Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

SCHOOL OF COMPUTING

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

1151CS115-COMPILER DESIGN

(VTUR15 Regulation)

Year/Semester: III/VI CSE

Syllabus

COURSE CODE	COURSE TITLE	L	T	P	C
1151CS115	COMPILER DESIGN	3	0	0	3

Course Category: Program Core

A. Preamble:

This Course describes the theory and practice of compilation, in particular, the lexical analysis, parsing and code generation and optimization phases of compilation, and design a compiler for a concise programming language.

B. Prerequisite Courses:

Sl. No	Course Code	Course Name
1	1151CS109	Theory of Computation

C. Related Courses:

Sl. No	Course Code	Course Name
1	1156CS601	Minor Project
2	1156CS701	Major Project

D. Course Outcomes:

Upon the successful completion of the course, students will be able to:

CO No's	Course Outcomes	Knowledge Level(Based on revised Bloom's Taxonomy)
CO1	Understand the major phases of compilation and to understand the knowledge of Lex tool & YAAC tool	K2
CO2	Develop the parsers and experiment the knowledge of different parsers design without automated tools	K3
CO3	Construct the intermediate code representations and generation	K3
CO4	Convert source code for a novel language into machine code for a novel computer	K3
CO5	Apply for various optimization techniques for dataflow analysis	K3

E. Correlation of COs with POs and PSOs:

COs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO 1	PSO 2	PSO 3
CO1	H	M	M		M		L		L			H	M	M	M
CO2	L	H	M	M	H				M				H	M	
CO3	L	H	H		M								L	M	
CO4	H	H	L		H		L						M	L	L
CO5	H	H	H		M				M			M	H	L	M

H- High; M-Medium; L-Low

F. Course Content:

UNIT I Introduction to Compilers 9

Compilers, Analysis of the Source Program, The Phases of a Compiler, Cousins of the Compiler, The Grouping of Phases, Compiler-Construction Tools.

LEXICAL ANALYSIS: Need and role of lexical analyser-Lexical errors, Input Buffering - Specification of Tokens, Recognition of Tokens, Design of a Lexical Analyzer Generator

UNIT II Syntax Analysis 9

Need and role of the parser- Context Free Grammars-Top Down parsing - Recursive Descent Parser - Predictive Parser - LL (1) Parser -Shift Reduce Parser - LR Parser - LR (0) item - Construction of SLR Parsing table -Introduction to LALR Parser, YACC- Design of a syntax analyser for a sample language

UNIT III Intermediate Code Generation 9

Intermediate languages – Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back patching – Procedure calls.

UNIT IV Code Generation 9

Issues in the design of code generator – The target machine – Runtime Storage management – Basic Blocks and Flow Graphs – Next-use Information – A simple Code generator – DAG representation of Basic Blocks

UNIT V Code Optimization and Run Time Environments 9

Introduction– Principal Sources of Optimization – Peephole Optimization- Optimization of basic Blocks – Introduction to Global Data Flow Analysis – Runtime Environments – Source Language issues – Storage Organization – Storage Allocation strategies – Access to non-local names – Parameter Passing**.**

G. Learning Resources

i. Text Books:

1. AlfredAho, Ravi Sethi, Jeffrey D Ullman, "Compilers Principles, Techniques and Tools", Pearson Education Asia, 2003.

ii. Reference Books:

1. Allen I. Holub "Compiler Design in C", Prentice Hall of India, 2003.
2. C. N. Fischer and R. J. LeBlanc, "Crafting a compiler with C", Benjamin Cummings, 2003.
3. J.P. Bennet, "Introduction to Compiler Techniques", Second Edition, Tata McGraw-Hill, 2003.
4. HenkAlblas and Albert Nymeyer, "Practice and Principles of Compiler Building with C", PHI, 2001.
5. Kenneth C. Louden, "Compiler Construction: Principles and Practice", Thompson Learning, 2003

iii. Online Recourses:

1. http://www.tutorialspoint.com/compiler_design/
2. http://nptel.ac.in/courses/106104123/Compiler%20DesignQuestions.pdf
3. http://www.vssut.ac.in/lecture_notes/lecture1422914957.pdf

INDEX

1151CS115-COMPILER DESIGN

UNIT 1
INTRODUCTION TO COMPILERS

TOPICS:

Compilers, Analysis of the Source Program, The Phases of a Compiler, Cousins of the Compiler, The Grouping of Phases, Compiler-Construction Tools.

LEXICAL ANALYSIS: Need and role of lexical analyser-Lexical errors, Input Buffering - Specification of Tokens, Recognition of Tokens, Design of a Lexical Analyzer Generator

TEXTBOOK:

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, "Compilers – Principles, Techniques and Tools", 2nd Edition, Pearson Education, 2007.

REFERENCES:

1. Randy Allen, Ken Kennedy, "Optimizing Compilers for Modern Architectures: A Dependence-based Approach", Morgan Kaufmann Publishers, 2002.
2. Steven S. Muchnick, "Advanced Compiler Design and Implementation, "Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint 2003.
3. Keith D Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann Publishers Elsevier Science, 2004.
4. Charles N. Fischer, Richard. J. LeBlanc, "Crafting a Compiler with C", Pearson Education,2008

FACULTY IN-CHARGE **HOD - CSE**

UNIT-I

Introduction to Compilers

Compilers, Analysis of the Source Program, The Phases of a Compiler, Cousins of the Compiler, The Grouping of Phases, Compiler-Construction Tools.

LEXICAL ANALYSIS: Need and role of lexical analyser-Lexical errors, Input Buffering - Specification of Tokens, Recognition of Tokens, Design of a Lexical Analyzer Generator

INTRODUCTION

Translators

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

Important role of translator are:

1 Translating the high level language program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the high level language.

Examples of widely used types of computer languages translators include interpreters, compilers and decompilers, and assemblers and disassemblers.

- If the translator translates a high-level language into another high-level language, it's called a translator or source-to-source compiler. Examples include Haxe, FORTRAN-to-Ada translators, CHILL-to-C++ translators, PASCAL-to-C translators, COBOL(DialectA)-to-COBOL(DialectB) translators.
- If the translator translates a high-level language into a lower-level language, it is called a compiler. Notice that every language can be either translated into a (Turing-complete) high-level or assembly language.
- If the translator translates a high-level language into an intermediate code which will be immediately executed, it is called an interpreter.
- If the translator translates target/machine code to source language, it is called a decompiler. Example: DCC, Boomerang Decompilers and Reverse Engineering Compiler(REC).
- If the translator translates assembly language to machine code, it is called an assembler. Examples include MASM, TASM and NASM.
- If the translator translates machine code into assembly language, it is called a disassembler. Examples include gdb, IDA Pro and OllyDbg.
- Translators that translate from a human-readable design specified in terms of rules and high-level functions into the equivalent logic gates and chip layout needed to achieve its manufacture belong to electronic design automation and hardware description language categories.

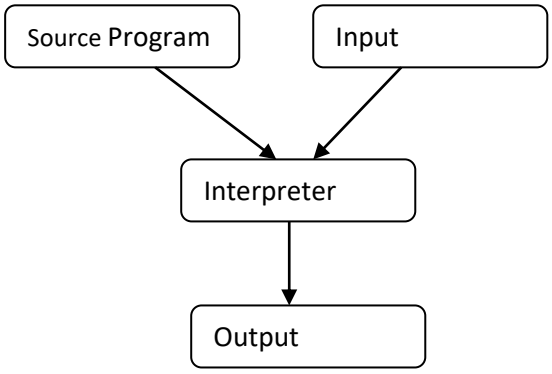
Compilation and Interpretation

Programming languages are usually implemented by interpreters or compilers, or some mix of both. Almost all language implementations are a mix of both.

There are two primary methods for translating high-level code:

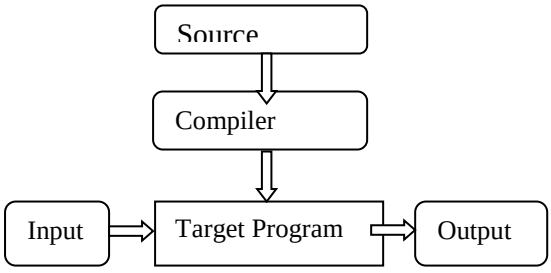
- Compilation
- Interpretation

A pure interpreter reads the source text of a program, analyzes it, and executes it as it goes. This is usually very slow--the interpreter spends a lot of time analyzing strings of characters to figure out what they mean. A pure interpreter must recognize and analyze each expression in the source text each time it is encountered, so that it knows what to do next. This is how most command shell languages work, including UNIX shells and Tcl.



Interpretation

A pure compiler reads the source text of a program, and translates it into machine code that will have the effect of executing the program when it is run. A big advantage of compilers is that they can read through and analyze the source program *once*, and generate code that you can run to give the same effect as interpreting the program.



- The “target program” is called the object code
- You translate once and run many times.

A compiler is a weird kind of interpreter, which "pretends" to interpret the program, and records what an interpreter would do. It then goes through its record of actions the interpreter would take, and spits out instructions whose effect is the same as what the interpreter would have done.

The compiler's spit out instructions that will do the "real work" that can only be done at runtime, because it depends on the actual data that the program is manipulating. For example, an if statement is always an if statement each time it's encountered, so that analysis can be done once. But which branch will be taken depends on the runtime value of an expression, so the compiler must emit code to test the value of the expression, and take the appropriate branch.

Most real interpreters are somewhere in between pure interpreters and compilers. They read through the source code for a program once, and translate it into an "intermediate representation" that's easier to work with--a data structure of some kind--and then interpret that.

Comparison of Compiler and Interpreter

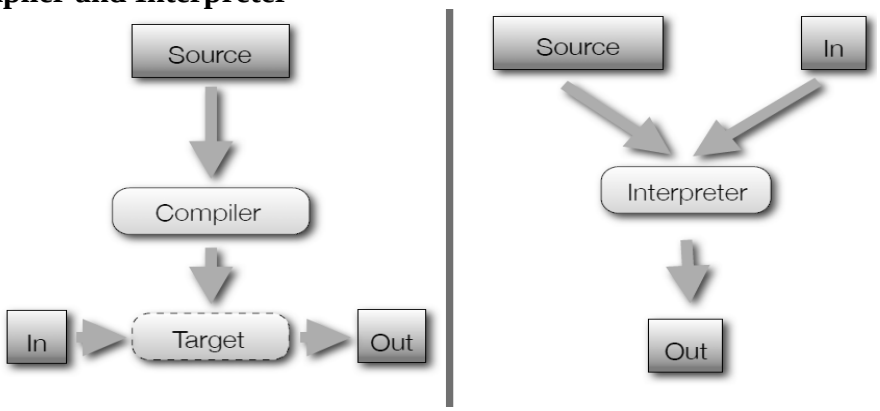


Figure: Comparison of Compiler and Interpreter

Compiler	*Interpreter*
Each statement translated once	Translate only if executed
Must compile	Run immediately
Faster execution	Allows more supportive environment
Only object code in memory when executing	Interpreter in memory
Object file likely large	Source likely smaller

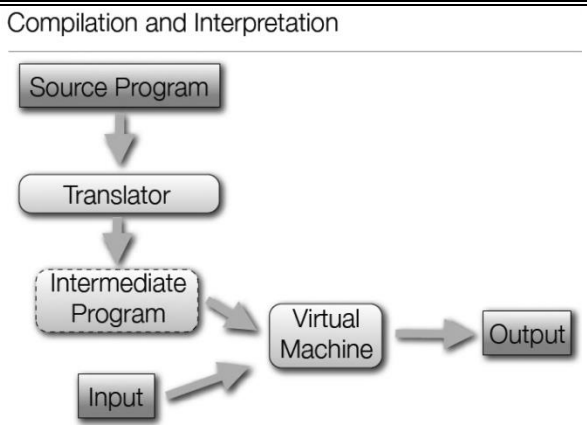


Figure Compilation and interpretation

- In the process of Compilation and interpretation, the translator can be a compiler or an interpreter. It is considered to be a compiler if:
    1. There is a thorough analysis of the program
    2. The transformation is non-trivial.
- This is exactly the process that Java uses.

Features of Compiler :

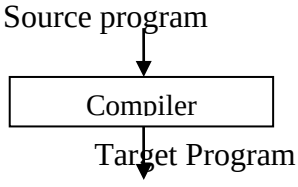
- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

Types of Compiler

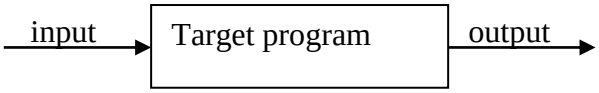
- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

Language Processors

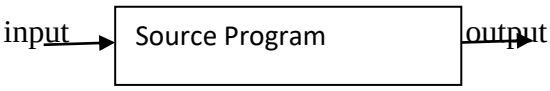
A compiler is a program that can read a program in one language (the *source* language) and translate it into an equivalent program in another language (the *target* language) see below figure. An important role of the compiler is to report any errors in the source program that it detects during the translation process.



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

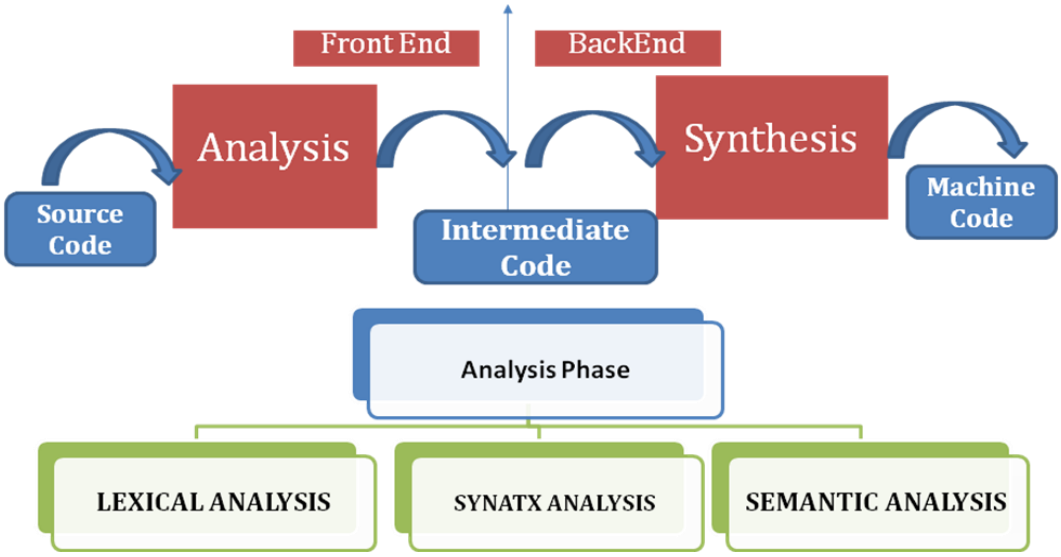


An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in below figure.



The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Analysis of Source Program - Architecture



Phases of a compiler

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)

b. Synthesis (Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the

analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.

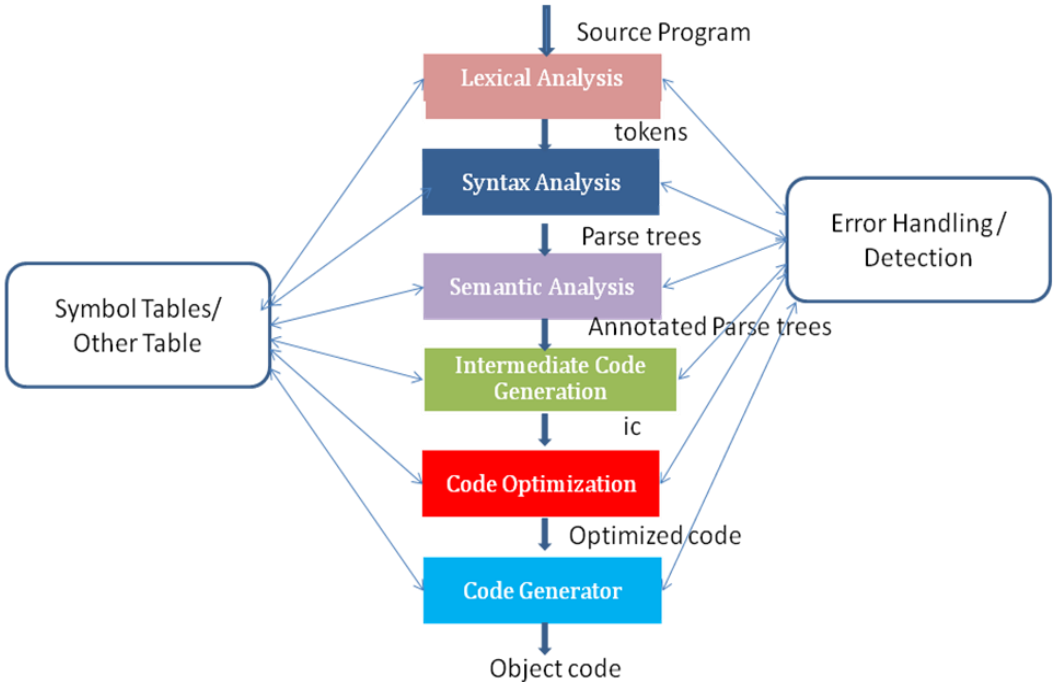
The analysis part also collects information about the source program and stores it in a data structure called a *symbol table,* which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end.*

The compilation process operates as a sequence of *phases,* each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in below figure.

The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program.



Lexical Analysis

- The first phase of a compiler is called *lexical analysis* or *scanning.* The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes.*

- For each lexeme, the lexical analyzer produces *token* of the form *(token-name, attribute-value)* as output that it passes on to the subsequent phase, syntax analysis.
- In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token.
- Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60 (1.1)

The characters in this assignment could be grouped into the following lexemesand mapped into the following tokens passed on to the syntax analyzer:

1. Position is a lexeme that would be mapped into a token **(id,** 1), where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component.
3. Initial is a lexeme that is mapped into the token **(id,** 2), where 2 points to the symbol-table entry for Initial.
4. + is a lexeme that is mapped into the token (+).
5. rate is a lexeme that is mapped into the token **(id,** 3), where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*).
7. 60 is a lexeme that is mapped into the token (60).

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure below shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

(i d , l) = (id, 2) (+) **(id,** 3) (*) (60) (1.2)

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators, respectively.

Syntax Analysis

- The second phase of the compiler is *syntax analysis* or *parsing.*
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in figure below**.**

- This tree shows the order in which the operations in the assignment

position = initial + **rate * 60**

are to be performed. The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.**

- The node labeled * makes it explicit that we must first multiply the value of **rate** by **60.**
- The node labeled + indicates that we must add the result of this multiplication to the value of initial.
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position.
- This ordering of operations is that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.
- The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

Semantic Analysis

- The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is *type checking.*
- The language specification may permit some type conversions called *coercions.* If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.
- Suppose that position , initial , and rate have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer. The type checker in the semantic analyzer in figure discovers that the operator * is applied to a floating-point number rate and an integer 60. In this case, the integer may be converted into a floating-point number. In above figure, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat,** which explicitly converts its integer argument into a floating-point number.

Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.
- This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

- We consider an intermediate form called *three-address code,*which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
t1= inttofloat(60)
t2 = id3 * t1
t 3  = id2 + t2                    (1.3)
id1=t3
```

First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.
- For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.
- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, **t3** is used only once to transmit its value to i d l so the optimizer can transform (1.3) into the shorter sequence
    - t1 = id3 * 60.0
    - id1 = id2 + t1(1.4)
- There are simple optimizations that significantly improve the running time of the target programwithout slowing down compilation too much.

Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code Optimization –The optimization process can be broadly classified into two types :

- **Machine Independent Optimization –** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
- **Machine Dependent Optimization –** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers Or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.
- For example, using registers Rl and **R2,** the intermediate code in (1.4) might get translated into the machine code

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2                    (1.5)
ADDF  R1, R1, R2
STF   id1, R1
```

The first operand of each instruction specifies a destination. The **F** in each instruction tells us that it deals with floating-point numbers.

The code in (1.5) loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0. The # signifies that 60.0 is to be treated as an immediate constant.

The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2. Finally, the value in register R1 is stored into the address of id1 , so the code correctly implements the assignment statement (1.1).

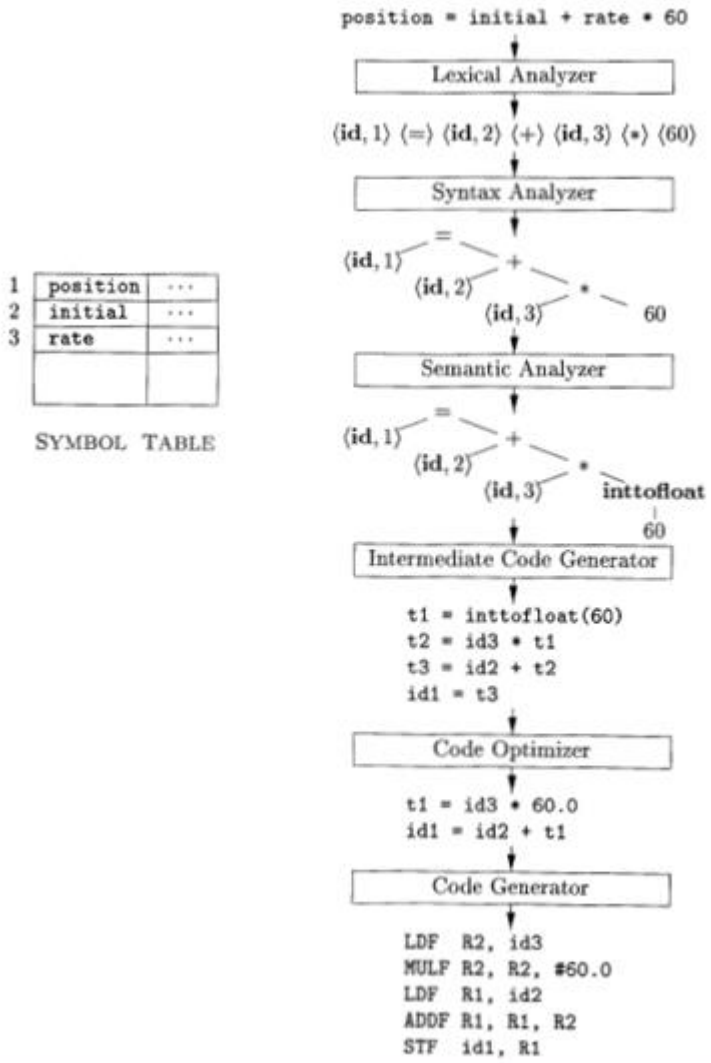
Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Example including all phases of Compiler:



Errors Encountered in Different Phases

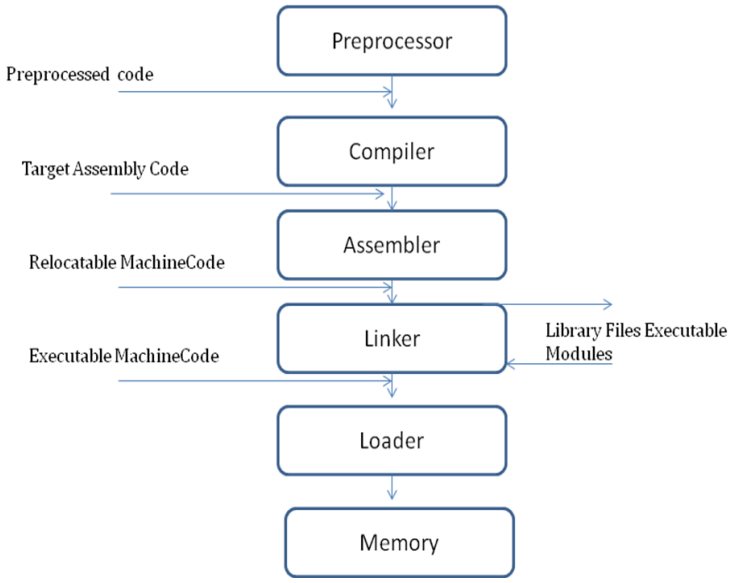
One of the most import functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by virtually all of the phases of a compiler. For example

1. The lexical analyser may be unable to proceed because the next token in the source program is misspelled (errors occur in separation of tokens)
2. The syntax analyser may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred. (errors occur during construction of syntax tree)
3. In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
4. The intermediate code generator may detect an operator whose operands have incompatible types.
5. The code optimizer, doing control flow analysis may detect that certain statements can never be reached (errors occur when the result is affected by the optimization)
6. The code generator may find a compiler-created constant that is too larger to fit in a word of the target machine.(it shows error when code is missing etc)

7. While entering information into the symbol table, the bookkeeping routine may discover an identifier that has been multiply declared with contradictory attributes

- Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors.
- Good error handling is difficult because certain errors can mask subsequent errors.

Cousins of Compiler



1) Preprocessor

It converts the HLL (high level language) into pure high level language. It includes all the header files and also evaluates if any macro is included. It is the optional because if any language which does not support #include and macro preprocessor is not required.

Functions include:

1. Macro processing - A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. File Inclusions -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

2) Compiler

It takes pure high level language as a input and convert into assembly code.

3) Assembler

It takes assembly code as an input and converts it into assembly code.

4) Linking and loading

It has four functions

1. **Allocation:**
It means get the memory portions from operating system and storing the object data.

2. **Relocation:**
It maps the relative address to the physical address and relocating the object code.

3. **Linker:**
   It combines all the executable object module to pre single executable file.
4. **Loader:**
   It loads the executable file into permanent storage.

The Grouping of Phases into Passes

Compiler can be grouped into front and back ends:

Front end: analysis (machine independent)

These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

Back end: synthesis (machine dependent)

It includes code optimization phase and code generation along with the necessary errorhandling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.
  Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

Compiler-Construction Tools

These are specialized tools that have been developed for helping implement various
phases of a compiler. The following are the compiler construction tools:

1. ***Parser generators*** that automatically produce syntax analyzers from a grammatical description of a programming language.

2. ***Scanner generators***that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. ***Syntax-directed translation engines*** that produce collections of routines for walking a parse tree and generating intermediate code.

4. ***Code-generator generators*** that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. ***Data-flow analysis engines*** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. ***Compiler-construction toolkits*** that provide an integrated set of routines for constructing various phases of a compiler.

NEED AND ROLE OF LEXICAL ANALYZER

- To identify the tokens, need some method of describing the possible tokens that can appear in the input stream.
- For this purpose, introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly, having decided what the tokens are, need some mechanism to recognize these in the input stream.
- This is done by the token recognizers, which are designed using transition diagrams and finite automata.
- A simple way to build a lexical analyzer is to construct a diagram that illustrates the structure of the tokens of the source language, and then to hand translate the diagram in to program for finding tokens.

ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis shown in fig.2.1

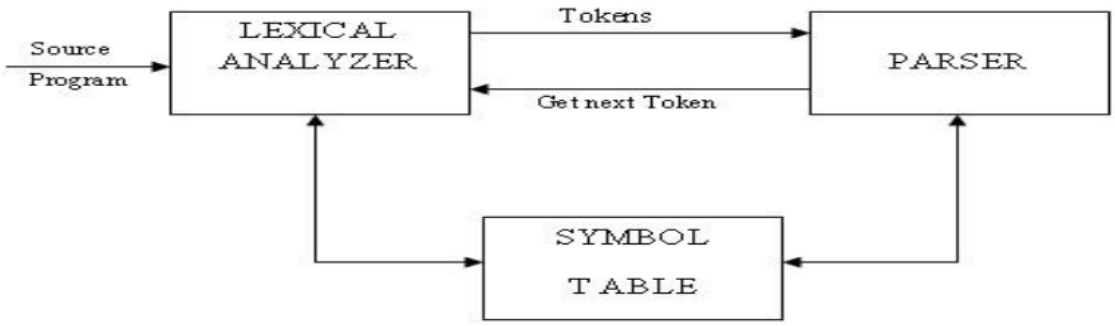


Figure: Interactions between the lexical analyzer and the parser

- 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.

1. One such task is striking out from the source program the commands and white spaces in the form of blank, tab and new line characters.
2. Another is correlating error message from the compiler with the source program.

Sometimes .lexical analyzers are divided into a cascade of two phases,

a) Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

LEXICAL ANALYSIS VS PARSING:

Table: 2.1 Difference between Lexical Analysis and Parsing

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).

ISSUES IN LEXICAL ANALYSIS:

The are several reasons for separating the analysis phase of compiling in to lexical and parsing.

1. Simpler design is the most consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

2. Compiler efficiency is improved by specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler speed up the performance of compiler.

3. Compiler portability is enhanced.

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.

Example: const pi=3.14;

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

Table2.2: Token with lexeme and pattern

A patter is a rule describing the set of lexemes that can represent a particular token in source program shown in Table 2.2.

Attributes for Tokens:

- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to subsequent phases of the compiler.
- The lexical analyzer collects information about tokens into their associated attributes.
- A token has usually only a single attribute-a pointer to the symbol-table entry in which the information about the token is kept shown in Table 2.3.

Example :E=M*C2;**

Lexeme	Tokens	Attribute values
E	id	pointer to symbol-table entry for E
=	assign_op	---
M	id	pointer to symbol-table entry for M
*	mult_op	---
C	id	pointer to symbol-table entry for C
**	exp_op	---
2	num	2

Table: 2.3 Token name and Attribute values E=M*C2**

LEXICAL ERRORS

- If the string "fi" is encountered in a C program for the first time in the context fi(a= =x)
  a lexical analyzer cannot tell whether "fi" is a misspelling of the keyword if or an undeclared function identifier.
- Since fi is a valid identifier, the lexical analyzer must return the tokens for an identifier and let some other phase of the compiler handle any error.
- But suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the patterns for tokens match a prefix of the remaining input.
- Perhaps the simplest recovery strategy is "panic mode" recovery. Delete successive characters from the remaining input the lexical analyzer can find a well-formed token.

Error-recovery actions are:

i. Delete one character from the remaining input.

ii. Insert a missing character in to the remaining input.

iii. Replace a character by another character.

iv. Transpose two adjacent characters.

INPUT BUFFERING

The readings of input are the general approaches to the implementation of a lexical analyzer.

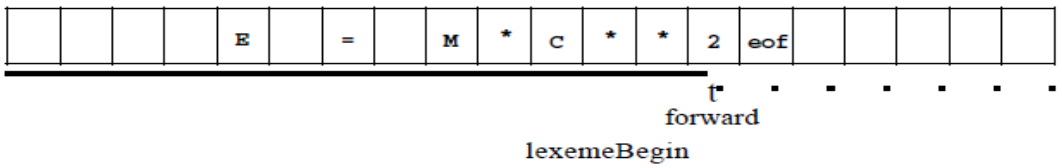
1. Use a lexical analyzer generator, such as lex compiler, to produce the lexical analyzer from a regular expression –based specification. In this case, the generator provides routines for reading and buffering the input.

2. Write the lexical analyzer in a conventional systems-programming language using the I/O facilities of that language to read the input.

3. Write the lexical analyzer is assembly language and explicitly manage the reading of input.

Buffer Pairs:

- The lexical analyzer 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 and other reasons, it is desirable for the lexical analyzer to read input from an input buffer. There are many schemes that can be used to buffer input. one class of schemes here.



A buffer divided into two N-character halves. N is the number of characters on one disk block. eg:1024.
Two pointers to the input are maintained:

1. Pointer **lexemeBegin,** marks the beginning of the current lexeme, whose extent are attempting to determine.

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 this chapter fig 2.3.

If forward at end of first half then
reload second half
forward+=1
else if forward at end of second half reload the first half
move forward to beginning of first half
else
forward+=1

Figure: Increment procedure for forward pointer

Sentinels:

- Except at the ends of the buffer halves the code requires two tests for each advance of the forward pointer.
- Reduce the two tests to one if extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character [eof] that cannot be part of the source program. The same buffer arrangement with the sentinels added.

switch (**forward++) {*
case of:
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;

Cases for the other characters

}

Figure: Increment procedure for forward pointer using sentinels

SPECIFICATION OF TOKENS

Alphabet or character class:

Alphabet denotes any finite set of symbols.EG:{0,1} is the binary alphabet.

String or sentence or word:

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

Length of a string:

Length of a string is the number of occurrences of symbols in string s denoted by |s|

Eg: s=abc=|s|=3

Empty string:

The empty string, denoted e, is the string of length zero Language : A language is any countable set of strings over some fixed alphabet.

Eg: The set of all string of zero or more a's.over an alphabet{a}

L={€,a,aa,aaa,….}

Empty set:

The empty set is the set containing only the empty string, denoted$^{\emptyset}$ or {€}.

Parts of a string:

1.Aprefix of string s is any string obtained by removing zero or more symbols from the end of s. For example, ban, banana, and e are prefixes of banana.

2. A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana, banana, and e are suffixes of banana.

3. A substring of s is obtained by deleting any prefix and any suffix from s. For instance, banana, nan, and e are substrings of banana.

4. The proper prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not e or not equal to s itself.

5. A subsequence of s is any string formed by deleting zero or more not necessarily

Consecutive positions of s. For example, baan is a subsequence of banana.

If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x. For example, if x = dog and y = house, then xy — doghouse.

Operations on Languages:

Union: union of L and M denoted LUM is the set of strings that are in either L or M or both.

LUM={S|S is in L or S is in M}

Concatenation:

Concatenation of L and M denoted L ,M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M.

LM={ST| S is in L and T is in M}

Kleen closure or star closure:

Star closure of L, denoted L* is the set of those strings that can be formed by taking any number of strings from L and concatenating all of them.

$L^*=U_{i=0}^{\infty} L^i$

Positive closure:

positive closure of L denoted L^+ is the set of those strings that can be formed by one or more concatenations of L

$L^+=U_{i=1}^{\infty} L^i$

Recognition of Tokens

Regular expressions

- A regular expression is any well-formed formula constructed over union, concatenation and closure.
- RE are important notation for specifying pattern.
- A RE is built up out of simpler regular expressions using a set of defining rules. Each RE r denotes a language L(r).

The rules that define the RE over alphabet ∑ are

1. € is a regular expression, and L(€) is {€}, that is, the language whose sole member is the empty string.
2. ifa is a symbol in E, then **a** is a regular expression, and L**(a) =** {a}, that is, the language with one string, of length one, with a in its one position.
3. Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively. then
   a. (r)|(s) is a regular expression denoting the language L(r) U L(s).

b. (r)(s) is a regular expression denoting the language L(r)L(s).

c. (r)* is a regular expression denoting (L(r))*.

d. r^+ is a regular expression denoting $L(r)^+$.

Regular set

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

Precedence of regular expression operators:

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) | has lowest precedence and is left associative.(Table 2.4)

EG: **(a)|((b)*(c))** by **a|b***c. denote the set of strings that are either a single a or are zero or more 6's followed by one c.

Algebraic Properties for regular expressions:

Axiom	**Description**
r\s = s\r	| is commutative
r\(s\t) = (r\s)\t	| is associative
r(st) = (rs)t	Concatenation is associative
r(s\t) = rs\rt; (s\t)r = sr\tr	Concatenation distributes over |
er = re = r	e is the identity for concatenation
r* = (r|e)*	e is guaranteed in a closure
r**=r*	* is idempotent

Table2.4 Axiom with Description

Example : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. Conventionally use italics for the symbols defined in regular definitions.

letter→ **A | B | - - - | Z | a | b | - - - | z |**

digit→ **0** | **1** |• • • | **9**

id → letter (letter| digit)*

Extensions of Regular Expressions

Zero or more instance

The unary operator * means "zero or more instance of"

Eg :L^*={ **ε**, L,LL, LLL….}

One or more instances:

The unary, postfix operator + represents the one or more instance of

Eg: a^+={a,aa,aaa….}

Zero or one instance.

The unary postfix operator ?means "zero or one occurrence."

r^* is equivalent to r + **ε**

Character classes.

The notation [abc] where a,b and c are alphabet symbols denotes are regular expressions a|b|c. Eg: **[a-z]** is shorthand for **a | b | . - - | z .**

Character not in a given set:

The carat character is used to form complements of the character classes.

Eg: [^a] any character that is not a.

Recognition of tokens:

To express pattern using regular expressions. Now, how to take the patterns for all the needed tokens and build a piece of code that examines 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
| term

Term →id
|number

- For relop , 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, assign the lexical analyzer the job stripping out white space, by recognizing the "token" defined by:

ws→ (blank/tab/newline)$^+$

- Here, blank, tab and newline are abstract symbols that use to express the ASCII characters of the same names.
- Token ws is different from the other tokens in that ,when recognize it, 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

Table: 2.5 Tokens, their patterns, and attribute values

TRANSITION DIAGRAM:

- Transition Diagram has a collection of nodes or circles, called states.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .
- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a symbol or set of symbols.
- If it is in one state s, and the next input symbol is a, for an edge out of state s labeled by a.
- If find such an edge ,advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state .it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used shown in below figure.

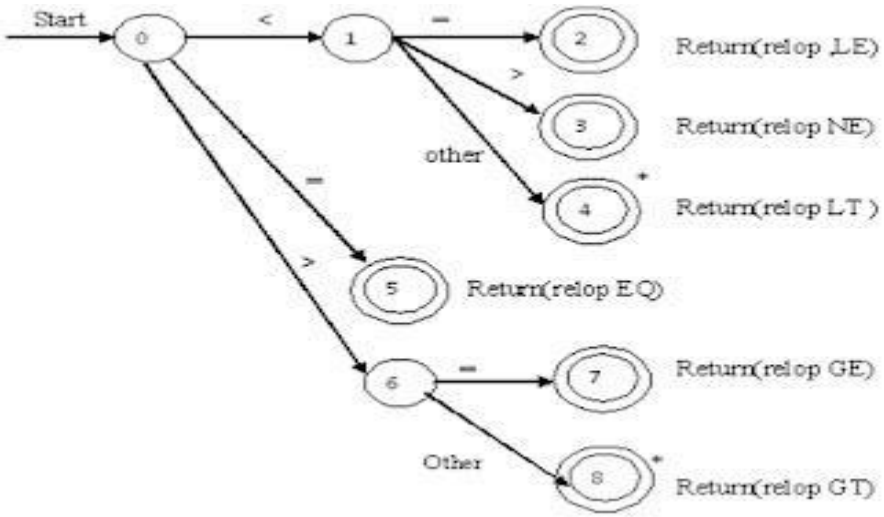


Figure: Translation diagram for relop

As an intermediate step in the construction of a LA, first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.

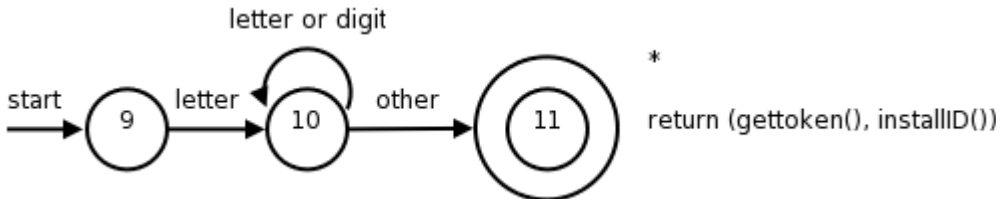


Figure: A transition diagram for id's and keywords

The above TD for an identifier, defined to be a letter followed by any no of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

If = if

Then = then

Else = else

Relop = < | <= | = | > | >=

Id = letter (letter | digit) *|

Num = digit |

AUTOMATA

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

• Call the recognizer of the tokens as a *finite automaton*.

• A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*

• This means that may use a deterministic or non-deterministic automaton as a lexical analyzer.

• Both deterministic and non-deterministic finite automaton recognizes regular sets.

• Which one?

– Deterministic – faster recognizer, but it may take more space

– Non-deterministic – slower, but it may take less space

– Deterministic automatons are widely used lexical analyzers.

• First, define regular expressions for tokens; Then convert them into a DFA to get a lexical analyzer for our tokens.

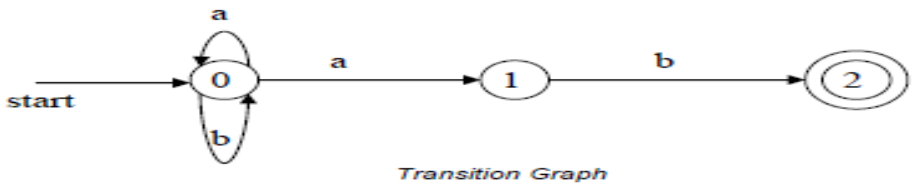
An automation in which the output depends only on the state of the machine is **called a Moore machine**.

An automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

Non-Deterministic Finite Automaton (NFA)

- A NFA has no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and € the empty string is a possible label.
- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
    - Q – finite set of states
    - Σ - a set of input symbols (alphabet)
    - δ move - a transition function move to map state-symbol pairs to sets of states. ∑U{€}
    - q0 - a start (initial) state
    - F- a set of accepting states (final states)
    - ε- Transitions are allowed in NFAs. In other words, can move from one state to another one without consuming any symbol.

Example:



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

The language recognized by this NFA is (a|b)*ab

Figure: A NFA & Transition Table

Deterministic Finite Automaton (DFA)

A Deterministic Finite Automaton (DFA) is a special form of a NFA.

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1, it has no transitions on input **ε** ,

2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, Σ, δ, qo, F), where

Q is a finite 'set of states', which is non -empty.

Σ is 'input alphabets', indicates input set.

qo is an 'initial state' and qo is in Q ie, qo, Σ, Q

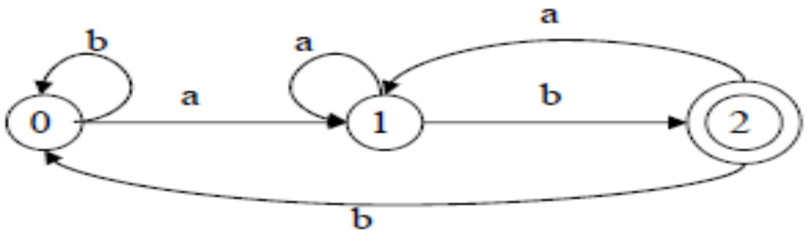
F is a set of 'Final states',

δ is a 'transmission function' or mapping function, using this function the next state can be determined.

• No state has ε- transition

• For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transition function is from pair of state-symbol to state (not set of states) in fig 2.6.

The DFA to recognize the language (a|b)* ab is as follows.



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

Figure: DFA & Transition Table

LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS-LEX

- There is a wide range of tools for construction of lexical analyzers. The majority of these tools are based on regular expressions.

- One of the traditional tools of that kind is Lex. Lex tool as the Lex compiler, and to its input specification as the Lex language.
- An input file call as lex.l is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c.
- The latter file is compiled by the C compiler into a file called a.out as always. The C-compiler output is a working lexical analyzer that can a stream of input characters and produce a stream of tokens in below figure

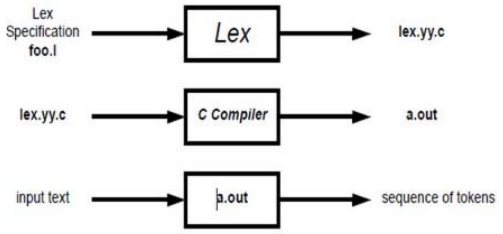


Fig. Creating lexical analyzer using lex

Structure of Lex programs

A Lex program consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The *declarations* section includes declarations of variables, manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.

2. The *translation rules* of a Lex program are statements of the form :

Pattern { Action }

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

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

- A lexical analyzer created by lex behaves in concert with a parser in the following manner. when activate by the parser.
- The lexical analyzer begins reading its remaining input ,one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions Pi.

- Then it executes action i. Typically action i will return control to the parser.
- However if it does not then the lexical analyzer proceeds to find more lexemes, until an action causes control to return to the parser.
- The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.
- The lexical analyzer returns a single quantity, the token to the parser.
- To pass an attribute value with information about the lexeme, can set a global variable called yylval.

Example1:

```
%{
#undef yywrap
#define yywrap() 1
 %}

%%
  [\n] {
printf("Hello World\n");
}
 %%
  main()
{
yylex(); //calling the rules section
}
```

Execution:

```
Flex hello.l
gcc lex.yy.c
./a.out
```

Output :

Hello World

Example 2:

```
%{
        /* definitions of manifest constants
```

```
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions
Delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
IntinstallID()  {/* funtion 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 */
}
IntinstallNum() { /* similar to installID, but puts numerical constants into a separate table */
}
```

DESIGN OF LEXICAL ANALYZER FOR A SAMPLE LANGUAGE

- The designing technique in generating a lexical-analyzer.
- Discuss two approaches, based on NFA's and DFA's.
- The program that serves as the lexical analyzer includes a fixed program that simulates an automaton.The rest of the lexical analyzer consists of components that are created from the Lex program.

The Structure of the Generated Analyzer

- Its components are:
    - A transition table for the automaton.
    - 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 by the automaton simulator

- To construct the automaton, begin by taking each regular-expression pattern in the Lex program and converting it to an NFA.
- A single automaton that will recognize lexemes matching any of the patterns in the program.So combine all the NFA's into one by introducing a new start state with ε-transitions to each of the start states of the NFA's Ni for pattern *Pi*

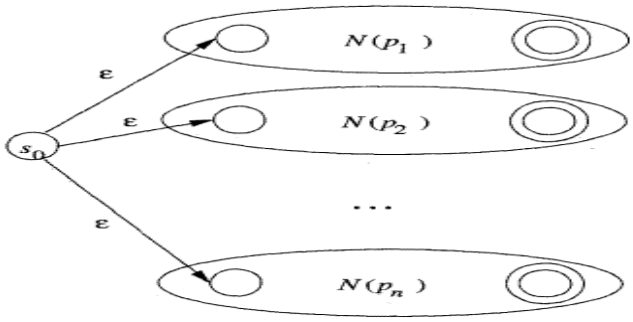


Fig :2.20 An NFA constructed from a Lex program

a { action A_1 for pattern P_1 }

abb { action A_2 for pattern P_2 }

a*b$^+$ { action A_n for pattern P_n}

Pattern Matching Based on NFA 's

- For pattern based matching the simulator starts reading characters and calculates the set of states.
- At some point the input character does not lead to any state or have reached the eof.
  - Since to find the longest lexeme matching the pattern proceed backwards from the current point (where there was no state) until reach an accepting state (i.e., the set of NFA states, N-states, contains an accepting N-state).
  - Each accepting N-state corresponds to a matched pattern.

The lex rule is that if a lexeme matches multiple patterns choose the pattern listed first in the lex-program.

Ex. Consider three patterns and their associated actions and consider processing the input aaba.

Pattern	**Actions to perform**
a	Action A_1
abb	Action A_2
a*b$^+$	Action A_3

Part-A

Questions and Answers

1. Define compiler?

A compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) and the compiler reports to its user the presence of errors in the source program.

2. What are the two parts of a compilation ?

The following are the parts of a compilation

i)Analysis phase

ii)Synthesis phase

3. What are the phases of compiler?

- Lexical analyser
- Syntax analyzer
- Semantic analyzer
- Intermediate code generation
- Code generation
- Code optimization and symbol table manager.

4. Define preprocessor & what are the functions of preprocessor?

Preprocessor produce input to the compilers (i.e.) the program will be divided in to the modules. They may perform the following functions.

i) Macro processing

ii) File inclusion

iii) Rational preprocessor

iv) Language extension

5. **What are the tools available in analysis phase?**

ii) Pretty printer

iii) Static checkers

iv) Interpreters.

6. Define pretty printers?

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

7. Define assembler and its types?

It is defined by the low level language is assembly language and high level language is machine language is called assembler.

- One pass assembler
- Two pass assembler

8. Give the types of a language processing system?

a) Preprocessors

b) Compilers

c) Assembler

d) Loaders and link editors

9. What are the functions performed in analysis phase?

a) Lexical analysis or Linear analysis
b) Syntax analysis or hierarchical analysis
c) Semantic analysis

10. What are the functions performed in synthesis phase?

i) Intermediate code generation
ii) Code generation
iii) Code optimization

11. Give the classification of processing performed by the semantic analysis?

a) Processing of declarative statements.
b) Processing of executable statements.

12. Give the properties of intermediate representation?

a) It should be easy to produce.
b) It should be easy to translate into the target program.

13. What is meant by lexical analysis?

It reads the characters in the program and groups them into tokens that are sequences of characters having a collective meaning. Such as an identifier, a keyword, a punctuation, character or a multi-character operator like ++.

14. What is meant by syntax analysis?

It processes the string of descriptors, synthesized by the lexical analyzer, to determine the syntactic structure of an input statement. This process is known as parsing. Output of the parsing step is a representation of the syntactic structure of a statement. It representation in the form of syntax tree.

15. What is meant by intermediate code generation?

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. It can have a variety of forms.This form called three-address code. It consists of sequence of instructions, each of which has at most three operands.

16. What is meant by semantic analysis?

This phase checks the source program for semantic errors and gathers type of information for the subsequent phase.

17. What do you meant by interpreter?

Certain other translators transform a programming language into a simplified language called intermediate code, which can directly executed using a program called an interpreter.

18. What do you meant by phases?

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

19. Write short notes on symbol table manager?

The table management or bookkeeping portion of the compiler keeps track of the names used by program and records essential information about each, such as its type (int,real etc.,) the data structure used to record this information is called a symbol table manger.

20. Write short notes on error handler?

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

21. Mention some of the cousins of the compiler

i) Preprocessors
ii) Assemblers
iii) Two pass assembly
iv) Loaders and Linker-editors.

22. What is front end and back end?

The phases are collected into a front end and a back end. The front end consists of those phases or parts of phases, that depends primarily on the source language and is largely independent of the target machine. The back ends that depend on the target machine and generally these portions do not depend on the source language.

23. What do you meant by passes?

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

24. List some compiler construction tools?

i) Parser generators
ii) Scanner generators
iii) Syntax-directed translation engine
iv) Automatic code generators
v) Data-flow engine

25. Explain any one compiler construction tool?

Scanner generators, these automatically generate lexical analyzers normally from a specification based on regular expressions. The resulting of lexical analyzer is in effect of finite automata.

26. What are issues available in lexical analysis?

i) Simpler design
ii) Compiler efficiency is improved by specialized buffering techniques for reading input characters and processing tokens and significantly speeds up the performance of a compiler.
iii) Compiler portability is enhanced.

***27.* What is the role of lexical analyzer?**

Its main task is to read the input characters and produce as output a sequence of tokens that theparser uses for syntax analysis.

28. Mention few cousins of compiler.

The following are the cousins of compilers
i. Preprocessors
ii. Assemblers
iii. Loaders
iv. Link editors.

29. What are the possible error recovery actions in lexical analyzer?

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

30. **Illustrate diagrammatically how a language is processed.**

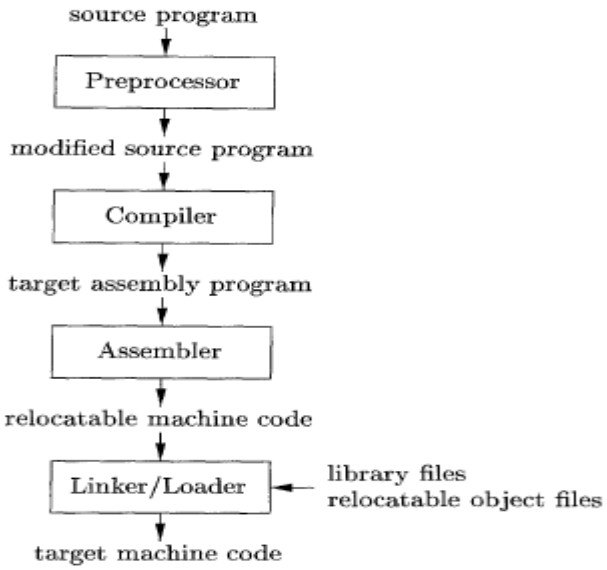


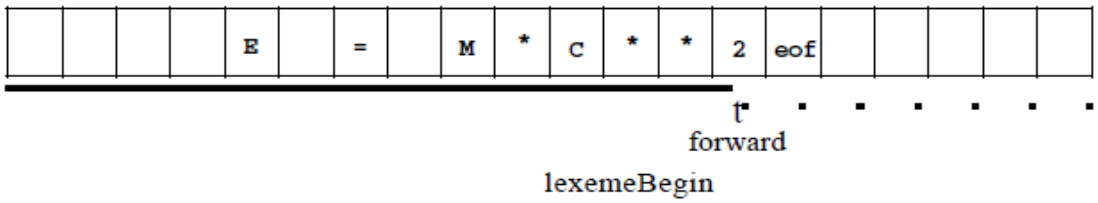
Figure 1.5: A language-processing system

1. What is the role of lexical analyzer

Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

2. Explain about input buffering.

Look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



3. Differentiate tokens, patterns, lexeme.

- Tokens- Sequence of characters that have a collective meaning.
- Patterns- There is 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 sequence of characters in the source program that is matched by the pattern for a token.

4. What are the possible error recovery actions in lexical analyzer?

1. Deleting an extraneous character

2. Inserting a missing character
3. Replacing an incorrect character by a correct character
4. Transposing two adjacent characters

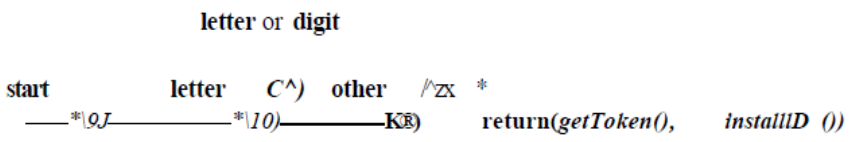
1. List the operations on languages.

- **Union -** L U M ={s | s is in L or s is in M}
- **Concatenation** – LM ={st | s is in L and t is in M}
- **Kleene Closure –** L* (zero or more concatenations of L)
- **Positive Closure –** L+ (one or more concatenations of L)

2. Write a regular expression for an identifier.

An identifier is defined as a letter followed by zero or more letters or digits.
The regular expression for an identifier is given as
letter (letter | digit)*

3. Give the transition diagram for an identifier.



4. Mention the various notational shorthand for representing regular expressions.

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

5. List the various error recovery strategies for a lexical analysis.

- Possible error recovery actions are:
- Panic mode recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters

6. Write a grammar for branching statements.

S→iEtS | iEtSeS |a
E→b

7. What is a lexeme? Define a regular set.

- A Lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- A language denoted by a regular expression is said to be a regular set

8. What is a sentinel? What is its usage?

A Sentinel is a special character that cannot be part of the source program. Normally use, eof‟ as the sentinel. This is used for speeding-up the lexical analyzer.

9. What is a regular expression? State the rules, which define regular expression?

Regular expression is a method to describe regular language

Rules:

1) €-is a regular expression that denotes {€} that is the set containing the empty string
2) If a is a symbol in €,then a is a regular expression that denotes {a}
3) Suppose r and s are regular expressions denoting the languages L(r) and L(s) Then,

a) (r)/(s) is a regular expression denoting L(r) U L(s).
b) (r)(s) is a regular expression denoting L(r)L(s)
c) (r)* is a regular expression denoting L(r)*.
d) (r) is a regular expression denoting L(r).

10. Construct Regular expression for the language L= {w € {a,b}/w ends in abb}

Ans: {a/b}*abb

11. What is recognizer?

Recognizers are machines. These are the machines which accept the strings belonging to certain language. If the valid strings of such language are accepted by the machine then it is said that the corresponding language is accepted by that machine, otherwise it is rejected.

12. Give the algebraic properties of regular expression?

AXIOM	**DESCRIPTION**
i) r/s	= s/r / is commutative
ii) r/(s/t)	=(r/s)/t / is associative
iii) (rs)t	=r(st) concatenation is associative
iv) r(s/t)	=rs/rt concatenation distributes over /
v) r**	=r* * is idempotent

13. Give the parts of a string?

Prefix of s, suffix of s, substring of s, proper prefix, proper suffix, proper substring and subsequence of s.

14. What are the systems referred to data flow engine?

i) Compiler-compilers
ii) Compiler-generators
iii) Translator writing systems.

15. What are the operations on language?

- Union
- Concatenation
- Kleene closure or star closure and
- Star closure.

16. Give the error recovery actions in lexical errors?

i) Deleting an extraneous character
ii) Inserting a missing character
iii) Replacing an incorrect character by a correct character.

17. What are the implementations of lexical analyzer?

a) Use a lexical analyzer generator, such as Lex compiler, to produce the lexical analyzer from a regular expression based specification
b) Write the lexical analyzer in a conventional systems-programming language using the I/O facilities of that language to read the input.
c) Write the lexical analyzer in assembly language and explicitly manage the reading of input.

21. **Define regular expression?**
It is built up out of simpler regular expression using a set of defining rules. Each regular expression „r" denotes a language L(r). The defining rules specify how L(r) is formed by combining in various ways the languages denoted by the sub expressions of r.

22. **Give the precedence of regular expression operator?**
i) The unary operator * has the highest precedence and is left associative.
ii) Concatenation has the second highest precedence and is left associative.
iii) / has the lowest precedence and is left associative.

23. Define the length of a string?
It is the number of occurrences of symbols in string, s" denoted by |s|.
Example: s=abc, |s| =3.

24. Give the rules in regular expression?
1) € is a regular expression that denotes {€}, that is the set containing the empty string.
2) If „a" is a symbol in Σ, then a is a regular expression that denoted {a} , i.e., the set containing the string a.
3) Suppose r and s are regular expression denoting the languages L(r) and L(s) .

25. **Define regular set?**
A language denoted by a regular expression is said to be a regular set.

26. Give the types of notational shorthand's of RE?
Zero or more instance
One or more instance
Character classes
Character not in a given set and unary operator?

27. **Define kleene closure or star closure and positive closure?**
Star closure of L, denoted L*, is the set of those strings that can be formed by taking any number of strings from L and concatenating all of them.
L* = Li
Positive closure of L, denoted L+, is the set of those strings that can be formed by one or more concatenations of L
L+ = Li

28. Define character class with example.
The notation [abc] where a, b, c are alphabet symbols denotes the regularexpression a/b/c.
Example:
[A-z] = a | b | c | -------| z
Regular expression for identifiers using character classes
[a – z A – Z] [A – Z a – z 0 – 9] *

29. Give the error recovery strategies in lexical analyzer.
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing of two adjacent character

30. Write the R.E. for the following language.

i. Set of statements over {a,b,c} that contain no two consecutive b‟s

(B/c) (A/c/ab/cb) *

ii. Set of statements over {a,b,c} that contain an even no of a‟s

((b/c)* a (b/c) * a)* (b/c)*

31. Describe the language denoted by the following R.E.

(0/1)*0(0/1)(0/1) ii. 0(0/1)*0

32. The set of all strings of 0's and 1's starting and ending with 0.

(00/11)*((01/10)(00/11)*(01/10)(00/11)*)

33. What are the tasks in lexical analyzer?

- One task is stripping out from the source program comments and white space in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

34. Define finite automata and its types with eg.

- A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.
- A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a finite automation.
    1. Deterministic (DFA)
    2. Non-deterministic (NFA)

35. What are the three parts of lexical program?

- Declarations

%%

- Translation rules

%%

- Auxiliary procedures

36. What are the four functions of regular expression to DFA?

- Nullable (n)
- Firstpos (n)
- Last (n)
- Followpos (n)

37. What are the models of LEX compiler?

- Lexeme
- FA simulator
- Transition table

38. Compare the features of DFA and NFA.

DFA	NFA
All transitions are deterministic Each transitions leads to exactly one state	transitions could be non-deterministic A transition could lead to a subset of states

For each state, transition on all possible symbols(alphabet)should be defined.	For each state, not all symbols necessarily have to be defined in the transition function
Accepts input if the last state is in F.	Accepts input if one of the last states is in F.
Sometimes harder to construct because of the number of states	Generally easier than a DFA to construct
practical implementation is feasible	Practical implementation has to be deterministic.

39. Why is buffering used in lexical analysis? What are the commonly used buffering methods?

The lexical analyzer reads the input source program and identifies the tokens from it. The input is read from the secondary storage. But reading the input in this way is costly. Hence the input source program are sentences are stored in the buffer. That is why buffering is required in lexical analysis.

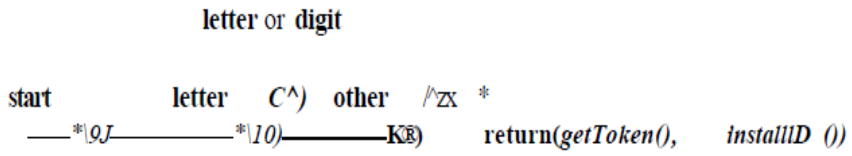
The one buffer and two buffer schemes are the commonly used buffering methods.

40. What are the issues of the lexical analyzer?

The are several reasons for separating the analysis phase of compiling in to lexical and parsing.

1. Simpler design is the most consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
2. Compiler efficiency is improved by specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler speed up the performance of compiler.
3. Compiler portability is enhanced.

41. Write the regular expression for the identifier and whitespace.



RE for white space is

r.e=(|\t\n)*

42. Write the RE for identifier and number.

RE for identifier is**R.E=letter(letter+digit)***

RE for integer number is **RE=digit.Digit***

PART-B & C Questions

1. (a)What is a compiler? Explain the various phases of compiler in detail, with a neat sketch.
   (b) Elaborate on grouping of phases in a compiler
2. (a) Describe the following software tools
   Structure Editors ii. Pretty printers iii. Interpreters
3. Write in detail about the cousins of the compiler.
4. Explain the role performed by lexical analysis of the compiler.
5. Explain the need of code optimization in compiler.
6. Explain the phases of compiler
7. Write the short notes on compiler construction tools.
8. Explain specification of tokens
9. Explain the cousins of compiler.
10. .Explain the various phases of a compiler in detail. Also write down the output for the following expression after each phase a:=b*c-d. (b) What are the phases of the compiler? Explain the phases in detail. Write down the output of each phase for the expression a: = b + c *50.
11. Describe the various phases of compiler and trace the program segment 4:*+=cba for all phases
12. Explain language processing system with neat diagram
13. Explain the need for grouping of phases.
14. Explain various Error encountered in different phases of compiler

Part 2 Unit I

1. Explain the language for specifying the lexical analyzer
2. Explain specification and recognition of tokens
3. Describe the error recovery schemes in the lexical phase of the compiler
4. Differentiate between lexeme, token and pattern
5. What are the issues in lexical analysis?
6. Explain in detail about the role of lexical analyzer with the possible error recovery actions
7. Elaborate the specification of tokens
8. Explain the role Lexical Analyzer and issues of Lexical Analyzer
9. (i)What are the issues in lexical analysis?
10. (ii)Elaborate in detail the recognition of tokens
11. Explain briefly about input buffering in reading the source program for finding the tokens
12. For the R.E. (a/b)*a(a/b). Draw the NFA. Obtain DFA form NFA. Minimize DFA using new construction. Write down the algorithm wherever necessary.
    a) (a/b)* abb
    b) (a/b)* a (a/b).
    c) (a/b)* a (a/b).
    d) (a/b)* a(a/b) (a/b)
    e) (a/b)* abb (a/b)*

13. Explain in detail about the role of lexical analyzer with the possible error recovery actions.
14. Construct a DFA directly from the regular expression (a|b)*abb without constructing NFA.
15. What are Lex and Lex specification ? How lexical analyzer is constructed using lex? Write a Lex program that recognizer the tokens
16. a)Construct NFA for (a/b)* a and convert into DFA
b)Differentiate between lexeme, token and pattern.
c)What are the issues in lexical analysis?