Compiler Design

Subject Code: 6CS63/06IS662

Part – A

UNIT – 1

Chapter 1

1. Introduction

1.1 Language Processors

A compiler is a program that can read a program in one language (source language) and translate it into an equivalent program in another language (target language). 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.

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.

A language-processing system typically involves – preprocessor, compiler, assembler and linker/loader – in translating source program to target machine code.

1.2 The structure of a Compiler

Analysis: source program to intermediate representation (front end)
Synthesis: intermediate representation to target program (back end)

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 phases of a compiler are: lexical analyzer (scanning) (linear analysis), syntax analyzer (hierarchical analysis) (parsing), semantic analyzer, intermediate code generator, machine-independent code optimizer, code generator and machine-dependent code optimizer

Symbol table manager and error handler are two independent modules which will interact with all phases of compilation. A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. Each phase can encounter errors. After detecting an error, a phase must somehow deal with that error, so that compilation must proceed, allowing further errors in the source program to be detected.

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.

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.

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.

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.

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.

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. Different compiler construction tools are: parser generators, scanner generators, syntax-directed translation engines, code-generator generators, data-flow analysis engines, compiler-construction toolkits.

1.3 The Evolution of Programming Languages

The move to higher-level languages
The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

Impacts on compilers
Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

1.4 The Science of Building a Compiler

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

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

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 obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

Finally, a compiler is a complex system; we must keep the system simple to assure 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 a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

1.5 Applications of Compiler Technology

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 in, 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, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Optimizations for computer architectures
The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies.* 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 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; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

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 harnessed effectively by the compiler to deliver real performance on an application.

Design of new computer architectures
In the early days of computer architecture design, compilers were developed after the machines were built. That 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.

Program translations
While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages.

Software productivity tools
Programs are arguably the most complicated engineering artifacts ever produced; they consist of many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs. An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Dataflow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

1.6 Programming Language Basics

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

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

Static scope and block structure
Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C + + , Java, and C # , also provide explicit control over scopes through the use of keywords like **public, private,** and **protected.** In this section we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces {and} to delimit a block; the alternative use of **b e g in** and **end** for the same purpose dates back to *Algol*.

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

Dynamic scope
Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called

procedure with such a declaration. Dynamic scoping of this type appears only in special situations.

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

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

Chapter 3

Lexical Analysis

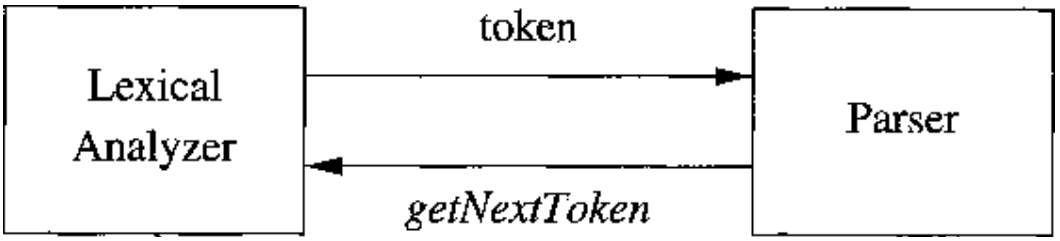
Lexical analysis reads characters from left to right and groups into tokens. A simple way to build lexical analyzer is to construct a diagram to illustrate the structure of tokens of the source program. We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a *lexical-analyzer generator* and compiling those patterns into code that functions as a lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patterns, not the entire program.

Three general approaches for implementing lexical analyzer are:

i. Use lexical analyzer generator (LEX) from a regular expression based specification that provides routines for reading and buffering the input.
ii. Write lexical analyzer in conventional language using I/O facilities to read input.
iii. Write lexical analyzer in assembly language and explicitly manage the reading of input.

The speed of lexical analysis is a concern in compiler design, since only this phase reads the source program character-by character.

3.1 The role of the lexical analyzer



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is 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.

It is the first phase of a compiler. It reads source code as input and sequence of tokens as output. This will be used as input by the parser in syntax analysis. Upon receiving 'getNextToken' from parser, lexical analyzer searches for the next token.

Some additional tasks are: eliminating comments, blanks, tab and newline characters, providing line numbers associated with error messages and making a copy of the source program with error messages.

Some of the issues are: simpler design, compiler efficiency is improved and compiler

portability is enhanced.

Tokens, patterns and lexemes

Token is a terminal symbol in the grammar for the source language. When the character sequence 'pi' appears in the source program, a token representing identifier is returned to the parser. A *token* is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

Pattern is a rule describing the set of lexemes that can represent a particular token in source programs. 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.

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

In many programming languages, the following classes cover most or all of the tokens:
1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the1 operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for tokens

A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept. The token names and associated attribute values for the statement

E = M * C ** 2

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>

Lexical errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error. For instance, if the string f i is encountered for the first time in a C

program in the context:

f i (a == f (x)) . ..

a lexical analyzer cannot tell whether f i is a misspelling of the keyword if or an undeclared function identifier. Since f i is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.
However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:
1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

3.2 Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

Look ahead of characters in the input is necessary to identify tokens. Specified buffering techniques have been developed to reduce the large amount of time consumed in moving characters. A buffer (array) divided into two N-character halves, where N=number of characters on one disk block ‘eof’ marks the end of source file and it is different from input character

E=M*C**2eof

Two pointers are maintained: beginning of the lexeme pointer and
forward pointer.

Initially, both pointers point to the first character of the next lexeme to be found. Forward pointer scans ahead until a match for a pattern is found. Once the next lexeme is determined, processed and both pointers are set to the character immediately past the lexeme. If the forward pointer moves halfway mark, the right N half is filled with new characters. If the

forward pointer moves right end of the buffer then left N half is filled with new characters. The disadvantage is look ahead is limited and it is impossible to recognize tokens, when distance between the two pointers is more than the length of the buffer.

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

Sentinels

It is an extra key inserted at the end of the array. It is a special, dummy character that can't be part of source program. With respect to buffer pairs, the code for advancing forward pointer is:

```
If forward is at the end of first half then begin reload
   second half
   forward=forward+1
end
elseif forward is at the end of second half then begin
   reload first half
   move forward to beginning of first half
     end
else
   forward=forward+1
```

Instead of this, we provide an extra character, sentinel at the end of each half of the buffer. Sentinel is not part of our source program and works as 'eof'. Now only one test is sufficient, that is, if forward='eof' or not.

3.3 Specification of tokens

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.*

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 *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 e, is the string of length zero.

The following string-related terms are commonly used:

1. A *prefix* 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.

String (S): Sentence/word: finite set/sequence of symbols
|S|=number of symbols in string S
eg: S=banana, then |S|=6
Prefix(S): S=banana, ban
Suffix(S): S=banana, nana
Substring(S): S=banana, ba,na,na
Є:empty string, S= Є then |S|=0, Ø=empty set
Language: set of strings, L
If x and y are strings then xy is concatenation
SЄ= ЄS=S
S0= Є
S1=S
S2=SS
S3=SSS
Si=Si-1S (if i>0)

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like 0, the *empty set,* or {e}, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.

Operations on languages

LUD: Union operation, where L=set of alphabets
{A..Z,a..z} and D=set of digits {0..9}
LD: Concatenation
L4:exponentiation: set of strings with 4 letters
L0= Є
Li=Li-1L
L*=all strings with Є: Kleene closure of L
D+: set of all strings of digits of one or more

1. *L* U *D* is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. *LD* is the set of strings of length two, each consisting of one letter followed by one digit.
3. L4 is the set of all 4-letter strings.
4. *L** is the set of all 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.

Regular Expressions

It is a notation which allows defining the sets precisely.
eg: L(LUD)* its regular expression is:

letter(letter|digit)*

Regular expression over alphabet ∑ has following rules:

Є is a regular expression, the set containing empty string a is a regular expression, if a belongs to ∑ that is {a} set containing the string a. Suppose r and s are regular expression denoting the languages L(r) and L(s) then,

(r|s) is a regular expression denoting L(r)UL(s)

rs is a regular expression denoting L(r)L(s)

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

(r) is a regular expression denoting L(r)

A language denoted by regular expression is said to be a regular set *, concatenation and | has highest to lowest precedence with left associative

If two regular expression 'r' and 's' denote the same language, we say 'r' and 's' are equivalent and write r=s

BASIS: There are two rules that form the basis:

1. e is a regular expression, and *L(e)* is {e}, that is, the language whose sole member is the empty string.
2. If *a* 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.

Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.1

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

1. (r)|(s) is a regular expression denoting the language *L(r)* U *L(s).*
2. (r)(s) is a regular expression denoting the language *L(r)L(s).*
3. (r)* is a regular expression denoting (L(r))*.
4. (r) is a regular expression denoting *L(r).* This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

The following table shows some of the regular expressions along with there possible regular sets:

Regular expression	**set**
a|b	{a,b}
(a|b)(a|b) or aa|ab|ba|bb	{aa,ab,ba,bb}
a*	{Є,a,aa,aaa,…}
(a|b)* or (a*b*)*	{Є, a,aa,b,bb,…}
a|a*b	{a,b,ab,aab,aaab,…}

Algebraic properties

- r|s=s|r
- r|(s|t)=(r|s)|t
- (rs)t=r(st)
- r(s|t)=rs|rt
- (s|t)r=sr|tr

- Єr=r
- r Є=r
- r*=(r| Є)*
- r**=r*

Regular definition

It is a sequence of definitions of the form
d1→r1
d2→r2 and so on
where di is distinct name and ri is regular expression
For example,
letter→A|B|..|Z|a|b|..|z
digit→0|1|..|9
id→letter(letter|digit)*

Extensions of regular expressions (Notational shorthands)

One or more instances: unary postfix operator '+'
eg: if 'r' is a regular expression denoting the language L(r) then
(r)+ is a regular expression denoting the language (L(r))+
a+ is a regular expression of set of all strings of one or more a's
r*=r+| Є
r+=rr*
Zero or one instance :unary postfix operator '?'
eg: r? means r|Є
If 'r' is a regular expression denoting the language L(r) then (r)? is a regular expression denoting the language L(r)U{Є}

Character class

[abc] denotes the regular expression a|b|c
[a-z] denotes the regular expression a|b|..|z

3.4 Recognition of tokens

In the previous section we learned how to express patterns using regular expressions. Now, we must study 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. Consider the following grammar, fragment/regular definitions:
stmt→if expr then stmt | if expr then stmt else stmt | Є
expr→term relop term | term
term→id | num
if → if
then → then
else → else
relop → <|<=|>|>=|=|<>
id → letter(letter|digit)*
num → digit+(.digit+)?(Є(+|-)?digit+)?
• Keywords cannot be used as identifiers

• Num represents unsigned int and real numbers
• The regular definition, ws,
delim → blank | tab | newline
ws → delim+

If 'ws' is found, the lexical analyzer does not return a token to the parser. Our goal is to construct a lexical analyzer to produce a pair consisting of token and attribute-value as output using the translation table

Transition diagrams

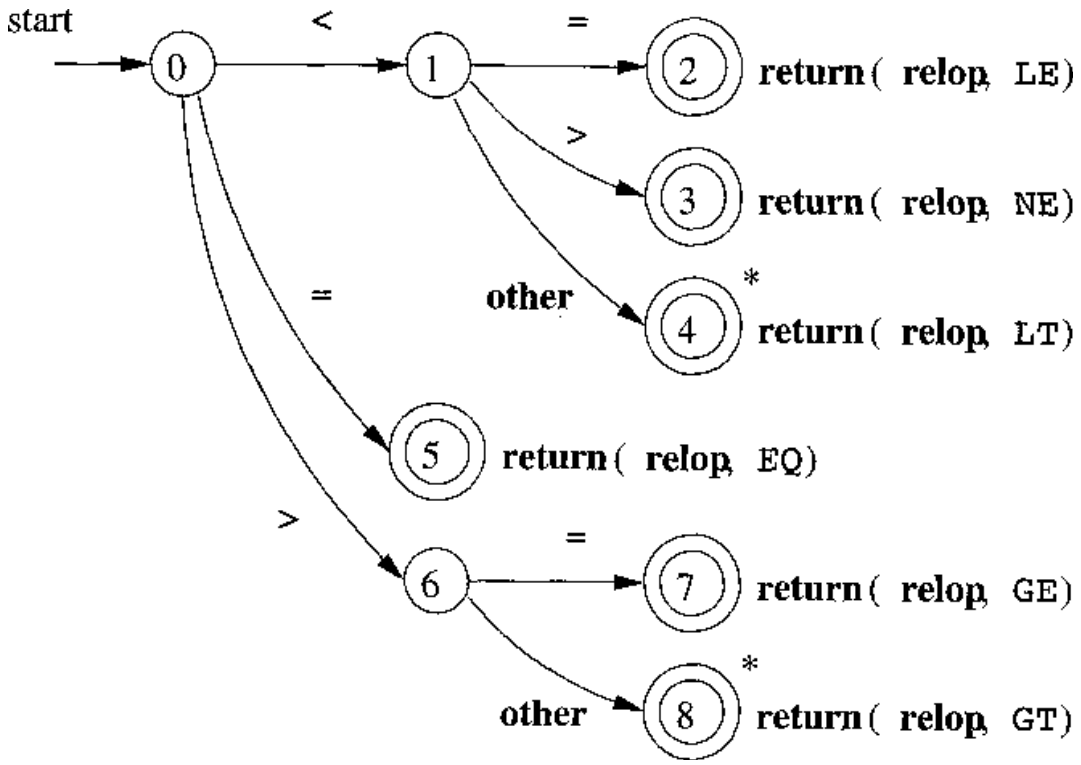
These are the flow charts, as an intermediate step in the construction of a lexical analyzer. This takes actions when a lexical analyzer is called by the parser to get the next token. We use transition diagram to keep track of information about characters that are seen as and when the forward pointer scans the input. Lexeme beginning pointer points to the character following the last lexeme found.

E=M* C**2eof

Transition diagrams have 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. We may think of a state as summarizing 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 state**,** and the next input symbol is *a,* we look for an edge out of state *s* labeled by *a* (and perhaps by other symbols, as well). If we find such an edge, we advance the *forward* pointer arid enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic,* meaning that there is never more than one edge out of a given state with a given symbol among its labels. We shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not 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.
2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.
3. 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.



Transition diagram for *relop*

```
TOKEN getRelopO
{
TOKEN retToken = new(RELOP);
while(1) { /* repeat character processing until a return
or failure occurs */
switch(state) {
case 0: c = nextCharQ;
if ( c == '<» ) state = 1;
else if ( c == '=' ) state = 5;
else if ( c == '>' ) state - 6;
else fail(); /* lexeme is not a relop */
break;
case 1: ...
case 8: retract();
retToken.attribute = GT;
return(retToken);
```

Implementation of *relop* transition diagram

Recognition of Reserved Words and Identifiers

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

1. 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. Of course, 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.

2. Create separate transition diagrams for each keyword; an example for the keyword **then**. Note that 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 **t h e n** in situations where the correct token was **id,** with a lexeme like t h e n e x t v a l u e that has **then** as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.

Architecture of a transition diagram based lexical analyzer

A sequence of transition diagrams can be converted into a program to look for the tokens. The size of the program is propositional to the number of states and edges in the diagrams. Each state gets a segment of code (module). Nextchar() is to read a next char from the input buffer. fail() calls an error recovery routine, when error is encountered. lexical_value returns tokens when id or num is found by install-id() and install_num() respectively. state&start keep track of the present and start state.

There are several ways that a collection of transition diagrams can be used to build a lexical analyzer. Regardless of the overall strategy, each state is represented by a piece of code. We may imagine a variable state holding the number of the current state for a transition diagram. A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

We could arrange for the transition diagrams for each token to be tried sequentially. Then, the function **fail()** resets the pointer **forward** and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords. We have to use these before we use the diagram for id, in order for the keywords to be reserved words.
We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier the next to keyword then, or the operator -> to -, for example.

The preferred approach is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern. This combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states into one start state, leaving other

transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex.