**Compiler:** (language processor)

# UNIT- I

# Introduction to Compiling

* Compiler is a program that read a program in one language (source language) and translates it into equivalent program in another language (target language).
* During compilation, role of compiler is to detect the errors in source code and report them to user.
* The target program is also called as executable program; it is called by user in order to produce outputs. We can also pass inputs to target program if necessary.



**Interpreter:** (language processor)

* Interpreter is another kind of language processor like compiler. Interpreter directly executes the operations in source program, in line by line on inputs supplied by user.

# Comparison between compiler and interpreter

|  |  |
| --- | --- |
| compiler | Interpreter |
| 1. Scans entire program and translates it aswhole into machine code. | 1. Translates program one statement at atime. |
| 2. Program which is compiled needs to berun by using another program. | 2. It takes source program along data andgives output |
| 3. Compiler doesn’t produce output. | 3. Interpreter produces output by line byline execution. |
| 4. Intermediate object code is generated. | 4. No intermediate object code isgenerated. |
| 5. Compiler is more efficient. | 5. Interpreter is less efficient. |
| 6. Design of compiler is hard. | 6. Design of interpreter is simple. |
| 7. programming languages like C, C++ usecompilers | 7. programming languages like python,ruby, use interpreter |

**Hybrid compiler:**

* Java language processors combine compilation and interpretation.
* Source program is first compiled and then interpreted with the help of JVM. Finally it produces output by supplying input to the process.
* Here JIT (just in time) compiler is used to execute some part of code, what will be modified in some program.

# Cross compiler:

* Cross compiler is a compiler capable of creating executable code for a platform other than one which the compiler is running.
* Example compiler runs on windows7 pc but generates code that runs on android Smartphone is cross compiler.

# Language processing system:

* In addition to compiler, several other programs may be required to create an executable target program.
* Source program divided into modules stored in separate files. The task of collecting source program is called pre-processor. Modified source program is then forwarded to compiler.
* Compiler produce an assembly program as its output, it is easier to debug.
* Assembly language is then processed by assembler. Assembler translates assembly language program into object code.
* Large programs compiled into pieces, so machine code linked with other object files and library files, those run on machines.
* Linker is program that links 2 object files containing compiled or assembled code to form a single file.
* Loader is program which collects all executable object files in memory for execution.

# Structure of Compiler :( Phases of Compiler)

* In compiler, we see that there are 2 parts
	1. Analysis 2. Synthesis
* Analysis part is also called as frontend and synthesis part is also called as backend.

## Analysis:

* It breaks up source program into pieces(lexemes)and check it with pattern for converting to tokens those tokens are needed for rest of compilation.
* It checks whether the source program is syntactically or semantically correct or not. If not it proceeds error information.
* Collected information should be stored in symbol table for the use of rest of compilation.
* Finally, it converts the source program into intermediate representation called intermediate code generator.

## Synthesis:

* It construct target program from the intermediate representation and put information in symbol table.
* It applies optimization, if optimization doesn’t expose any new problems.

# PHASES OF COMPILATION:

* Compiler operates in phases, each of which transforms source program from one representation to another.
* Analysis part is carried out in 4 phases and Synthesis part is carried out in 2 phases.
	1. Lexical analysis
	2. Syntax analysis
	3. Semantic analysis
	4. Intermediate code generation
	5. Code optimization
	6. Code generation
* Two other activities are along with these 6 phases those are symbol table and error handler.
* In the above 6 phases, code optimization phase should be optional.



## Lexical analysis:

* Lexical analysis is also called as scanning.
* It scans the source program and broken up into group of strings called token, in the form

<token name, attribute value>.

* This information should be useful for next phases. Example: Assignment statement in source program

Position= initial +rate\*60

* 1. Position is a lexeme converted into token <id, 1>
	2. Assignment symbol (=) is a lexeme converted into token <=>
	3. Initial is a lexeme mapped into token <id, 2>

4. + is a lexeme mapped into token <+>

1. Rate is a lexeme mapped into token <id, 3>
2. \* is a lexeme mapped into token <\*>
3. 60 is a lexeme mapped into token <60>
* Token names =, +, \* are abstract symbols for assignment, addition and multiplication operators.

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

## Syntax Analysis:

* Syntax analysis is also called as parsing.
* It creates tree like structure that shows the grammatical structure of token stream.
* In syntax tree interior node represent operations and children of node represent variables of operation or operands.
* Here first rate is multiplied with 60 then result will be added to initial. Final result will be copied to position.

## Semantic analysis:

* It uses syntax tree and information in symbol table to check the source program is semantically correct or not.
* Type checking is important in semantic analysis, because it checks that each operator has matching operands.
* In semantic analysis, language specifications (permit) allow some type conversions called coercions.
* Initially position, initial and rate are declared to input and 60 is an integer. Type checker discovers operation \* applied to real and integer. In this case, integer is converted to real by coercion.

## Intermediate code generator:

* After semantic analysis, many compilers generate intermediate representation for source program.
* Intermediate representation is easy to produce and it is easy to translate into target program.
* Intermediate representation can be done with the help of three address code.

t1 = int to real (60) t2 = id3 \* t1

t3 = id2 + t2 id1 = t3

1. **Code optimization:** (it is optional)
* Code optimization improves the intermediate code to get better target program.
* It reduces the unnecessary lines in the code without disturbing the meaning of the program.
* These optimizations improve the running time of the target program.

t1 = id3 \* 60.0 id1 = id2 + t1

## Code generators:

* Code generator takes intermediate representation of source program and maps it into target language.
* If target language is machine code, registers or memory locations are selected for each variable used by the program.

LDF R2, id3 MULF R2, #60.0 LDF R1, id2 ADDF R1, R2 STF id1, R1

R1 and R2 are Registers and F indicates floating point

## Symbol table management:

* It is an essential function of compiler to record identifiers used in source program.
* It collects information about various attributes of each identifier. These attributes provide information like name, type and scope of identifier.
* It is a data structure contains record for each identifier.

## Error handler:

* It detects the errors and reporting these errors with informative messages.
* Syntax and semantic analysis phases usually handle large fraction of errors in compilation.
* It must report the place in source program where error is detected, because of this user has better idea about error.

# Grouping of phases into passes:

* Compiler operates in phases, grouping of one or more phases into a module is called pass.
* Pass read the input program and writes an output program.
* Normally phases are grouped into 2 – passes, those are front end phases and back end phases.

1st pass – front end phases - Lexical analysis

* Syntax analysis
* Semantic analysis
* Intermediate code generation 2nd pass – backend phases - Code optimization (optional)
* Code generation
* In sometimes, phases of compilation can be grouped into one pass. All phases are in that pass.
* We produce compilers for different source languages for one target machine by combining different front ends with back end for target machine.
* Similarly, we produce compilers for different target machines by combining different front end with back ends for target machine.

# Compiler construction tools:

* Compiler construction tools are some specialized tools for helping in implementation of various phases of compilation.
* Some commonly used compiler construction tools are
1. **Scanner generator:** These generators produce lexical analyzers by taking regular expression as input.

**Example:** LEX scanner generator used in Linux.

1. **Parser generator:** These generators produce syntax analyzers by taking context free grammar as input.

**Example:** YACC parser generator used in Linux.

1. **Syntax directed translation engines:** It produces intermediate code by taking parse tree as input. Translation is done for each node of tree.
2. **Code generators:** It produces machine code by taking intermediate code as input.
3. **Data flow analysis engines: -** It is useful in code optimization, for collecting information about values from one part to other part of program.
4. **Compiler construction tools kits: -** These provide set of routines for constructing various phases in compilation.

# Cousins of compiler:

* Cousins of compiler means the context in which the compiler operates.
* Contexts are basically programs such as
	1. Pre-processor
	2. Assembler
	3. Loader
	4. Link editors or linker
* These all are parts of language processing system.

# Lexical Analysis

**Role of Lexical Analyzer:**

* Lexical analyser reads input characters in the source program group them into lexemes and produce sequence of tokens as output.
* Output of lexical analyser (sequence of tokens) sent to parser for syntax analysis.
* Lexical analyser interacts with symbol table to collect information about tokens before passing to parser.
* Interaction between lexical analyser and parser is implemented by method getNextToken() issued by parser.
* Some lexical analysers do additional tasks like stripping out comments and whitespaces. Another one is relating error messages to source program with line numbers.
* It is divided into 2 processes. One for deleting comments and whitespace and another one is for producing sequence of tokens.

# Lexical Analysis Vs Parsing(Syntax Analysis)

* There are number of reasons for which the lexical analyser is separated from parsing

## Simplicity of design:

* By separating lexical and syntax analysis, we can simplify at least one of this phases.
* We can design new language by separating these two phases will give better idea about complete design.

## Compiler efficiency is improved:

* Separate lexical analyser allows us to apply special techniques for lexical task, not parsing.
* In addition, we provide special buffering techniques add ed to lexical phase to speed up compiler.

## Compiler portability is enhanced:

* By separating lexical and syntax phases, input device specific particular can be restricted to lexical analyzer.
* Because of the separating lexical and syntax phases, one phase doesn’t disturb the work of other.

# Lexeme, Pattern and Token:

* Some terminologies used in lexical analysis
1. Lexeme: sequence of characters in source program Example: int i, num, t2;
2. Pattern: set of rules that describe token Example: L (L|D)\*
3. Token: it describe the category of input string Example: identifier, keyword…. Etc
* Most of the token used in many programming languages are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| Token | Informal description | Sample Lexeme | Pattern |
| if | characters i, f | if | if |
| else | characters e, l, s, e | else | else |
| comparison | < or > or <= or>= or == or != | <=, != | <|>|<=|>=|==|!= |
| id | letter followed by letter or digit | Pi, score, d2 | L (L|D)\* |
| number | any numeric constant | 3.14, 0, 2e23 | D(.D)?(E[+-]?D)? |
| literal | anything between “ “ | “core dumped” |  |

* Pattern for each keyword is same as keyword itself.

# Attributes of Tokens:

* When more than one lexeme can match pattern, lexical analyzer provides additional information in the form of attribute about particular lexeme that matched
* Attribute value for identifiers is pointer to symbol table entry for identifiers Example: Token names and associated attribute values for E=M\*C\*\*2

<id, 1><=><id, 2><\*><id, 3><\*\*><num, 2>

# Lexical errors:

* It is hard for lexical analyzer to tell, without the help of components that there is a source code error.
* If string fi encountered first time in c, lexical analyzer can’t tell whether fi is misspelling of keyword if or undeclared function identifier.

If(a==f(x)).......

* Lexical analyzer return token id to parser and parser handle an error due to transpose of letters
* A situation in which lexical analyzer is unable to proceed then simplest recovery strategy is panic mode recovery.
* In panic mode ,we delete successive characters form remaining input until find a token
* Other possible error recovery actions are
	1. Delete one character from remaining input
	2. Insert missing character into remaining input
	3. Replace character by other character
	4. Transpose 2 adjacent characters

# Input Buffering:

* Input buffering is special technique developed to reduce amount of time taken to process large characters during compilation.
* In Input buffering schema, lexical analyzer scans data from left to right one character at a time
* In this schema, Buffers are used and each buffer size is N. If less the number of characters in input file than N then use special character eof to end of source file.
* If uses 2 pointers to keep track of the portion of input
1. Pointer LexemeBegin, marks the beginning of current lexeme.
2. Pointer Forward, Scans forward until a pattern match is found.
* There are 2 methods used in this input buffering
1. One buffer schema
2. Two buffer schema

## One buffer schema:

* In this one buffer schema, only one buffer is used to store input string.
* Problem in this schema is if lexeme is very long then it cross boundary, to scan rest we reset the buffer. it will overwrite the starting part of lexeme.
* To overcome the above problem 2 buffer schema is introduced.

## Two buffer schema:

* In this schema, two buffers are used to store input string.
* Here first buffer and second buffer scanned alternatively
* While end of the one buffer is reached other buffer is filled.
* Once lexeme is determined, forward is set to character at its right end. after lexeme is
* recorded as token in symbol table then LexemeBegin set to character immediately after lexeme just found
* Each lexeme is identified by scanning one character that character must be removed before matching to pattern.

## Sentinels:

* In the above schema, we make 2 tests before reading character. Those are
1. What character is read?
2. Check whether it is last character in buffer or not
* To overcome the above conditions we use sentinels. Sentinels are special characters that can’t be part of program. It avoids testing of end of buffer.
* The natural choice for sentinels is eof.



Switch(forward++){

Case eof: if(forward is end of 1st buffer){

Reload 2nd buffer

Forward =beginning of 2nd buffer

}

else if(forward is end of 2nd buffer){ Reload 1st buffer

Forward =beginning of 1st buffer

}

else /\*eof indicate ending of input\*/ terminate lexical analysis

break;

}

# Specification of Tokens:

* To specify tokens regular expressions are used .when a pattern is matched by some regular expression then token can be recognized.

## Strings and Languages:

* Alphabet is finite set of symbols. Examples of symbols are letters, digits and punctuation.
* String over an alphabet is finite sequence of symbols drawn from alphabet, in language theory the terms “Sentence and word “are used for “String”.
1. Length of string is denoted by |S|
2. Empty string can be denoted by ε
3. Empty set of strings is denoted by ∅

## Operations on languages:

* In lexical, most important operations on language are union, concatenation and closure.
* Language is a collection of strings. These are important operations used in language.
* Keen closure means set of strings you can get by concatenation of zero or more L and denoted by L\*.
* Positive closure is same as keen closure except L0 item. It can be done by one or more L, denoted by L+.

|  |  |
| --- | --- |
| Operation | Definition and notation |
| Union of L and M | LUM {s|s is in L or s is in M} |
| Concatenation of L and M | LM {st|s is in L and t is in M} |
| Keen closure of L | L\*=⋃i=1 Li |
| Positive closure of L | U0221EL+=⋃i=1 Li |

Example: let L be set of letters {A, B.... Z, a, b... z} and let D be set of digits {0, 1....9} then by performing various operations above discussed.

1. L U D is set of letters and digits - 62 strings possible
2. L D is set of strings a letter followed by digits - 520 string possible
3. L4 is set of strings with 4 letters
4. L\* is set of strings of letters including ε
5. L+ is set of strings of letters without ε

# Regular Expression:

* Regular expression is a way to represent string and words of given language.
* We were able to describe identifiers by set of letters and digits by the language operators union, concatenation closure as shown below.

Letter (Letter | digit)\*

* Larger regular expressions are built from smaller one in one of the following ways.
* Suppose r and s are regular expression then
1. ( r) |(s) is regular expression denoting language L(r) U L(s)
2. (r) (s) is regular expression denoting language L(r) L(s)

3. r\* is regular expression denoting language (L(r))\*

4. r is regular expression denoting language L(r)

* Some algebraic laws for regular expressions used in language are shown below

|  |  |
| --- | --- |
| Law | Description |
| r|s =s|r | | is commutative |
| r|(s|t)=(r|s)|t | | is associative |
| r(s|t)=rs|rt; (s|t)r=sr|tr | concatenation is distributive |
| εr=rε=r | ε is identify for concatenation |
| r\* =(r|ε )\* | ε is guaranteed in closure |
| r\*\*=r\* | \* is idempotent |

# Regular Definitions:

* Regular definition is a sequence of definitions used to express regular expression from the given alphabets
* Regular definition is in the below form

d1→ r1 d2→ r2

. .

. .

dn→ rn

Here di is new symbol, not in alphabet but ri is regular expression over alphabet Ʃ⋃{d1,d2, }

Example1: Regular definition for identifier from letters and digits are shown below letter → A|B|...|Z|a|b |z

digit → 0|1| |9

id → letter(letter|digit)\*

Example2: regular definition for unsigned number from digits shown below digit → 0|1| |9

digits → digit digit\* optional Fraction → .digits|ε

optional Exponent → (E(+|-|ε)digits)|ε

number → digits optional Fraction optional Exponent

* The above 2 examples are the general forms of the regular definitions.

# Notations or Extensions of Regular Expressions:

* There are few notations useful in specification of lexical analyzers.
1. One or more instances
2. Zero or more instances
3. Character classes

## One or more instances:

* Unary postfix operator + represents positive closures of regular expression and its language.

 + Operator has same precedence and associativity as operator \*.

## Zero or one instances:

* Unary postfix operator ? Means zero or one instances i.e., r? equivalent to r|ε.
* The ? Operator has same precedence and associativity as \* and +.
1. **Character class:**
* Regular expressions a1|a2|…an where ai's where symbols of alphabet can replace by [a1, a2,

… an].

* Example: Consecutive uppercase, lowercase letters or digits we can replace by [a1-an]. i.e [abc] for a|b|c and [a-z] for a|b| ….|z.

**Recognition of Tokens:**

* With the help of patterns we can recognize the tokens. The patterns for tokens are described using regular definitions, are shown below:

digit → [0-9] digits → digit+

number → digits[.digits]?(E[+-].digits)? letter → [A-Z a-z]

id → letter(letter|digit)\* If → if

then → then else → else

relop → <|>|<=|>=|=|<>

* Finally we assume that keywords are represented words i.e., they are not identifiers even though they match with identifier patterns.
* In addition, we assign token ws for identifying the blanks, tabs and newlines. Pattern for ws is ws -> (blank| tab| newline)+
* In order to indicate the instance of token relop , we use symbolic constants, LT, LE and soon as Attribute value for relational operators.

|  |  |  |
| --- | --- | --- |
| **User name** | **Token name** | **Attribute value** |
| any ws | -- | -- |
| if | if | -- |
| then | then | -- |
| Any id | Id | Pointer to table entry |
| Any number | number | Pointer to table entry |
| **<** | relop | LT |
| **>** | relop | GT |
| **< =** | relop | LE |
| **>=** | relop | GE |
| **=** | relop | EQ |
| **<>** | relop | NE |

**Transition Diagrams:**

* In contribution of lexical analysis, we first convert patterns into stylized flow charts called Transition diagram.
* Transition diagrams have collection of nodes or circles called States. Each state represents condition that occurs during scanning for lexeme that matches one of several patterns.
* Edges are diverted from one state to another in transition diagrams. Each edge is labelled by symbol or set of symbols.
* Some significant conventions about transition diagram are:
	1. Certain states are said to be accepting or found, these states are indicated by double circle.
	2. In addition it is necessary to remove forward pointers one positions, because after one extra position only, we can decide it as one lexeme. Place \* near to accepting state.
	3. One state is designated as start state. It is indicated by edge. Transition diagram always begin with starting state.

**Transition diagram for relational operators:**

* Example: Transition diagrams that recognize lexemes matching the token relop shown below.
* We begin with state 0, if we see < as first symbol among lexemes that match pattern relop, we can only look at <, <> or <=.
* We go to state 1 then look the next character, if it is = then recognize lexeme <= then enter state2 and return token relop with attribute LE. Like that all relop with attributes is stored in symbol table.

# Recognition as Reserved words and Identifiers:

* Keywords like if or then are reserved; these are not identifiers even though they look like identifiers.
* We typically use below transition diagram for identifier lexemes and keywords like if, then, else….
* There are 2 ways that we can handle reserved words those look like identifiers.
1. Install reserved words in symbol table initially. When we find identifier, call installID to place it in symbol table. If identifier already in symbol table means it is a reserved word. If not it is an identifier.
2. Create separate transition diagrams for each keyword. If any identifier matches with keyword in middle of scanning. In situation that we must prioritize tokens, because of this reserved words recognized before the identifiers. Prioritize means based on order they arranged.

# Transition diagram for number (or constant) and whitespace:

* Transition diagram for token number is most complex diagram we have seen.



* Begin in state 12, if we see digit go to 13.in state 13 we can read any number with additional digits. We see a number in the form integer (123) enter state 20 then return token number with pointer in symbol table.
* If we see dot in state 13, we has “optional fraction” move to 14 then 15. In 15 we can read any fraction with additional digits. If we see E in 15 goto16 and 17 finally moves 18. In 18 any exponent with digits is accepted.
* Transition diagrams for white space are shown below. It includes blanks, tabs and new line characters. These white spaces are not part of any token in design.

# Architecture of Transition diagram based lexical analyzer:

* There are several ways that collection of transition diagram can be used to build lexical analyzer.
* We may imagine a variable state holding number of current states for a transition diagram.
* A switch based on the value of STATE takes us to code for each of the possible states in that diagram.
* Below program will explain how switch based STATE is used in transition diagram in token relop.

Token getRelop ()

{

Token retToken = new (RELOP); While (1) {

/\*repeat until return of failure occurs\*/ Switch (state) {

Case 0: c = nextchar(); if(c == '<') state = 1;

else if (c == '=') state = 5; else if (c == '>') state = 6;

else fail(); /\*lexeme is not Relop\*/ break;

Case 1: ....

....

....

Case 8: retract (); retToken.attribute = GT; return(retToken);

}

}

}

* Let’s consider the ways of constructing one transition diagram for all lexemes in entire lexical analyzer.
1. Transition diagrams for each token to be tried “sequentially”. Then function fail() resets forward pointer and starts next transition diagram. It is used for individual keywords before going to identifiers.
2. We could run various transition diagrams “in parallel”, feeding next input character to all of them. In this strategy one diagram finds lexeme that matches its pattern, while

one or more other diagrams are still processing because of this some identifier like THENEXT taken as keyword THEN.

1. The preferred approach is to combine all transition diagrams into one. we allow the diagrams to read input until there is no possible next state then take longest lexeme that matched any pattern.

# Lexical-analyzer generator Lex:

1. Various tools have built for constructing lexical analyzers using regular expression to describe patterns for tokens.
2. A tool called Lex or Flex is used for construction of lexical analyzer. Basically it is UNIX utility.

## Use of lex:

1. In Lex tool, we will write input file with lex.l extension then it would be forwarded to Lex compiler.
2. Lex compiler transforms lex.l to C program file called lex.yy.c. later this file is compiled by C compiler into a file called a.out



1. This a.out file will take stream of input characters and produce a stream of tokens.
2. if any value is shared between lexical analyser and parser then place the value in global variable yylval.

## Structure of Lex programs:

1. Lex program has the following form: declarations

%%

translation rules

%%

auxiliary functions

1. Declarations section includes declaration of variables manifest constant and regular definitions.
2. Translation rules each have form pattern { action }
3. Each pattern is regular expression, which use regular definitions of the declaration section. Action are fragments of code, what will be done after recognise any pattern.
4. Auxiliary functions are used in the actions and these can be compiled separately and loaded with lexical analyser.
5. Finally lexical analyser returns single value (token name) to the parser. if any additional information is needed by the parser can be provided by yylval global variable.

**Example:** Let us take a Lex Program that recognises the token.

* In that program, declaration part consists a pair of brackets % { and % } anything with in braces is copied directly to file lex.yy.c.
* It is common place for definitions of manifest constants like LT, IF and soon. Regular definition is used in later part of declaration.
* In translation rules we will place patterns and actions that should be taken when particular pattern is matched.
* In auxiliary function section, we see 2 functions InstallID() and InstallNum(). Everything in auxiliary function section is copied directly to file lex.yy.c, but used in actions.
* Finally some patterns in the middle are ws for whitespaces, if, then, else are keywords, id for identifier and number for any type of numeric.

%{

/\*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-Z a-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 installlD() {

/\*function to install lexeme into symbol table\*/

}

Int installNUM(){

/\*similar to install ID. But put numbers in separate table\*/

}

* Keywords like if match with patterns IF and ID in such as situations pattern listed first are taken in to consideration.
* The actions taken when id is matched are given below.
* Function installID() is called to place lexeme in symbol table.
* Function returns a pointer to symbol table; which is placed in global variable yylval, it can be used as a parser .function set 2 variable automatically by lexical analyzer.
* yytext is pointer to beginning of lexme
* yyleng is length of lexeme found
* Token name ID is returned to parser.

# Conflict resolution in Lex:

* When conflict occurs by matching input to one or more patterns, can be resolved by the given 2 rules
	1. Always prefer longer prefix than shorter prefix
	2. If longest prefix matches two or more patterns prefer pattern listed first in Lex program.

# Lookahead operator:

* Lex automatically reads one character ahead of last character that forms selected lexeme and then removes last characters
* Sometimes we want certain pattern to be matched to input only when it is followed by certain other character. To maintain that we use slash after certain pattern. This slash is also called lookahead operator.

**Example:** let us take if condition as an example. The structure of if condition as follows IF(condition)THEN…….

If keyword is always followed by left parenthesis, some condition that contain parenthesis, right parenthesis and letter.lex rule for above if condition as follows

IF/ \(.\*/) {letter}

The rule says that pattern matches lexeme is just 2 letters IF. The slash says that additional pattern follows but not matches the lexeme. \( means literal meaning and .\* indicates any string without newline . again\) means closing of literal followed by a letter.

# Design of lexical analyzer generator:

* Design of lexical analyzer generator explains how lex is architected based on NFA’S and DFA’S.
* It explains how the lexical analyzer is worked based on input buffering, lex compiler and automata simulator.

## Structure of generated analyzer:

* Generated analyzer includes a fixed programs that simulates an automation, here automata is open, whether it is deterministic or non deterministic.
* Rest of the generator analyzer consists of components these are created from lex program by lex itself.



* The components are
1. transition table for automata .
2. functions ,those are passed directly through lex to output.
3. actions in input program, which are executed at appropriate time by automata simulator .
* To construct automata, each regular expression in lex program is converting to an NFA.
* We need single automation that recognizes lexemes matched by any pattern in lex program. So, combine all NFA's into one by introducing a state with Ԑ. Transactions to all are shown below.

**Example:** Here a, abb, a\*b+ are 3 patterns with some conflicts like abb matches both 2nd and 3rd patterns. But it is lexeme for 2nd pattern based on 1st in the list of patterns in program. To recognize that, construct NFA's for 3 patterns.





To recognize all patterns of these lexemes method, we construct one NFA by combine all with the help of an additional state 0 with three Ԑ transactions.



# Pattern Matching based on NFA's:

* If lexical analyzer construct above NFA then it read input from lexeme begin and moves forward pointer in input.
* NFA simulation reaches a point on input where there are no next states, then decide longest prefix that is lexeme matching some pattern.
* We look backwards in the sequence of states until we find one or more accepting states. If more than one state is there, select first in list of patterns and perform action associated with that pattern.

**Example:** Suppose we take input aaba in that NFA simulation, first starting with Ԑ closure of starting state zero which is {0,1,3,7} then proceed states character by character in input. After reading 4th (input symbol (Character)), we have empty states and there is no transaction from state 8 on input symbol as we need to took backwards, state 8 is a matched state with pattern a\*b+ with lexeme abb as longest prefix; then perform action for that pattern.



# DFA for Pattern Matching:

* If lexical analyzer constructs equivalent DFA for NFA of all patterns, determine the first pattern which is accepted by the given DFA states.

**Example:** Here we start with starting state {0, 1, 3, 7} by taking input symbol we move from one state to other. After reaching {6,8} state it accepts 2 patterns abb and a\*b+. But by consider first in list we choose abb is matched then perform action for pattern abb. Finally state {6, 8} is the state in DFA will accept the pattern abb, which is longest prefix for the given input abba.

# Implementing Lookahead Operator:

* Lookahead operator / is need for some patterns like r1/r2, here r1 is one pattern and it need some additional part r2 to describe the r1pattern.
* Consider a statement if(condition), to identify pattern if it need some additional part condition followed by if. To maintain this type of statements we will use lookahead operator(/).
* We treat / as Ԑ, so we do not look for / on input. If one Ԑ is in NFA then easy to finding correct state else it is very difficult to finding correct states.
* If NFA recognizes prefix xy for matching RE then end of lexeme is not more to accepting state because of ending it enters to state S. It is intermediate state between starting and ending states.
1. State S means Ԑ transaction on /.
2. There is a path from starting state to S. It indicated by x.
3. There is a path from S to accepting state. It indicated by y.



## LEX PROGRAM

**Experiment:** Lex program to count number of vowels and consonants in given string.

## Procedure:

1. Download Lex tool from internet for supporting OS.
2. Install the tool.
3. Copy the path of bin folder in Lex tool and pate it in environmental variables.
4. Download GCC tool from internet for supporting OS.
5. Install the tool and copy the path then paste it in environmental variable.
6. Open notepad, type program and save it as "name.l".

## Program:

%{

int v=0,c=0;

%}

%%

[aeiou AEIOU] v++; [A-Z a-z] c++;

%%

main()

{

printf("Enter String:"); yylex();

printf("Number of Vowels : %d",v); printf("Number of Consonants : %d",c);

}

int yywrap()

{

return 1;

}

## Output:

F:> Flex name.l F:> GCC lex.yy.c F:> a.exe

Enter String:

CSE Students Number of Vowels: 2

Number of Consonants: 9