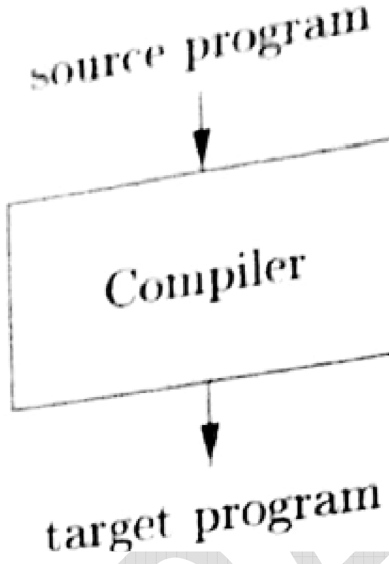
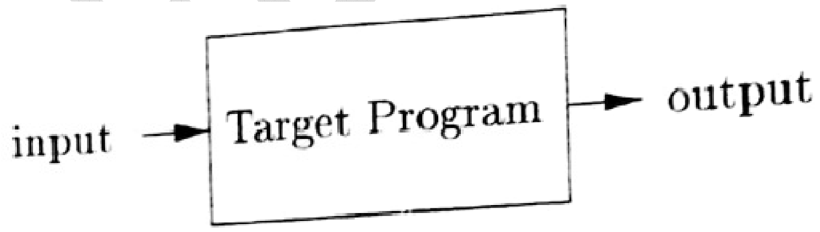
COMPILER DESIGN
MODULE-3

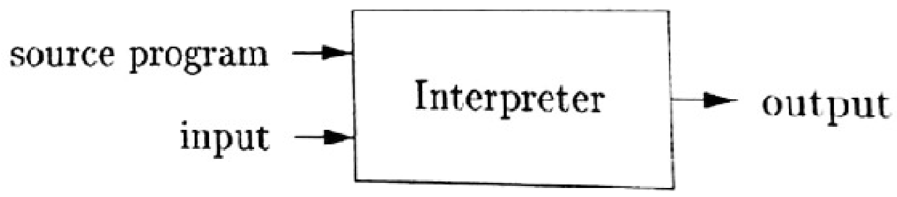
A compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program inanother language — the *target* language; An important role of the compiler is to report any errors in the source program that it detects during the translation process



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



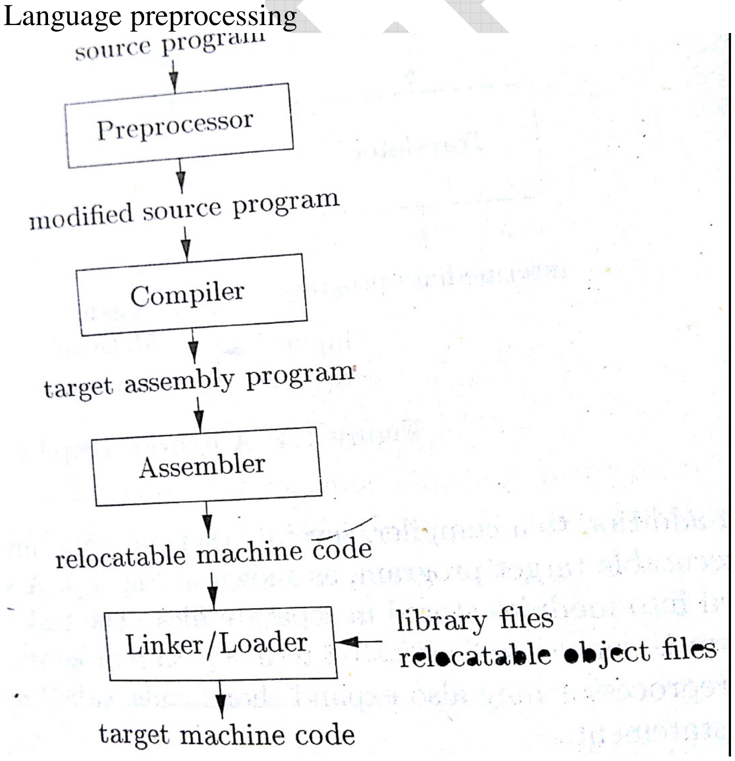
An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in



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

Compiler	Interpreter
A compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program inanother language — the *target* language	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	Interpreter is slower in mapping inputs to outputs
Error diagnostics is less than interpreter	An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement

Language preprocessing

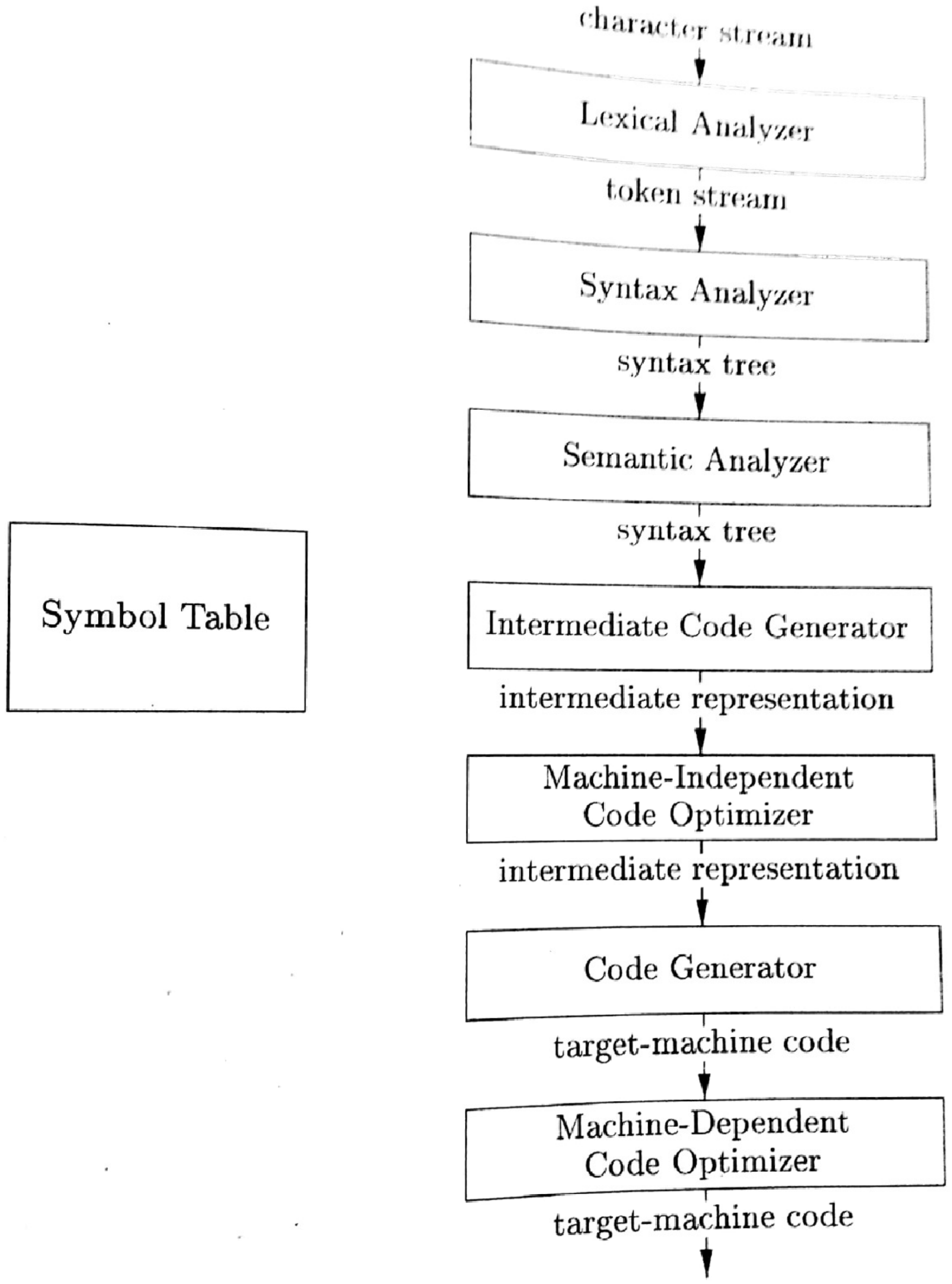


In addition to a compiler, several other programs may be required to create an executable target program, A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor.* The preprocessor may also expand shorthands, called macros, into source language statements. The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output. Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

The Structure of a Compiler

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

Phases of compiler



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 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.

For example, suppose a source program contains the assignment statement
p o s i t i o n = i n i t i a l + r a t e * 60
shows the representation of the assignment statement after lexical analysis as the sequence of tokens
(i d , l) (=) (id, 2) (+) **(id,** 3) (*) (60)

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 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,* 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. The language specification may permit some type conversions called *coercions.*

Intermediate Code Generation

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. 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 consists of the three-address code sequence. There are several points worth noting about three-address instructions. 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 .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.

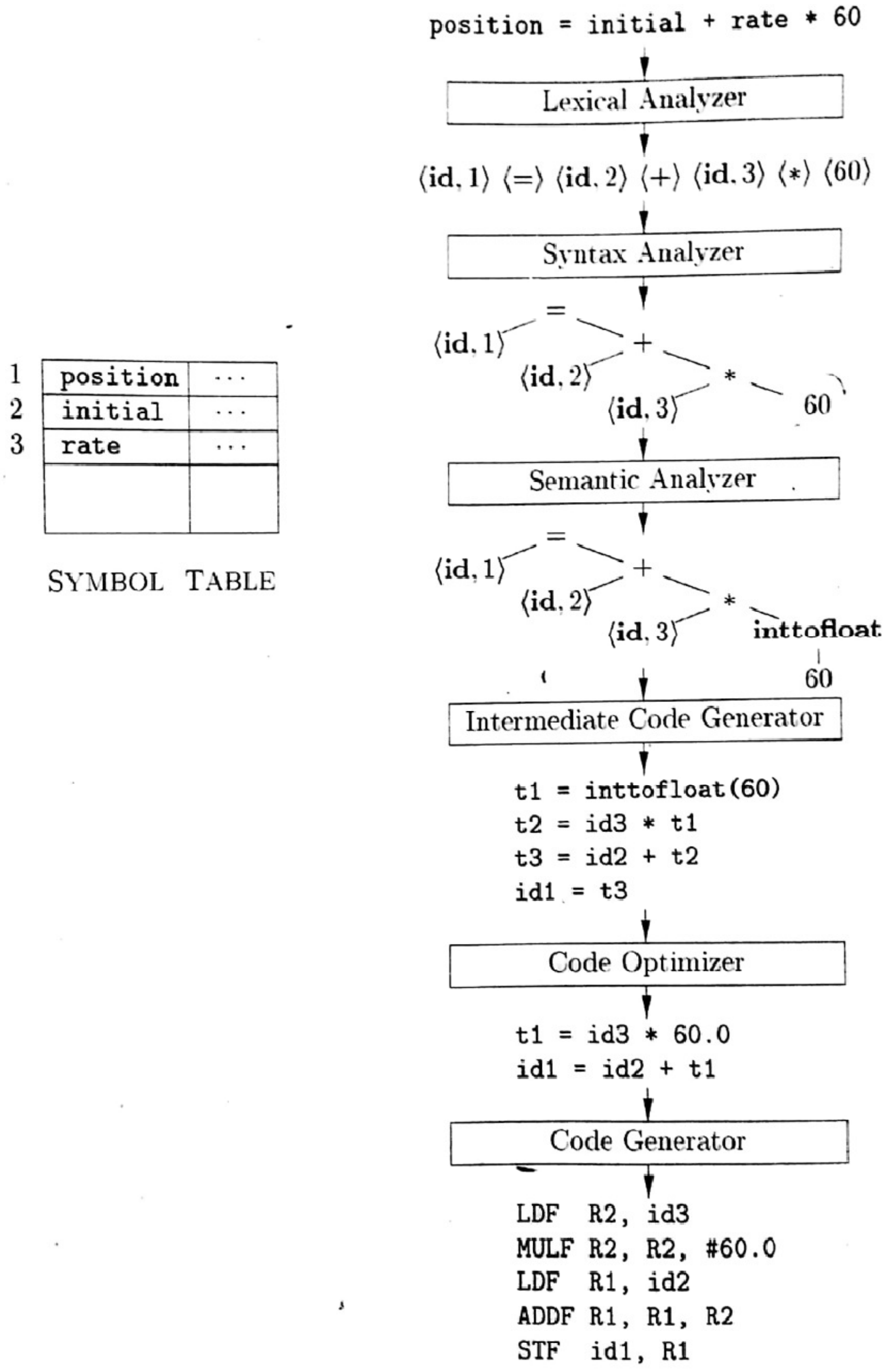
Code Optimization

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

```
t1 = id3 * 60.0
id1 = id2 + t1
```

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
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



1	position	···
2	initial	···
3	rate	···

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

```
t1 = id3 * 60.0
id1 = id2 + t1
```

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

The Grouping of Phases into Passes
The discussion of phases deals with 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 machinestore or retrieve data from that record quickly.

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.

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.
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.

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 lowlevel

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 byhow 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.

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 he 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:

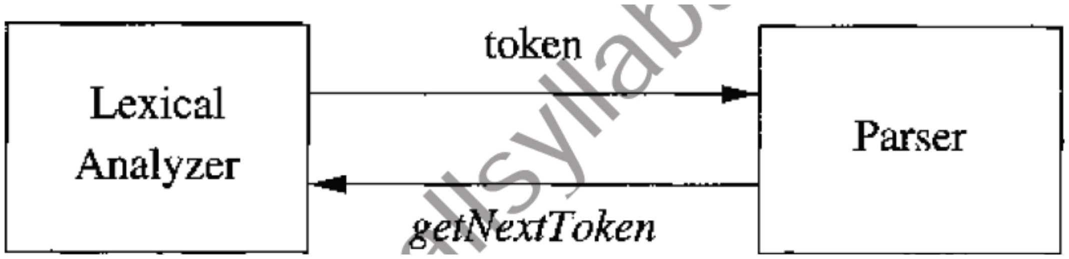
i. Use lexical analyzer generator (LEX) from a regular expression based specification that provides routines for reading and buffering the input.

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.

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.

Ex: Identify the lexeme and write the pattern and token for a=b*10

Lexemes are a,=,b,*,10

Pattern for identifiers =letter followed by letter or digit (or)

Letter(Letter|digit)*

Pattern for numbers=any digits 0-9 (or) digit+

Token for identifirs is <id, pointer to symbol table entry>

Token for numbers=<Num, Value>

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>

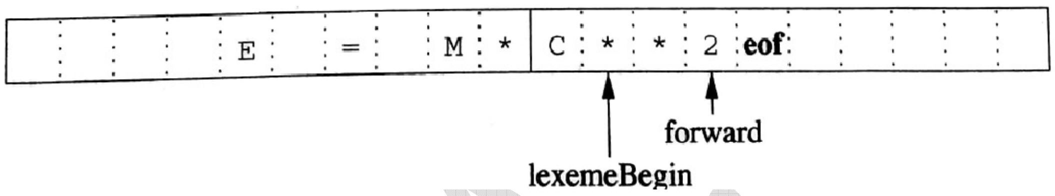
Lexical errors

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.

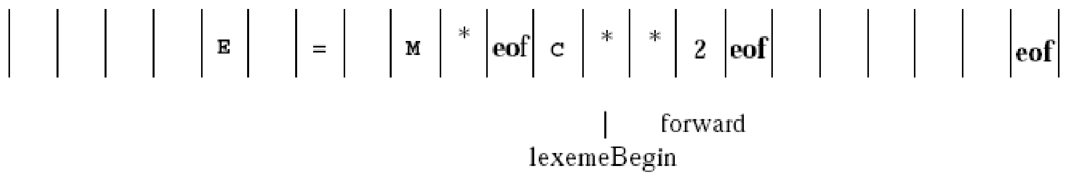
3.2 Input Buffering
Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme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

Lookahead code with sentinels

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

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."

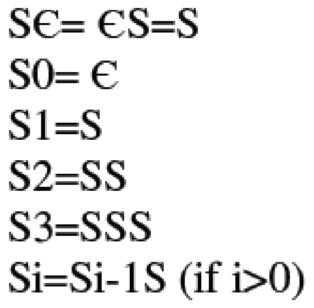
The length of a string *s,* usually written |s|, is the number of occurrences of symbols in *s.* For example, banana is a string of length six. The *empty string,* denoted e, is the string of length zero.

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.

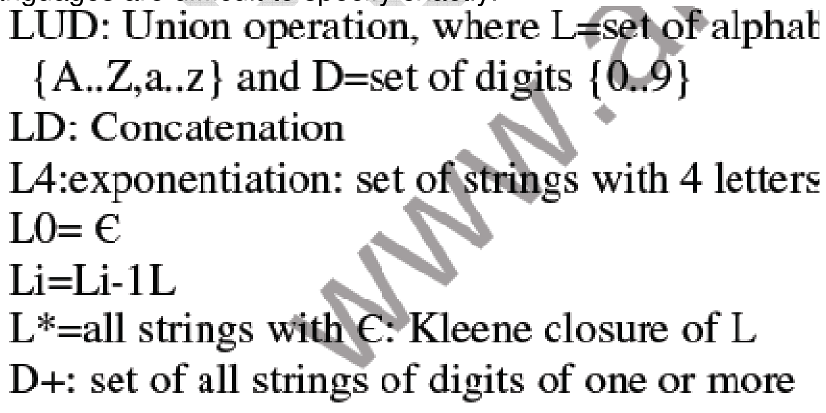
2. A *suffix* of string *s* is any string obtained by removing zero or more symbols from the

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
_:empty string, S= _ 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.



LUD: Union operation, where L=set of alphab
{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

Operations on languages
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:
_ 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*	{ ε,a,aa,aaa,…}
(a|b)* or (a*b*)*	{ ε, 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
• ε r=r
• r ε =r
• r*=(r| ε)*
• 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+| ε
r+=rr*
Zero or one instance :unary postfix operator '?'
eg: r? means r| ε
If 'r' is a regular expression denoting the language L(r) then (r)? is a regular expression denoting the language L(r)U{ε}

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 | _
expr->term relop term | term
term->id | num

if ->if
then -> then
else -> else
relop -> <|<=|>|>=|=|<>
letter->[a-zA-Z]
digit->[0-9]
id -> letter(letter|digit)*
digits->digit+
num ->digits(.digits)?(E(+|-)?digits)?
• 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

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**	

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 *labelled* 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:

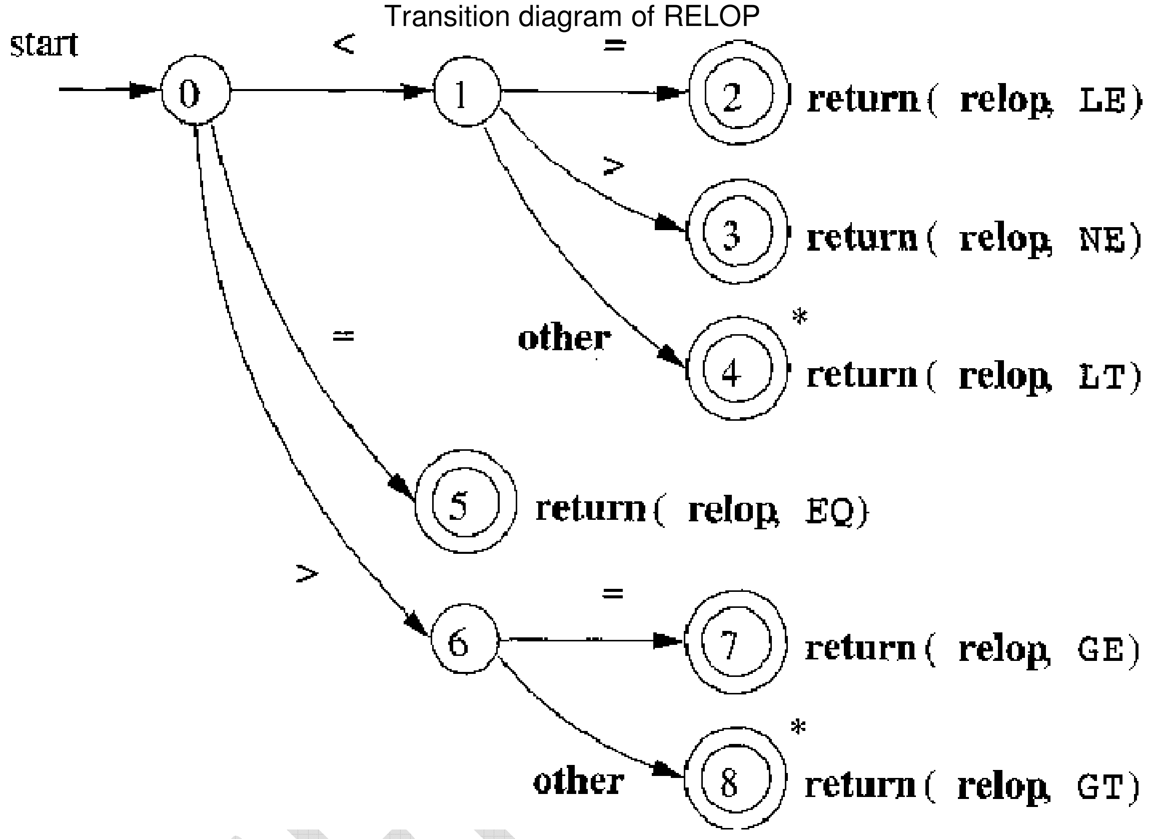
1. Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always indicate an accepting state by a double circle, and if there is an action 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 of RELOP



Transition diagram for *relop*

TOKEN getRelop()
{
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

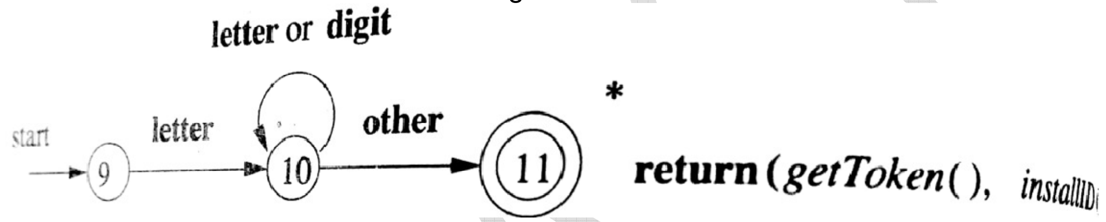
There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry

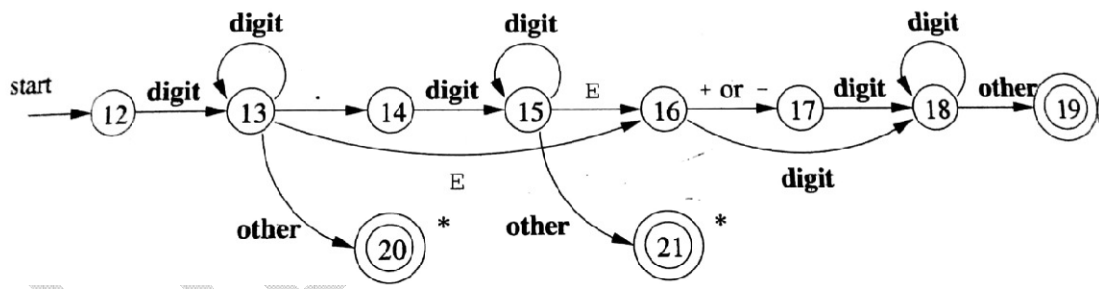
indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier 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.

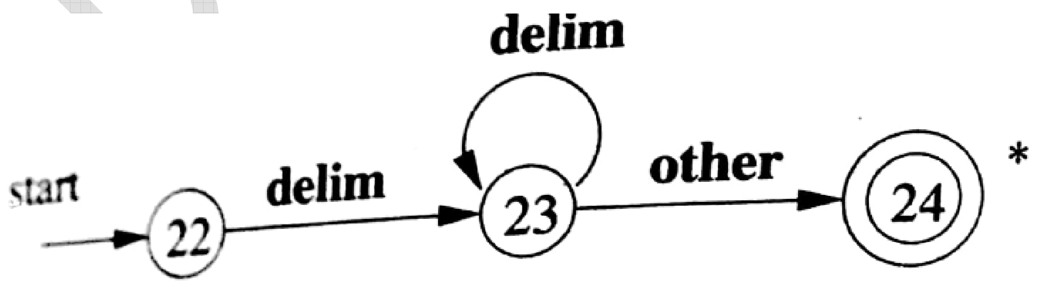
Transition diagram of identifier



Transition diagram of unsigned number



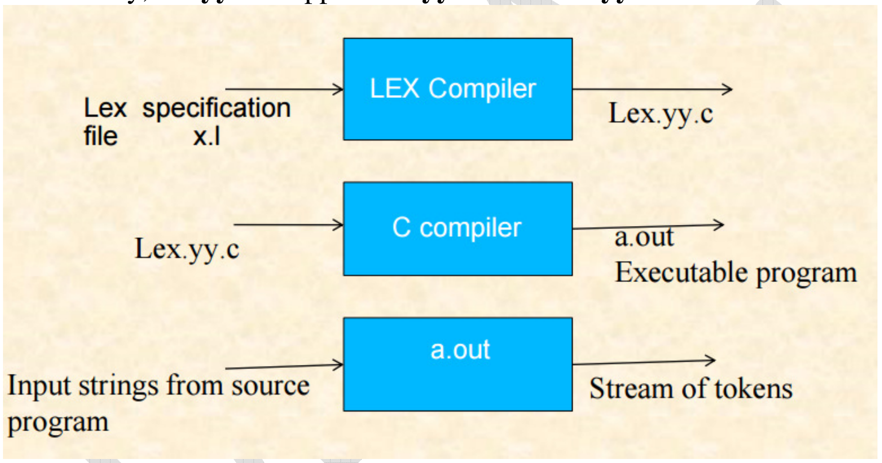
Transition diagram of white space



The Lexical-Analyzer Generator Lex

The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler*. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called l e x . y y . c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections; here we only learn the Lex language.

Use of Lex
An input file, which we call l e x . l , is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms l e x . l to a C program, in a file that is always named l e x . y y . c. The latter file is compiled by the C compiler into a file called
a . o u t , as always. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens. The normal use of the compiled C program, referred to as a. out. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable y y l v a l , 2 which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.
incidentally, the **yy** that appears in **yylval** and **lex.yy.c** refers to the **Yacc** parsergenerator,



Structure of Lex Programs
A **Lex** program has the following form: The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions,**.** The translation rules each have the form
Pattern { Action }
Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of **Lex** using other languages have been created.
The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer. The lexical

analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns *Pi.* It then executes the associated action *Ai.* Typically, *Ai* will return to the parser, but if it does not (e.g., because *Pi* describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable y y l v a l to pass additional information about the lexeme found, if needed.
The action taken when *id* is matched is threefold:
1. Function i n s t a l l l D ( ) is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable y y l v a l , where it can be used by the parser or a later component of the compiler. Note that i n s t a l l l D ( ) has available to it two variables that are set automatically by the lexical analyzer that Lex generates:
(a) yytext is a pointer to the beginning of the lexeme, analogous to lexemeBegin
(b) yyleng is the length of the lexeme found.
3. The token name ID is returned to the parser.

Conflict Resolution in Lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:
1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

The Lookahead Operator

Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

```
%{
    /* 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);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```