found, and the lexeme matched is the string between the lexeme-beginning and forward pointers.

Usually, the Lex specification is such that some pattern, possibly an error pattern, will always match. If no pattern matches, however, we have an error condition for which no provision was made, and the lexical analyzer should transfer control to some default error recovery routine.

Example 3.18

A simple example illustrates the above ideas. Suppose we have the following Lex program consisting of three regular expressions and no regular definitions,

```
a      { action  $A_1$  for pattern  $p_1$  }
abb    { action  $A_2$  for pattern  $p_2$  }
a*b+  { action  $A_3$  for pattern  $p_3$  }
```

The three tokens above are recognized by the automata of Fig.(a). We have simplified the third automaton somewhat from what would be produced by Algorithm . As indicated above, we can convert the NFA's of Fig.(a) into one combined NFA W shown in (b).

Let us now consider the behavior of N on the input string *aaba* using our modification of Algorithm. Figure 3.36 shows the sets of states and patterns that match as each character of the input *aaba* is processed. This figure shows that the initial set of states is {0, 1, 3, 7}. States 1, 3, and 7 each have a transition on *a*, to states 2, 4, and 7, respectively. Since state 2 is the accepting state for the first pattern, we record the fact that the first pattern matches after reading the first *a*.

However, there is a transition from state 7 to state 7 on the second input character, so we must continue making transitions. There is a transition from state 7 to state 8 on the input character *b*. State 8 is the accepting state for the third pattern. Once we reach state 8, there are no transitions possible on the next input character *a* so we have reached termination. Since the last match occurred after we read the third input character, we report that the third pattern has matched the lexeme *aab*.

The role of $action_i$ associated with the pattern p_i in the Lex specification is as follows. When an instance of p_i is recognized, the lexical analyzer executes the associated program $action_i$. Note that $action_i$ is not executed just because the NFA enters a state that includes the accepting state for p_i ; $action_i$ is only executed if p_i turns out to be the pattern yielding the longest match.

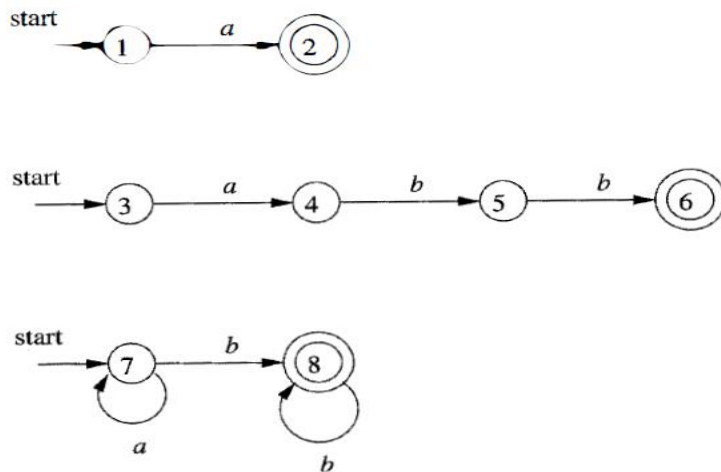


Figure 3.51: NFA's for **a**, **abb**, and **a*b⁺**

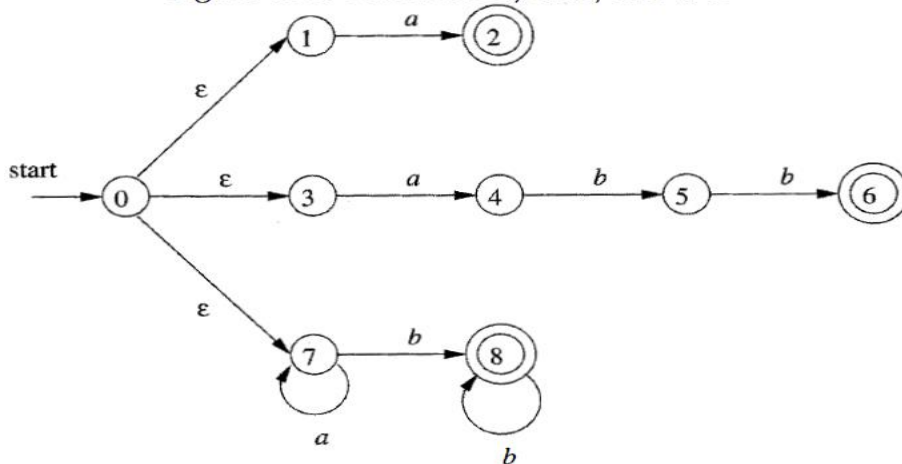
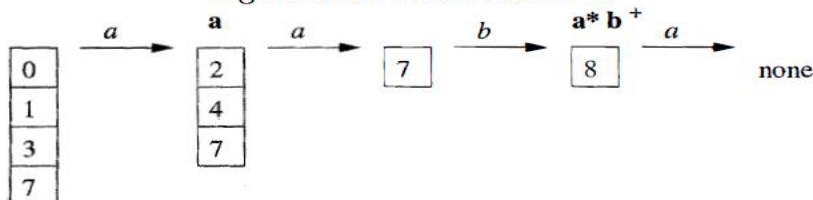


Figure 3.52: Combined NFA

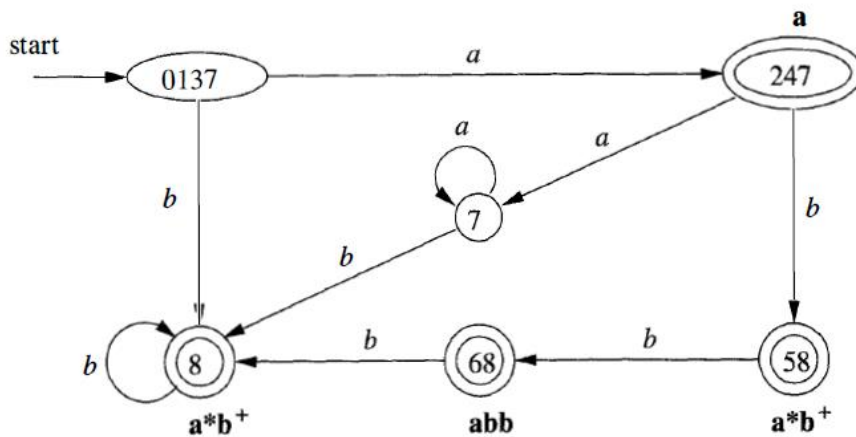


Sequence of sets of states entered when processing input *aaba*

DFA's for Lexical Analyzers

Another approach to the construction of a lexical analyzer from a Lex specification is to use 3 DFA to perform the pattern matching+ The only nuance is to make sure we find the proper pattern matches. The situation is completely analogous to the modified simulation of an NFA just described. When we convert an NFA to a DFA using the subset construction Algorithm 3.2, there may be several accepting states in a given subset of nondeterministic states.

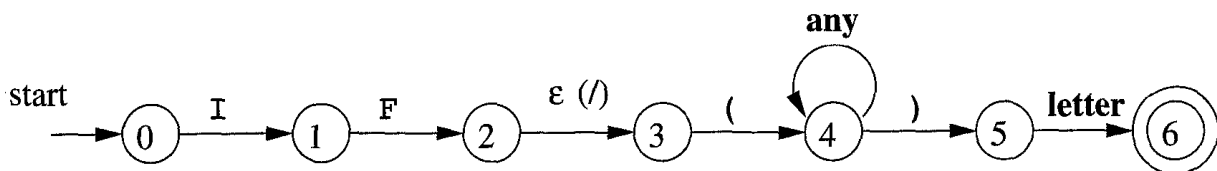
In such a situation, the accepting states corresponding to the pattern listed first in the Lex specification has priority. As in the NFA simulation, the only other modification we need to perform is to continue making state transitions until we reach a state with no next state(i.e., the state 0) for the current input symbol, To find the lexeme matched, we return to the last input position at which the DFA entered an accepting state.



Transition graph for DFA handling the patterns **a**, **abb**, and **a*b⁺**

Recall from Section 3.4 that the lookahead operator *!* is necessary in some situations, since the pattern that denotes a particular token may need to describe some trailing context for the actual lexeme. When converting a pattern with */* to an NFA, we can treat the */* as if it were *c*, so that we do not actually look for */* on the input. However, if a string denoted by this regular expression is recognized in the input buffer, the end of the lexeme is not the position of the NFA's accepting state. Rather the end occurrence of the state of this NFA having a transition on the (imaginary) */*.

Example 3. The NFA recognizing the pattern for IF given in Example 3.12 is shown in Fig. State 6 indicates the presence of keyword IF; however, we find the token IF by scanning backwards to the last occurrence of state 2.

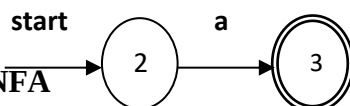


NFA recognizing the keyword IF

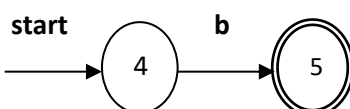
Refer the lex program that is question number 8.

17. Write notes on regular expression to NFA. Construct Regular expression to NFA for the sentence (a/b)*a. (10) May/June 2016

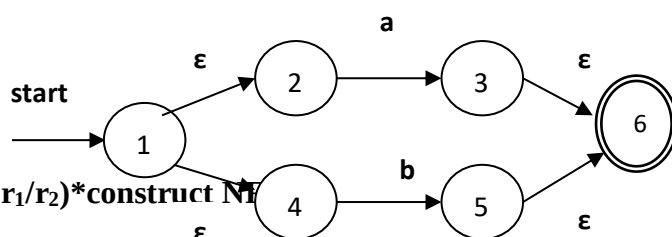
For r_1 construct NFA



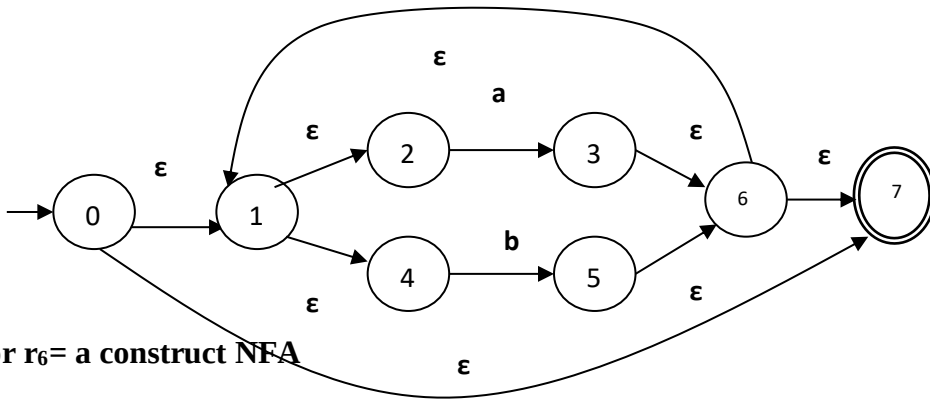
For r_2 construct NFA



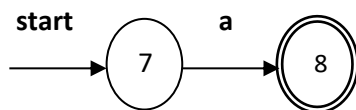
For $r_3 = r_1 / r_2$ construct NFA



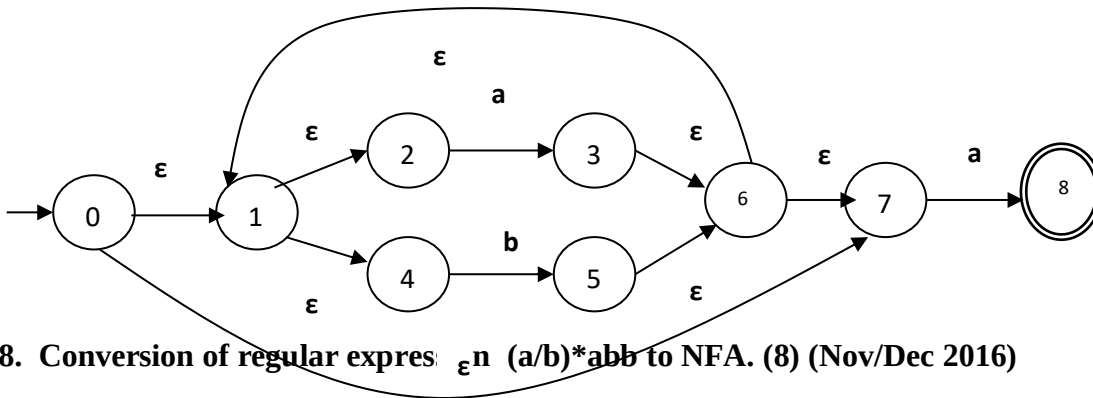
For $r_3 = (r_1/r_2)^*$ construct NFA



For $r_6 = a$ construct NFA

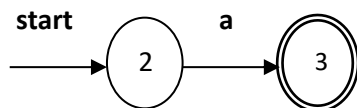


For $r_7 = r_5.r_6$ construct NFA

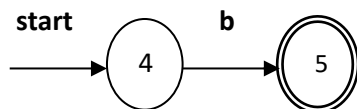


18. Conversion of regular expressions $(a/b)^*abb$ to NFA. (8) (Nov/Dec 2016)

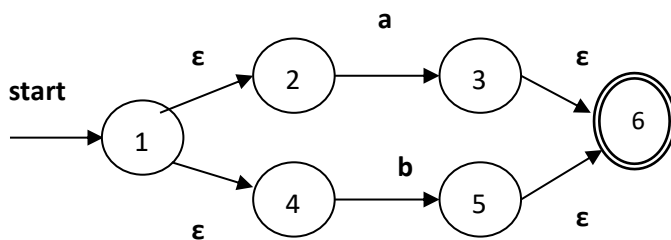
For r_1 construct NFA



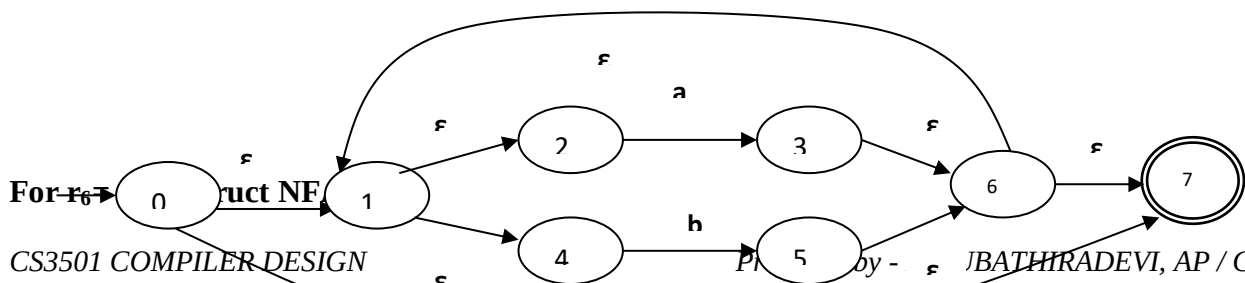
For r_2 construct NFA

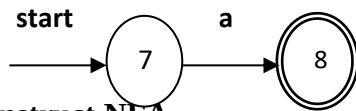


For $r_3 = r_1/r_2$ construct NFA

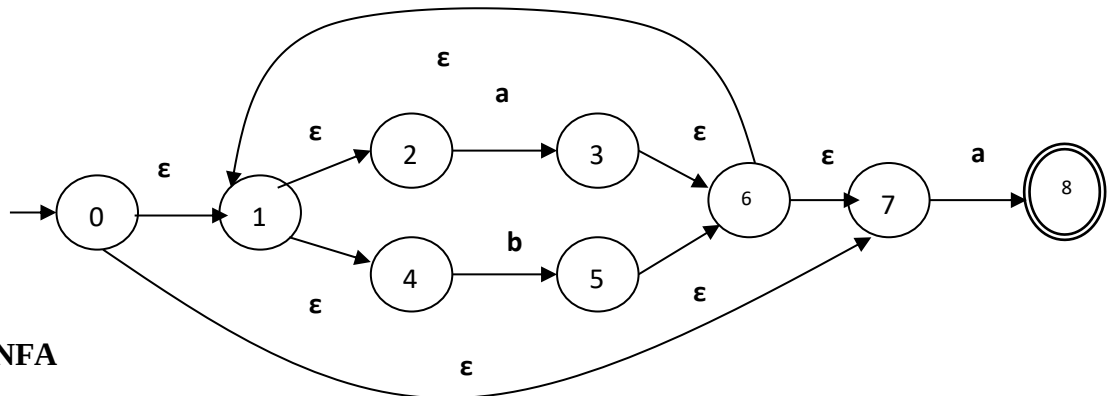


For $r_3 = (r_1/r_2)^*$ construct NFA

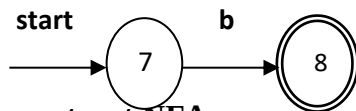




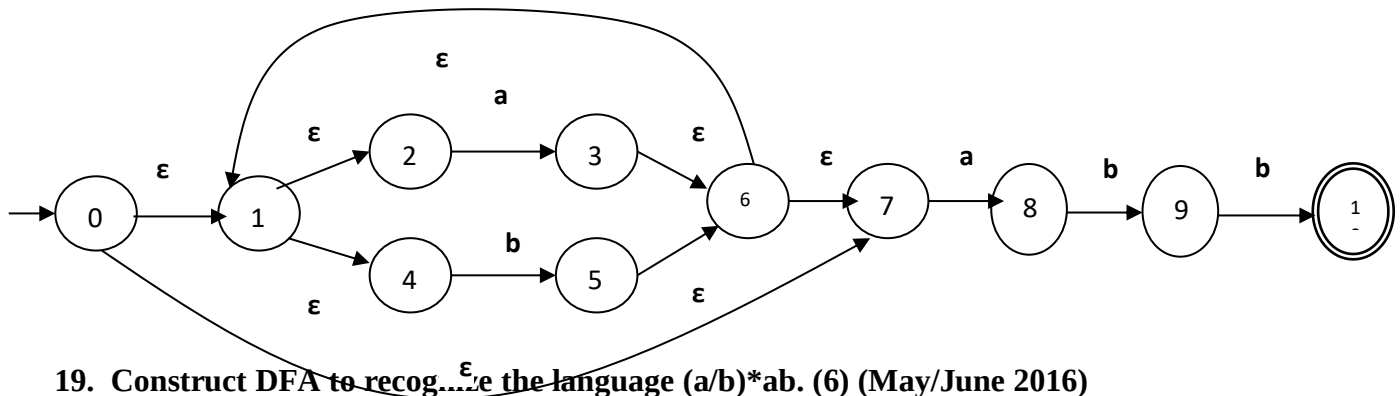
For $r_7 = r_5.r_6$ construct NFA



For $r_8 = b$ construct NFA



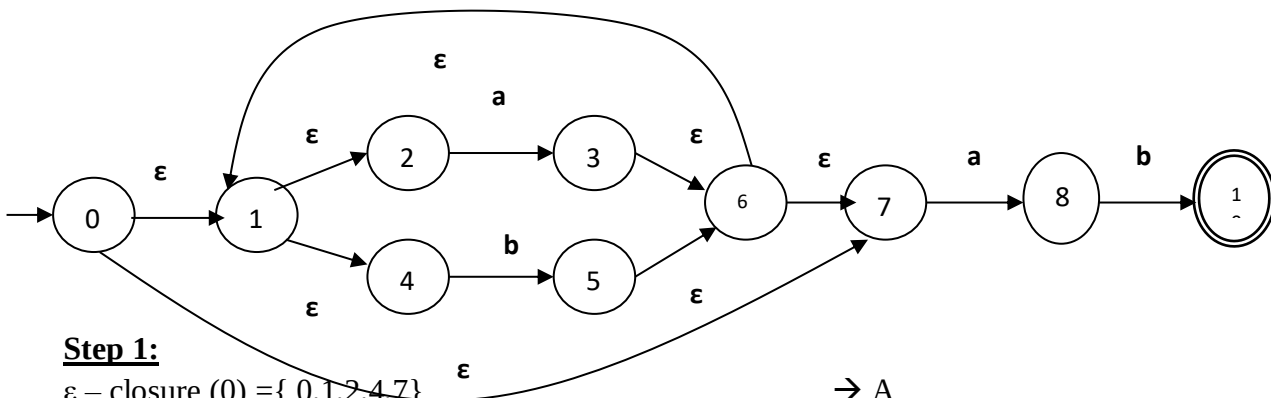
For $r_9 = r_7.r_8$ construct NFA



19. Construct DFA to recognize the language $(a/b)^*ab$. (6) (May/June 2016)

Solution:

The NFA for the RE $(a/b)^*ab$ is shown below,



Step 1:

ϵ -closure (0) = { 0, 1, 2, 4, 7 } $\rightarrow A$

Now process the state A with 'a' and 'b'.

Step 2:

$\delta(A, a) = \epsilon$ -closure ($\delta [A, a]$)

= ϵ -closure ($\delta [\{ 0, 1, 2, 4, 7 \}, a]$)

= ϵ -closure (3, 8)

= { 3, 6, 7, 1, 2, 4, 8 }

$\rightarrow B$

$\delta(A, b) = \epsilon$ -closure ($\delta [A, b]$)

= ϵ -closure ($\delta [\{ 0, 1, 2, 4, 7 \}, b]$)

= ϵ -closure (5)

= { 5,6,7,1,2,4 }

→ C

Step 3:

$\delta(B,a) = \epsilon - \text{closure}(\delta[B,a])$

= $\epsilon - \text{closure}(\delta[\{ 3,6,7,1,2,4,8\},a])$

= $\epsilon - \text{closure}(3,8)$

= { 3,6,7,1,2,4,8 }

→ B

$\delta(B,b) = \epsilon - \text{closure}(\delta[B,b])$

= $\epsilon - \text{closure}(\delta[\{ 3,6,7,1,2,4,8\},b])$

= $\epsilon - \text{closure}(5,9)$

= { 5,6,7,1,2,4,9 }

→ D

Step 4:

$\delta(C,a) = \epsilon - \text{closure}(\delta[C,a])$

= $\epsilon - \text{closure}(\delta[\{ 5,6,7,1,2,4\},a])$

= $\epsilon - \text{closure}(3,8)$

= { 3,6,7,1,2,4,8 }

→ B

$\delta(C,b) = \epsilon - \text{closure}(\delta[C,b])$

= $\epsilon - \text{closure}(\delta[\{ 5,6,7,1,2,4\},b])$

= $\epsilon - \text{closure}(5)$

= { 5,6,7,1,2,4 }

→ C

Step 5:

$\delta(D,a) = \epsilon - \text{closure}(\delta[D,a])$

= $\epsilon - \text{closure}(\delta[\{ 5,6,7,1,2,4,9\},a])$

= $\epsilon - \text{closure}(3,8)$

= { 3,6,7,1,2,4,8 }

→ B

$\delta(D,b) = \epsilon - \text{closure}(\delta[D,b])$

= $\epsilon - \text{closure}(\delta[\{ 5,6,7,1,2,4,9\},b])$

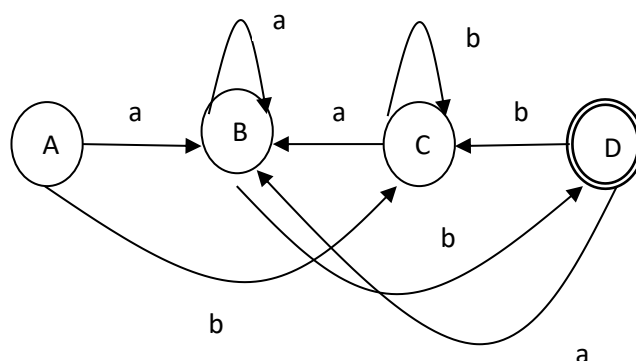
= $\epsilon - \text{closure}(5)$

= { 5,6,7,1,2,4 }

→ C

The transition table for DFA is given below,

	a	b
A	B	C
B	B	D
C	B	C
*D	B	C



TWO MARKS

1. Define compiler.(Nov/Dec09)

A compiler is a program that reads a program written in one language –the **source language** and **translates** it into an equivalent program in another language-the **target language**. The compiler **reports** to its user the **presence of errors** in the source program. Example: Turbo C, common for both C & C++

2. What is the need for a Compiler?

We need Compilers, because the source program does not understand by the Computer. So, it has to convert into machine understandable language. So we use Compilers for this purpose.

We cannot use the same compiler for all computers. Because every HLL has its own syntaxes.

3. What are the two types of analysis of the source programs by compiler? (April/May 10)

Or What are the two parts of a compilation? Explain briefly. MAY/JUNE-2016

Analysis and **Synthesis** are the two parts of compilation.

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The synthesis part constructs the desired target program from the intermediate representation.

4. What is a language processing system?(Nov/Dec 2008)

The program which translate the program written in a programming language by the user into an executable program is known as language processors.

5. List the subparts or phases of analysis part.

Analysis consists of three phases:

- Linear Analysis.
- Hierarchical Analysis.
- Semantic Analysis.

6. What is linear analysis?

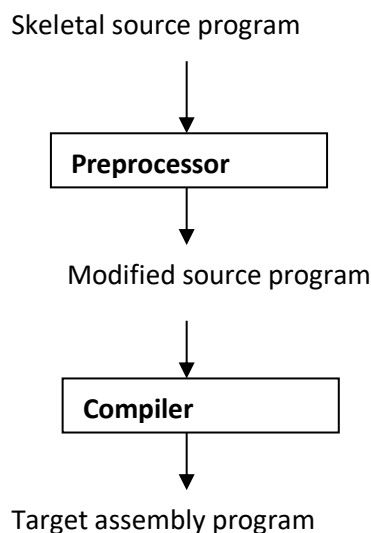
Linear analysis is one in which the stream of characters **making up the source program** is read from left to right and grouped **into tokens** that are sequences of characters having a collective meaning.

Also called lexical analysis or scanning.

7. Describe the error recovery schemes in a lexical phase of a compiler.(April/May 2015)

- Panic mode
- Statement mode
- Error productions
- Global correction

8. Illustrate diagrammatically how a language is processed. MAY/JUNE-2016



9. What are the phases of a compiler?

Phases of a Compiler

2. Lexical analysis (“scanning”)
 - Reads in program, groups characters into “tokens”
3. Syntax analysis (“parsing”)
 - Structures token sequence according to grammar rules of the language.
4. Semantic analysis
 - Checks semantic constraints of the language.
5. Intermediate code generation
 - Translates to “lower level” representation.
6. Program analysis and code optimization
 - Improves code quality.
7. Final code generation.

10. What is a symbol table? (Nov/Dec 2007) (Nov/Dec 2016)

A symbol table is a data structure containing a **record for each identifier**, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.

11. Mention some of the cousins of a compiler. [May/June 2012]

Cousins of the compiler are:

- ☐ Preprocessors
- ☐ Assemblers
- ☐ Loaders and Link-Editors

12. What is grouping of phases?[Nov/Dec 2013][May/June-14]

Grouping of Phases

- Front end : machine independent phases
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Some code optimization
- Back end : machine dependent phases
- Final code generation
- Machine-dependent optimizations

13. List the phases that constitute the front end of a compiler.

The front end consists of those phases or parts of phases that depend primarily on the source language and are largely independent of the target machine. These include

- ☐ Lexical and Syntactic analysis
- ☐ The creation of symbol table
- ☐ Semantic analysis
- ☐ Generation of intermediate code

A certain amount of code optimization can be done by the front end as well. Also includes error handling that goes along with each of these phases.

14. Mention the back-end phases of a compiler.

The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language. These include

- Code optimization
- Code generation, along with error handling and symbol- table operations.

15. Define compiler-compiler.

Systems to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems.

Largely they are oriented around a particular model of languages, and they are suitable for generating compilers of languages similar model.

16. What are compiler construction tools? (Nov/Dec 04,05) (Nov/Dec 2016)

The following is a list of some compiler construction tools:

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

17. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully.

Mainly performs type checking.

18. Define translator.

Translator converts the high level programming code to a machine understandable code.

19. What do you mean by assembler?

Assembler translates assembly language program, which uses mnemonic names for operation codes and data addresses into the machine language.

20. What do you mean by preprocessor? (May/June07)

A translator whose source language and object language are high- level language is termed as preprocessor.

21. Define linker.

A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

22. Define loader.

A loader is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program.

23. Define macro processor.

A macro processor is a system program that **replaces the macro calls** in an assembly language program **by the macro definition**. During this text replacement process, the formal arguments in the macro definition are converted into the actual arguments in the macro call.

24. Define phase.

A phase is a logically cohesive operation that takes as input one representation of source program and produces as output another representation of the source program.

25. Differentiate compiler and interpreter. (May/June2008)

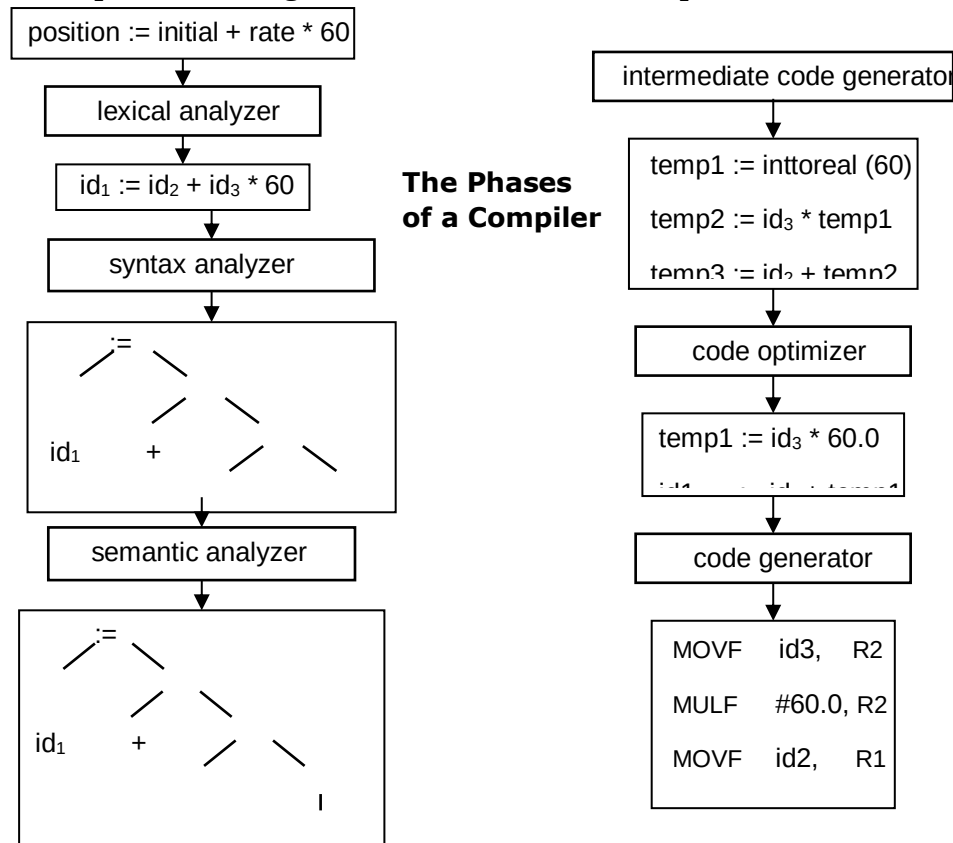
Compiler	Interpreter
Merit: In the process of compilation the program is analyzed only once and then the code is generated. Hence compiler is efficient than interpreter.	Demerit: The source program gets interpreted every time it is to be executed, and every time the source program is analyzed. Hence interpretation is less efficient than compiler.

The compiled produce object code.	The interpreters do not produce object code.
Demerit: The compiler has to be present on the host machine when particular program needs to be compiled. Demerit : The compiler is a complex program and it requires large amount of memory.	Merit: The interpreters can be made portable because they do not produce object code. Merit: Interpreters are simpler and give us improved debugging environment.

26. Define pretty printers?

A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For the comments may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

27. Explain with diagram how a statement is compiled.



28. Write short notes on error handler?

The error handler is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. So that as many errors as possible can be detected in one compilation.

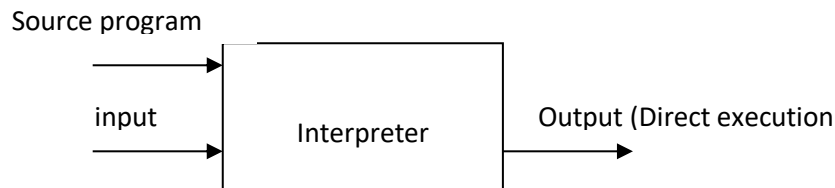
29. What do you meant by passes?

A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by a subsequent pass. In an implementation of a compiler, portions of one or more phases are combined into a module called pass.

30. Define Interpreters. [April/May 2011]

Interpreters

- An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input.
- It does not produce the object code rather each time the program needs execution. The model for interpreter is as shown in figure



- Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter.
- The process of interpretation can be carried out in following phases.
 1. Lexical analysis
 2. Syntax analysis
 3. Semantic analysis
 4. Direct execution.

31. What are the functions of preprocessor? [Nov/Dec 2007]

Preprocessor perform the following functions

1. Macro processing
2. File Inclusion
3. "Rational Preprocessors
4. Language extension

32. Define concrete and abstract syntax with example. [May/June 2009]

- Concrete syntax is the surface level of a language (think, strings)
- Abstract syntax is the deep structure of a language (think, trees/terms)
- The abstract syntax defines the way the programs look like to the evaluator/compiler.
- Parsers convert concrete syntax into abstract syntax and have to deal with ambiguity
- It consists of a set of rules (productions) that define the way programs look like to the programmer

33. Define panic mode.

Panic mode error recovery is deleting the successive characters from the remaining input until the lexical analyzer can find a well formed token.

34. What are the phases included in front end of a compiler? What does the front end produce? May -11

The phases of front end of compiler are lexical analysis, syntax analysis and semantic analysis. The front end produces an intermediate code.

35. Define the term cross compiler. Nov/Dec 05

There may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler.

36. Difference between phase and pass.

Phase

The process of compiler is carried out in various steps.

These steps are referred as phases.

The phases of compilation are lexical analysis, syntax analysis, intermediate code generation, code generation code optimization.

Pass

Various phases are logically grouped together to form a pass.

The process of compilation can be carried out in single pass or in multiple passes.

37. What are the factors affecting number of passes in compiler?

Various factors affecting the number of passes in compiler are

1. Forward reference
2. Storage limitations
3. Optimization.

38. What are machine dependent and machine independent phases?

The Machine dependent phases are code generation and code optimization phases.

The Machine independent phases are lexical analyzers, syntax analyzers, semantic analyzers.

39. What are the different parameter passing mechanisms?

- 1) Call-by-value
- 2) Call-by-reference

40. Define Scope of declaration.

The scope of a declaration of a variable x is the region of the program in which the use of x is referred to this declaration.

41. Define Dynamic scope of a variable.

The dynamic scope refers to the fact anything to be known only during execution. A use of name x refers to the declaration of x in the most recently called procedure containing the declaration of x.

42. What are the classifications of a compiler?

Compilers are classified as:

2. Single-pass
3. Multi-pass
4. Load-and-go
5. Debugging or optimizing.

43. What are the Error Recovery actions in Lexical Analyzer? (Nov/Dec 2008)

1. Deleting an extraneous character
2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. transposing two adjacent characters.

44. State any two reasons as to why phases of compiler should be grouped. ? (May-June 2014)

Grouping of phases facilitates efficient intermediate code generation used portability.

45. Why is buffering used in lexical analysis? What are the commonly used buffering methods? (May-June 2014)

Buffers are used to reduce the overhead required to process an input character.

46. What are the issues in lexical analysis?(May/June2007)

Simpler design will improve performance

- Compiler efficiency is improved
- Compiler portability is enhanced.

1. Write a regular expression for an identifier and whitespace. (Nov/Dec 2013)

Identifier

An identifier is defined as a letter followed by zero or more letters or digits.

The regular expression for an identifier is given as

R.E=letter (letter | digit)*

white space

spaces and tabs are generally ignored by the compiler, expect to serve as delimiters in most language and are not put out as tokens.

Delim → blank|tab|newline

Ws → delim+

2. Mention the various notational shorthands for representing regular expressions.

- One or more instances (+)
- Zero or one instance (?)
- Character classes ([abc] where a,b,c are alphabet symbols denotes the regular expressions a | b | c.)
- Non regular sets

3. What is the function of a hierarchical analysis?

Hierarchical analysis is one in which the tokens are grouped hierarchically into nested collections with collective meaning. Also termed as Parsing.

4. What does a semantic analysis do?

Semantic analysis is one in which certain checks are performed to ensure that components of a program fit together meaningfully.

Mainly performs type checking.

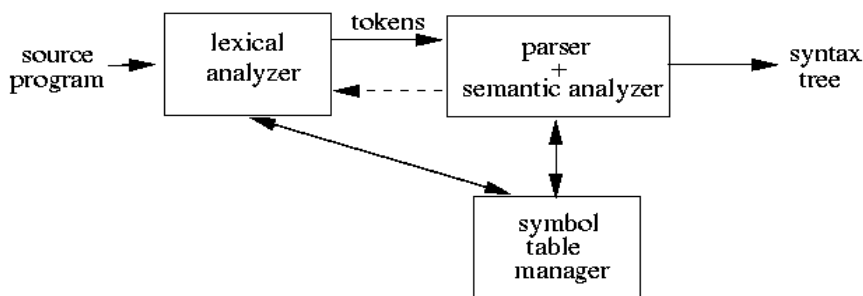
5. List the various error recovery strategies for a lexical analysis. [May/ June 2012]

Possible error recovery actions are:

- Panic mode recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters

6. What are roles and tasks of a lexical analyzer? [Nov/Dec 2011]

Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.



Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

7. What is the purpose of scanner?

The lexical analyzer or **scanner** reads one character at a time from the source program and divides the sequence of characters into a logically cohesive element called **tokens**.

8. Define sentence.

Collection of finite sequence of characters from the character set is called a sentence.

9. Define terminal symbol and non-terminal symbol

Terminal symbol can be defined as a basic symbol in the language. For ex if ; , + , switch, etc.,

Non-terminals are special symbols that denote set of strings.

10. What do you mean by LEX?

LEX is a software tool used for automatically writing Lexical Analyzers. LEX source is a specification of tokens in the form of a list of regular expressions together with an action for each regular expression.

Structure of LEX Program

declarations

%%

translation rules

%%

auxiliary functions

11. What do you mean by look ahead symbol?

The next input symbol for reduction/ derivation is called a look ahead symbol.

12. Define regular expression.

Regular expressions are the useful notations for defining the class of languages or regular sets. Regular expressions over an alphabet Σ can be constructed from the following rules.

- 1) ϵ is the regular expression denoting $\{\epsilon\}$, language containing only the empty string.
- 2) For each a in Σ a is a regular expression denoting $\{a\}$, the language having the only string a .
- 3) If R and S are regular expressions denoting the languages L_R and L_S respectively, then
 - i) $(R) | (S)$ is a regular expression denoting $L_R \cup L_S$
 - ii) $(R) . (S)$ is a regular expression denoting $L_R . L_S$
 - iii) $(R)^*$ is a regular expression denoting L_R^*

13. What are the tasks done by scanner other than token generation?

- Stripping out the comments
- Stripping out the white spaces
- Correlating error messages from the compiler with the source program
- Keep track of number of new line characters found, to provide a line number with an error message
- Making a copy of the source program with error messages marked in it

14. What are the reasons for separating the analysis phase to lexical analysis and parser?
[May/June 2013] [May/June 2009]

- Simpler design
- Improving compiler efficiency
- Enhancing compiler portability.

15. Define attribute.

The information about the token is called an *attribute*. The attributes are available in the corresponding tables.

16. Define panic mode.

Panic mode error recovery is deleting the successive characters from the remaining input until the lexical analyzer can find a well formed token.

17. Write down the possible error recovery actions taken by lexical analyzer.

- Deleting an extraneous character.
- Inserting a missing character.
- Replacing an incorrect character by a correct character.
- Transposing two adjacent characters.

18. Define character class and language

Any finite set of symbols is called a *character class*.

Example: (i)[a-z] denotes the RE $a|b|...|z$

(ii)[abc] denotes the RE $a|b|c$

Language denotes any set of strings over some fixed alphabet.

Example: $L = \{0^n 1^n | n > 0\}$

19. What are the operations supported by a string?

Prefix, suffix, sub string, proper prefix, proper suffix, proper sub string and subsequence.

20. What do you mean by subsequence?

Any string formed by deleting zero or more characters from any position (ie., not necessarily contiguous symbols) from S.

21. What is the purpose of the following notations $*$, $+$ and $?$

These operators are unary postfix operators.

- $*$ Represents zero or more occurrences
- $+$ Represents one or more occurrences
- $?$ Represents zero or one occurrences

22. Explain the notation useful for character class

The notation for representing a character class is $[]$. For example $[a-z]$ denotes the regular expression $a / b / c / d / e \dots / z$

23. What is the purpose of transition diagram?

Transition diagram describe the actions take place when analyzer is called by the parser to get the next token.

24. Define the purpose of `gettoken( )` function.

This function identifies the identified word is a key word or not. If it is a keyword it returns the same word. Otherwise it returns the token.

25. Define the purpose of `installid( )` function.

The `installid` function returns the attribute value of a token returned by `gettoken` function. If the lexeme is already available it will return a pointer. Otherwise it will install the lexeme, create a new pointer and return a pointer.

26. Define Look ahead operator.

The operator (normally `'/'`) used in the regular expressions to specify the lexical analyzers correctly is called look-ahead operator.

27. Define Finite automata.

The transition diagram constructed as a recognizer for a language is called as Finite Automata. The transition diagrams represent the regular expressions and accept any string x as input, answers **yes** if x is a sentence of the language generated by the regular expression or answers **no** $\square$

28. Define NFA. May/June-14

A finite automata which has more than one transition from some state for one input symbol is termed as Non-deterministic finite automata.

29. Define DFA. May/ ne-14

A finite automata is said to be deterministic if

- i) it has no transitions on input
- ii) for each state s and input symbol $'a'$, there is at most one edge labeled $'a'$ leaving s .

30. Write down the structure of LEX program.

A LEX source program consists Auxiliary Definitions and Translation Rules.

declarations

`%%`

translation rules

`%%`

auxiliary functions

Auxiliary definitions are the statements of the form

$$D_1 = R_1$$

$$D_2 = R_2$$

.....

$D_n = R_n$ where each D_i is a distinct name and each R_i is a regular expression chosen from $\square\square\square\square D_1 \square\square D_2 \square\square\square D_3 \dots \square\square D_{i-1}$, ie the characters or previously defined names.

Translation rules are the statements of the form

$$P_1 \{ A_1 \}$$

$$P_2 \{ A_2 \}$$

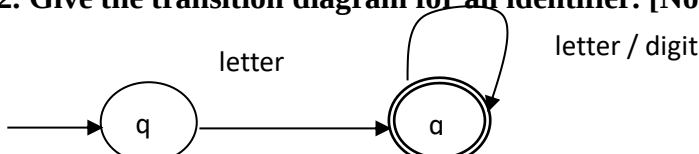
.....

$P_n \{ A_n \}$ where each P_i is a regular expression called pattern over the alphabet consisting of $\square$ and the auxiliary definition names. Each A_i is a program fragment describing the action to be taken when token P_i is found.

31. Define concrete and abstract syntax with example. [May/June 2009]

- Concrete syntax is the surface level of a language (think, strings)
- Abstract syntax is the deep structure of a language (think, trees/terms)
- The abstract syntax defines the way the programs look like to the evaluator/compiler.
- Parsers convert concrete syntax into abstract syntax and have to deal with ambiguity
- It consists of a set of rules (productions) that define the way programs look like to the programmer

32. Give the transition diagram for an identifier. [Nov/Dec 2011]



33. Why is buffering used in lexical analysis? What is the commonly used buffering methods? [May/June-14]

- Use a lexical analyzer generator, such as Lex compiler to produce the lexical analyzer from a regular expression based specification.
- Write the lexical analyzer in a conventional system-programming language, using the I/O facilities of that language to read the input.
- Write the lexical analyzer in assembly language and explicitly manage the reading of input
- the commonly used buffering methods are sentinel, buffer pairs.

34. Define LEXEME.(May/June 2014)

Collection or group of characters forming tokens is called Lexeme

35. Define tokens, patterns and lexemes. (Nov/Dec 2016)

Tokens

Sequence of characters that have a collective meaning.

A token is a string of characters, categorized according to the rules as a symbol (e.g., keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.). The process of forming tokens from an input stream of characters is called **tokenization**.

LEXEME: Collection or group of characters forming tokens is called Lexeme.

PATTERN: A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const

if	if	if
relation	<,<=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and “ except”

36. List the operations on languages. May/June2016

- ☐ **Union** - $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
- ☐ **Concatenation** – $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
- ☐ **Kleene Closure** – L^* (zero or more concatenations of L)
- ☐ **Positive Closure** – L^+ (one or more concatenations of L)

37. Write a grammar for branching statements. May/June2016

A grammar for branching statements

```

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

```

38. Divide lexical analysis phase.

1. Scanning Phase: This does simple tasks.
2. Lexical analysis phase: This does complex tasks.

39. What are the issues in lexical analysis?

1. Separation of phases
2. Compiler efficiency is improved
3. Compiler portability is enhanced
4. The lexical analyzer has to recognize the longest possible string.
5. Skipping comments
6. Symbol table interface.

40. Define tokens.

Token is a sequence of characters that can be treated as a single logical.

41. List 4 software tools that generate parser.

LEX, YACC, BISON, LEMON

41. Define Recognizer or Automata.

A recognizer for a language is a program that takes a string X, and answers “yes” if x is a sentence of that language, and “no” otherwise.

42. Algebraic properties of regular expressions.

$R|S = S|R$ – Commutative

$R|(S|T) = (R|S)T$ – Associative

$(RS)T = R(ST)$ – Concatenation is associative

$R^{**} = R^{*-*}$ is idempotent

43. What is Buffer Pair?

To reduce the amount of overhead required to process an input character, a buffer has been used. It is divided into 2 N-character halves where N is the number of characters on one disk block.

44. What is Sentinel?

The is a special character used to represent the end of the buffer which cannot be part of the source program.

:	:	E:	:	:	M:	*	:	eof	:	C	:	:	:	:	eof
---	---	----	---	---	----	---	---	-----	---	---	---	---	---	---	-----

45. What are equivalent states?

Two states P and Q are said to be equivalent if the transition from the state P on input on input X and the transitions from the state Q on input x are reaching the accepting state F.

46. Define String.

A language is a finite sequence of symbols drawn from that alphabet or letters. The strings are synonymously called as words.

The length of a string is denoted by $|S|$

The empty string can be denoted by ϵ .

The empty set of strings is denoted by Φ .

Example: banana

47. What is Kleene's closure?

The Kleene closure of a language L is written as

L^* = Zero or more occurrence of language L.