JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8602-COMPILER DESIGN

(Regulation 2017)

Year/Semester: III/06/ CSE

2020 – 2021

Prepared by

Mrs.R.Dayana

Assistant Professor/CSE

UNIT I INTRODUCTION TO COMPILERS 9

Structure of a compiler – Lexical Analysis – Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens – Lex – Finite Automata – Regular Expressions to Automata – Minimizing DFA.

INTRODUCTION TO COMPILERS

Translator:

It is a program that translates one language to another.

$$\begin{tikzcd}\text{source code} \arrow[r] & \boxed{\text{Translator}} \arrow[r] & \text{target code}\end{tikzcd}$$

Types of Translator:

1.Interpreter
2.Compiler
3.Assembler

1.Interpreter:

It is one of the translators that translate high level language to low level language.

$$\begin{tikzcd}\text{high level language} \arrow[r] & \boxed{\text{Interpreter}} \arrow[r] & \text{low level language}\end{tikzcd}$$

During execution, it checks line by line for errors.
Example: Basic, Lower version of Pascal.

2.Assembler:

It translates assembly level language to machine code.

$$\begin{tikzcd}\text{assembly language} \arrow[r] & \boxed{\text{Assembler}} \arrow[r] & \text{machine code}\end{tikzcd}$$

Example: Microprocessor 8085, 8086.

3.Compiler:

It is a program that translates one language(source code) to another language (target code).

$$\begin{tikzcd}\text{source code} \arrow[r] & \boxed{\text{Compiler}} \arrow[r] & \text{target code}\end{tikzcd}$$

It executes the whole program and then displays the errors.

Example: C, C++, COBOL, higher version of Pascal.

Difference between compiler and interpreter:

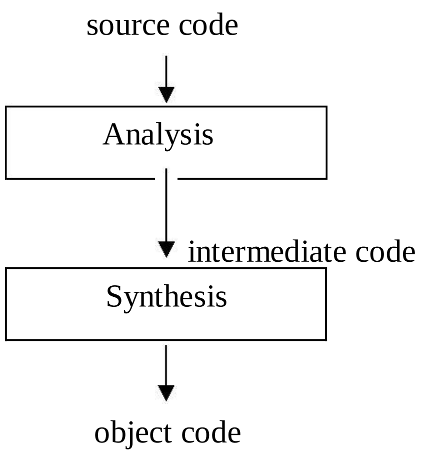
Compiler	Interpreter
It is a translator that translates high level to low level language	It is a translator that translates high level to low level language
It displays the errors after the whole program is executed.	It checks line by line for errors.
Examples: Basic, lower version of Pascal.	Examples: C, C++, Cobol, higher version of Pascal.

PARTS OF COMPILATION

There are 2 parts to compilation:
1. Analysis
2. Synthesis

Analysis part breaks down the source program into constituent pieces and creates anintermediate representation of the source program.

Synthesis part constructs the desired target program from the intermediate representation.



Software tools used in Analysis part:

1) **Structure editor:**
- Takes as input a sequence of commands to build a source program.
- The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- For example , it can supply key words automatically - while …. do and begin….. end.

2) Pretty printers :
- A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- For example, comments may appear in a special font.

3) Static checkers :

- A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- For example, a static checker may detect that parts of the source program can never be executed.

4) Interpreters :

- Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
- An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
- Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

PHASES OF COMPILER

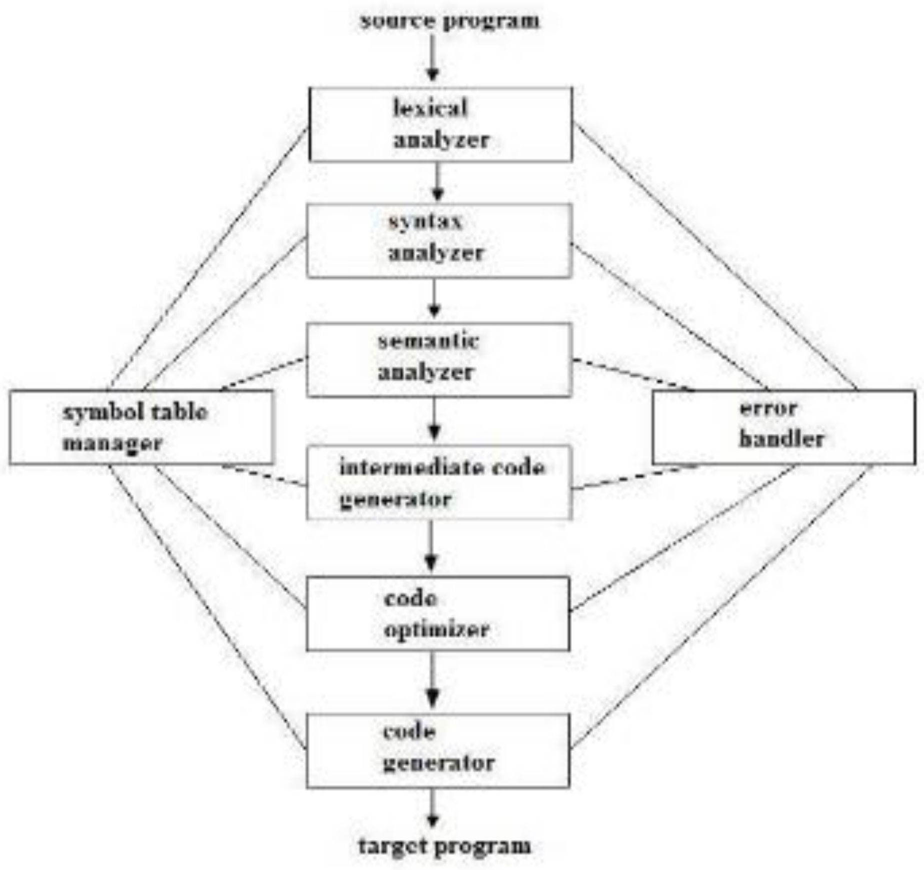
A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:
1) Lexical analysis 2) Syntax analysis
3) Semantic analysis
4) Intermediate code Generation
5) Code optimization
6) Code generation

Sub-Phases:
1) Symbol table management 2) Error handling

Lexical analysis:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token** : It represents a logically cohesive sequence of characters such as keywords,operators, identifiers, special symbols etc.
  Example: a +b =20
  Here, a,b,+,=,20 are all separate tokens.
  Group of characters forming a token is called the **Lexeme**.
- The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.



Syntax analysis:

- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.
- Syntax tree:
  It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For a=b+c*2, syntax tree is

$$\begin{tikzcd}
 & = \arrow[dl, no head] \arrow[dr, no head] & & & \\
a & & + \arrow[dl, no head] \arrow[dr, no head] & & \\
 & b & & * \arrow[dl, no head] \arrow[dr, no head] & \\
 & & c & & 2
\end{tikzcd}$$

Semantic analysis:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

Intermediate code generation:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- The three -address code consists of a sequence of instructions, each of which has atmost three operands.
  Example: t1=t2+t3

Code optimization:

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

Code generation:

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

Symbol table management:

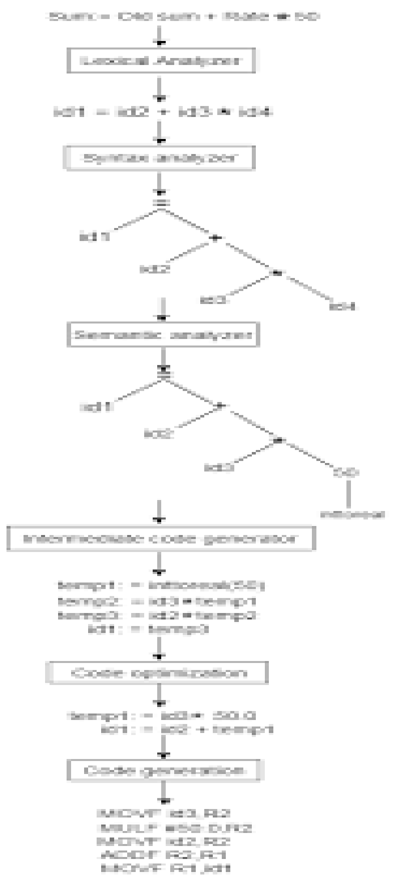
- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Error handling:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.

- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and duringtype conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2. The figure shows the representation of this statement after each phase.

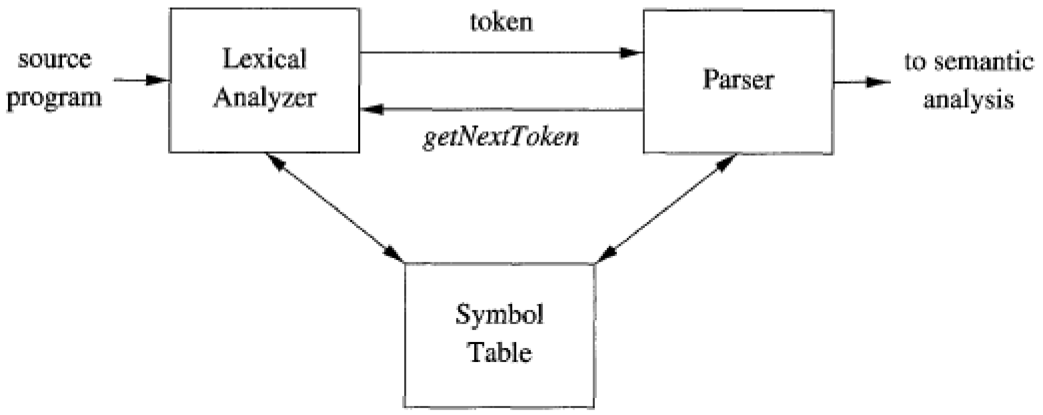


LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

THE ROLE OF THE 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. It is common for the lexical analyzer to interact 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 These interactions are suggested in Fig. 3.1. Commonly, 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.



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.

Issues in lexical analysis:

1. **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. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. **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 can speed up the compiler significantly.
3. **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:

Token:

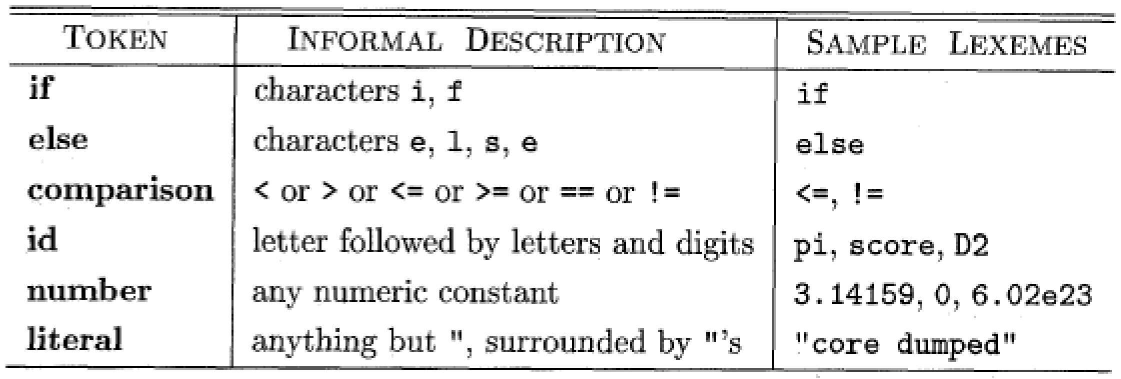
A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.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

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:

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.



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

1.Tokens are treated as terminal symbols in the grammar for the source language using boldface names to represent tokens.
2.The lexemes matched by the pattern for the tokens represent the strings of characters in the source program that can be treated together as a lexical unit
3.In most of the programming languages keywords, operators identifiers , constants , literals and punctuation symbols are treated as tokens.
4. A pattern is a rule describing the set of lexemes that can represent a particular token in the source program.
5. In many languages certain strings are reserved i.e their meanings are predefined and cannot be changes by the users
6. If the keywords are not reserved then the lexical analyzer must distinguish between a keyword and a user-defined identifier

Attributes for tokens:

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse. We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that

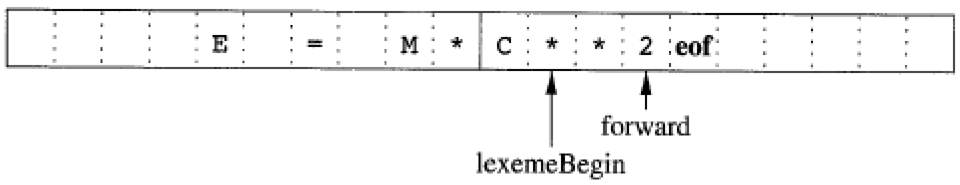
combines several pieces of information. The most important example is the token id, where we need to associate with
the token a great deal of information. Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example : The token names and associated attribute values for the Fortran statement
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 >
Note that in certain pairs, especially operators, punctuation, and keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

Errors in lexical analysis:
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: 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.
Panic mode :
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.

<u>INPUT BUFFERING</u>:
During lexical analyzing , to identify a lexeme , it is important to look ahead atleast one additional character. 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 alternatively reloaded



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 into 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:

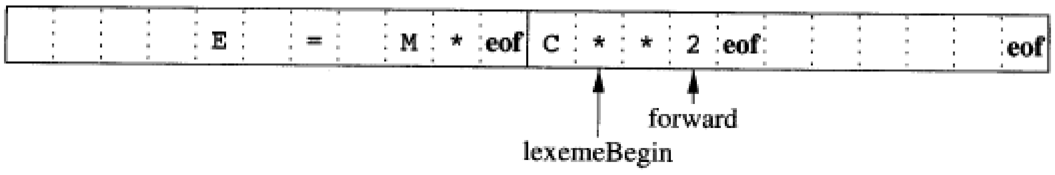
I. **Pointer lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. **Pointer forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.
Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. 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.
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.

```
if forward at end of first half then begin
      reload second half;
      forward := forward + 1
end
else if forward at end of second half then begin
      reload first half;
      move forward to beginning of first half
end
else forward := forward + 1;
```

Fig. 3.4. Code to advance forward pointer.

Sentinels:

If we use the scheme of Section 3.2.1 as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). 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. Figure 3.5 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.



```
forward := forward + 1;
if forward↑ = eof then begin
        if forward at end of first half then begin
             reload second half;
             forward := forward + 1
        end
        else if forward at end of second half then begin
             reload first half;
             move forward to beginning of first half
        end
        else  /* eof within a buffer signifying end of input */
             terminate lexical analysis
end
```

Fig. 3.6. Lookahead code with sentinels.

<u>SPECIFICATION OF TOKENS</u> :

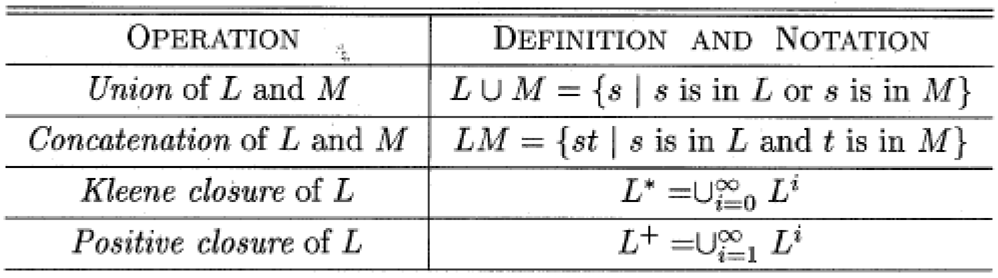
Regular languages are an important notation for specifying lexeme patterns

Strings and Languages:

- An alphabet is any finite set of symbols ex: Letters, digits and punctuation
- The set {01) is the binary alphabet
- A string over an alphabet is a finite sequence of symbols drawn from the alphabet
- The length of the string s,represented as |s| , is the number of occurrences of symbols in s .
- The empty string denoted as € is the string of length 0
- A language is any countable set of strings over some fixed alphabet ex: abstract languages
- If x and y are strings ten the concatenation of x and y denoted by xy is the string formed by appending y to x   for ex if x=cse and y=department , then xy=csedepartment .

Operation on Languages:

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. 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 Lo, the "concatenation of L zero times," is defined to be {E), and inductively, L~ is Li-'L. Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term Lo. That is, E will not be in L+ unless it is in L itself.



OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Example ': Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z) and let D be the set of digits {0,1,.. .9). We may think of L and D in two, essentially equivalent, ways. One way is that 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 the Fig:

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 df 520 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 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.

Regular expressions:

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters.

In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure. This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if letter- is established to stand for any letter or the underscore, and digit- is established to stand for any digit, then we could describe the language of C

identifiers by:

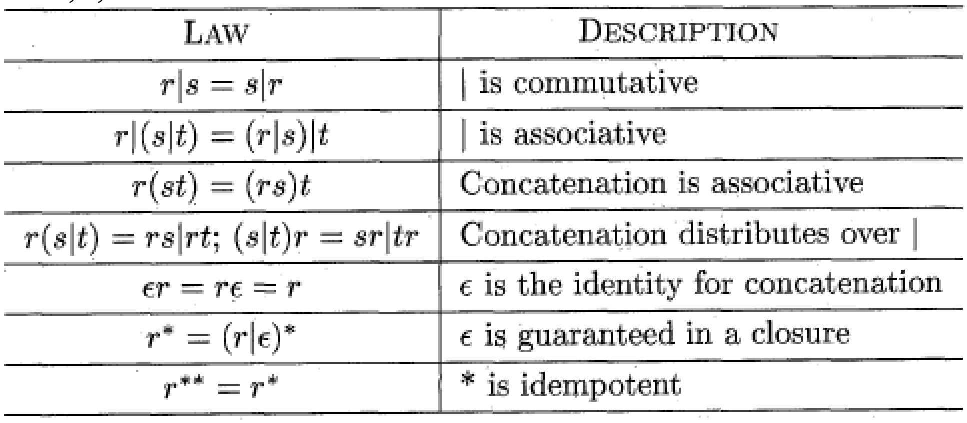
letter- (letter I digit)*

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of letter- with the remainder of the expression signifies concatenation. The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet C and the languages that those expressions denote.

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 C, 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.'
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)1 (9) 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. As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:
a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative.
Under these conventions, for example, we may replace the regular expression (a) I ((b) * (c)) by a/ b*c. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.
Example : Let C = {a, b}.
1. The regular expression a1 b denotes the language {a, b}.
2. (a1 b) (alb) denotes {aa, ab, ba, bb), the language of all strings of length two over the alphabet C. Another regular expression for the same language is aalablbal bb.
3. a* denotes the language consisting of all strings of zero or more a's, that is, {E, a, aa, aaa, . . . }.
4. (alb)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {e, a, b, aa, ab, ba, bb, aaa, . . .}. Another regular expression for the same language is (a*b*)*.
5. ala*b denotes the language {a, b, ab, aab,aaab,. . .), that is, the string a and all strings consisting of zero or more a's and ending in b. A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (alb) = (bla). There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure below shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.



LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If C is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

where:

1. Each di is a new symbol, not in C and not the same as any other of the d's
 2. Each ri is a regular expression over the alphabet C U {dl, d2,. . . , di-l).

By restricting ri to C and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over C alone, for each ri. We do so by first replacing uses of dl in r2 (which cannot use any of the d's except for dl), then replacing uses of dl and d2 in r3 by rl and (the substituted) 7-2, and so on. Finally, in rn we replace each di, for i = 1,2,. . . ,n - 1, by the substituted version of ri, each of which has only symbols of C.

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

letter- + A (B I . - . [Z 1 a 1 b l . - - l z 1 -
digit -+ 01 1 1 - - . 1 9
id + letter- (letter- I digit)*

Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns.

Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification lexical analyzers. The references to this chapter contain a discussion of some regularexpression variants in use today.

1. One or more instances. The unary, postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then (r)+d enotes the language (~ (r)) 'T. he operator
has the same precedence and associativity as the operator *. Two useful algebraic laws, r* = r+Jc and rf = rr* = r*r relate the Kleene closure and positive closure.
2. Zero or one instance. The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to rlc, or put another way, L(r?) = L(r) U (€1. The ? operator has the same precedence and associativity as
* and +.
3. Character classes. A regular expression allazl. .. lan, where the ai's are each symbols of the alphabet, can be replaced by the shorthand [ala2 . . . a,]. More importantly, when a1 , a2, . . . , a, form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by al-a,, that is, just the first and last separated by a hyphen. Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a | b |. . . |z.

<u>RECOGNITION OF TOKENS:</u>

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. Our discussion will make use of the following running example.

stmt -> **if** *expr* **then** *stmt*
I **if** *expr* **then** *stmt* **else** *stmt*
I other
expr -> *term* **relop** *term*
I *term*
term -> **id**
I **number**

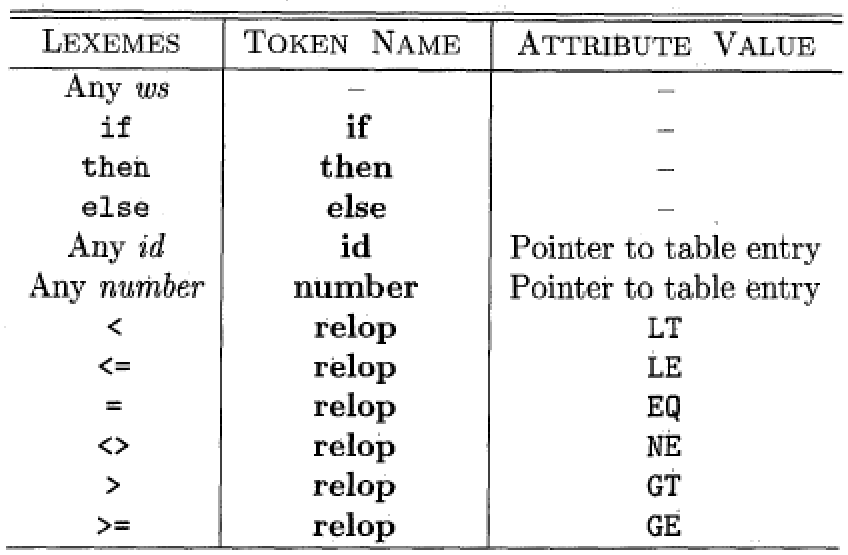
This syntax is similar to that of the language Pascal, in that then appears explicitly after conditions.
For relop, we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes. The terminals of the grammar, which are if, then, else, relop, id, and number, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions

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 i f , then, and else, as well as lexemes that match the patterns for relop, id, and number. To simplify matters, we make the common assumption that keywords are also reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers. In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" ws defined by:

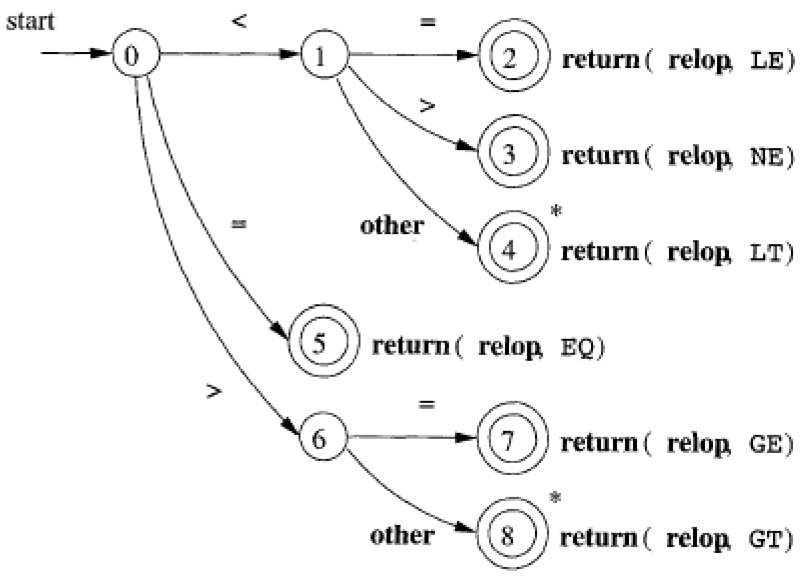
ws -+ (blank I tab (newline)+

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



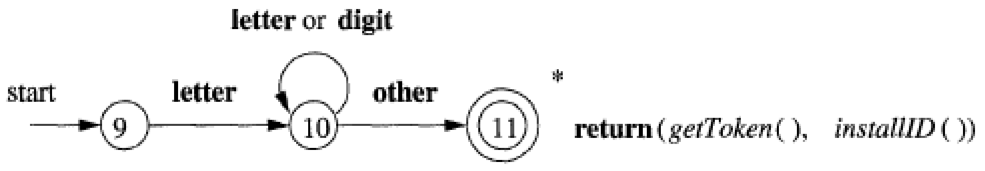
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Transition diagram:
As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams.
State:
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.
Edges:
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 s, 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. Starting in Section 3.5, 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.
Example: Transition diagrams for relational operations.



Recognizing keywords and identifiers :
Recognizing keywords and identifiers presents a problem. Usually, keywords like if or then are reserved (as they are in our running example), so they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like that of Fig. 3.14 to search for identifier lexemes, this diagram will also recognize the keywords i f , then, and else of our running example.

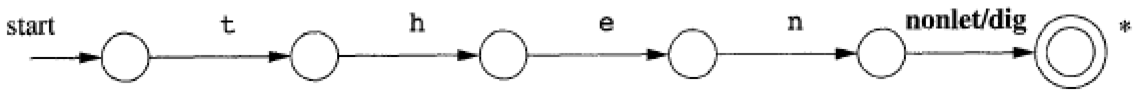
letter or digit:



A transition diagram for id's and keywords

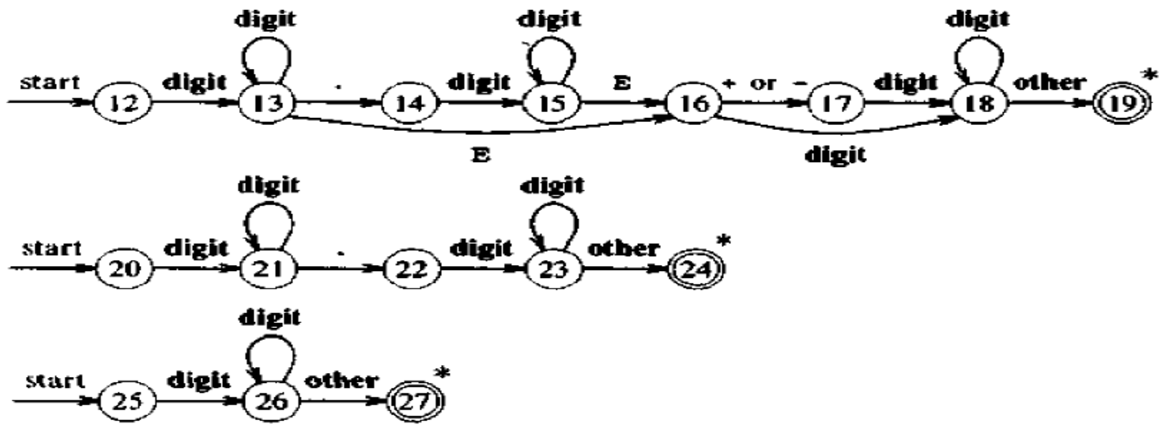
There are two ways that we can handle reserved words that look like identifiers:
1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 3.14. 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 is shown in Fig. 3.15. 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 then in situations where the correct token was id, with a lexeme like thenext value 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.



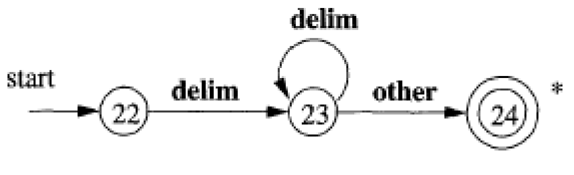
Hypothetical transition diagram for the keyword then

The transition diagram for a token number is given below



- If a dot is seen we have an fraction number. Three diagrams in which first one is the recognition of number which has fractional and exponent part or only exponent part.
- Second diagram is the recognition of number which has only fractional part.
- Third diagram is the recognition of number which has only decimal value.

A transition diagram for whitespace is given below



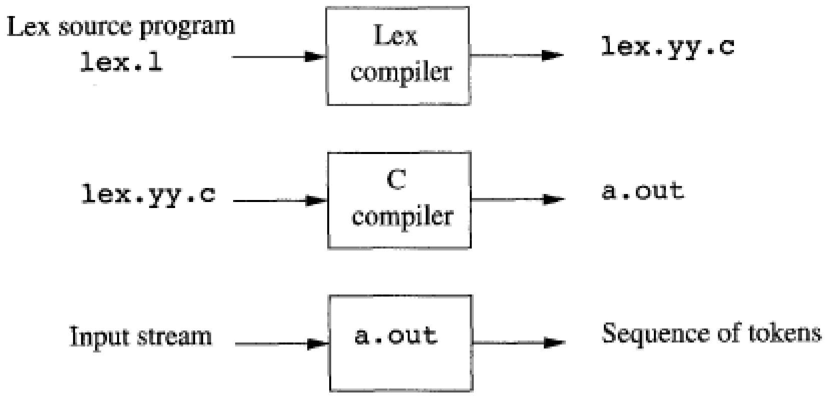
In the diagram we look for one or more whitespace characters represented by delim in the diagram –typically these characters would be blank,tab,newline

THE LEXICAL- ANALYZER GENERATOR - LEX

In this section, we introduce a tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. 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 lex . yy . 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 lex1 is written in the lex language and describes the lexical analyzer to be generated . The Lex compiler transforms lex1 to a c program in a file that is always named lex.yy.c



Structure of a lex program:
A Lex program has the following form:
declarations
%%
translation rules
%%
auxiliary functions
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.
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 as follows :

1. 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.
2.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.
3.The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

Lex program for token :
%{
/* definitions of manifest constants
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim [\t\nl

```
ws (delim)+
letter [A-Za-z]
digit [o-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(REL0P); }
“<=”     {yylval = LE; return(REL0P); }
“=”      {yylval = EQ ; return(REL0P); }
“<>”     {yylval = NE; return(REL0P);}
“>”      {yylval = GT; return(REL0P);}
“>=”     {yylval = GE; return(REL0P);}
%%
int installID0 {/* 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 numerical
                              constants into a separate table */
}
```

In the declarations section we see a pair of special brackets, %(and %). Anything within these brackets is copied directly to the file lex . yy . c, and is not treated as a regular definition.The manifest constants are placed inside it

Also the languages occur as a sequence of regular definitions .

Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces. Thus, for instance, delim is defined to be a shorthand for the character class consisting of the blank, the tab, and the newline; the latter two are represented, as in all UNIX commands, by backslash followed by t or n, respectively

In the auxiliary-function section, we see two such functions, installID( )and installNum() . Like the portion of the declaration section that appears between everything in the auxiliary section is copied directly to file lex. yy . c, but may be used in the actions.

First, ws, an identifier declared in the first section, has an associated empty action. If we find whitespace, we do not return to the parser, but look for another lexeme. The second token has the simple regular expression pattern

i f . Should we see the two letters if on the input, and they are not followed by another letter or digit (which would cause the lexical analyzer to find a longer prefix of the input matching the pattern for id), then the lexical analyzer consumes these two letters from the input and returns the token name IF, that is, the integer for which the manifest constant IF stands. Keywords then and else are treated similarly. The fifth token has the pattern defined by id. Note that, although keywords like i f match this pattern as well as an earlier pattern, Lex chooses whichever pattern

is listed first in situations where the longest matching prefix matches two or more patterns. The action taken when id is matched is given as follows:
I. Function installID( ) 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 yylval, where it can be used by the parser or a later component of the compiler. Note that installID () 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
(b) yyleng is the length of the lexeme found.
3. The token name I D is returned to the parser.

FINITE AUTOMATA:

We shall now discover how Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as finite automata. These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
2. Finite automata come in two flavors:
(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and E, the empty string, is a possible label.
(b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

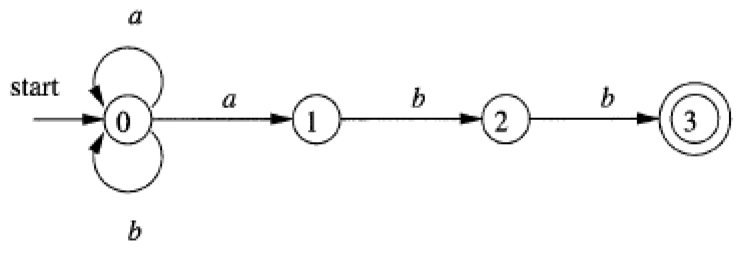
Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the regular languages, that regular expressions can describe

NDFA:

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols C, the input alphabet. We assume that E, which stands for the empty string, is never a member of C.
3. A transition function that gives, for each state, and for each symbol a set of next states.
4. A state so from S that is distinguished as the start state (or initial state).
5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

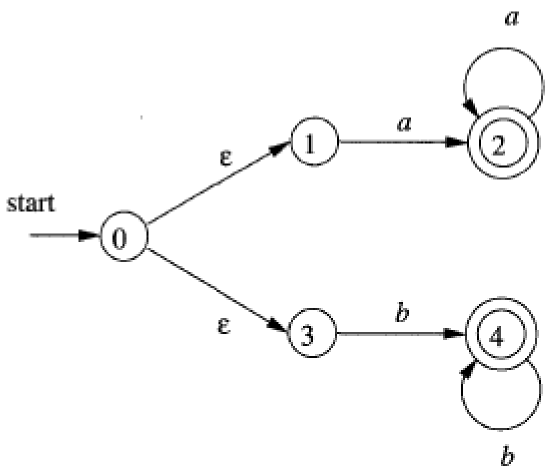
We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a



DFA:

A deterministic finite automaton (DFA) is a special case of an NFA where:
1. There are no moves on input
2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets. While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.



MINIMIZATION OF DFA:

Two method of solving

1) **by using table filling algorithm**

Example:

Minimize the given regular expression (a|b)*abb.

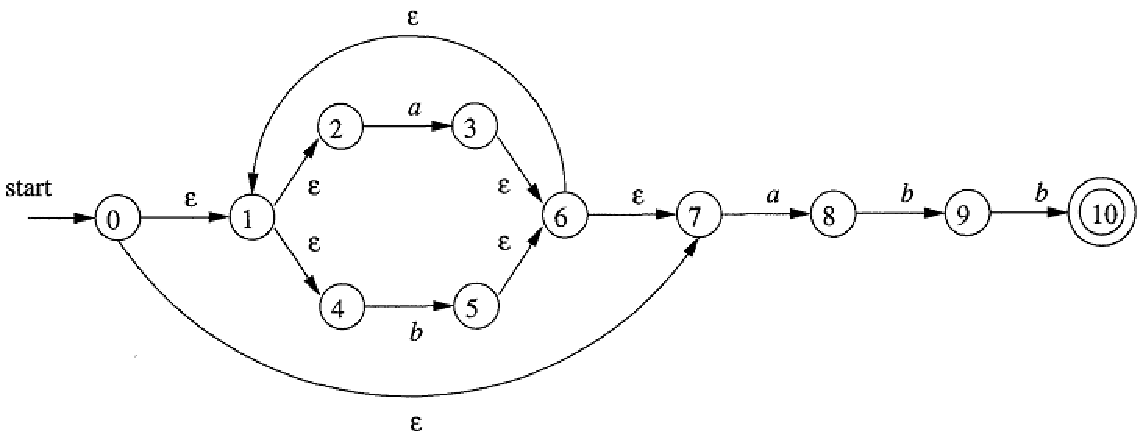


Figure 3.34: NFA N for **(a|b)*abb**

ε – closure (0) = {0, 1, 2, 4, 7 } = A

Dtran[A, a] = ε – closure (move(A, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[A, b] = ε – closure (move(A, b))
= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

Dtran[B, a] = ε – closure (move(B, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[B, b] = ε – closure (move(B, b))
= ε – closure (5, 9)
= {1, 2, 4, 5, 6, 7, 9 }
= D

Dtran[C, a] = ε – closure (move(C, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[C, b] = ε – closure (move(C, b))
= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

Dtran[D, a] = ε – closure (move(D, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[D, b] = ε – closure (move(D, b))
= ε – closure (5, 10)
= {1, 2, 4, 6, 7, 10 }
= E

Dtran[E, a] = ε – closure (move(E, a))
= ε – closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8 }
= B

Dtran[E, b] = ε – closure (move(E, b))

= ε – closure (5)
= {1, 2, 4, 5, 6, 7 }
= C

NFA State	DFA State	a	b
{0, 1, 2, 4, 7 }	A	B	C
{1, 2, 3, 4, 6, 7, 8 }	B	B	D
{1, 2, 4, 5, 6, 7 }	C	B	C
{1, 2, 4, 5, 6, 7, 9 }	D	B	E
{1, 2, 4, 6, 7, 10 }	E	B	C

Fig. Transition table Dtran for DFA D

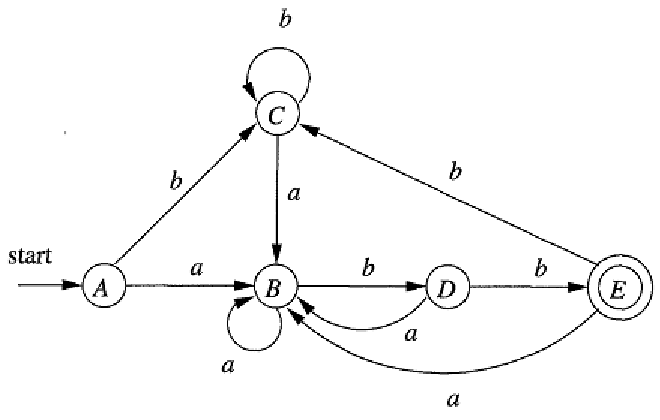


Figure 3.36: Result of applying the subset construction to

Finally optimize the above one by using table filling algorithm. Refer Class notes.