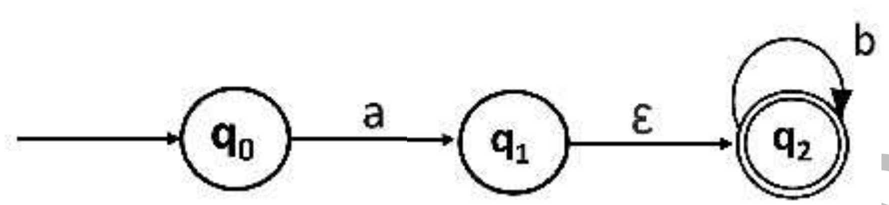
Eliminating ε Transitions

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi ∈ Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:

Convert the following NFA with ε to NFA without ε.



www.binils.com

Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

1. ε-closure(q0) = {q0}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

Now the δ' transition on each input symbol is obtained as:

1. δ'(q0, a) = ε-closure(δ(δ^(q0, ε),a))
2. = ε-closure(δ(ε-closure(q0),a))
3. = ε-closure(δ(q0, a))
4. = ε-closure(q1)
5. = {q1, q2}
6.
7. δ'(q0, b) = ε-closure(δ(δ^(q0, ε),b))
8. = ε-closure(δ(ε-closure(q0),b))

www.binils.com for Anna University | Polytechnic and Schools

9. = ε-closure(δ(q0, b))
10. = Φ

Now the δ' transition on q1 is obtained as:

1. δ'(q1, a) = ε-closure(δ(δ^(q1, ε),a))
2. = ε-closure(δ(ε-closure(q1),a))
3. = ε-closure(δ(q1, q2), a)
4. = ε-closure(δ(q1, a) ∪ δ(q2, a))
5. = ε-closure(Φ ∪Φ)
6. = Φ
7.
8. δ'(q1, b) = ε-closure(δ(δ^(q1, ε),b))
9. = ε-closure(δ(ε-closure(q1),b))
10. = ε-closure(δ(q1, q2), b)
11. = ε-closure(δ(q1, b) ∪ δ(q2, b))
12. = ε-closure(Φ ∪q2)
13. = {q2}

www.binils.com

The δ' transition on q2 is obtained as:

1. δ'(q2, a) = ε-closure(δ(δ^(q2, ε),a))
2. = ε-closure(δ(ε-closure(q2),a))
3. = ε-closure(δ(q2, a))
4. = ε-closure(Φ)
5. = Φ
6.
7. δ'(q2, b) = ε-closure(δ(δ^(q2, ε),b))
8. = ε-closure(δ(ε-closure(q2),b))
9. = ε-closure(δ(q2, b))
10. = ε-closure(q2)
11. = {q2}

Nowwewillsummarizeallthecomputedδ'transitions: 1.

δ'(q0, a) = {q0, q1}

Download Binils Android App in Playstore Download Photoplex App

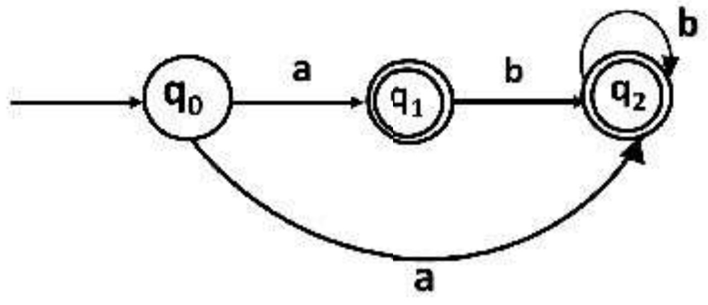
2. δ'(q0, b) = Φ
3. δ'(q1, a) = Φ
4. δ'(q1, b) = {q2}
5. δ'(q2, a) = Φ
6. δ'(q2, b) = {q2}

The transition table can be:

→q0	{q1, q2}	Φ
*q1	Φ	{q2}

www.binils.com

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



FINITE AUTOMATA

Finite automata can be represented by input tape and finite control.

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

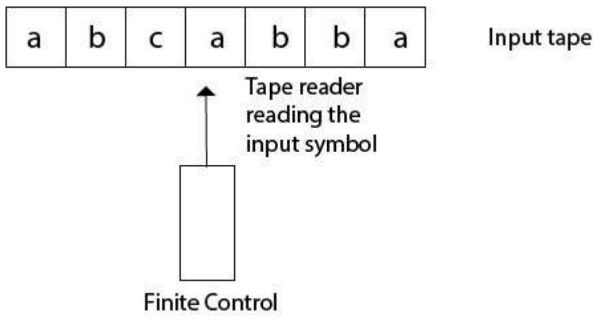


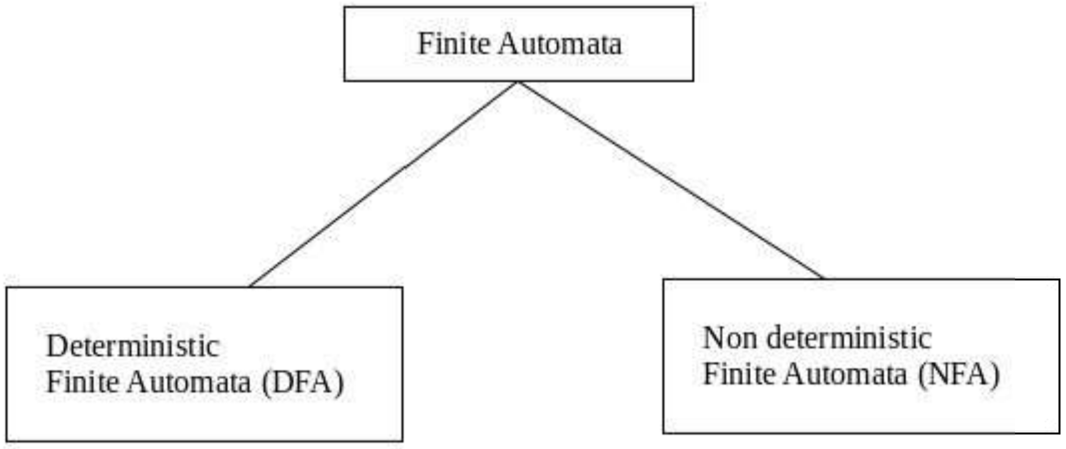
Fig :- Finite automata model

www.binils.com

Types of Automata:

There are two types of finite autorata:

1. DFA(deterministic finite automata)
2. NFA(non-deterministic finite automata)


Finite Automata
Deterministic Finite Automata (DFA)
Non deterministic Finite Automata (NFA)

DFA

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

NFA

NFAstandsfornon-deterministicfiniteautomata.Itisusedtotransmitanynumberofstatesfora particular input. It can accept the null move.

Some important points about DFA and NFA:

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

Transition Diagram

Atransitiondiagramorstatetransitiondiagramisadirectedgraphwhichcanbeconstructedasfollows:

- ThereisanodeforeachstateinQ,whichisrepresentedbythecircle.
- Thereisadirectededgefromnodeqtonodeplabeledaifδ(q,a)=p.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

www.binils.com

Some Notations that are used in the transition diagram:

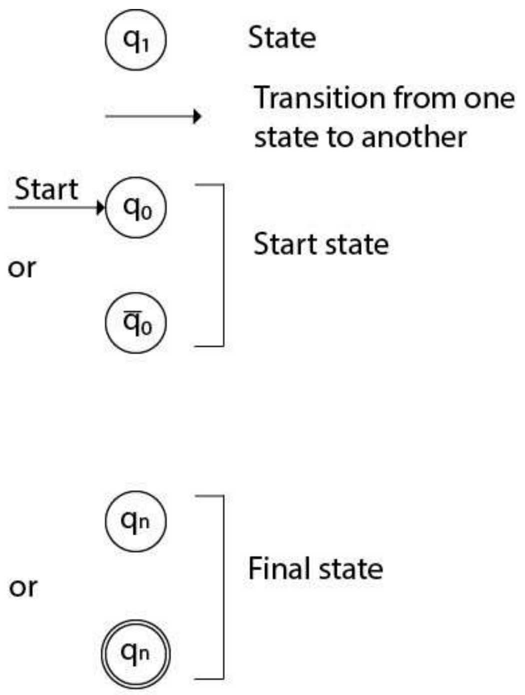


Fig:- Notations

www.binils.com

There is a description of how a DFA operates:

1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

2. The string w is said to be accepted by the DFA if r ∈ F that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if r ∉ F.

Example :

DFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:

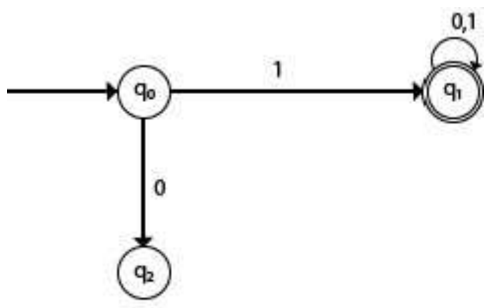


Fig: Transition diagram

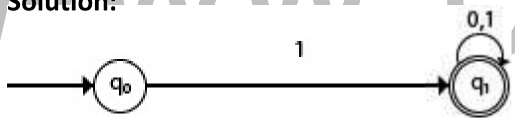
The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q 1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 11 1......., that means all string starts with 1.

Example :

NFA with ∑ = {0, 1} accepts all strings starting with 1.

www. binils.com

Solution:



The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111......, that means all string starts with 1.

Transition Table

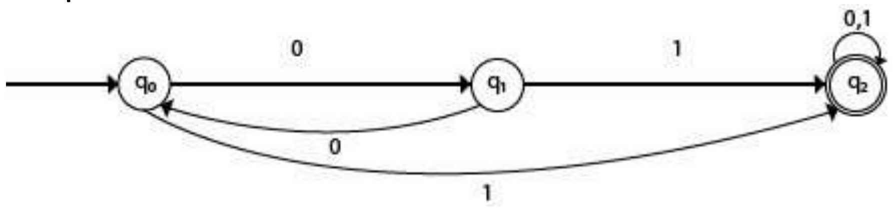
The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

A transition table is represented by the following things:

- Columns correspond to input symbols.
- Rows correspond to states.
- Entries correspond to the next state.

- The start state is denoted by an arrow with no source.
- The accept state is denoted by a star.

Example:



Solution:

Transition table of given DFA is as follows:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q2
q1	q0	q2
*q2	q2	q2

Explanation:

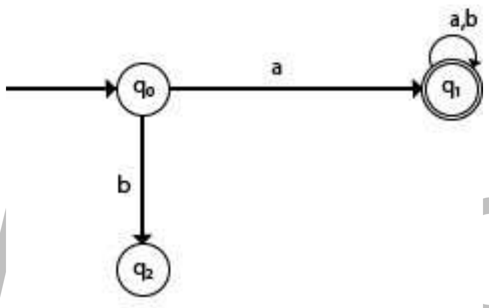
- In the above table, the first column indicates all the current states. Under column 0 and 1, the next states are shown.
- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q1 and on input 1 the next state will be q2.
- In the second row, when the current state is q1, on input 0, the next state will be q0, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q2, and on 1 input the next state will be q2.
- The arrow marked to q0 indicates that it is a start state and circle marked to q2 indicates that it is a final state.

Download Binils Android App in Playstore Download Photoplex App

DFA (Deterministic finite automata)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.



www.binils.com

Fig:- DFA

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: final state
5. δ: Transition function

Transition function can be defined as:

1. $\delta: Q \times \Sigma \rightarrow Q$

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

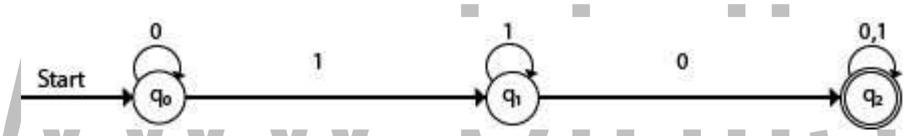
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example :

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 ={q0}
4. F = {q2}

Solution:

Transition Diagram:



www.binils.com

Transition Table:

PresentState	NextstateforInput0	Next State of Input 1
→q0	q0	q1
q1	q2	q1
*q2	q2	q2

NFA (Non-Deterministic finite automata)

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ε transition.

In the following image, we can see that from state q0 for input a, there are two next states q1 and q2, similarly, from q0 for input b, the next states are q0 and q1. Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deterministic finite automata.

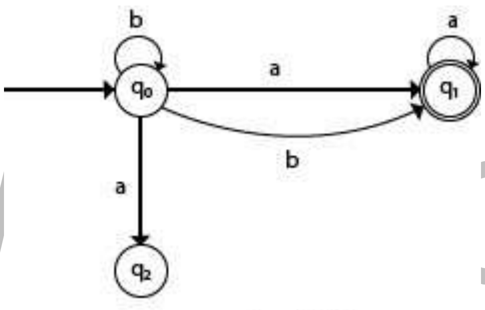


Fig:- NDFA

www.binils.com

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows: δ: Q x ∑

$\rightarrow 2^Q$

where,

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: final state
5. δ: Transition function

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

Example :

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 ={q0}
4. F = {q2}

Solution:

Transition diagram:

www.binils.com

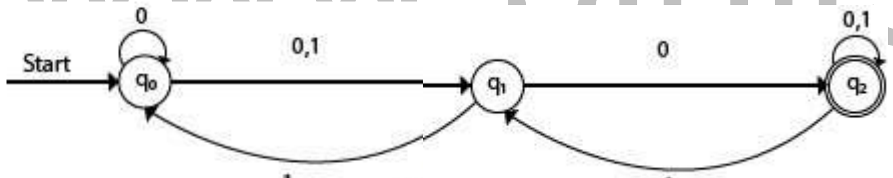


Fig: NFA

Transition Table:

PresentState	Next stat for Input 0	Next State of Input 1
→q0	q0, q1	q1
q1	q2	q0
*q2	q2	q1, q2

INTRODUCTION TO COMPILERS

INTRODUCTION TO LANGUAGE PROCESSING:

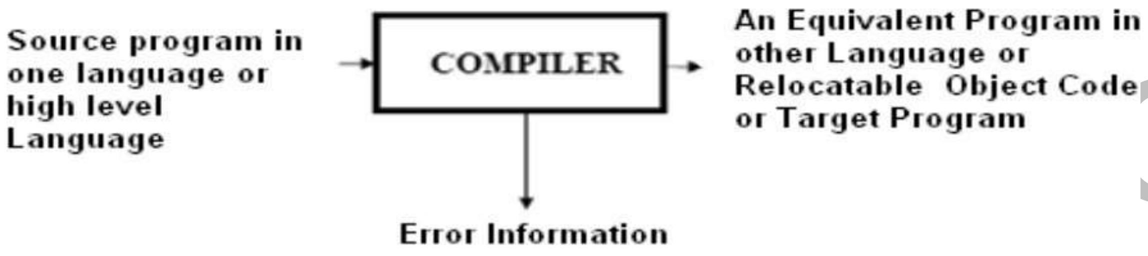
As Computers became inevitable and indigenous part of human life, and several languages with different and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software‘s have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGE TRANSLATORS :

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target] language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level language program.

☐ Two commonly Used Translators are Compiler and Interpreter

1. **Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.


Source program in one language or high level Language
COMPILER
An Equivalent Program in other Language or Relocatable Object Code or Target Program
Error Information

om

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

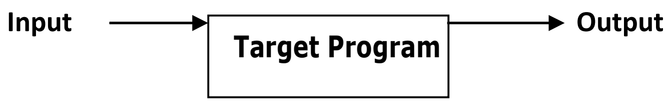


Figure: Running the target Program

2. **Interpreter:** An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by theuser.

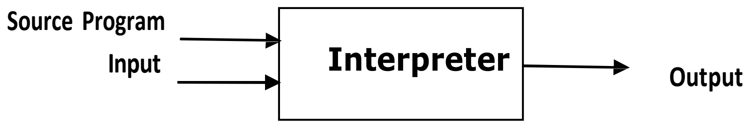


Figure : Running the target Program

LANGUAGE PROCESSING SYSTEM:

Basedontheinputthetranslatortakesandtheoutputitproduces,alanguagetranslatorcanbecalledas any one of the following.

Preprocessor: Apreprocessortakestheskeletalsourceprogramasinputandproducesanextended versionof it,whichistheresultantofexpandingtheMacros,manifestconstantsifany,andincluding headerfilesetcin thesourcefile.Forexample,theCpreprocessorisamacroprocessorthatisused automaticallybytheC compilertotransformoursourcebeforeactualcompilation.Overandabovea preprocessor performs the followingactivities:

- Collectsallthemodules,filesincaseifthesourceprogramisdividedintodifferentmodules stored at different files.
- Expands short hands / macros into source languagestatements.

Compiler: Is a translator that takes as input a source program written in high level language and converts itintoits equivalenttargetprograminmachinelanguage.Inadditiontoabovethecompileralso

- Reportstoitsuserthepresenceoferrorsinthesourceprogram.
- Facilitatestheuserinrectifyingtheerrors,andexecutethecode.

www.binils.com

Assembler: Is a program that takes as input an assembly language program and converts it into its equivalent machine languagecode.

Loader / Linker: Thisisaprogramthattakesasinput arelocatablecodeandcollectsthelibrary functions, relocatableobjectfiles,andproducesitsequivalentabsolutemachinecode.

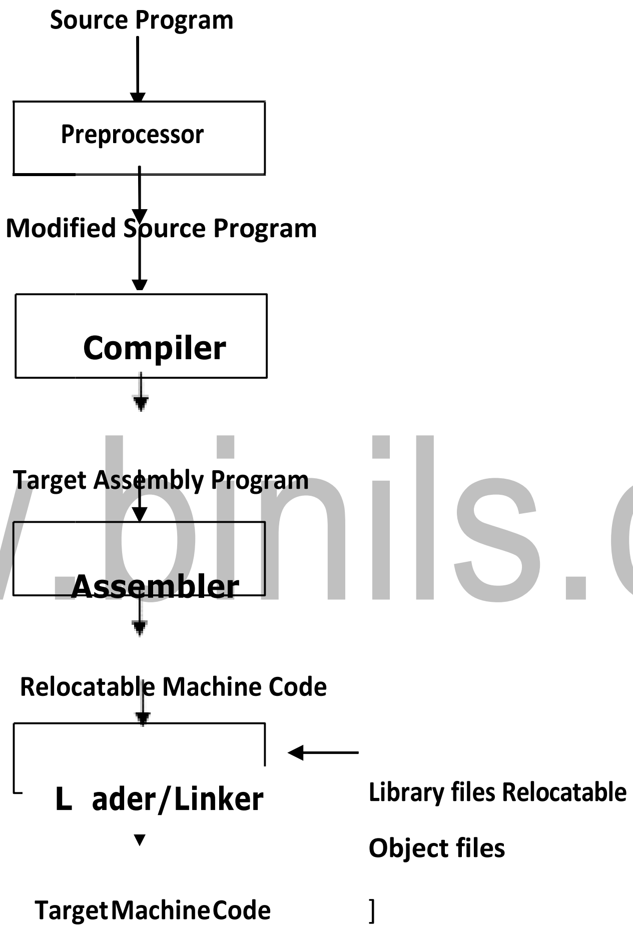
Specifically,

- **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placingthealteredinstructionsanddatainmemoryattheproperlocations.
- **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needsthem.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.



www.binils.com

Figure : Context of a Compiler in Language Processing System

TYPES OF COMPILERS:

Basedonthespecificinputittakesandtheoutputitproduces,theCompilerscanbeclassifiedintothe following types;

TraditionalCompilers(C,C++,Pascal): TheseCompilersconvertasourceprograminaHLLintoits equivalent in native machine code or object code.

www.binils.com for Anna University | Polytechnic and Schools

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code for another machine.

Incremental Compilers: These compilers separate the source into user defined–steps; Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into object code of another platform**.**

www.binils.com

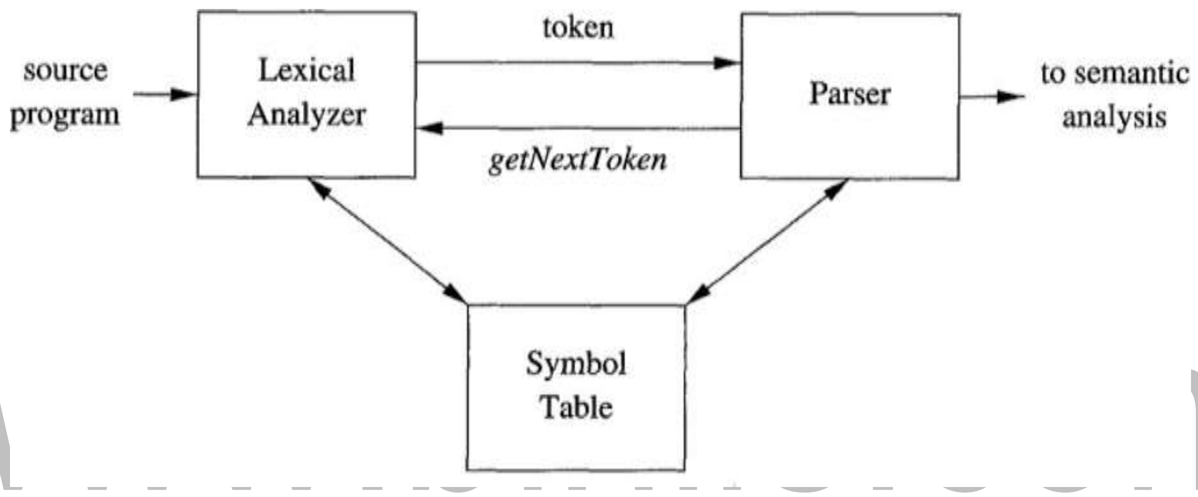
Download Binils Android App in Playstore Download Photoplex App

www.binils.com for Anna University | Polytechnic and Schools

LEXICAL ANALYSIS

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 tokens for each lexeme in the source program. This 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.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.



www.binils.com

Figure : Lexical Analyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

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. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match]. 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.

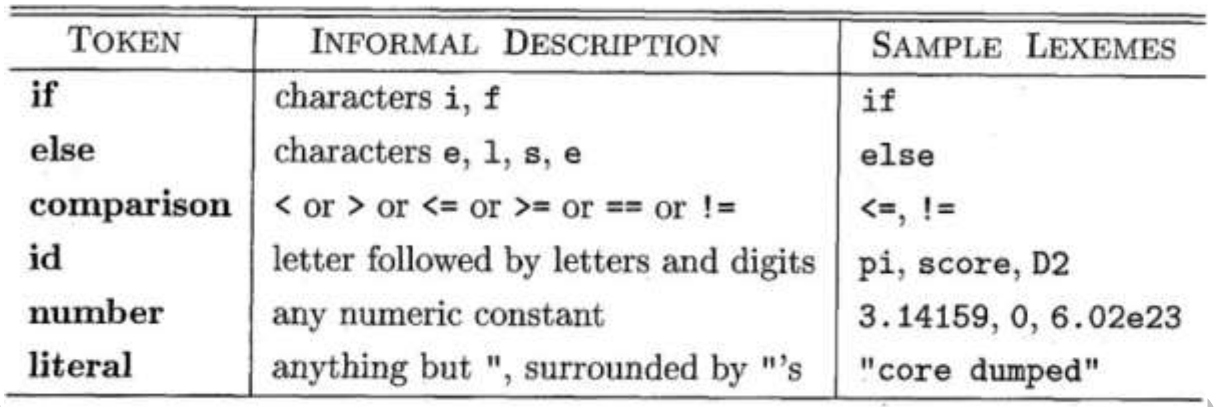
Download Binils Android App in Playstore Download Photoplex App

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

Example: In the following C language statement , printf

("Total = %d\n‖, score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖** is a lexeme matching **literal [or string]**.



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"`

www.binils.com

Figure 1.7: Examples of Tokens

LEXICAL ANALYSIS Vs 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.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

2. **Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. **Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

INPUT BUFFERING:

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single- character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

www.binils.com

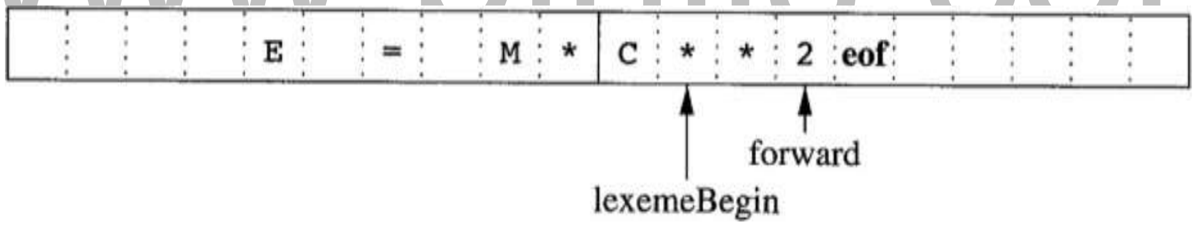


Figure : Using a Pair of Input Buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting todetermine.

2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of thischapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

www.binils.com

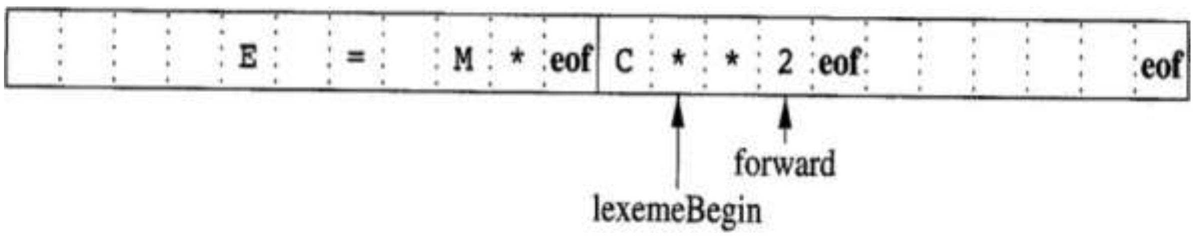


Figure : Sentential at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end. Figure 1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of

www.binils.com for Anna University | Polytechnic and Schools

a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++ )
{
        case eof: if (forward is at end of first buffer )
                  {
                     reload second buffer;
                     forward = beginning of second buffer;
                  }
                  else if (forward is at end of second buffer )
                  {
                     reload first buffer;
                     forward = beginning of first buffer;
                  }
                  else   /* eof within a buffer marks the end of input */
                     terminate lexical analysis;
            break;
}
```

Figure: use of switch-case for the sentential

www.binils.com

Download Binils Android App in Playstore Download Photoplex App

www.binils.com for Anna University | Polytechnic and Schools

PHASES OF A COMPILER

To reduce the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions. Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)

Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram depicts the phases of a compiler through which it goes during the compilation. Therefore a typical Compiler is having the following Phases:

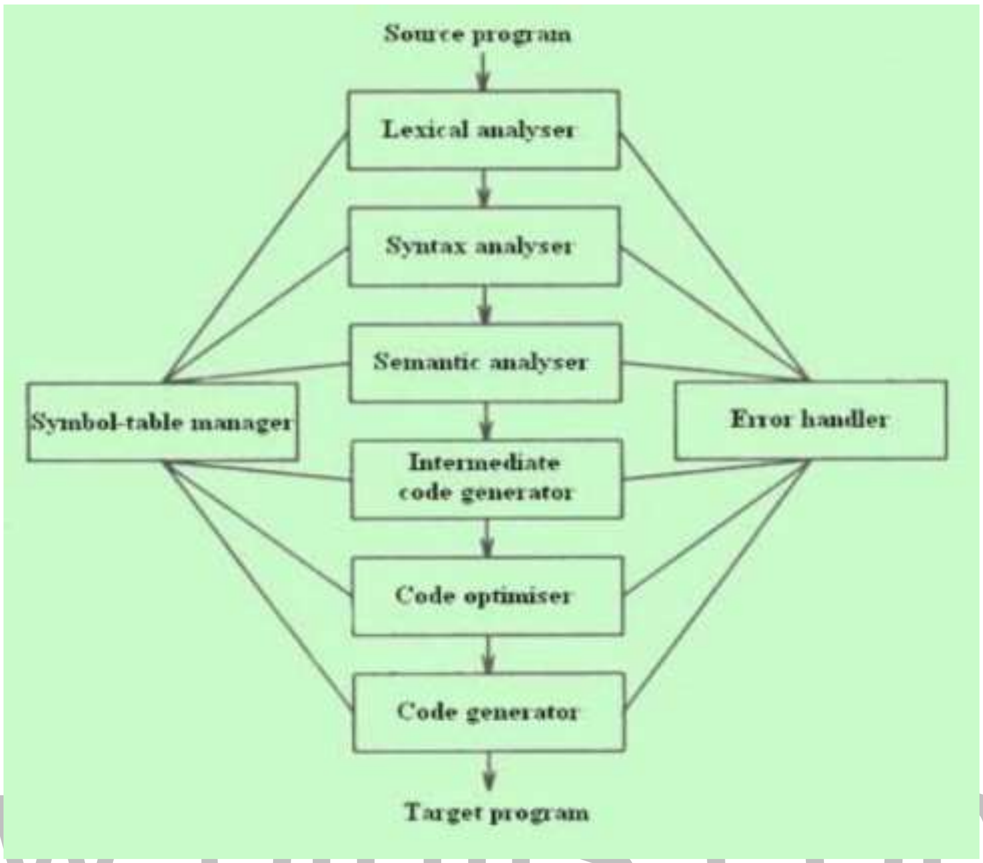
www.binils.com

1. Lexical Analyzer (Scanner),
2. 2. Syntax Analyzer(Parser),
3. 3.Semantic Analyzer,
4. 4.Intermediate Code Generator(ICG),
5. 5.Code Optimizer(CO) ,and
6. 6.Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some cases. The description is given in next section.

The Phases of compiler divided into two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

Download Binils Android App in Playstore Download Photoplex App



www.binils.com

Figure : Phases of a Compiler

PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THE FRONT-END & BACK-END OF A COMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-**

end, and **The Back-end**. This division is due to their dependence on either the Source Language or the Target machine. This model is called an Analysis & Synthesis model of a compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front- end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and those portions don‘t dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

www.binils.com

- o Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier, a Keyword, a punctuation mark, a multi character operator like := .
- o The character sequence forming a token is called a **lexeme** of the token.
- o The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn‘t exist.
- o Also removes the Comments, and unnecessary spaces.
  - The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves are tokens.
- The interior node of this tree represents the stream of tokens that logically belongs together.
- It means it checks the syntax of program elements.

SEMANTIC ANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

- The Syntactically and Semantically correct structures are produced here in the form of a Syntax tree or DAG or some other sequential representation like matrix.

www.binils.com

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- It should be easy to produce,and Easy to translate into the target program. Example intermediate code forms are:
- Three addresscodes,
- Polishnotations,etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

- Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.
- Sometimes the data structures used in representing the intermediate forms may also bechanged.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

- Memory locations are selected for each variable used, and assignment of variablesto registers is done.

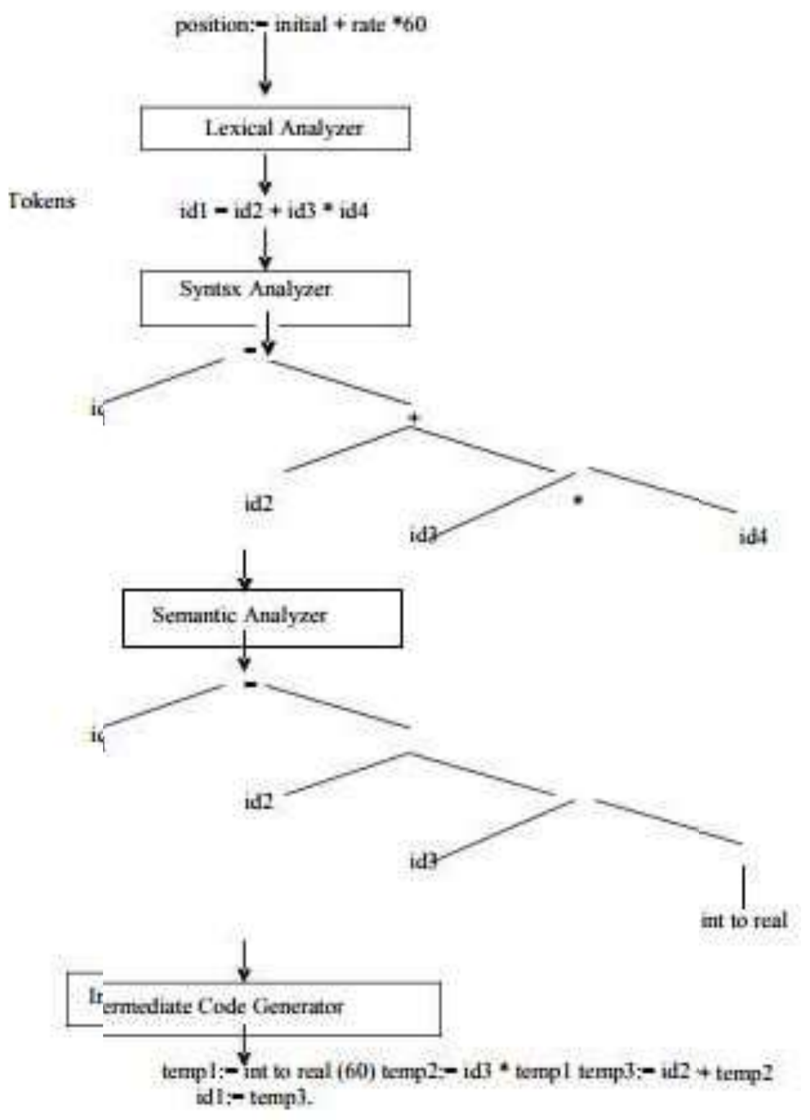
- Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

www.binils.com

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

The input source program is **Position=initial+rate*60**


position:= initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntax Analyzer
=
id
+
id2
*
id3
id4
Semantic Analyzer
=
id
id2
id3
int to real
Intermediate Code Generator
temp1:= int to real (60) temp2:= id3 * temp1 temp3:= id2 + temp2
id1:= temp3.

ls.com

Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, }

In a regular expression, x^+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, . }

Operations on Regular Language

The various operations on regular language are:

www.binils.com

Union: If L and M are two regular languages then their union L U M is also a union.

L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

L ∩ M = {st | s is in L and t is in M}

Kleen closure: If L is a regular language then its Kleen closure L1* will also be a regular language.

L* = Zero or more occurrence of language L.

Example :

Write the regular expression for the language accepting all combinations of a's, over the set ∑ = {a}

Solution:

All combinations of a's means a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is we expect the set of {ε, a, aa, aaa, }. So we give a regular expression for
this as:

1. R = a*

www.binils.com for Anna University | Polytechnic and Schools

That is Kleen closure of a.

Example :

Write the regular expression for the language accepting all combinations of a's except the null string, over the set ∑ = {a}

Solution:

The regular expression has to be built for the language

1. L = {a, aa, aaa,.. }

This set indicates that there is no null string. So we can denote regular expression as:

R = a^+

Example 3:

Write the regular expression for the language accepting all the string containing any number of a's and b's.

Solution:

www.binils.com

The regular expression will be:

1. r.e. = (a + b)*

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab, . }, any combination of a and b.

The (a + b)* shows any combination with a and b even a null string.

Examples of Regular Expression

Example :

Write the regular expression for the language accepting all the string which are starting with 1 and ending with 0, over ∑ = {0, 1}.

Solution:

In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:

1. R = 1 (0+1)* 0

Download Binils Android App in Playstore Download Photoplex App

Example :

Write the regular expression for the language starting and ending with a and having any having any combination of b's in between.

Solution:

The regular expression will be:

1. R = a b* b

Example :

Write the regular expression for the language starting with a but not having consecutive b's.

Solution: The regular expression has to be built for the language:

1. L = {a, aba, aab, aba, aaa, abab, .. }

The regular expression for the above language is:

1. R = {a + ab}*

www.binils.com

Example :

Write the regular expression for the language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.

Solution: As we know, any number of a's means a* any number of b's means b*, any number of c's means c*. Since as given in problem statement, b's appear after a's and c's appear after b's. So the regular expression could be:

Conversion of RE to FA

To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.

Step 2: Convert this NFA with ε to NFA without ε.

Step 3: Convert the obtained NFA to equivalent DFA.

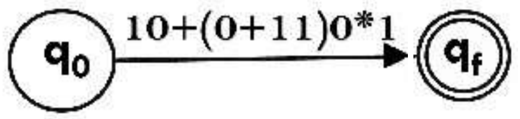
Example :

Design a FA from given regular expression 10 + (0 + 11)0* 1.

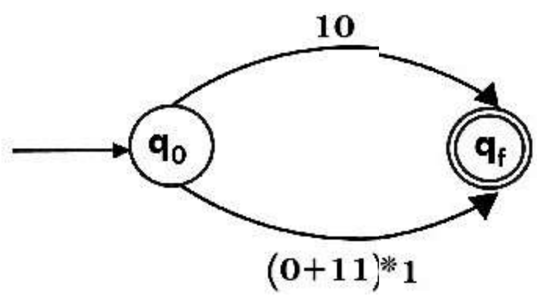
www.binils.com for Anna University | Polytechnic and Schools

Solution: First we will construct th ε transition diagram for a given regular expression.

Step 1:

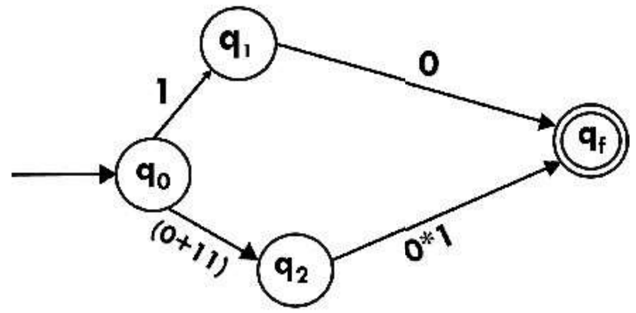


Step2:



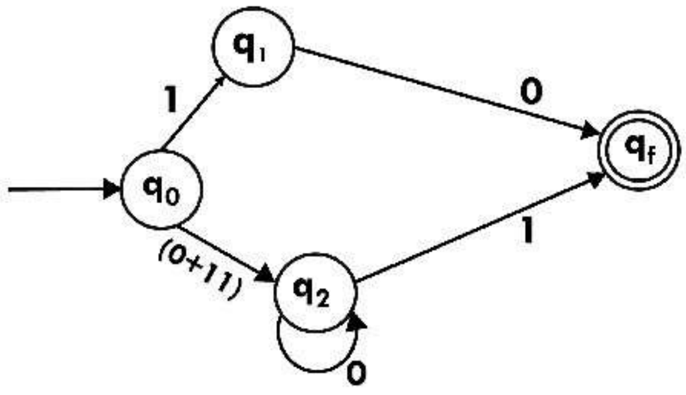
www.binils.com

Step3:

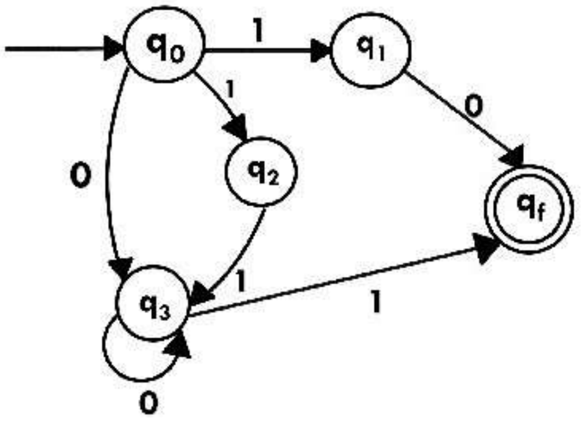


Step4:

Download Binils Android App in Playstore Download Photoplex App



Step 5:



www.binils.com

Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
→q0	q3	{q1, q2}
q1	qf	φ
q2	φ	q3
q3	q3	qf

*qf	φ	φ

The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	φ
[q2]	Φ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	Φ	φ

www.binils.com

SPECIFICATION OF TOKENS

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

LEX the Lexical Analyzer generator

Lex is a tool used to generate lexical analyzer, 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, it is a c program given for C Compiler, gives the Object code. Here we need to know how to write the Lex language. The structure of the Lex program is given below.

Structure of LEX Program : A Lex program has the following form:

```
Declarations
%%
Translation rules
%%
Auxiliary functions definitions
```

www.binils.com

The declarations section : includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. It appears between %{. . .%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the

form Pattern {Action}

The auxiliary function definitions section includes the definitions of functions used to install identifiers and numbers in the Symbol tale.

LEX Program 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           [o-91
id             {letter} ({letter} | {digit}) *
number          {digit}+ (\ . {digit}+)? (E [+-I]?{digit}+)?
%%
{ws}             {/* no action and no return */}
if               {return(1F) ; }

then            {return(THEN) ; }
else            {return(ELSE) ; }
(id)            {yylval = (int) installID(); return(1D);}
(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 installID0() {/* 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 numerical constants into a separate table */}
```

Figure : Lex Program for tokens common tokens

www.binils.com