USN ☐☐☐☐☐☐☐☐☐☐

CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 1 – March 2025

Sub:	**Compiler Design**					Sub Code:	BCS613C	Branch:	CSE
Date:	26.03.2025	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VI/ A, B, C		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	a)Describe the science of building a compiler.	[6]	CO1	L2

* Modeling in Compiler Design and Implementation
*The Science of Code Optimization
* design objectives

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 for the compilers, while balancing the need of generality and power against simplicity and efficiency.
Some of most fundamental models are

- Finite-state machines and regular expressions: These models are useful for describing the lexical units of programs (keywords, identifiers etc) and for describing the algorithms used by the compiler to recognize those units.
- Context-free grammars: used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs.
- 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 compiler makes to produce code that is more efficient than the normal code. But 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. It is more important because massively parallel computers require substantial optimization, or their performance suffers largely.

The use of mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers.

Compiler optimizations must meet the following design objectives:

- **The optimization must be correct**: that is, the compiler should preserve the meaning of the compiled program: No optimizing compiler is completely error-free. Thus, the most important objective in writing a compiler is that it is correct.

- **The optimization must improve the performance of many programs**: The compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

- **The compilation time must be kept reasonable**: we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as machines get faster. Often, a program is first developed and debugged without program optimizations. This reduces compilation time.

- **The engineering effort required must be manageable**: Finally, a compiler is a complex system; we must keep the system simple so that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes huge amount of effort to create a correct and effective optimization. We must prioritize the optimizations and implement only those that lead to the greatest benefits on source programs.

b)Illustrate the concept of passes in compiler design

[4] CO1 L3

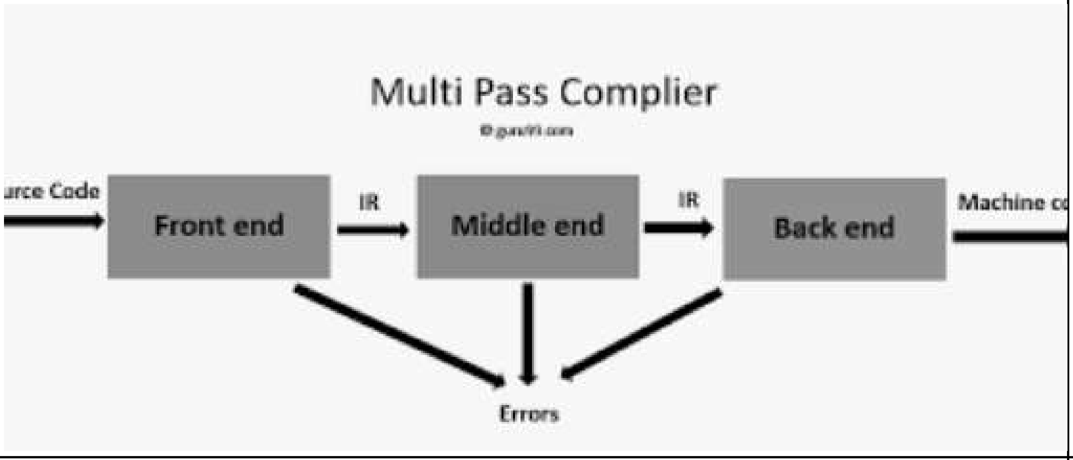
*Single pass

*Multi pass

The phases are the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.

For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Compilers for different source language which are based on well designed intermediate representations can be produced for the same target machine by allowing the frontend of a particular language to interact with the backend of the target machines. Similarly, we can produce compilers for different target machines, by combining a front end of the compiler with the back ends of different target machines.



2

a)Describe the applications of compiler technology.

[5] CO1 L2

*Implementation of High-Level Programming Languages -
*Object orientation
*Optimizations for Computer Architectures
*Parallelism
*Memory Hierarchies
*Design of New Computer Architectures

Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.

Generally, higher-level programming languages are easier to program, but are less efficient, that is, the target programs run more slowly.

Programmers using a low-level language have more control over a computation and can, produce more efficient code. Unfortunately, lower-level programs are harder to write, less portable, more prone to errors, and harder to maintain.

Example : The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution.

When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers, for which the **register** keyword was used This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature. In fact, programs that use the register keyword may lose efficiency, because programmers cannot make the best decision on low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specific machine architecture. Hardwiring low- level resource-management decisions

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction.

C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization.

- They support user-defined aggregate data types, such as arrays and structures
- high-level control flow, such as loops and procedure invocations.

If we take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as **data-flow optimizations**, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

Both these features make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called methods in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages.

- ➢ The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type.

- All array accesses are checked to ensure that they lie within the bounds of the array.
- Java has no pointers and does not allow pointer arithmetic.
- It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use.

While all these features make programming easier, they

include a run-time overhead. Compiler optimizations

have been developed to reduce the overhead, for

example,

- Eliminating unnecessary range checks
- Allocating objects that are not accessible beyond a procedure on the stack instead of the heap.
- Effective algorithms also have been developed to minimize the overhead of garbage collection.

Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques:

- **parallelism** : Parallelism can be found at several levels: at the instruction level, where multiple operations are executed simultaneously and at the processor level, where different threads of the same application are run on different processors.
- **Memory hierarchies**: Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence. In some cases, the machine includes a hardware scheduler that can change the order of the instruction so that parallelism can be increased. In the program the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible.. Whether the hardware reorders the instructions or not compilers can rearrange the instructions to make instruction-level parallelism more effective. Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue multiple operations in parallel. Intel **IA64** is a well-known example of such an architecture.

All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Memory Hierarchies

A memory hierarchy consists of several levels of storage with different **speeds and sizes**, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be used effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines**.** A processor usually has a small number of registers consisting of hundreds of bytes **smallest in size**, several levels of caches containing kilobytes to megabytes,

physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond have the maximum size.

It is possible to improve the effectiveness of the memory hierarchy by

- Changing the layout of the data
- Changing the order of instructions accessing the data.
- Changing the layout of code to make instruction caching more effective

Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. This has changed. Since programming in highlevel languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier. These architectures were known as CISC

(Complex Instruction-Set Computer).

Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include

- Data flow machines: (flow is based on the availability of data the instruction that all the data available is executed
- Vector machines(Single instruction operates on multiple data simultaneously)
- VLIW (Very Long Instruction Word) :machines: makes use of instruction parallelism
- SIMD (Single Instruction, Multiple Data) arrays of processors,
- Systolic arrays (processing units arranged like a matrix and are called cells. each cell shares its information with its cell, operations are triggered when the data arrives
- Multiprocessors with shared memory, and
- Multiprocessors with distributed memory.

The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology. Since entire systems can fit on a single chip, the focus is now on application specific processors. Application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

b) Discuss about language processors with block diagrams.	[5]	CO1	L2

Compiler

Interpreter

Hybrid compiler

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 An important role of the compiler is to report any errors in the source program that it detects during the translation process.

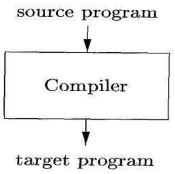


Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

Figure 1.2: Running the target program

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

Interpreter

An interpreter executes the source program statement by statement. It give better error diagnostics than a compiler.

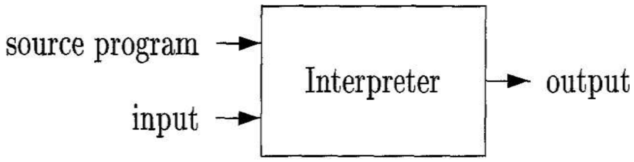


Figure 1.3: An interpreter

Example : Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of

inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

1.1. LANGUAGE PROCESSORS

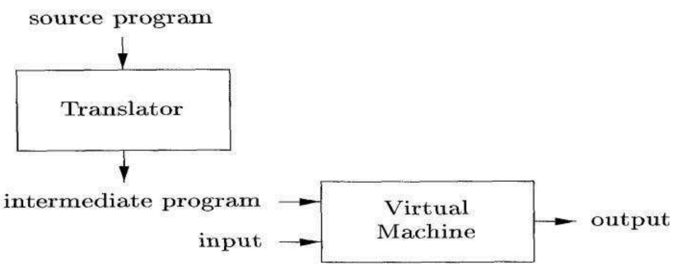
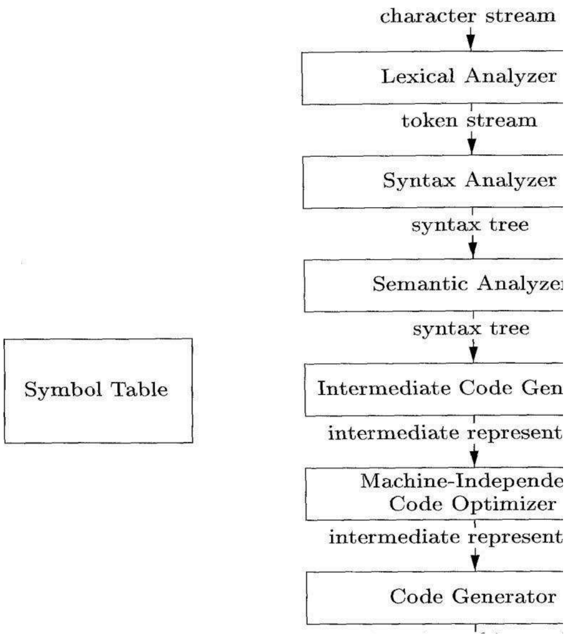


Figure 1.4: A hybrid compiler

3	Illustrate the importance of each phase of a compiler with pi = bp-fp* 60. A compiler maps a source program into a semantically equivalent target program. There are two parts to this mapping: analysis and synthesis. **Analysis part:** It breaks up the source program into components 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 or semantically not correct, 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**: It 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 and the synthesis part is called 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 Fig. 1.6. 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 Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing. 	[10]	CO1	L3

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 as output a token of the form (token-name, attribute-value) 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.

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.

Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check if the source program is semantically correct. It also gathers type information and saves it in either the syntax tree or the symbol table, to be used during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

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. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. 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.

Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. It could be improved in terms of faster code, or other objectives like shorter code, or target code that consumes less power.

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.

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. The symbol table is a data structure containing a record for each variable name. It contains fields that stores 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.

The attributes in the symbol table may provide information about

- The storage allocated for a name
- The type associated with a variable name
- The scope of a name (where in the program its value may be used)
- In the case of procedure names, it stores the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

4

a) Explain the concept of input buffering in lexical analyzer..

[5] CO2 L2

In the process of reading the character the lexical analyzer has to look one or more characters beyond the next lexeme before we can be sure of the right lexeme.

For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id.** In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, a two-buffer scheme is used, that handles large lookaheads safely.

Buffer Pairs

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

	E			M	*	c	*	*	2	eof	

Figure 3.3 using a pair of input buffers

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

Two pointers to the input are maintained:

1. Pointer lexemeBegin, 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 right end of the lexeme. For example In Fig. 3.3, we see forward has passed the end of the next lexeme, ** (the Fortran exponentiation operator),

and must be retracted one position to its left.The lexeme is recorded as an attribute value of a token. After the token is returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. we shall never overwrite the lexeme in its buffer before determining it.

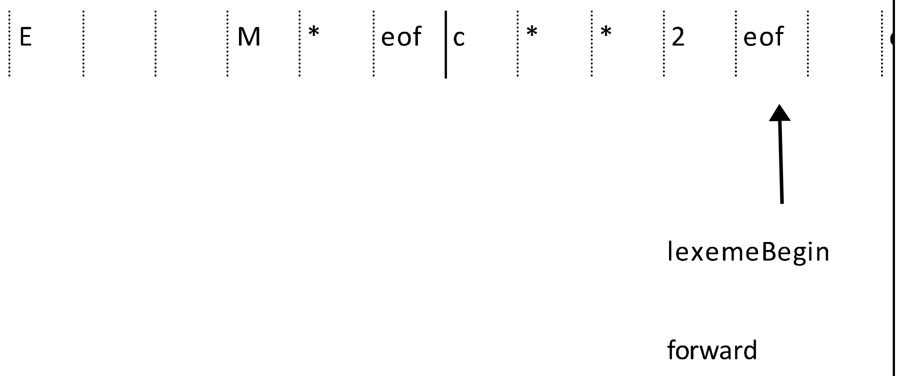
Sentinels

In the buffer pair scheme each time we advance forward ie, for each character read, we make two tests:

- For the end of the buffer, and
- To determine what character is read.

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof.**

Figure **3.4** shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end. The following algorithm shows how forward is advanced.



Buffer with sentinels at the end

b) Demonstrate the process of specification of tokens in lexical analyzer.	[5]	CO2	L3

An **alphabet** is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1) is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems.

.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string."

The length of a string i*s,* usually written |s | is the number of occurrences of symbols in s.

For example, banana is a string of length six. The empty string, denoted ∈ **,** is the string of length zero.

A **language** is any countable set of strings over some fixed alphabet.

Abstract languages like **Ø,** the empty set ∈, or , the set containing only the empty string, are included under this definition. The set of all syntactically well-formed C programs are examples of language.

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.

The empty string is the identity under concatenation; that is, for any string s, ∈S = S∈ = s.

If we think of concatenation as a product, we can define the 'exponentiation" of strings as follows.

Define s^o to be ∈, and

for all *i* > 0, define s^i to be $s^{i-1}s$.

Since ∈S = S, it follows that s^1 = s. Then s^2 = ss, s3 = sss, and so on.

Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure.

Operations	Definition and notations
Union of L and M	L U M={s| s is in L or s is in
Concatenation of L and M	LM={st| s is in L and t is in M
Kleene closure of L	$L^{*}= U^{\infty}_{i=0} L^{i}$
Positive cldure of L	$L+=U^{\infty}_{i=1} Li$

Union is the string taken from either of the languages.
The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.
The (Kleene) closure of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times. Note that L^o, the "concatenation of L zero times,"

Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term L^o. That is,

∈ will not be in L+.

Example 3.3 : Let L be the set of letters **{A, B,** . . . , Z, **a, b,** . . . , *z)* and let *D* be the set of digits **{0,1,**.. .9). *L* and *D* are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that *L* and *D* are languages, all of whose strings happen to be of length one.

Here are some other languages that can be constructed from languages *L* and *D,* using the operators of Fig. 3.6:

1. L *U D* is the set of letters and digits - the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. *LD* is the set df 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. *L** is the set of ail strings of letters, including e, the empty string.
5. *L(L U D)** is the set of all strings of letters and digits beginning with a letter.
6. *D+* is the set of all strings of one or more digits.

5	a) write the regular definition for **identifier** and **number.**	[5]	CO2	L2

Regular Definitions

For notational convenience, we can give names to certain regular expressions and use those names in subsequent expressions and they are called as regular definitions.

If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form where:

Each di is a new symbol, not in C and not the same as any other of the d's, and

Each ri is a regular expression over the alphabet r Σ {dl, d2,. . . , di-l).

By restricting ri to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone.

The regualar definitions of identifiers can be rewritten as

letter_ ->[A-Za-z_] digit->[0-9]

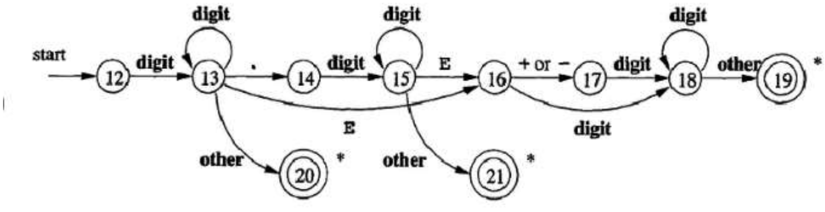
id->letter_(letter|digit)*

The regular Definition of numbers can be rewritten as

digit->[0-9] digits->digit+

number->digits(.digits)?(E[+-]? digits)?

b) Demonstrate the **unsigned numbers** with the help of a transition diagram.



[5]

CO2

L3

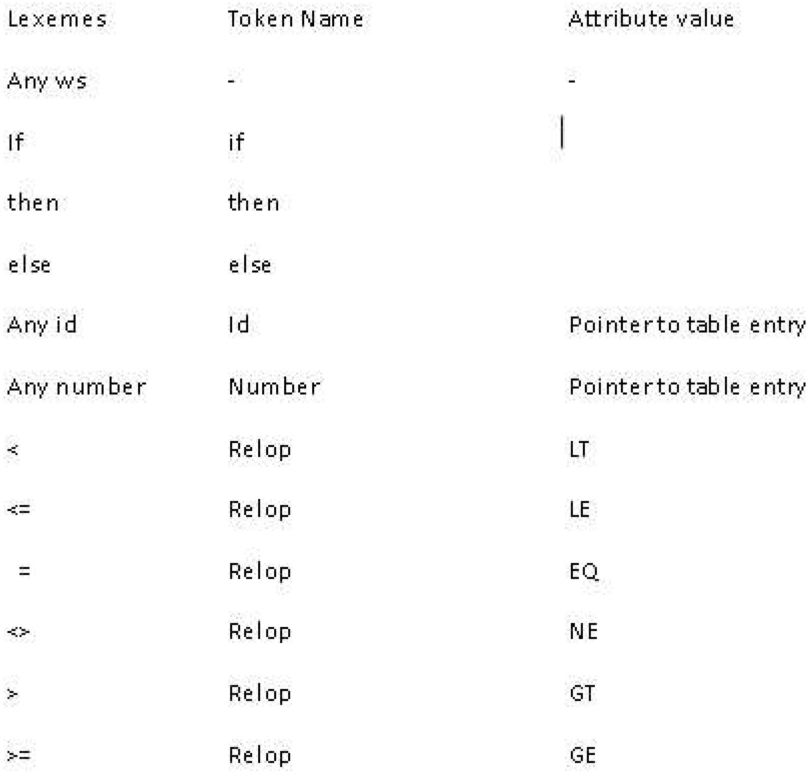
There are two ways to handle reserved words that look like identifiers:

Install the reserved words in the symbol table initially:
A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to installID places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function getToken examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme

	represents - either id or one of the keyword tokens that was initially installed in the table. Create separate transition diagrams for each keyword: an example for the keyword then is shown in Fig. Such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id,For example when there is a lexeme like thenextvalue that has then as a proper prefix it should return the token as then here the token is an id. If this approach is used, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.			
6	a) Explain Recognition of tokens with an example. Patterns are Expressed using regular expressions. The patterns of all needed tokens have to to build a piece of code that matches the input string and finds a lexeme that matches a pattern Consider the Grammar Given below stmt->if expr then stmt | if expr then stmt else stmt |∈ expr-> term relop term | term term ->id | number Grammar for branching statements The grammar given above describes a simple form of branching statements and conditional expressions. The patterns for these tokens are described using regular definitions, as follows. Digit->[0-9] digits-> digit+ number-> digits (.digits)? (E[+-]? digits)? letter->[A-Za-z] id->letter(letter|digit)* if->if then->then else->else relop-> < |> |<= |>= |= |<> for this language, the lexical analyzer will recognize the keywords if , then, and else, as well as lexemes that match the patterns for relop, id, and number. in addition, the lexical analyzer have to strip out whitespace, by recognizing the "token" ws defined below 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	[5]	CO2	L2

recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.

Fig. 3.12.shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value, For the six relational operators, symbolic constants LT, LE, and so on are used as the attribute value, in order to indicate the instance of the token relop that was found.



Lexemes	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	EQ
<>	Relop	NE
>	Relop	GT
>=	Relop	GE

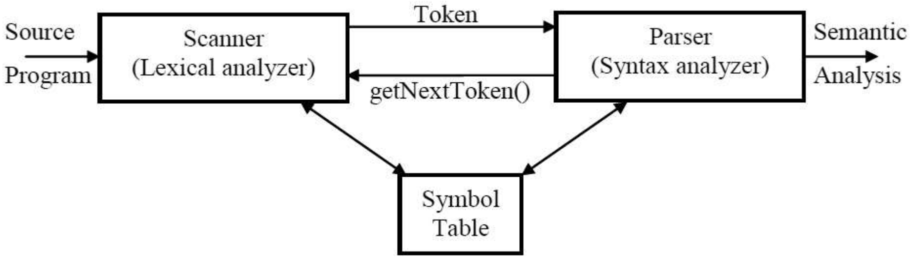
Figure 3.10 Tokens their patterns and attribute values

Transition Diagrams

Transition diagram is the intermediate step in the construction of a lexical analyzer. Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning . A state is a summary of all we need to know about what characters we have seen between the lexemeBegin pointer and the forward pointer.

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in some states, and the next input symbol is a, we look for an edge out of state s labeled by a. If we find such an edge, we advance the forward pointer to enter the state of the transition diagram to which that edge leads. All the transition diagrams are deterministic.

Some important conventions about transition diagrams are:

	Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may consist of all positions between the LexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state. In addition, if it is necessary to retract the forward pointer one position then we shall additionally place a * near that accepting state. If is necessary to retract forward by more than one position, that many number of *'s have to be attached to the accepting state. One state is designated the start state, or initial state; it is indicated by an edge, labeled "start ," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read. The transition diagram to recognize the lexemes that match the token relop . The relational operators that are considered are <,<= <>,=, >, >=			
	b) Explain the role of lexical analyzer. As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. The lexical analyzer interacts with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser. These interactions are given in Fig. 3.1. The interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.  Interaction between the Lexical analyzer and the parser	[5]	CO2	L2

Since the lexical analyzer is the part of the compiler that reads the source text, it performs certain other tasks like

Stripping out comments and whitespace (blank, newline, tab, and other characters that are used to separate tokens in the input).

Correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.

If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Lexical analyzers are divided into a cascade of two processes:

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.

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

Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

Simplicity of design: It is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

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

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

Tokens, Patterns, and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the

	keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings. A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.			

CI CCI HOD

CO-PO Mapping		**Blooms Level**	**Modules covered**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**	**PSO4**
CO 1	Understand the different phases of compiler design techniques .	**L2**	**1**	3	3	2	3	0	0	0	0	0	0	0	2	0	0	2	2
CO 2	Analyze the working of lexical Analyser in design of compilers.	**L4**	**2**	3	3	2	3	0	0	0	0	0	0	0	2	0	0	2	2
CO 3	Design syntax analyser using top down and bottom up approaches.	**L6**	**3**	3	3	2	3	0	0	0	0	0	0	0	2	0	0	2	2
CO 4	Illustrate syntax directed translation for a given grammar.	**L3**	**4**	3	3	2	3	0	0	0	0	0	0	0	2	0	0	2	2
CO 5	Explain intermediate code representation and code generation of compilers.	**L2**	**5**	3	3	2	3	0	0	0	0	0	0	0	2	0	0	2	2

COGNITIVE LEVEL	REVISED BLOOMS TAXONOMY KEYWORDS

L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PROGRAM OUTCOMES (PO), PROGRAM SPECIFIC OUTCOMES (PSO)				CORRELATION LEVELS	
PO1	Engineering knowledge	PO7	Environment and sustainability	0	No Correlation
PO2	Problem analysis	PO8	Ethics	1	Slight/Low
PO3	Design/development of solutions	PO9	Individual and team work	2	Moderate/ Medium
PO4	Conduct investigations of complex problems	PO10	Communication	3	Substantial/ High
PO5	Modern tool usage	PO11	Project management and finance		
PO6	The Engineer and society	PO12	Life-long learning		
PSO1	Develop applications using different stacks of web and programming technologies				
PSO2	Design and develop secure, parallel, distributed, networked, and digital systems				
PSO3	Apply software engineering methods to design, develop, test and manage software systems.				
PSO4	Develop intelligent applications for business and industry				