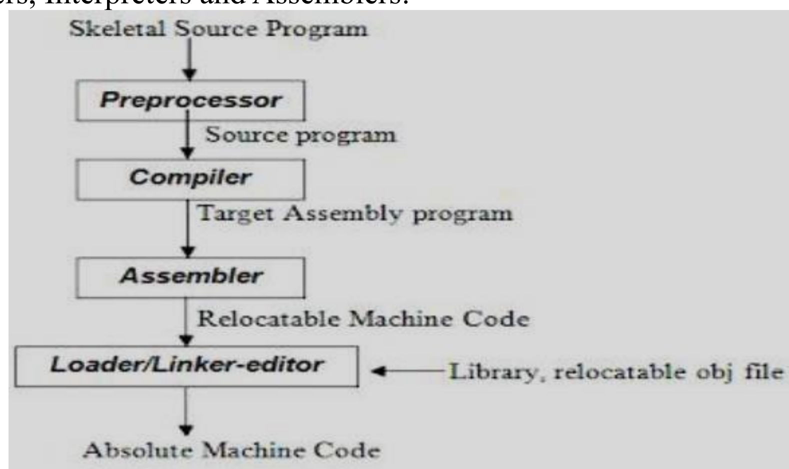
Module I

Language Processors, the structure of a compiler, the science of building a compiler, programming language basics.

Lexical Analysis: The Role of the Lexical Analyzer, Input Buffering, Recognition of Tokens, The Lexical-Analyzer Generator Lex, Finite Automata, From Regular Expressions to Automata, Design of a Lexical-Analyzer Generator, Optimization of DFA-Based Pattern Matchers.

Introduction of Language Processing System

- Programming languages are used to instruct user ideas to the computer
- computer can't understand natural language, i.e. it can understand only binary / machine language (0's and 1's)
- language translator is a system software that converts a code from one form of a language to another form of language.
  Example: Compilers, Interpreters and Assemblers.



Language Processing System

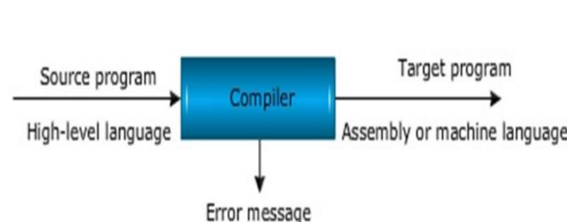
Preprocessor:

A preprocessor produce input to compilers. They may perform the following functions.

1. **Macro processing:** A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. ***File inclusion:*** A preprocessor may include header files into the program text.
3. ***Rational preprocessor:*** these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. ***Language Extensions:*** These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

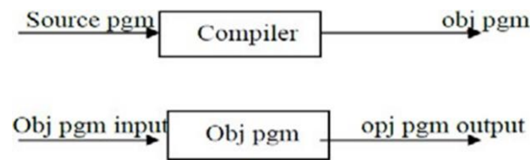
COMPILER:

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



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

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



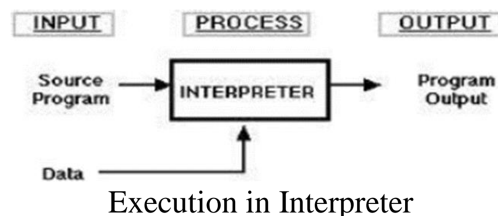
Execution process of source program in Compiler

ASSEMBLER:

Programmers found it difficult to write or read programs in machine language.
They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER:

An interpreter is a program that appears to execute a source program as if it were machine language



Execution in Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

- Lexical analysis
- Synatx analysis
- Semantic analysis
- Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.
Type of object that denotes a variation 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.

LOADER AND LINK-EDITOR:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute.
This would waste core by leaving the assembler in memory while the user’s program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problems of wasted translation time and memory. System programmers developed another component called loader “A loader is a program that places programs into memory and prepares them for execution.”

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

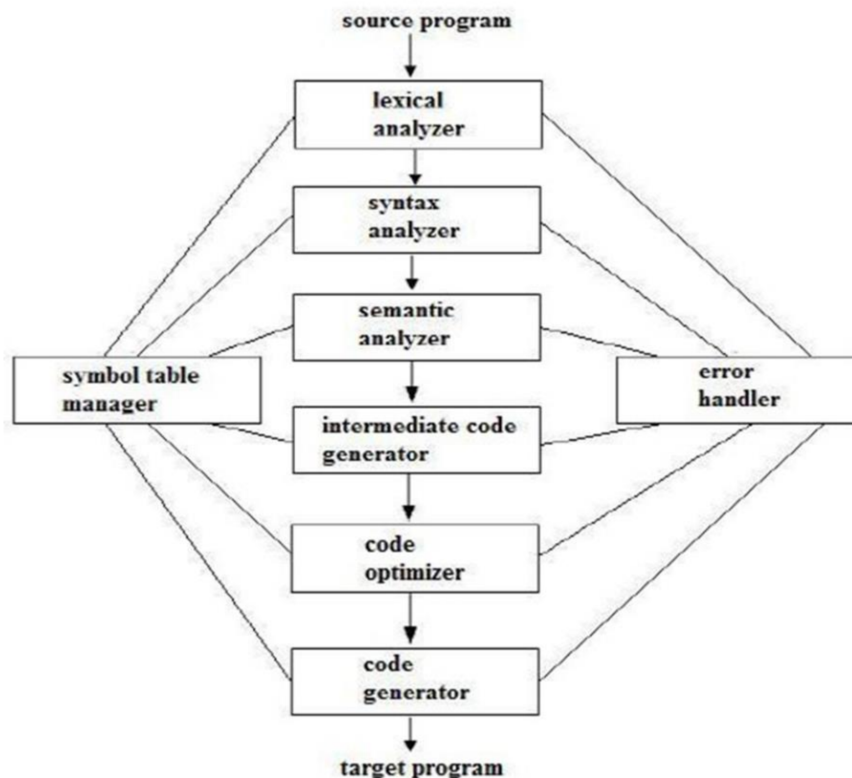
TRANSLATOR:

A translator is a program that takes as input a program written in one language and produces as output a program in another language.
Beside program translation, the translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers.
Important role of translator are: Translating the HLL program input into an equivalent ML program.
Providing diagnostic messages wherever the programmer violates specification of the HLL.

LIST OF COMPILERS:

Ada compilers
ALGOL compilers
BASIC compilers
C# compilers
C compilers
C++ compilers
COBOL compilers
Common Lisp compilers
ECMAScript interpreters
Fortran compilers
Java compilers
Pascal compilers
PL/I compilers
Python compilers
Smalltalk compilers

Structure of Compiler



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Phases of a Compiler:

- Phase is sequence of statements or steps, that takes source program in one representation and generates code in another representation
- Compiler operates in phases, as it is difficult to generate the target code directly
- A compiler is basically divided into two parts. They are
  - Analysis part
  - Synthesis part
- The analysis part is often called front-end of a compiler and synthesis part is often called back-enc of a compiler
- Analysis part breaks up the source program into constituent pieces and create an intermediate representation of the source code known as intermediate code
- Analysis part is also collects information about the source program and stores it in a data structure called a symbol table
- Analysis part includes 3 phases of a compiler, that is lexical analyzer, syntax analyzer and semantic analyzer.
- Synthesis part constructs desired target code from an intermediate and the information in the symbol table
- Synthesis part includes 3 phases of a compiler, that is intermediate code generation, code optimization and code generation
- Intermediate code generation will act as interface between analysis and Synthesis parts.

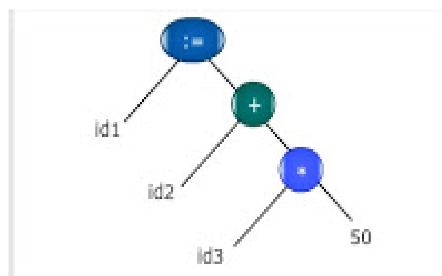
1. **Lexical Analyzer:**
   - It is a first phase of a compiler
   - Lexical Analyzer is also known as scanner
   - Lexical Analyzer reads the stream of characters from left to right and groups the characters into meaningful sequences called lexeme

Example: result = number1 + number2 * 50

- After lexical analyzer, the above statement would be
- <id,1> <=> <id,2> <+> <id,3> <*> <number>
- In this representation the token names=,+,* are abstract symbols for the assignment, addition, multiplication respectively.

2. **Syntax Analyzer**
   - It is a second phase of compiler
   - It is also known as parser or hierarchical analysis
   - Parser collects the sufficient tokens which are generated by lexical analyzer as its input and generates parse tree as its output
   - Parser checks the source code i.e., whether it is syntactically correct or not. Here the meaning of a source code is not checked
   - The role of a syntax analyzer is to check the syntax errors in a source program

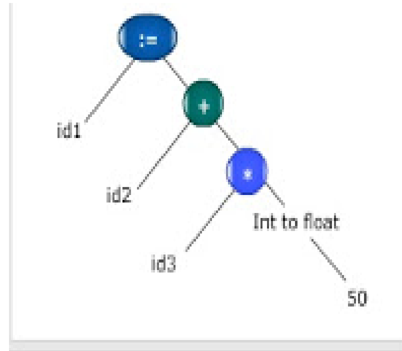
Example:



id1: = id2+id3*50
where id1, id2, id3, 50 are external nodes
and :=,+,* are internal nodes

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

3. **Semantic Analyzer:**
   - Semantic Analyzer takes parse tree as its input and construct meaningful parse tree as its output
   - The semantic analyzer uses the parse tree and the information in the symbol table to check the source code for meaning



4. **Intermediate Code Generation**
   - Intermediate is the interface between front-end and back-end of a compiler
   - Intermediate code generation takes a parse tree as its input and construct one or more intermediate representations, which can have a variety of forms

Example:

t1:=int to float(50)
t2:=id3*t1
t3:=id2+t2
t4:=t3
id1=t4

where t1,t2,t3,t4 are temporary variables generated by a compiler automatically

- There are certain properties, which should be possessed by the three address code and those are
- Each three address instruction has at the most one operator in addition to the assignment.
- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some Three address instructions may have fewer than three operands for example first and last instruction of above given three address code. i.e.,

t1:=int to float(10)
id1:=t3

5. **Code Optimization:**
   - It is optional phase of a compiler
   - The process of reducing the number of instructions without changing the meaning of source program is known as code optimization
   - Code Optimization takes intermediate code as its input and generates optimized intermediate codes as its output if possible
   - The role of a code optimization is to improve the intermediate code so that the running time of the target program can be significantly improved

Example:

t1:=id3*50.0
id1:=id2+t1

6. **Code Generation:**

- It is a final phase of a compiler
- The code generator takes intermediate or optimized intermediate code and required information in symbol table as its input and maps it into the targe code

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

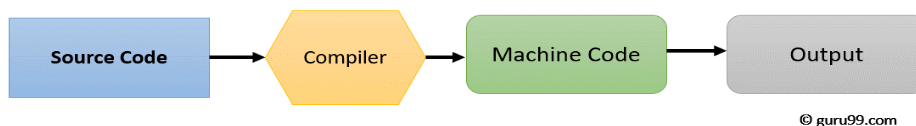
- If the output of a code generator is machine code, registers or memory location are selected for each of the variables used by the program
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task

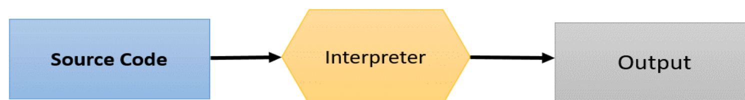
Example:

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

Compare interpreter and compiler

Differences between Interpreter and Compiler	
Interpreter	**Compiler**
Interpreter translates just one statement of the program at a time into machine code.	Compiler scans the entire program and translates the whole of it into machine code at once.
An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.	A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.
An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.	A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.
Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy.	A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler.
Interpreters are used by programming languages like BASIC, Python, PHP and Ruby for example.	Compliers are used by programming languages like C, C++ and Java for example.
It doesn't create an intermediate object (.obj) code.	It usually generates intermediate code in the form of the object file (.obj).
Slower execution of control statements as compared to the compiler.	Faster execution of control statements as compared to the interpreter.





P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influenced over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions. These models are useful for de-scribing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to de-scribe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. Similarly, trees are an important model for representing the structure of programs and their translation into object code.

The Science of Code Optimization

The term "optimization" in compiler design refers to the attempts that a com-piler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

Programming Language Basics

1 The Static/Dynamic Distinction
2 Environments and States
3 Static Scope and Block Structure
4 Explicit Access Control
5 Dynamic Scope
6 Parameter Passing Mechanisms
7 Aliasing

The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time.* On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time.*

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

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

Environments and States

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

It may be less clear that the location denoted by x can change at run time.

Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

program. Later languages, such as C + + , Java, and C # , also provide explicit control over scopes through the use of keywords like **public, private,** and **protected .**

Explicit Access Control

Classes and structures introduce a new scope for their members. If *p* is an object of a class with a field (member) *x,* then the use of *x* in *p.x* refers to field *x* in the class definition. In analogy with block structure, the scope of a member declaration *x* in a class *C* extends to any subclass *C*, except if *C* has a local declaration of the same name *x.*

Through the use of keywords like **public, private,** and **protected,** object-oriented languages such as C + + or Java provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

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

Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

Parameter Passing Mechanisms

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

Call - by - Value

Call - by - Reference

Aliasing

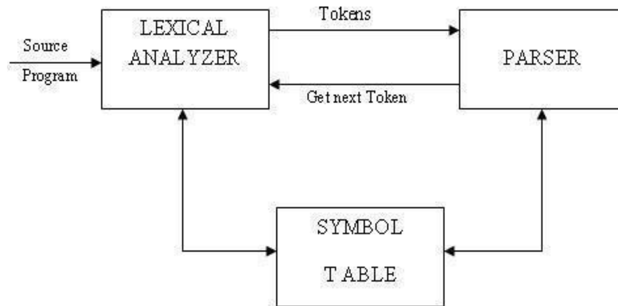
There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any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

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

Role of the Lexical Analyzer

- The LA is the first phase of a compiler.
- Lexical analysis is called as linear analysis or scanning.
- In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
-



- Upon receiving a "get next token" command form the parser, the lexical analyzer reads the input character until it can identify the next token.
- The LA return to the parser representation for the token it has found.
- The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.
- LA may also perform certain secondary tasks as the user interface.
- One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters.
- Another is correlating error message from the compiler with the source program.

Token, Lexeme, Pattern:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: A set of strings in the input for which the same token is produced as output.
This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Description of token:

Token	lexeme	pattern
const	const	const
if	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

LEXICAL ANALYSIS VS PARSING:

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc.	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence).
The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

Input buffering

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

- The lexical analyser scans the characters of the source program one at a time to discover tokens.
- Often, however, many characters beyond the next token may have to be examined before the next token itself can be determined.
- For this reason, it is desirable for the lexical analyser to read input from the input buffer.
- Fig shows a buffer divided into two halves of, say 100 characters each.
- One pointer marks beginning of the token being discovered.
- A look ahead pointer scans ahead of the beginning point, until token is discovered.

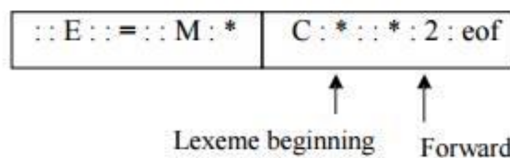


Fig. 1.9 An input buffer in two halves

- We view the position of each pointer as being between the character last read and the character next to be read.
- In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.
- The distance which the lookahead pointer may have to travel past the actual token may be large.
- For example, in a PL/I program we may see:
  DECLARE (ARG1, ARG2… ARG n)
- Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.
- In either case, the token itself ends at the second End.
- If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.
- Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- In the above example, if the look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens.
- While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

BUFFER PAIRS

- A buffer is divided into two N-character halves, as shown below
- Each buffer is of the same size N, and 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.

Recognition of Tokens

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols. Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|

Digits - 0 | 1 | 2 | …. | 9

letter (letter / digit)*

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt -> if expr then stmt

| If expr then else stmt

| є

Expr --> term relop term

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

|term
Term -->id

For relop ,we use the comparison operations of languages like Pascal or SQL where = is “equals” and < > is “not equals” because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit   -->[0,9]
digits  -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter  -->[A-Z,a-z]
id      -->letter(letter/digit)*
if      --> if
then    -->then
else     -->else
relop   --></>/<=/>=/==/< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the “token” we defined by:
ws --> (blank/tab/newline)+

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

Lexeme	Token Name	Attribute Value
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any Id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

The Lexical-Analyzer Generator Lex

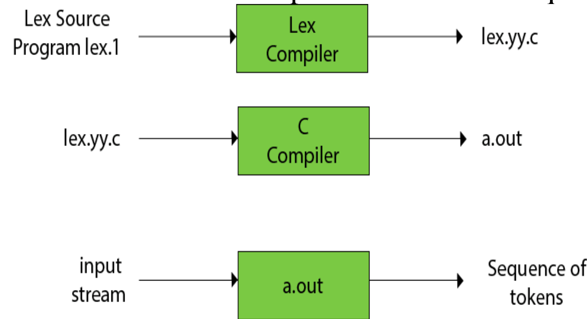
LEX Tool:

- Lex is a program / that generates lexical analyzer.
- It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language.
- Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

- A Lex program is separated into three sections by %% delimiters.

The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }

- **Definitions** include declarations of constant, variable and regular definitions.
- **Rules** define the statement of form p1 {action1}, p2 {action2}....pn {action}.
- Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
- **User subroutines** are auxiliary procedures needed by the actions.
- The subroutine can be loaded with the lexical analyzer and compiled separately.

Finite Automata

The finite automaton can be represented as input tape and finite control as shown in the Figure:
i) **Input tape:** is a linear tape having some cells that can hold an input symbol from Σ.
ii) **Finite control:** The finite control indicates the current state and it decides on 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 any instance only one input symbol is read.

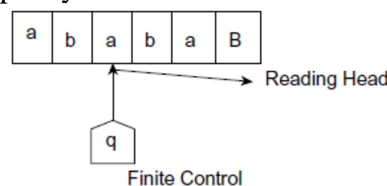


Fig. 2.1 *Finite Automaton*

The reading head examines the read symbol, and the head moves to the right with or without changing the state. When the entire string is read, if finite control is in final state, the string is accepted; else it is rejected. Finite automaton can be represented by the transition diagram, in which the vertices represent the states and edges represent transitions. We can model the real-word problems as a finite automaton and this helps in understanding the behaviour and in analysing the behaviour.

The different types of Finite Automata are as follows −

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- Finite Automata without output
    - Deterministic Finite Automata (DFA).
    - Non-Deterministic Finite Automata (NFA or NDFA).
    - Non-Deterministic Finite Automata with epsilon moves (e-NFA or e-NDFA).
- Finite Automata with Output
    - Moore machine.
    - Mealy machine.

From Regular Expressions to Automata

Regular expressions and finite automata are equivalent in their descriptive power. That is, for any given regular expression, we can construct its equivalent finite automaton that recognizes the language it describes and vice versa.

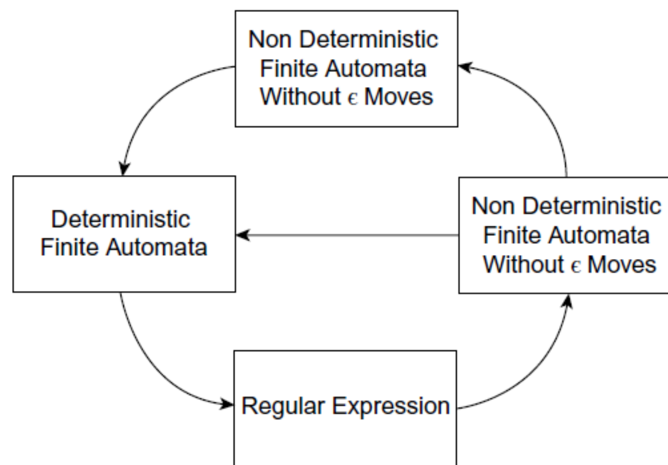


Fig. 3.1 *Equivalance of FA & RE*

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

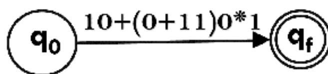
Table 3.1 *Relation between FA, Rs & RE*

S. No.	Finite automata	Regular set	Regular expression
1.		ϕ	Φ
2.		{ε}	ε
3.		{a}	a
4.		{a, b}	a + b a | b
5.		{ab}	a.b
6.		{a, aa,…., ba, bba, … baba…}	b*a(a*bb*a)* or (b*aa*b)*b*a
7.		{є, a, aa, aaa,…….}	a*
8.		{a, aa, aaa, ….}	a^+
9.		{є, aa,$(aa)^2$, …}	(aa)*
10.		{b, bbb, $b(bb)^2$, …..}	b(bb)*

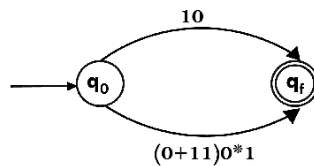
Example: Construct NFA for the regular expression **$10+(0+11)0^{*}1$**

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

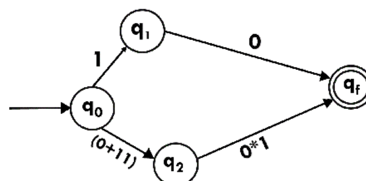
Step 1:



Step 2:

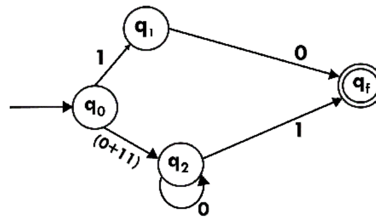


Step 3:

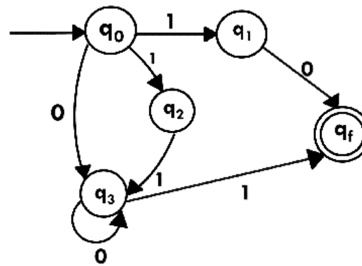


P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Step 4:



Step 5:



Design of a Lexical-Analyzer Generator

1 The Structure of the Generated Analyzer

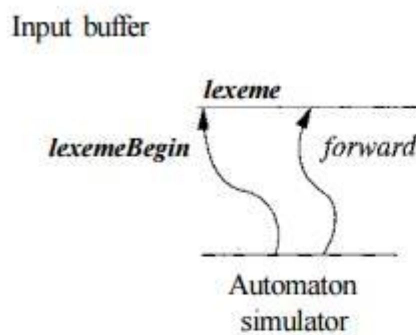
2 Pattern Matching Based on NFA's

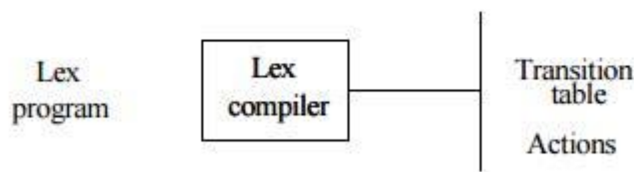
3 DFA's for Lexical Analyzers

4 Implementing the Lookahead Operator

1. The Structure of the Generated Analyzer

Figure Overviews the architecture of a lexical analyzer generated by Lex. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.





P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

These components are:

A transition table for the automaton.

Those functions that are passed directly through Lex to the output.

The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

2. Pattern Matching Based on NFA's

If the lexical analyzer simulates an NFA such as that of Fig, then it must read input beginning at the point on its input which we have referred to as *lexemeBegin.* As it moves the pointer called *forward* ahead in the input, it calculates the set of states it is in at each point.

Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus, we are ready to decide on the longest prefix that is a lexeme matching some pattern.

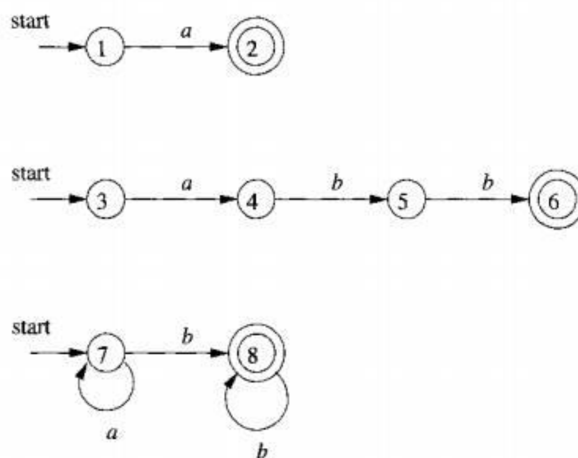
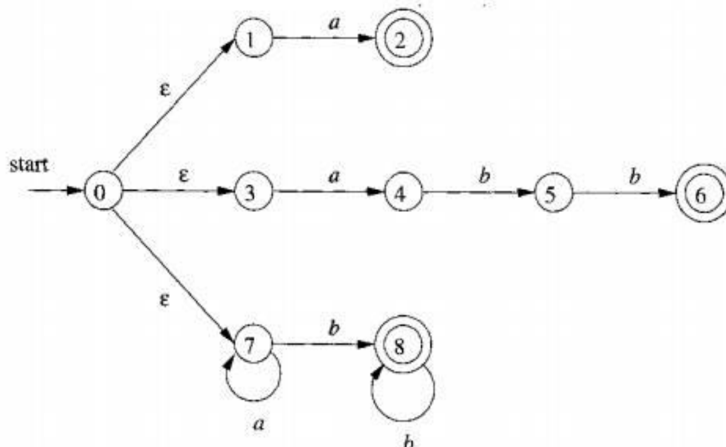


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$



•

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

3. DFA's for Lexical Analyzers

Another architecture, resembling the output of Lex, is to convert the NFA for all the patterns into an equivalent DFA, using the subset construction. Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented, and make that pattern the output of the DFA state.

We use the DFA in a lexical analyzer much as we did the NFA. We simulate the DFA until at some point there is no next state (or strictly speaking, the next state is 0, the *dead state* corresponding to the empty set of NFA states). At that point, we back up through the sequence of states we entered and, as soon as we meet an accepting DFA state, we perform the action associated with the pattern for that state.

4. Implementing the Lookahead Operator

The **Lex** lookahead operator / in a **Lex** pattern is sometimes necessary, because the pattern for a particular token may need to describe some trailing context in order to correctly identify the actual lexeme. When converting the pattern to an NFA, we treat the / as if it were ε, so we do not actually look for a / on the input. However, if the NFA recognizes a prefix *xy* of the input buffer as matching this regular expression, the end of the lexeme is not where the NFA entered its accepting state. Rather the end occurs when the NFA enters a state *s* such that

1. *s* has an ε-transition on the /,
2. There is a path from the start state of the NFA to state *s* that spells out *x*.
3. There is a path from state *s* to the accepting state that spells out *y*.
4. *x* is as long as possible for any *xy* satisfying conditions **1-3.**

Optimization of DFA-Based Pattern Matchers.

Optimization of DFA-Based Pattern Matchers

1 Important States of an NFA

2 Functions Computed From the Syntax Tree

3 Computing unliable, firstpos, and lastpos

4 Computing followpos

5 Converting a Regular Expression Directly to a DFA

6 Minimizing the Number of States of a DFA

7 State Minimization in Lexical Analyzers

8 Trading Time for Space in DFA Simulation

1. Important States of an NFA

The important states of the NFA correspond directly to the positions in the regular expression that hold symbols of the alphabet. It is useful, as we shall see, to present the regular expression by its *syntax tree,* where the leaves correspond to operands and the interior nodes correspond to operators.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

2. 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,* defined as follows. 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 subexpression represented by *n* has e in its language. That is, the subexpression can be "made null" or the empty string, even though there may be other strings it can represent as well.

2. *firstpos(n)* is the set of positions in the subtree rooted at **n** that correspond to the first symbol of at least one string in the language of the subexpression rooted at **n.**

3. *lastpos(n)* is the set of positions in the subtree rooted at *n* that correspond to the last symbol of at least one string in the language of the subexpression 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 = axa2 ••• an in L ((r) #) such that for some i, there is a way to explain the membership of x in L ((r) #) by matching to position p of the syntax tree and ai+i to position g.

3 Computing nullable, firstpos, and lastpos

We can compute nullable, firstpos, and lastpos by a straightforward recursion on the height of the tree. The basis and inductive rules for nullable and firstpos are summarized in Fig. 3.58. The rules for lastpos are essentially the same as for firstpos, but the roles of children c\ and c2 must be swapped in the rule for a cat-node.

4 Computing *followpos*

Finally, we need to see how to compute *followpos.* There are only two ways that a position of a regular expression can be made to follow another.

1 . I f n i s a cat-node with left child *c * and right child **C 2 ,** then for every position *i* in *lastpos(ci*), all positions in *firstpos(c2)* are in *followpos(i).*

2. If *n* is a star-node, and *i* is a position in *lastpos(n),* then all positions in *firstpos(n)* are in *followpos(i).*

5. Converting a Regular Expression Directly to a DFA

1. Construct a syntax tree T from the augmented regular expression (r) #

2. Compute nullable, firstpos, lastpos, and followpos for T.

Construct Dstates, the set of states of DFA D, and Dtran, the transition function for D. The states of D are sets of positions in T. Initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is firstpos(no), where node no is the root of T. The accepting states are those containing the position for the endmarker symbol #. •

6. Minimizing the Number of States of a DFA

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

There can be many DFA's that recognize the same language. Not only do these automata have states with different names, but they don't even have the same number of states. If we implement a lexical analyzer as a DFA, we would generally prefer a DFA with as few states as possible, since each state requires entries in the table that describes the lexical analyzer.

7. State Minimization in Lexical Analyzers

To apply the state minimization procedure to the DFA's generated, we must begin with the partition that groups to-gether all states that recognize a particular token, and also places in one group all those states that do not indicate any token.

8. Trading Time for Space in DFA Simulation

The simplest and fastest way to represent the transition function of a DFA is a two-dimensional table indexed by states and characters. Given a state and next input character, we access the array to find the next state and any special action we must take, e.g., returning a token to the parser. Since a typical lexical analyzer has several hundred states in its DFA and involves the ASCII alphabet of 128 input characters, the array consumes less than a megabyte.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)