

UNIT I

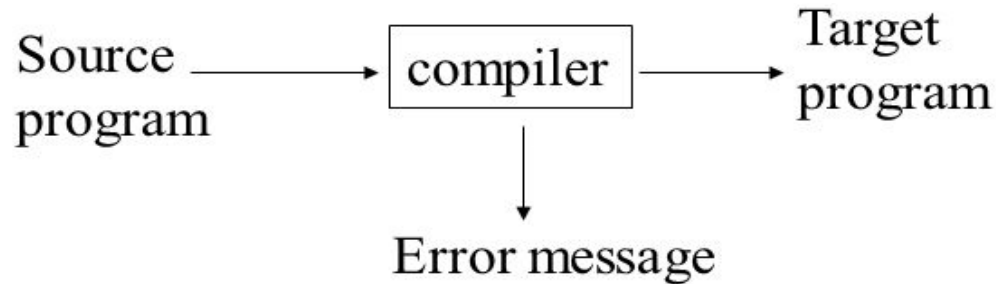
Introduction to Compilers

Translator

- ✓ A translator is a programming language processor that converts a computer program from one language to another.
- ✓ It discovers and identifies the error during translation.

Compiler

- ✓ A compiler is a computer program which helps to transform source code written in a high-level language into low-level machine language.
- ✓ It translates the code written in one programming language to some other language without changing the meaning of the code.
- ✓ The compiler also makes the end code efficient which is optimized for execution time and memory space.



Interpreter

An **interpreter** is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages.



Example

Input

Output

a := 3;
b := 4;
write (a*b)

Compiler

MOV A,='3'
MOV B,='4'
LOD 1,A
MUL 1,B
STO 1,TEMP
PUSH TEMP
CALL WRITE

a := 3;
b := 4;
write (a*b)

Interpreter

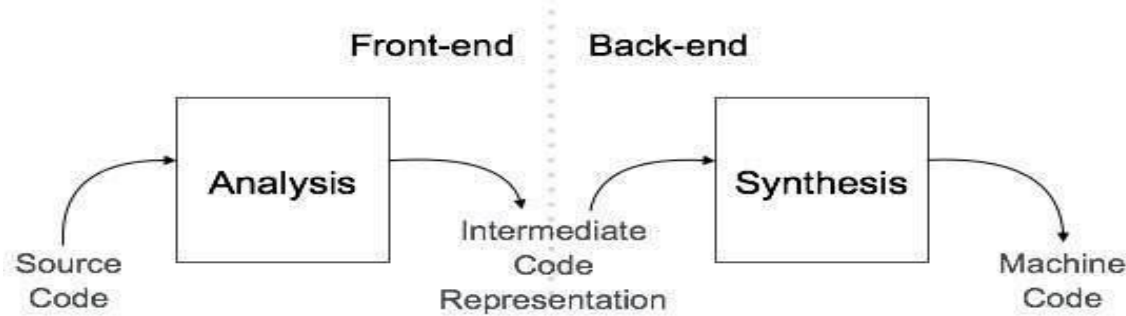
12

Comparison of Compiler and Interpreter

Compiler	Interpreter
Compiler takes Entire program as input	Interpreter takes Single instruction as input
Intermediate object is generated	No Intermediate object is generated
Conditional control statements are executes faster	Conditional control statements are executes slower
Memory requirement is more	Memory requirement is less
Program need not be compiled every time	Every time higher level program is converted into lower level program
Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted(if any)
Example: C, C++ and JAVA	Example: Basic and Python

Parts of the Compiler or Architecture of Compiler

A compiler can broadly be divided into two parts based on the way they compiler



Analysis Phase

- ✓ Known as the front-end of the compiler
- ✓ The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.
- ✓ The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

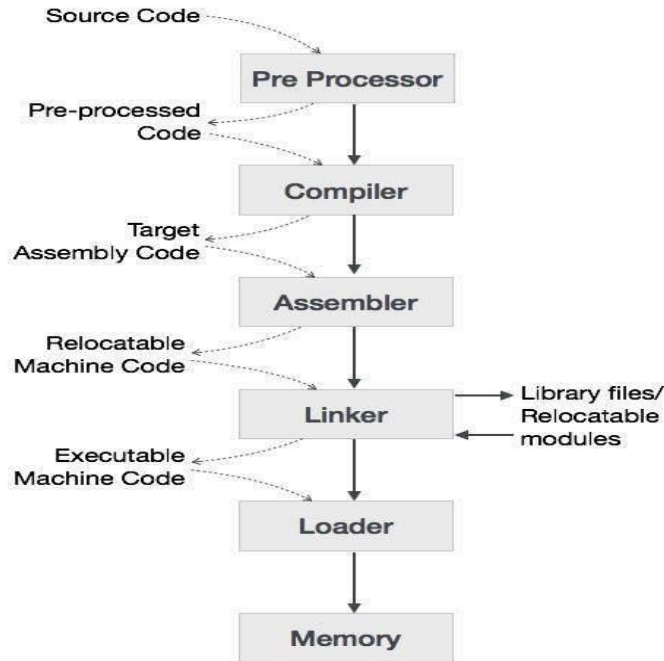
Synthesis Phase

- ✓ Known as the back-end of the compiler.
- ✓ The **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

Language Processing Systems or Cousins of Compiler

www.studymaterialz.in

- ✓ Any computer system is made of hardware and software.
- ✓ The hardware understands a language, which humans cannot understand. So write programs in high-level language, which is easier to understand and remember.
- ✓ These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



Preprocessor: The preprocessor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.

Compiler: A compiler is a computer program which helps to transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code

Assembler: It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.

Linker: The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.

Loader: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.

Compiler Construction Tools

- ✓ Compiler construction tools were introduced as computer-related technologies spread all over the world. They are also known as a compiler- compilers, compiler- generators or translator.
- ✓ These tools use specific language or algorithm for specifying and implementing the component of the compiler.

Scanner generators: This tool takes regular expressions as input. For example LEX for Unix Operating System.

Syntax-directed translation engines: These software tools offer an intermediate code by using the parse tree. It has a goal of associating one or more translations with each node of the parse tree.

Parser generators: A parser generator takes a grammar as input and automatically generates source code which can parse streams of characters with the help of a grammar.

Automatic code generators: Takes intermediate code and converts them into Machine Language

Data-flow engines: This tool is helpful for code optimization. Here, information is supplied by user and intermediate code is compared to analyze any relation. It is also known as data-flow analysis. It helps you to find out how values are transmitted from one part of the program to another part.

Advantages

- ✓ Modification of user program can be easily made and implemented as execution proceeds.
- ✓ Type of object that denotes a various may change dynamically.
- ✓ Debugging a program and finding errors is simplified task for a program used for interpretation.
- ✓ The interpreter for the language makes it machine independent.

Disadvantages

- ✓ The execution of the program is slower.
- ✓ Memory consumption is more.

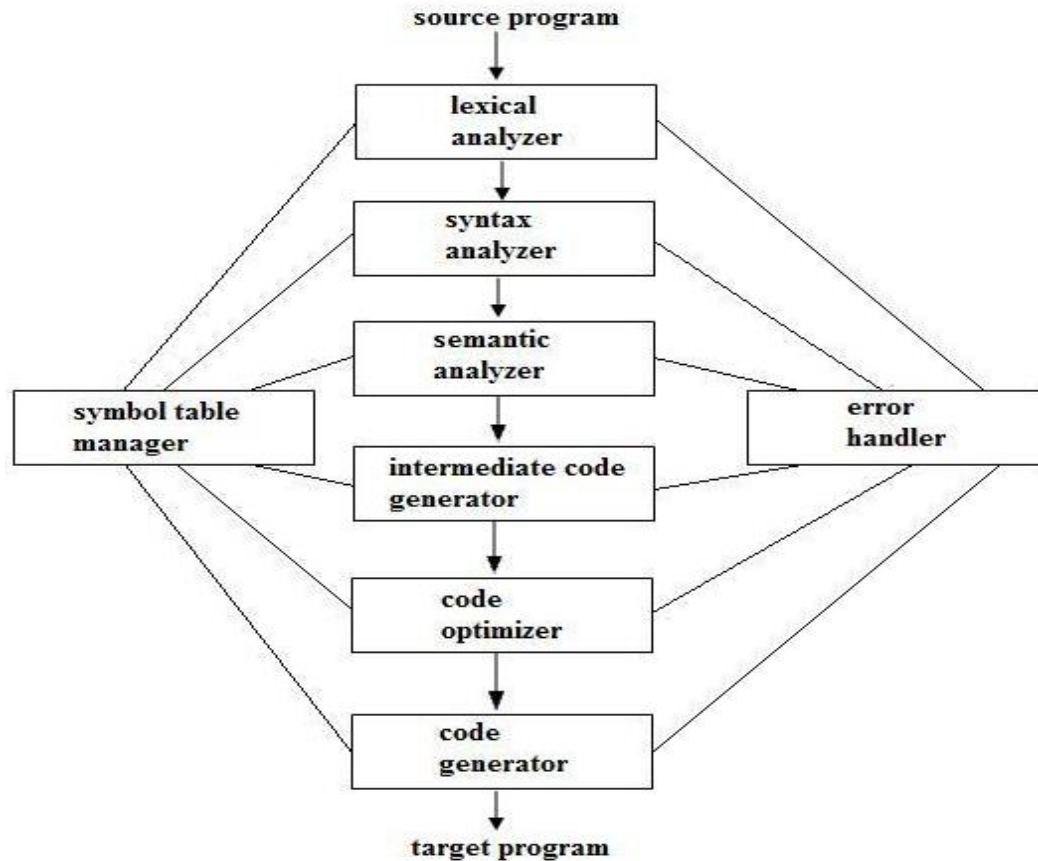
Topic

Structure of Compiler



Phases of a compiler

- ✓ A compiler operates in phases.
- ✓ A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below
- ✓ There are two phases of compilation.
 - a. Analysis (Machine Independent/Language Dependent)
 - b. Synthesis(Machine Dependent/Language independent)
- ✓ Compilation process is partitioned into no-of-sub processes called '**phases**'.
- ✓ In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.
- ✓ The symbol table, which stores information about the entire source program, is used by all phases of the compiler.



Lexical Analysis:

- ✓lexical analysis or scanning forms the first phase of a compiler.
- ✓The lexical analyzer reads the stream of characters which makes the source program and groups them into meaningful sequences called lexemes.
- ✓For each lexeme, the lexical analyzer produces tokens as output.
- ✓A token format is shown below.

<token-name, attribute-value>

- ✓These tokens pass on to the subsequent phase known as syntax analysis.
- ✓The token elements are listed below:

Token-name: This is an abstract symbol used during syntax analysis.

Attribute-value: This point to an entry in the symbol table for the corresponding token.

- ✓Information from the symbol-table entry 'is needed for semantic analysis and code generation.

For example, let us try to analyze a simple arithmetic expression evaluation in Lexical context www.studymaterialz.in

position = initial + rate * 60

The individual units in the above expression can be grouped into lexemes

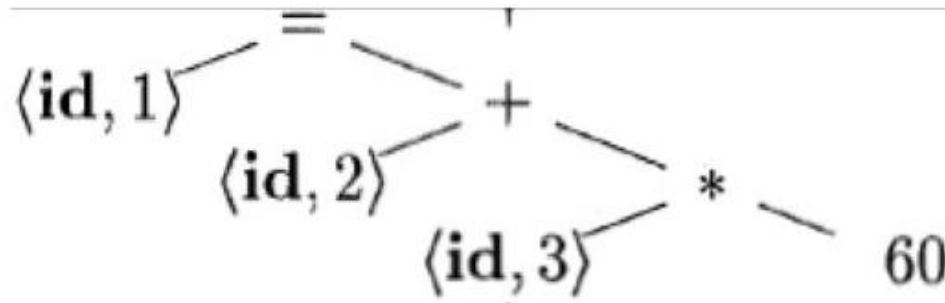
Lexeme	Token
position	Identifier
=	Assignment Symbol
initial	Identifier
+	Addition Operator
rate	Identifier
*	Multiplication Operator
60	Constant

The expression is seen by Lexical analyzer as

<id,1> <=> <id,2> <+> <id,3> <*> <60>

Syntax Analysis

- ✓ Syntax analysis forms the second phase of the compiler.
- ✓ The list of tokens produced by the lexical analysis phase forms the input and arranges them in the form of tree-structure (called the syntax tree). This reflects the structure of the program. This phase is also called parsing.
- ✓ The syntax tree consists of interior node representing an operation and the child of the node representing arguments. A syntax tree for the token statement is as shown in the above example.
- ✓ Operators are considered as root nodes of this syntax tree. In the above case = has left and a right node. The left node consists of <id,1> and the right node is again parsed and the immediate operator is taken as right node

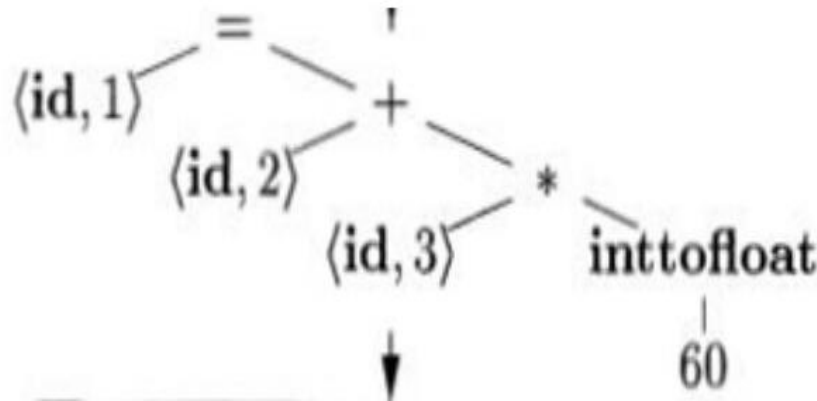


- ✓ Semantic analysis forms the third phase of the compiler. This phase uses the syntax tree and the information in the symbol table to check the source program for consistency with the language definition.
- ✓ This phase also collects type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- ✓ Type checking forms an important part of semantic analysis. Here the compiler checks whether each operator has matching operands
- ✓ **Coercions(some type conversions)** may be permitted by the language specifications. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Coercion exists in the example quoted

position = initial + rate * 60

- ✓ Suppose position, initial and rate variables are declared as float. Since the <id, 3> is a floating point, then 60 is also converted to floating point.
- ✓ The syntax tree is modified to include the conversion/semantic aspects. In the example quoted 60 is converted to float as inttofloat



Intermediate Code Generation

- ✓ Intermediate code generation forms the fourth phase of the compiler. After syntax and semantic analysis of the source program, many compilers generate a low level or machine-like intermediate representation, which can be thought as a program for an abstract machine.
- ✓ This intermediate representation must have two important properties:
 - (a) It should be easy to produce
 - (b) It should be easy to translate into the target machine.
- ✓ The above example is converted into three-address code sequence

There are several points worth noting about three-address instructions.

1. First each three-address assignment instruction has at most one operator on the right side. The multiplication precedes the addition in the source program
2. Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction.
3. Third, some "three-address instructions" like the first and last in the sequence have fewer than three operands.

t1 = inttofloat (60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3

Code Optimization

- ✓ Code Optimization forms the fifth phase in the compiler design.
- ✓ This is a machine-independent phase which attempts to improve the intermediate code for generating better (faster) target code.
- ✓ For example, a straightforward algorithm generates the intermediate code using an instruction for each operator in the tree representation that comes from the semantic analyzer.
- ✓ The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the int-to-float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.
- ✓ Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform the above three-address code sequence into a shorter sequence.

t1 = id3 * 60.0

id1 = id2 + t1

Code Generator

- ✓ Code Generator forms the sixth phase in the compiler design. This takes the intermediate representation of the source program as input and maps it to the target language.
- ✓ The intermediate instructions are translated into sequences of machine instructions that perform the same task. A critical aspect of code generation is the assignment of registers to hold variables.
- ✓ Using R1 & R2 the intermediate code will get converted into machine code.

LDF R2, id3

MULF R2, R2, #60.0

LDF R1, id2

ADDF R1, R1, R2

STF id1, R1

- ✓ The first operand of each instruction specifies a destination.
- ✓ The F in each instruction tells us that it deals with floating-point numbers.
- ✓ The # signifies that 60.0 is to be treated as an immediate constant.

Symbol-Table Management

- ✓ An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- ✓ These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.
- ✓ The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

Error Handling

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

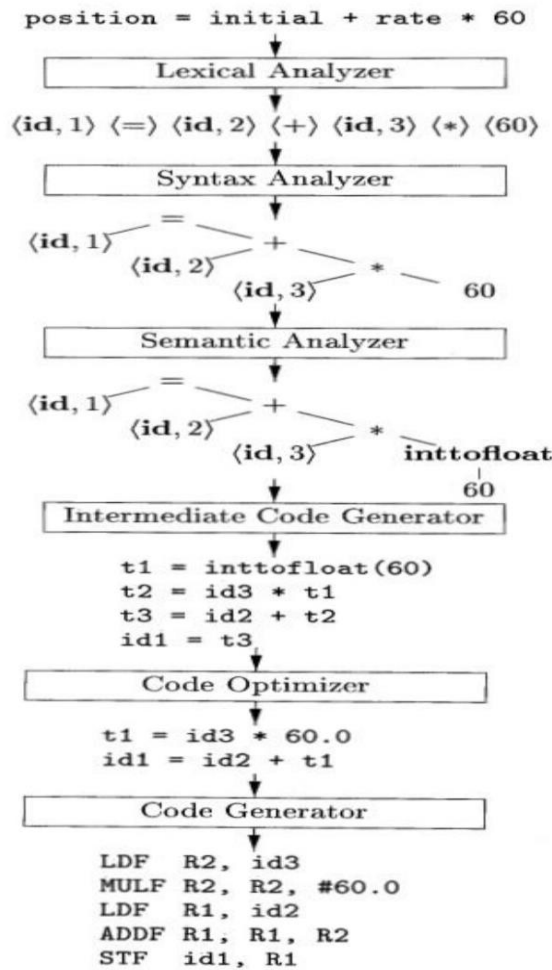
The Grouping of Phases into Passes

- ✓ Activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- ✓ For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- ✓ Code optimization might be an optional pass.
- ✓ Back-end pass consisting of code generation for a particular target machine.
- ✓ Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine.
- ✓ With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine.
- ✓ Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines

- 1
- 2
- 3

position	...
initial	...
rate	...

SYMBOL TABLE



Topic

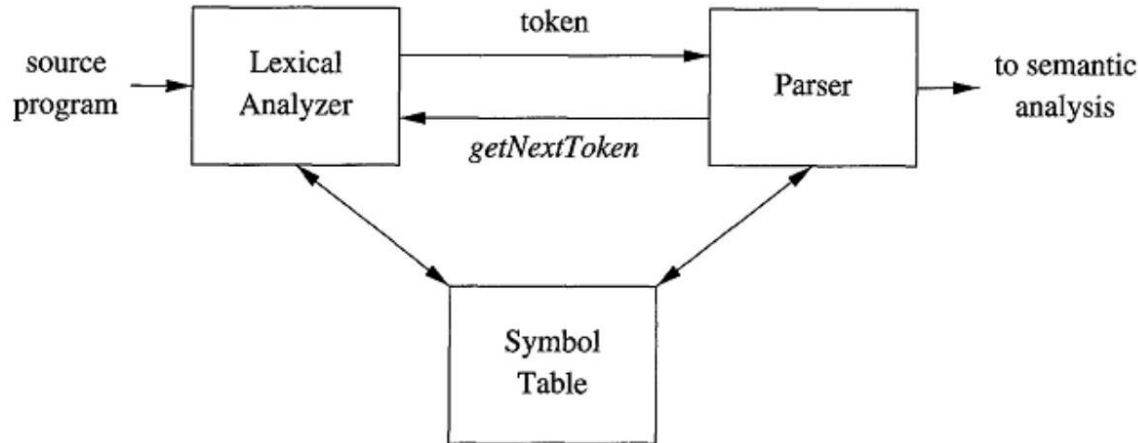
Lexical Analysis – Role of Lexical Analyzer

Over View of Lexical Analysis

- ✓ Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences.
- ✓ The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- ✓ If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer.
- ✓ It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
- ✓ Programs that perform lexical analysis are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error.
- ✓ It reads character streams from the source code, checks for legal tokens, and pass the data to the syntax analyzer when it demands.

Role of Lexical Analyzer

- ✓ As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- ✓ The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well



- ✓ The interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser. Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.
- ✓ Sometimes, lexical analyzers are divided into a cascade of two processes:
- a. Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
 - b. Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis versus Parsing

- ✓ There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.
1. Simplicity of design is the most important consideration.
 2. Compiler efficiency is improved.
 3. Compiler portability is enhanced.

Token

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.

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.

Lexeme

A lexeme is a sequence of characters in the source program that matches the pattern for a token

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

Attributes for Tokens

- ✓ When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phase's additional information about the particular lexeme that matched.
- ✓ For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.

Example: The token names and associated attribute values for the Fortran statement

E = M * C ** 2

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

< assign-op >

<id, pointer to symbol-table entry for M>

<mult-op>

<id, pointer to symbol-table entry for C>

<exp-op>

<number, integer value 2 >

Lexical Errors

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

fi(a == f(x)) . . .

- ✓ A lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier.
- ✓ Since fi is a valid lexeme for the token id, here the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- ✓ The simplest recovery strategy is "panic mode" recovery. Delete successive characters from the remaining input, until the lexical analyzer can find a well formed token at the beginning of what input is left.

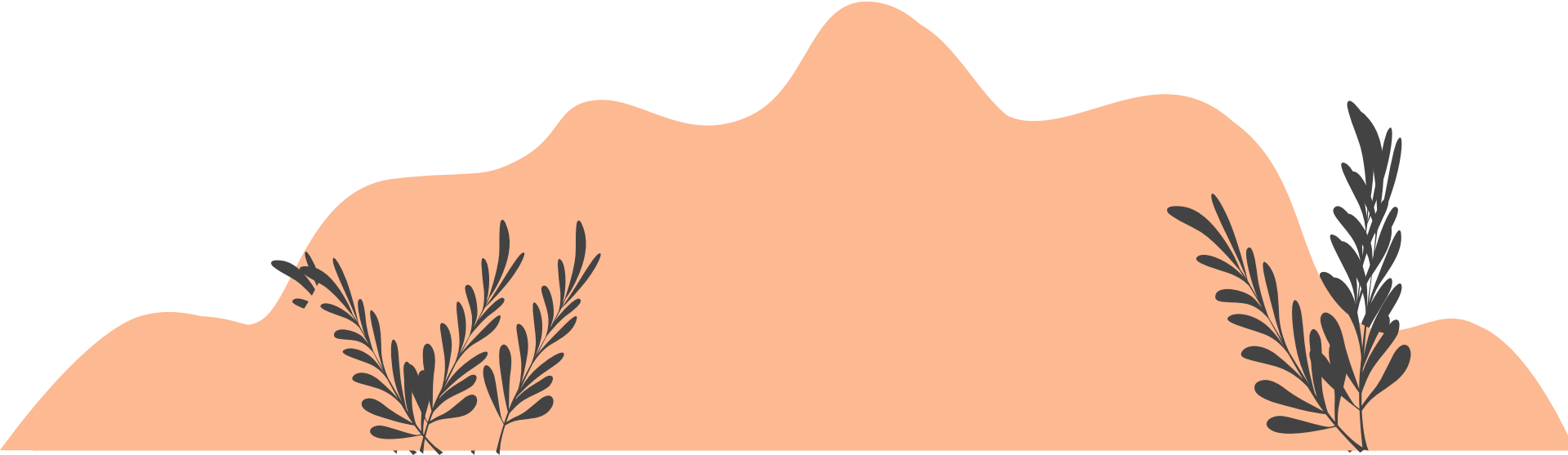
Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.



Topic

Input Buffering



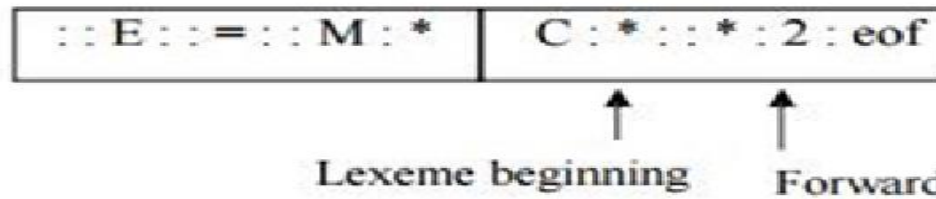
- ✓ The lexical analyzer scans the characters of the source program one at a time to discover tokens.
- ✓ Source program spends a considerable amount of time in the lexical analysis phase. So speed of lexical analysis is concern.
- ✓ Lexical analyzer needs to look ahead several characters before a match can be announced.
- ✓ The two buffer input scheme is useful when look-ahead on the input is needed to identify tokens.

i) Buffer Pairs

ii) Sentinels

Buffer Pairs

- ✓ In this buffer pair method, buffer is divided into two N-character halves
- ✓ Each buffer is of the same size N,
- ✓ N is usually the number of characters on one block. E.g., 1024 or 4096 bytes.
- ✓ Using one system read command we can read N characters into a buffer.
- ✓ If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.



Two pointers to the input are maintained:

1. Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found.
 - ✓ Once the next lexeme is determined, forward is set to the character at its right end.
 - ✓ The string of characters between the two pointers is the current lexeme.
 - ✓ After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found

Algorithm - Advancing forward pointer

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

Procedure AdvanceForwardPointer

begin

if forward at end of first half then

begin

reload second half;

forward := forward + 1

end

else if forward at end of second half then

begin

reload second half;

move forward to beginning of first half

end

else

forward := forward + 1;

end if

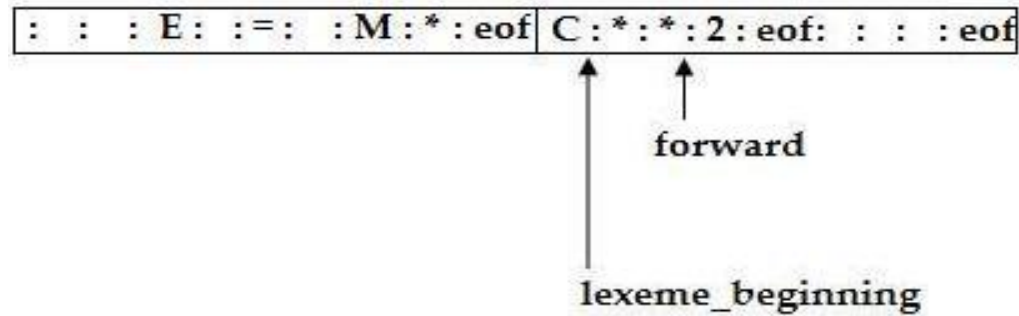
end Procedure AdvanceForwardPointer

Disadvantages of this scheme

- ✓ This scheme works well most of the time, but the amount of lookahead is limited.
- ✓ For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read

Sentinels

- ✓ To combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- ✓ The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- ✓ The sentinel arrangement is as shown below:



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

**Code to advance forward pointer:
Procedure LookAheadwithSentinel
begin**

```
    forward := forward + 1;
if forward ↑ = eof then
begin
    if forward at end of first half then
    begin
        reload second half;
        forward := forward + 1
    end
    else if forward at end of second half then
    begin
        reload first half;
        move forward to beginning of first half
    end
    else
        terminate lexical analysis
    end if
end if
```

end Procedure LookAheadwithSentinel

Advantages

- ✓ Most of the time, It performs only one test to see whether forward pointer points to an *eof*.
- ✓ Only when it reaches the end of the buffer half or *eof*, it performs more tests.
- ✓ The average number of tests per input character is very close to 1.

Topic

Specification of Tokens

Introduction

- ✓ To specify the tokens regular expression are used. When a pattern is matched by some regular expression then token can be recognized.
- ✓ Regular expressions are used to specify the patterns. Each pattern matches a set of strings.
- ✓ 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. Symbols are the collection of letters and characters.
- ✓ 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.
- ✓ The length of a string S , usually written as $|S|$, is the number of occurrences of symbols in S .
- ✓ The empty string, denoted ϵ , is the string of length zero
- ✓ For example, banana is a string of length six.

The following string-related terms are commonly used:

A prefix of string S - Any string obtained by removing zero or more symbols from the end of string S. For example, ban is a prefix of banana.

A suffix of string S - Any string obtained by removing zero or more symbols from the beginning of S. For example, nana is a suffix of banana.

A substring of S - Obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.

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.

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:

Union of L and M = $L \cup M$

$$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$$

Concatenation L and M = LM

$$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$$

Kleen closure of L = L^*

$$L^* = \bigcup_{i=0} L^i$$

Positive closure of L = L^+

$$L^+ = \bigcup_{i=1} L^i$$

Regular Expressions

- ✓ Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols.
- ✓ The grammar defined by regular expressions is known as **regular grammar**.
- ✓ The language defined by regular grammar is known as **regular language**.
- ✓ Each regular expression r denotes a language $L(r)$.

Rules that define the regular expressions over some alphabet Σ

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
 - i) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$
 - ii) $(r).(s)$ is a regular expression denoting the language $L(r).L(s)$.
 - iii) $(r)^*$ is a regular expression denoting $(L(r))^*$.

Example: letter (letter | digit) *

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 $r = s$.

Algebraic Properties Of Regular Expression

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

- i) $|$ is commutative: $r|s = s|r$
- ii) $|$ is associative: $r|(s|t) = (r|s)|t$
- iii) Concatenation is associative: $(rs)t = r(st)$
- iv) Concatenation is distributive: $r(s|t) = rs|rt$
- v) ϵ is identity element for concatenation $\epsilon.r = r.\epsilon = r$
- vi) Closure is idempotent: $r^{**} = r^*$

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

- $d_1 \rightarrow r_1$ Each d_i is a distinct name.
- $d_2 \rightarrow r_2$ Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.
- $d_3 \rightarrow r_3$
-
- $d_n \rightarrow r_n$

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

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

digit $\rightarrow 0 \mid 1 \mid \dots \mid 9$

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

Notations Of Regular Expression

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

1. One or more instances (+): The unary postfix operator + means “one or more instances of” regular expression. $(r)^+$ is a regular expression that denotes $(L(r))^+$

2. Zero or more instance (*): The operator “*” denotes zero or more instances of regular expressions. $(r)^*$ is a regular expression that denotes $(L(r))^*$

3. Zero or one instance (?): The unary postfix operator ? means “zero or one instance of”. $(r)?$ is a regular expression that denotes the language $L(r) \cup \{\epsilon\}$

4. Character Classes: The character class such as $[a - z]$ denotes the regular expression $a \mid b \mid c \mid d \mid \dots \mid z$.

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.

Topic

Recognition of Tokens

Recognition of Tokens

Consider the following grammar fragment

stmt $\rightarrow$ **if** **expr** **then** **stmt** | **if** **expr** **then** **stmt** **else** **stmt** | ϵ

expr $\rightarrow$ **term** **relop** **term** | **term**

term $\rightarrow$ **id** | **num**

The components $G=\{V,T,P,S\}$ for the above grammar is,

Variables= {stmt, expr, term}

Terminals= {if, then, else, relop, id, num}

Start symbol=stmt

The terminals generate the following regular definition,

if $\rightarrow$ if

then $\rightarrow$ then

else $\rightarrow$ else

relop $\rightarrow$ <|<=|=|<>|>|>=

id $\rightarrow$ letter(letter|digit)*

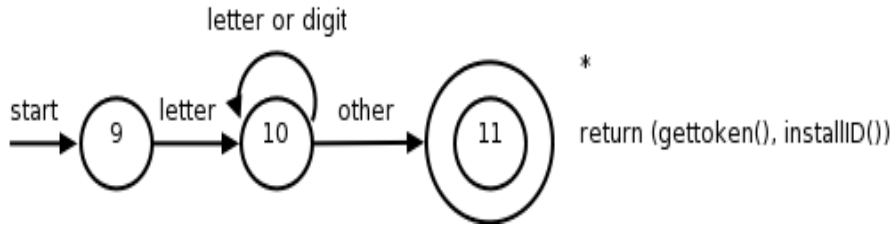
num $\rightarrow$ digit⁺ (.digit⁺)?(E(+|-)?digit⁺)?

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num

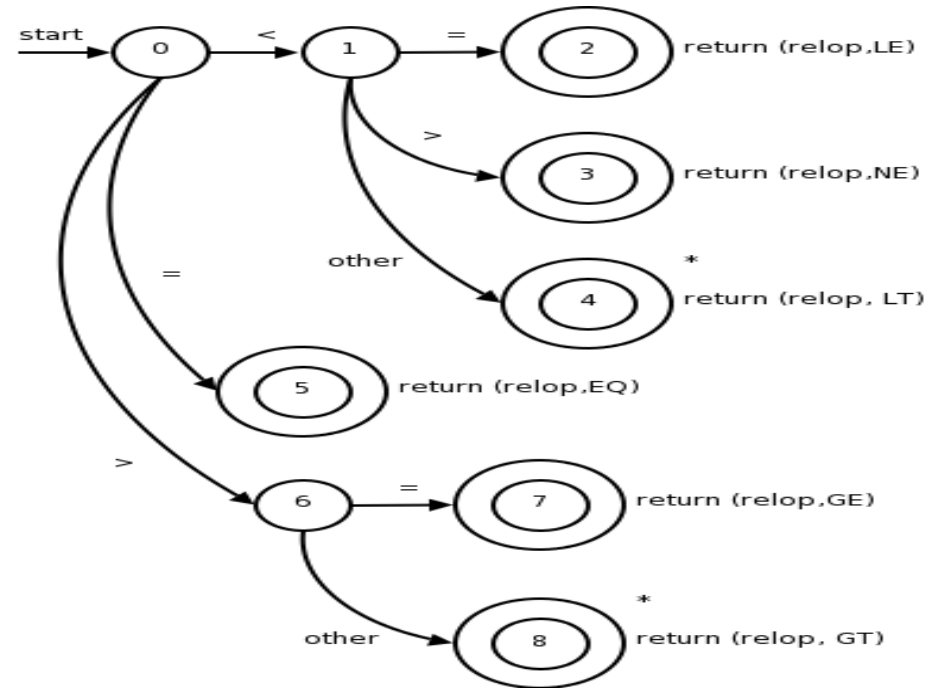
Transition diagrams

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.

Example: Transition diagram for identifier



Example: Transition diagram for relopertor



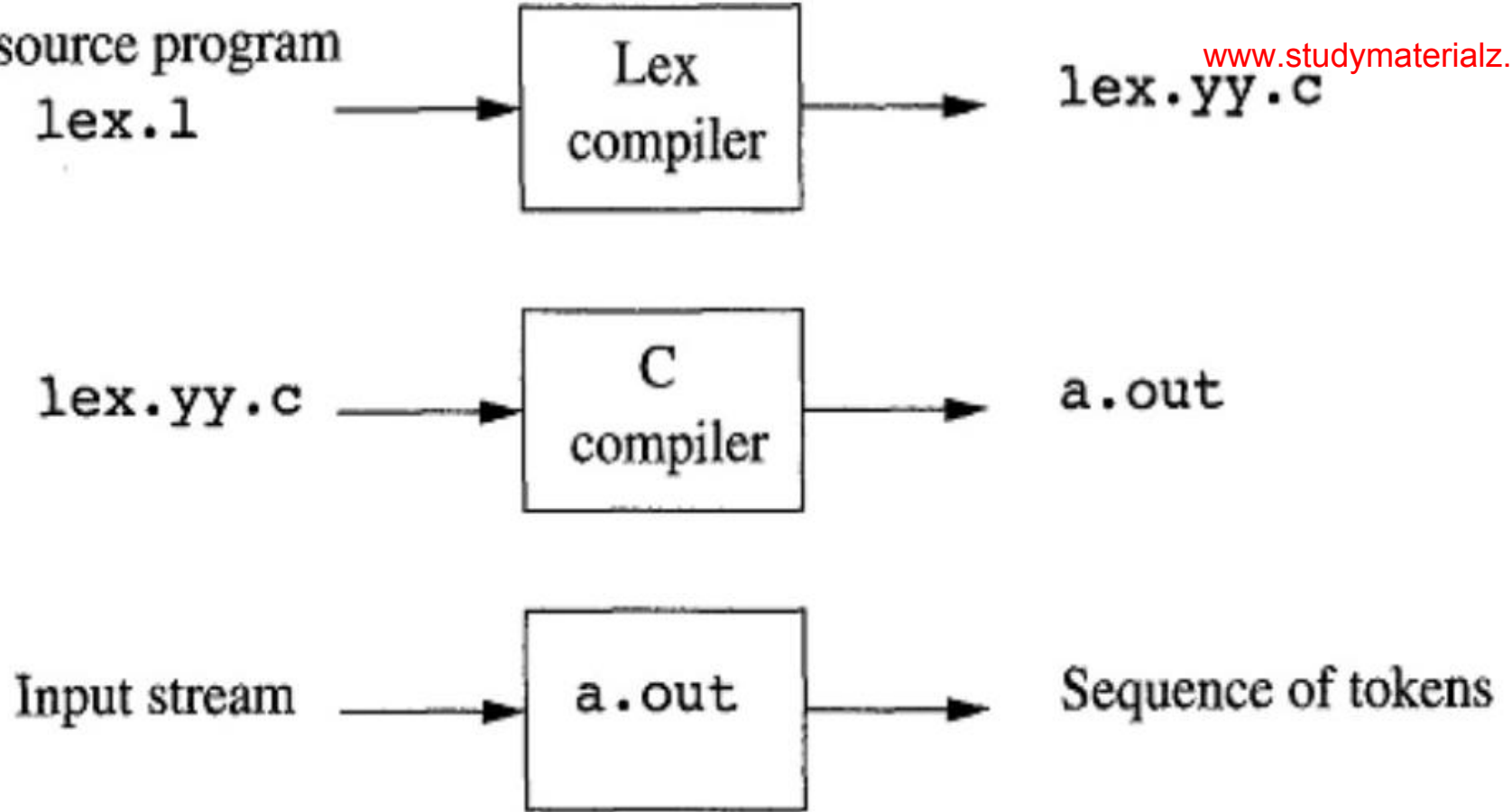
Topic

Lex

- ✓ 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.
- ✓ The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`,

Creating a lexical analyzer with Lex

- ✓ An input file, which we call `lex.l`, is written in the Lex language and describes the lexical analyzer to be generated.
- ✓ The Lex compiler transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`. The latter file is compiled by the C compiler into a file called `a.out`, as always.
- ✓ The C compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.



Structure of Lex Programs

A Lex program has the following form:

declarations

%%

translation rules

%%

auxiliary functions

- ✓ The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.
- ✓ The translation rules of a Lex program are statements of the form :

p1 {action 1}

p2 {action 2}

p3 {action 3}

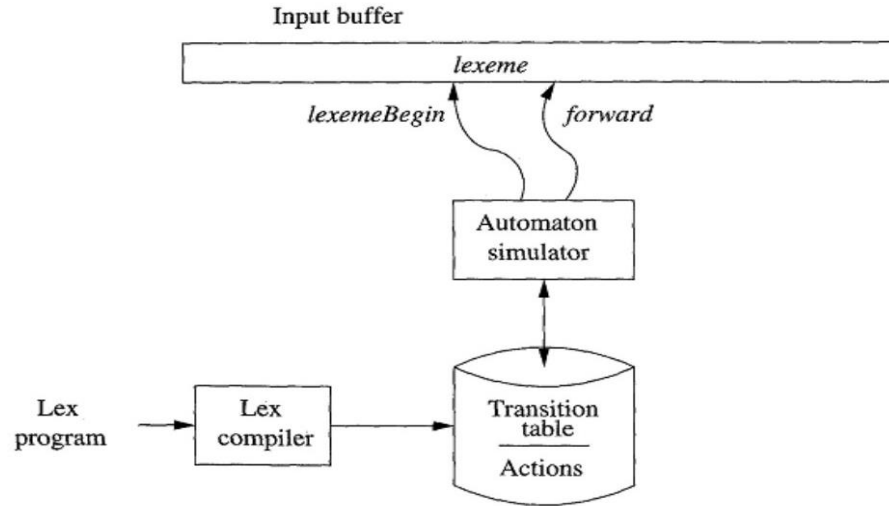
... ..

where each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

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

Design of a Lexical- Analyzer Generator

The following figure gives overview of the architecture of a lexical analyzer generated by Lex as follows.



These components are:

1. A transition table for the automaton.
2. Those functions that are passed directly through Lex to the output
3. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator

Topic

Finite Automata

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

A FA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

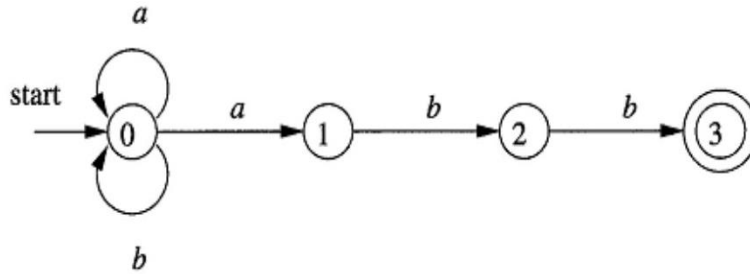
Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states Q .
2. A set of input symbols Σ , the input alphabet. We assume that ϵ , which stands for the empty string, is never a member of Σ .
3. A transition function δ that gives, for each state, and for each symbol in a set of next states.
4. A state q_0 from Q that is distinguished as the start state (or initial state).
5. A set of states F , a subset of Q that is distinguished as the accepting states (or final states).

This NFA graph is very much like a transition diagram, except:

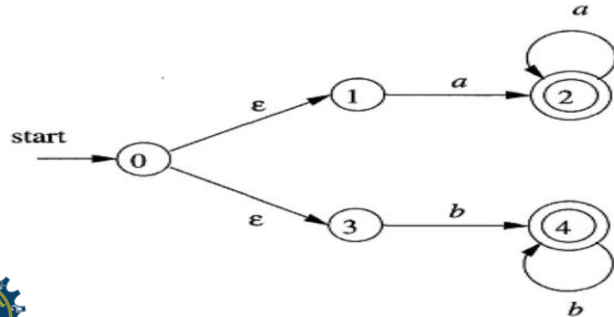
- The same symbol can label edges from one state to several different states, and
- An edge may be labeled by ϵ , the empty string, instead of, or in addition to, symbols from the input alphabet.



Deterministic Finite Automata

A deterministic finite automaton (DFA) is a special case of an NFA where:

- There are no moves on input ϵ , and
- For each state s and input symbol a , there is exactly one edge out of s labeled a .

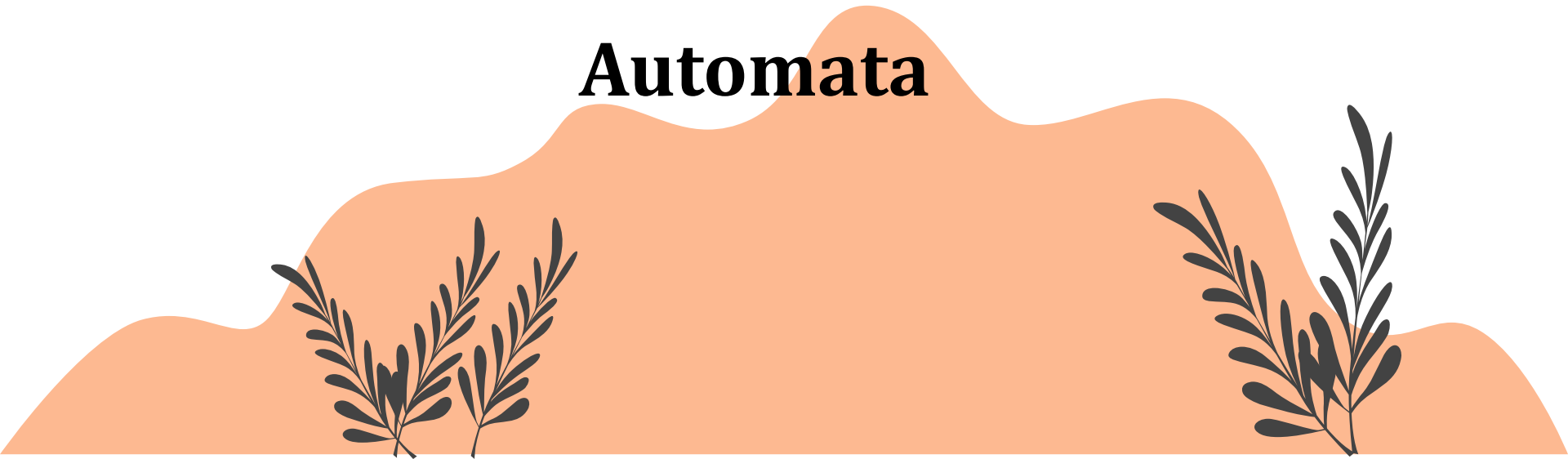




Topic

Regular Expressions to

Automata



Implementation of that software requires the simulation of a DFA. Because an NFA often has a choice of move on an input symbol or on ϵ its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

The steps are

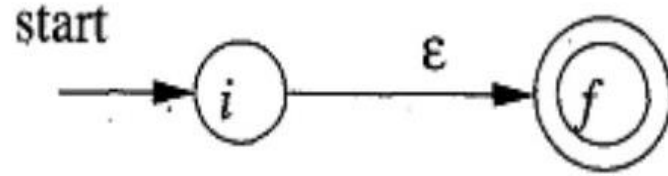
1. Construction of NFA from regular expression.
2. Conversion of NFA from DFA

Construction of NFA from a regular expression

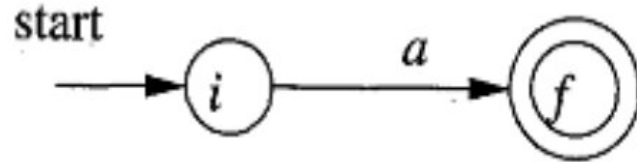
- ✓ McNaughton-Yamada-Thompson algorithm to convert a regular expression to an
- ✓ NFA.
- ✓ The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression.
- ✓ There are 2 rules for construction of NFA for sub expressions.

i) Basic rules for handling sub expressions with no operators

For expression ϵ constructs the NFA



For any sub expression a in Σ , construct the NFA as



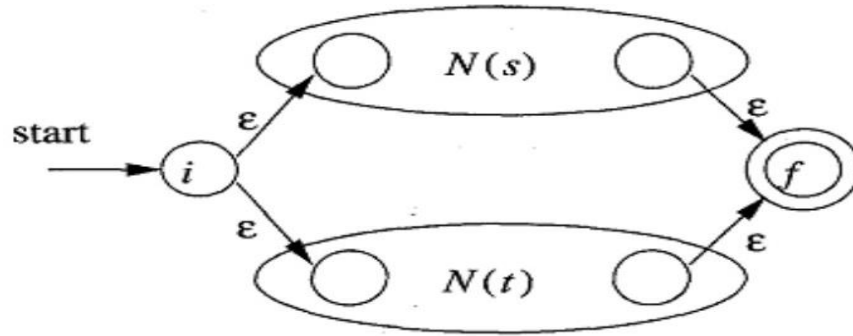
Where, i is a new state start state of this NFA,
 f is another new state accepting state for the NFA

ii) Induction Rules

- ✓ Used to construct larger NFA's from the NFA's for immediate sub expressions of given expressions.

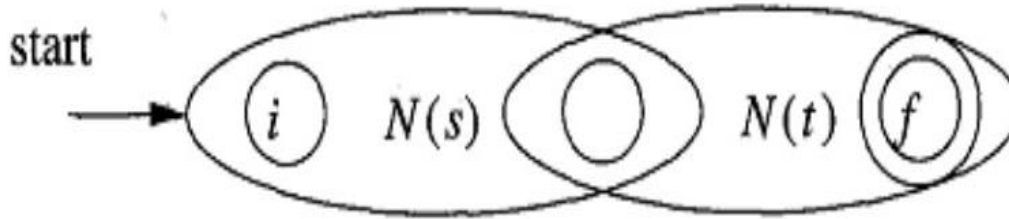
a) UNION

Suppose $r = s|t$. Then $N(r)$, the NFA for r , is constructed from $N(s)$ & $N(t)$. $N(r)$ accepts $L(s)UL(t)$, which is same as $L(r)$.



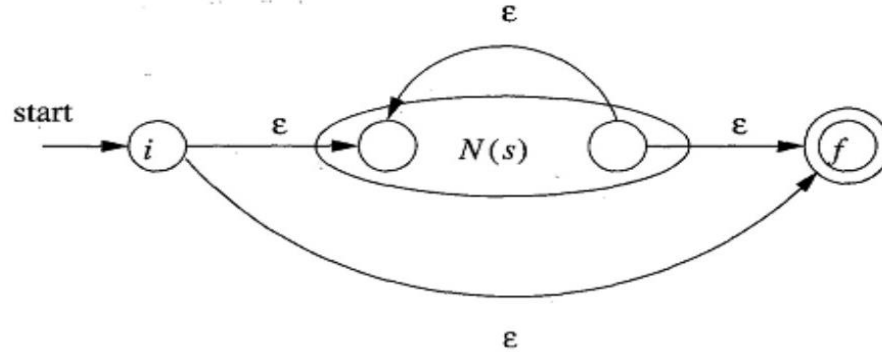
b) CONCATENATION

Suppose $r = st$. The start state of $N(s)$ becomes the start state of $N(r)$, and the accepting state of $N(t)$ is the only accepting state of $N(r)$. Thus, $N(r)$ accepts exactly $L(s)L(t)$, and is a correct NFA for $r = st$.



c) CLOSURE

Suppose $r = s^*$. Here, i and f are new states, the start state and lone accepting state of $N(r)$. $N(r)$ accepts ϵ , which takes care of the one string in $L(s)_0$, & accepts all strings in $L(s)_1$, $L(s)_2$ and so on



Finally, suppose $r = (s)$. Then $L(r) = L(s)$, and we can use the NFA $N(s)$ as $N(r)$.

The properties of $N(r)$ are as follows:

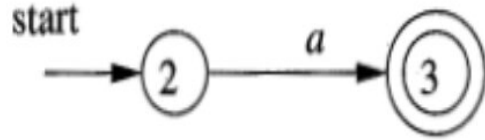
1. $N(r)$ has at most twice as many states as there are operators and operands in r .
2. $N(r)$ has one start state and one accepting state.
3. Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in C or two outgoing transitions, both on ϵ .

Problem

Construct an NFA for $r = (a \mid b)^* a b$

The following figure shows parse tree for r

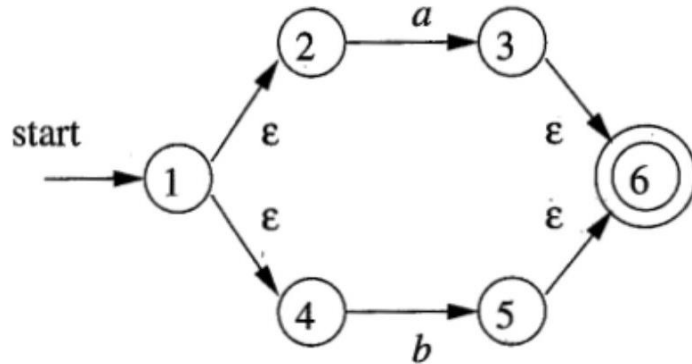
i) NFA for a



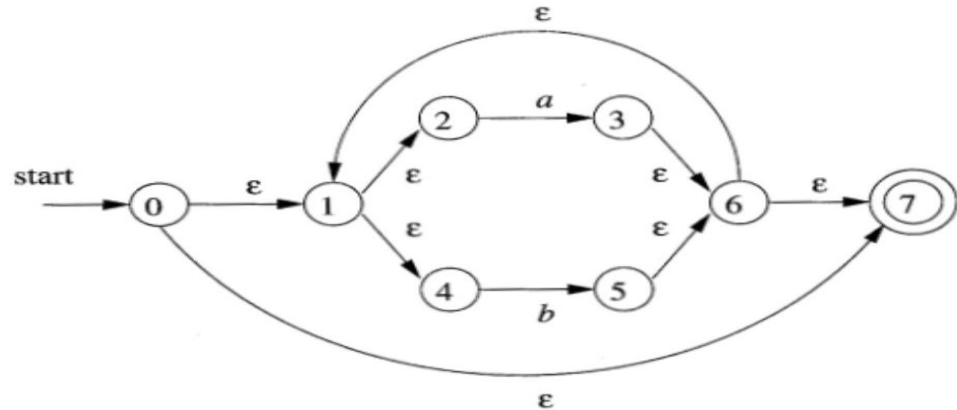
ii) NFA for a



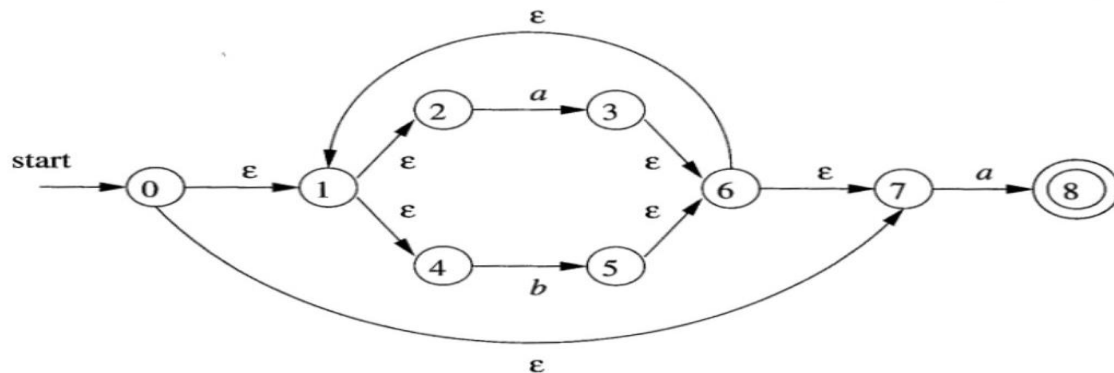
iii) NFA for $a \mid b$



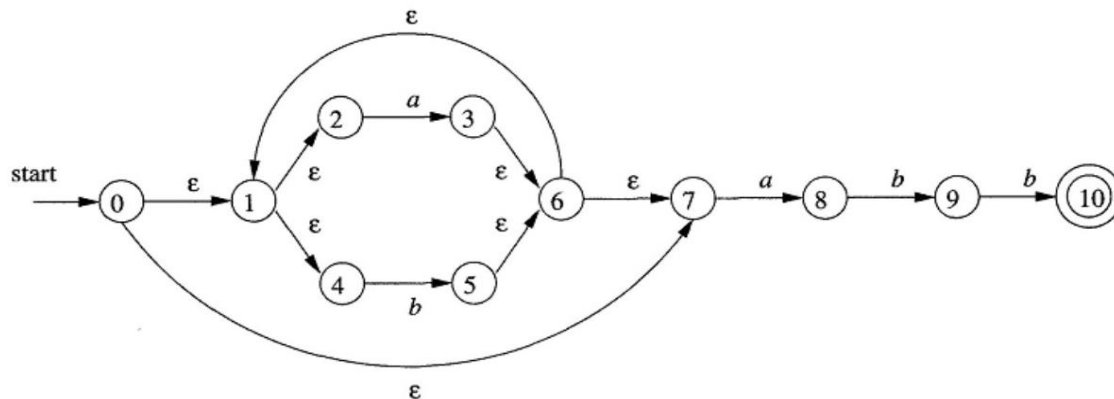
iv) NFA for $(a|b)^*$



v) NFA for $(a|b)^*a$



vi) NFA for $(a|b)^*abb$



Topic

Minimizing DFA

The general idea behind the subset construction algorithm is that each state of the constructed DFA corresponds to a set of NFA states.

The Subset Construction Algorithm

1. Create the start state of the DFA by taking the ϵ -closure of the start state of the NFA.
2. Perform the following for the new DFA state: For each possible input symbol:
 - a) Apply move to the newly-created state and the input symbol; this will return a set of states.
 - b) Apply the ϵ -closure to this set of states, possibly resulting in a new set.This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA.

DFA Minimization

- ✓ The process of converting a given DFA into an equivalent DFA with a minimum number of states is considered to be a minimization of DFA.
- ✓ It contains the minimum number of states.
- ✓ The DFA in its minimal form is called as a **Minimal DFA**

Minimization of DFA Using Equivalence Theorem

Step 1:

Eliminate all the dead states and inaccessible states from the given DFA (if any).

Dead State

All those non-final states which transit to itself for all input symbols in Σ are called as dead states.

Inaccessible State

All those states which can never be reached from the initial state are called as inaccessible states.

Step 2:

- ✓ Draw a state transition table for the given DFA.
- ✓ Transition table shows the transition of all states on all input symbols in Σ .

Step 3:

- ✓ Now, start applying equivalence theorem.
- ✓ Take a counter variable k and initialize it with value 0.
- ✓ Divide Q (set of states) into two sets such that one set contains all the non-final states and other set contains all the final states.
- ✓ This partition is called P_0 .

Step 4:

- ✓ Increment k by 1.
- ✓ Find P_k by partitioning the different sets of P_{k-1} .
- ✓ In each set of P_{k-1} , consider all the possible pair of states within each set and if the two states are distinguishable, partition the set into different sets in P_k .
- ✓ Two states q_1 and q_2 are distinguishable in partition P_k for any input symbol 'a'.

Step 5

- ✓ Repeat step-04 until no change in partition occurs.
- ✓ In other words, when you find $P_k = P_{k-1}$, stop.

Step 6:

- ✓ All those states which belong to the same set are equivalent.
- ✓ The equivalent states are merged to form a single state in the minimal DFA.

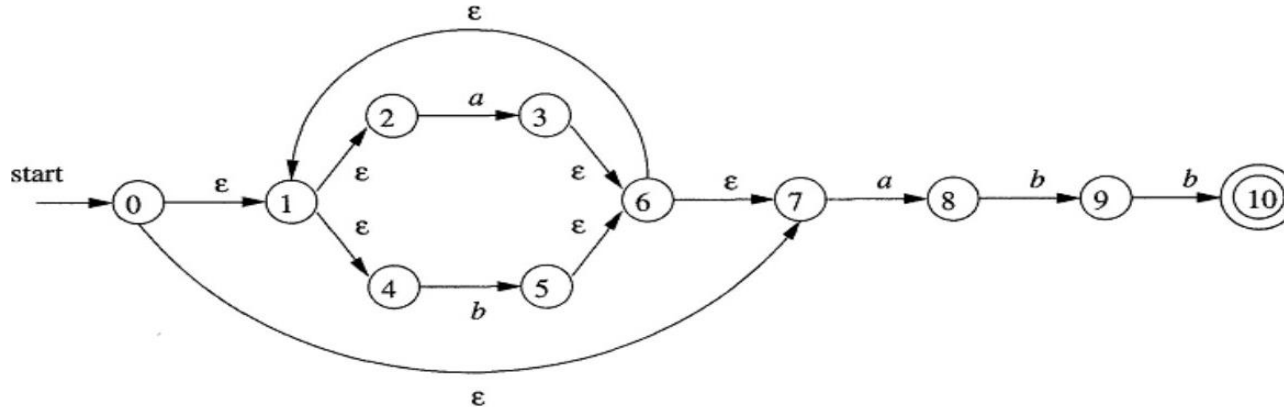
Number of states in Minimal DFA = Number of sets in P_k

Example:

Construct NFA for the RE $(a|b)^*abb$ and also convert it into Minimized DFA

Step 1: NFA Construction using Thompson Method

$(a|b)^*abb$



Step 2: Convert NFA into DFA using Subset construction Algorithm

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

$$\text{Dtran}[A, a] = \epsilon - \text{closure}(\text{move}(A, a))$$

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

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

$$= B$$

$$\begin{aligned}\text{Dtran}[A, b] &= \varepsilon - \text{closure}(\text{move}(A, b)) \\ &= \varepsilon - \text{closure}(5) \\ &= \{1, 2, 4, 5, 6, 7\} \\ &= C\end{aligned}$$

$$\begin{aligned}\text{Dtran}[B, a] &= \varepsilon - \text{closure}(\text{move}(B, a)) \\ &= \varepsilon - \text{closure}(3, 8) \\ &= \{1, 2, 3, 4, 6, 7, 8\} \\ &= B\end{aligned}$$

$$\begin{aligned}\text{Dtran}[B, b] &= \varepsilon - \text{closure}(\text{move}(B, b)) \\ &= \varepsilon - \text{closure}(5, 9) \\ &= \{1, 2, 4, 5, 6, 7, 9\} \\ &= D\end{aligned}$$

$$\begin{aligned}\text{Dtran}[C, a] &= \varepsilon - \text{closure}(\text{move}(C, a)) \\ &= \varepsilon - \text{closure}(3, 8) \\ &= \{1, 2, 3, 4, 6, 7, 8\} \\ &= B\end{aligned}$$

$$\begin{aligned}\text{Dtran}[C, b] &= \varepsilon - \text{closure}(\text{move}(C, b)) \\ &= \varepsilon - \text{closure}(5) \\ &= \{1, 2, 4, 5, 6, 7\} \\ &= C\end{aligned}$$

$$\begin{aligned}\text{Dtran}[D, a] &= \varepsilon - \text{closure}(\text{move}(D, a)) \\ &= \varepsilon - \text{closure}(3, 8) \\ &= \{1, 2, 3, 4, 6, 7, 8\} \\ &= B\end{aligned}$$

$$\begin{aligned}\text{Dtran}[D, b] &= \varepsilon - \text{closure}(\text{move}(D, b)) \\ &= \varepsilon - \text{closure}(5, 10) \\ &= \{1, 2, 4, 6, 7, 10\} \\ &= E\end{aligned}$$

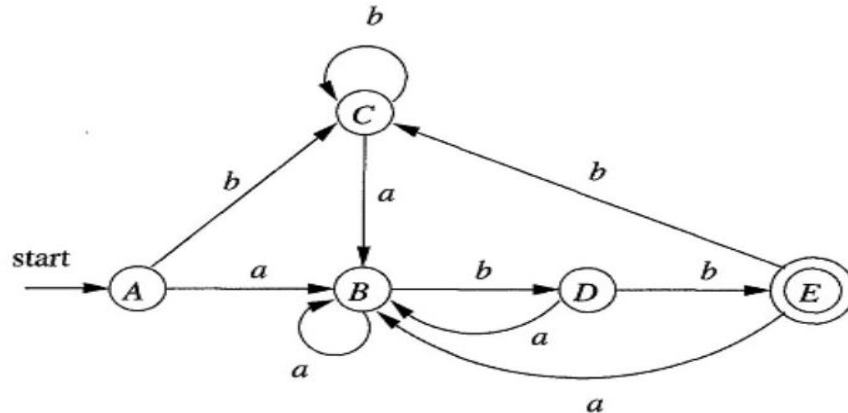
$$\begin{aligned}\text{Dtran}[E, a] &= \varepsilon - \text{closure}(\text{move}(E, a)) \\ &= \varepsilon - \text{closure}(3, 8) \\ &= \{1, 2, 3, 4, 6, 7, 8\} \\ &= B\end{aligned}$$

$$\begin{aligned}\text{Dtran}[E, b] &= \varepsilon - \text{closure}(\text{move}(E, b)) \\ &= \varepsilon - \text{closure}(5) \\ &= \{1, 2, 4, 5, 6, 7\} \\ &= C\end{aligned}$$

Step 3: DFA Transition Table

NFA State	DFA State	a	b
{0,1,2,4,7}	->A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,6,7,10}	*E	B	C

Step 4: DFA Transition Diagram



Step 5: DFA Minimization

[A B C D E]



Split the above set into Final and Nonfinal States

[A B C D] [E]



Split it into equal states and non-equal states

[A B C D] [E]

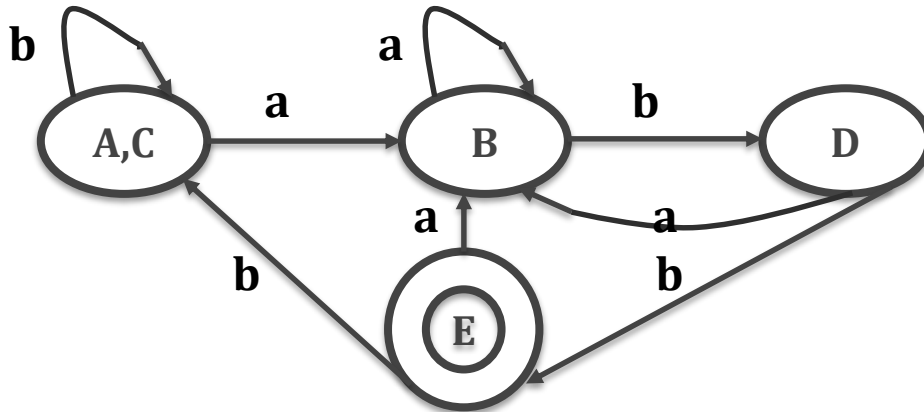
A & C are equal states. Because both are moves to same state while reading the input string

[A C] [B D] [E]

Step 6: Minimized DFA Transition Table

DFA State	a	b
->(A,C)	B	A
B	B	D
D	B	E
*E	B	A

Step 7: Minimized DFA Transition Diagram



Important states of an NFA

- ✓ To begin our discussion of how to go directly from a regular expression to a DFA, we must first dissect the NFA construction and consider the roles played by various states. We call a state of an NFA important if it has a non- ϵ out-transition.
- ✓ During the subset construction, two sets of NFA states can be identified if they:
 1. Have the same important states, and
 2. Either both have accepting states or neither does.
- ✓ By concatenating a unique right endmarker # to a regular expression r , we give the accepting state for r a transition on #, making it an important state of the NFA for $(r)\#$.
- ✓ By using the augmented regular expression $(r)\#$, any state with a transition on # must be an accepting state.
- ✓ To present the regular expression by its syntax tree, where the leaves correspond to operands and the interior nodes correspond to operators.
- ✓ An interior node is called
 - 1) a cat-node is labeled by the concatenation operator (dot)
 - 2) or-node is labeled by union operator |
 - 3) star-node is labeled by star operator *

- ✓ Leaves in a syntax tree are labeled by ϵ or by an alphabet symbol. To each leaf not labeled ϵ , we attach a unique integer. We refer to this integer as the position of the leaf and also as a position of its symbol.

Functions Computed From the Syntax Tree

- ✓ To construct a DFA directly from a regular expression, we construct its syntax tree and then compute four functions: nullable, firstpos, lastpos, and followpos.
 - ✓ Each definition refers to the syntax tree for a particular augmented regular expression $(r) \#$.
1. nullable(n) is true for a syntax-tree node n if and only if the sub expression represented by n has ϵ in its language.
 2. firstpos(n) is the set of positions in the sub tree rooted at n that correspond to the first symbol of at least one string in the language of the sub expression rooted at n .
 3. lastpos(n) is the set of positions in the sub tree rooted at n that correspond to the last symbol of at least one string in the language of the sub expression rooted at n .
 4. followpos(p), for a position p , is the set of positions q in the entire syntax tree such that there is some string $x = a_1a_2... a_n$, in $L((r)\#)$ such that for some i , there is a way to explain the membership of x in $L((r)\#)$ by matching a_i to position p of the syntax tree and a_{i+1} to position q .

Computing nullable, firstpos, and lastpos:

The basis & induction rules for nullable(n), firstpos(n) lastpos(n) & followpos(n) is as follows.

NODE n	nullable(n)	firstpos(n)	Lastpos(n)
A leaf labeled ϵ	true	Φ	Φ
A leaf with position i	false	{i}	{i}
An or-node $n = c_1 \mid c_2$	nullable(c_1) or nullable(c_2)	firstpos(c_1) U firstpos(c_2)	lastpos(c_1) U lastpos(c_2)
A cat-node $n = c_1 c_2$	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) firstpos(c_1) U firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) U lastpos(c_2) else lastpos(c_2)
A star-node $n = c_1^*$	True	firstpos(c_1)	Lastpos(c_1)

Computing followpos:

- ✓ There are only two ways that a position of a regular expression can be made to follow another.
- 1. If n is a cat-node with left child $C1$ and right child $C2$, then for every position i in $\text{lastpos}(c1)$, all positions in $\text{firstpos}(c2)$ are in $\text{followpos}(i)$.
- 2. If n is a star-node, and i is a position in $\text{lastpos}(n)$, then all positions in $\text{firstpos}(n)$ are in $\text{followpos}(i)$.

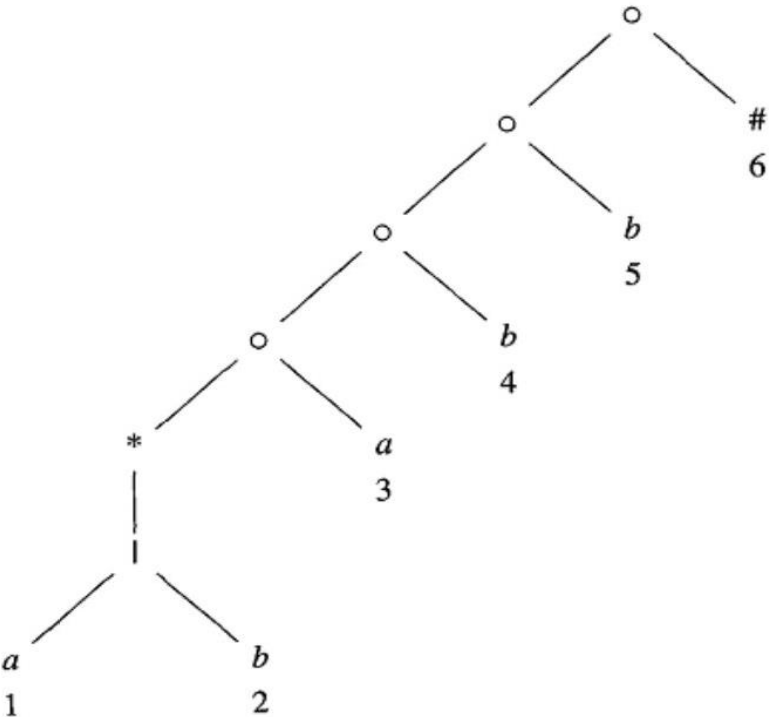
The followpos is almost an NFA without ε -transitions for the underlying regular expression, and would become one if we:

1. Make all positions in firstpos of the root be initial states,
2. Label each arc from i to j by the symbol at position i , and
3. Make the position associated with end marker $\#$ be the only accepting state

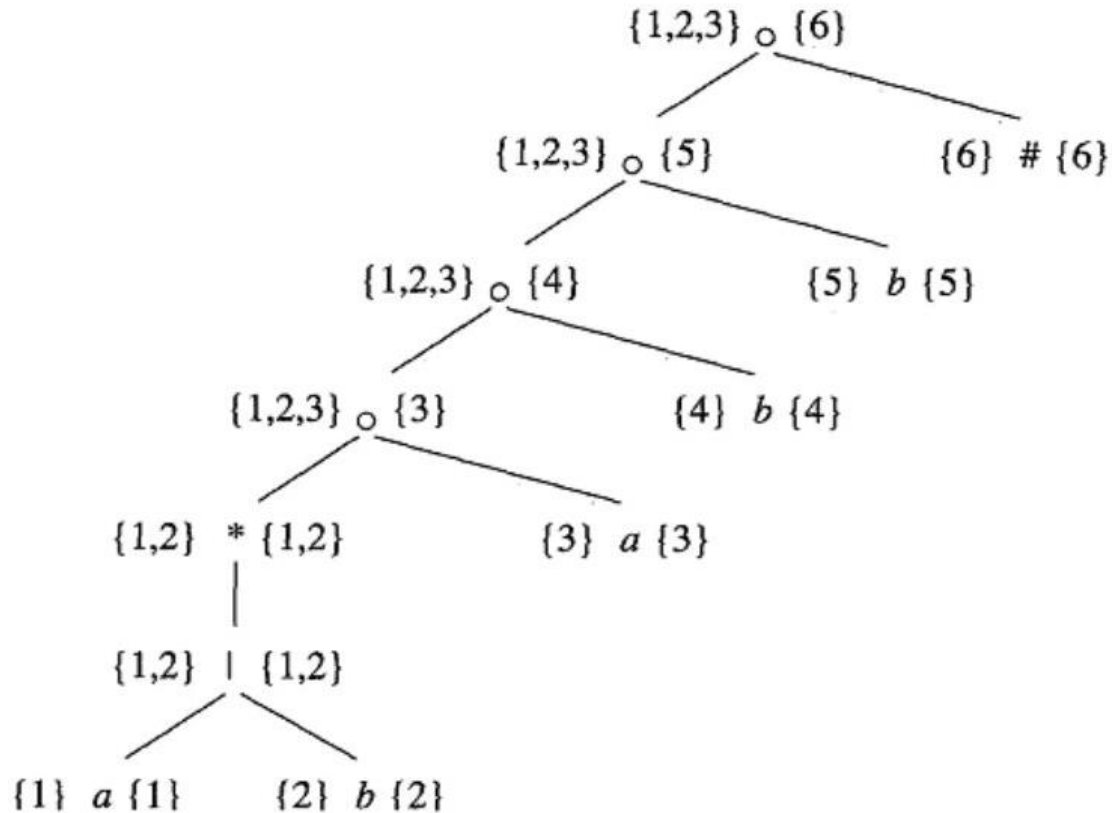
Example

Construct optimized DFA for the RE $(a|b)^*abb$

Step 1: Syntax Tree



Step 2: Computing firstpos and lastpos



Step 3: Computing followpos

NODE n	$followpos(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

Step 4: Directed Graph

