T. Y. B. Sc.
COMPUTER SCIENCE
SEMESTER-VI

NEW SYLLABUS
CBCS PATTERN

COMPILER CONSTRUCTION

Dr. Ms. MANISHA BHARAMBE

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

Syllabus ...

1. Introduction (4 Lectures)

- Definition of Compiler, Aspects of Compilation
- The Structure of Compiler
- Phases of Compiler:
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - Code Optimization
  - Code Generation
- Error Handling
- Introduction to One Pass and Multipass Compilers, Cross Compiler, Bootstrapping

2. Lexical Analysis (Scanner) (4 Lectures)

- Review of Finite Automata as a Lexical Analyzer, Applications of Regular Expressions and Finite Automata (Lexical Analyzer, Searching using RE), Input Buffering, Recognition of Tokens
- LEX: A Lexical Analyzer Generator (Simple Lex Program)

3. Syntax Analysis (Parser) (14 Lectures)

- Definition, Types of Parsers
- Top-Down Parser:
  - Top-Down Parsing with Backtracking: Method and Problems
  - Drawbacks of Top-Down Parsing with Backtracking
  - Elimination of Left Recursion (Direct and Indirect)
  - Need for Left Factoring and Examples
- Recursive Descent Parsing:
  - Definition
  - Implementation of Recursive Descent Parser Using Recursive Procedures
- Predictive [LL(1)] Parser (Definition, Model)
  - Implementation of Predictive Parser [LL(1)]
  - FIRST and FOLLOW
  - Construction of LL(1) Parsing Table
  - Parsing of a String using LL(1) Table
- Bottom - Up Parsers
- Operator Precedence Parser - Basic Concepts
- Operator Precedence Relations form Associativity and Precedence
  - Operator Precedence Grammar
  - Algorithm for LEADING and TRAILING (with Examples)
  - Algorithm for Operator Precedence Parsing (with Examples)
  - Precedence Functions

- Shift Reduce Parser:
  - Reduction, Handle, Handle Pruning
  - Stack Implementation of Shift Reduce Parser (with Examples)
- LR Parser:
  - Model,
  - Types [SLR (1), Canonical LR, LALR] - Method and Examples
- YACC:
  - Program Sections
  - Simple YACC Program for Expression Evaluation

4. System Directed Definition (7 Lectures)

- Syntax Directed Definitions (SDD)
- Inherited and Synthesized Attributes
- Evaluating an SDD at the Nodes of a Parse Tree, Example
- Evaluation Orders for SDD's
- Dependency Graph
- Ordering the Evaluation of Attributes
- S - Attributed Definition
- L - Attributed Definition
- Application of SDT
- Construction of Syntax Trees
- The Structure of a Type
- Translation Schemes:
  - Definition
  - Postfix Translation Scheme

5. Code Generation and Optimization (7 Lectures)

- Compilation of Expression:
  - Concepts of Operand Descriptors and Register Descriptors with Example.
  - Intermediate Code for Expressions - Postfix Notations, Triples, Quadruples and Expression Trees
- Code Optimization:
  - Optimizing Transformations: Compile Time Evaluation, Elimination of Common Sub Expressions, Dead Code Elimination, Frequency Reduction, Strength Reduction
- Three Address Code
- DAG for Three Address Code
- The Value - Number Method for Constructing DAG's
- Definition of Basic Block, Basic Blocks and Flow Graphs
- Directed Acyclic Graph (DAG) representation of Basic Block
- Issues in Design of Code Generator

CHAPTER 2

Lexical Analysis (Scanner)

Objectives...

- To understand Concept of Lexcial Analysis
- To Design the Lexical Analyzer
- To understand Lex

2.0 INTRODUCTION

- Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.
- Lexical analysis is the act of breaking down source text into a set of words called tokens. Each token is found by matching sequential characters to patterns.
- A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.
- The scanner or Lexical Analyzer (LA) performs the task of reading a source text as a file of characters and dividing them up into tokens.
- All the tokens are defined with regular grammar and the lexical analyzer identities strings as tokens and sends them to syntax analyzer for parsing.
- A typical lexical analyzer or scanner is shown in Fig. 2.1.

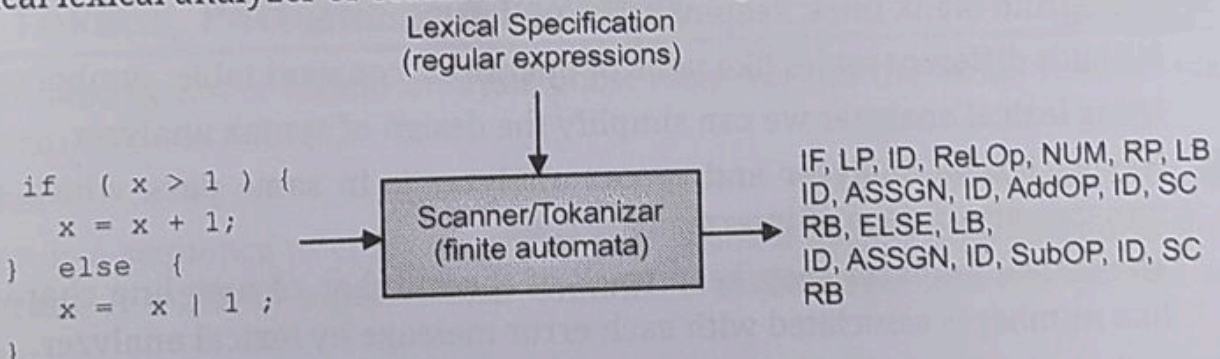


Fig. 2.1: A Lexical Analyzer

- A lexical analyzer consists of an implementation of the finite-state automation. Finite Automata (FA) is a state machine that takes a string of symbols as input and changes its state accordingly.

2.1 BASIC CONCEPTS IN SCANNER

[April 16, 17]

- Lexical analysis is the first phase of the compiler also known as a scanner. It converts the high level input program into a sequence of tokens.
- The main task accomplished by lexical analysis is to identify the set of valid words of language that occurs in an input stream.
- The program which does lexical analysis is called scanner, (see Fig. 2.2).

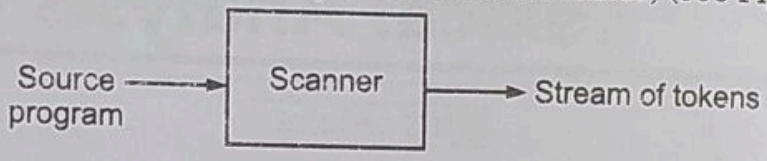


Fig. 2.2: Working of Scanner

- The scanner scans the input program character by character and groups the characters into the lexical units called tokens.
- Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).
- A program that performs lexical analysis may be termed a lexer, tokenizer or scanner although scanner is also a term for the first stage of a lexer.

2.1.1 Role of Lexical Analyzer

[April 16]

- Lexical analyzer acts as an interface between the input source program to be compiled and the later stages of the compiler.
- Lexical analysis means to separate words from the statements from the source code and using rules and regulations i.e. pattern, it converts words in to tokens.
- These words are called as lexeme and the program which performs this task is called as Lexical analyzer.

Need of Lexical Analyzer:

1. Lexical analyzer can perform functions like deleting comments, extra blank spaces and blank lines, keeping track of line numbers.
2. It builds different tables like table of operators, constant table, symbol table etc.
3. Using lexical analyzer we can simplify the design of syntax analyzer.
4. Having lexical analyzer and syntax analyzer is in same pass, which avoids the unnecessary storage of intermediate code.
5. The lexical analyzer may keep track of the number of new line characters, so a line number is associated with each error message by lexical analyzer.

- The main task of lexical analyzer is to read the input program and breaking it up into a sequence of tokens. These tokens are used by the syntax analyzer.
- The interaction between lexical analyzer and parser or syntax analyzer is shown in Fig. 2.3.

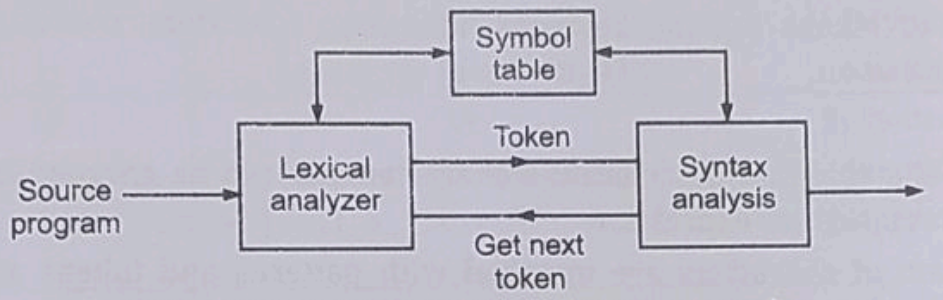


Fig. 2.3: Lexical Analyzer and Parser Interaction

- Lexical analyzer sends a sequence of tokens to the parser and parser sends acknowledge "get next token to lexical analyzer. Lexical analyzer is called by the parser whenever it needs a new token.
- Sometimes, lexical analyzers are divided into following two processes:
  1. **Scanning:** It is a simple process that does not tokenize the input, such as deletion of comments and compaction of consecutive white space characters into one.
  2. **Lexical Analysis:** It is a complex process, which gives output of the scanner as a token.
- There are number of reasons the analysis phase and parsing phase are separated. These reasons are explained below:
  1. **Efficiency:** Separating lexical analysis from syntax analysis increases design efficiency. Compiler efficiency is improved. In lexical analysis, buffering technique is used for reading the input, which speeds up the compiler significantly.
  2. **Simplicity:** By having LA as a separate phase, the complier design is simplified. By separating lexical and syntax analysis phase, logic becomes too simple e.g., if comments and whitespaces (blank, tab, newline) are removed in lexical analysis phase, the parsing becomes so easy and parser design is not complex.
  3. **Portability:** Portability is enhanced. Due to input/output and character set variations, lexical analyzers are not always machine independent. We can take care of input alphabet peculiarities at this level.

2.1.2 Tokens, Patterns and Lexemes

[April 16, Oct. 16]

- When talking about lexical analysis, most often we use the terms lexeme, token and pattern.

[Oct. 16]

1. **Token:**

- Token is a sequence of characters that represents a unit of information in the source program. A token is a pair which contains a token name and an optional attribute value.
- Token is a group of characters with logical meaning. Token is a logical building block of the language.
- There is a set of strings in the input for which the same token produced is output. The tokens are input to the parser.

- Examples of tokens include keyword, identifier, operators, constant, punctuations symbols and so on.

2. **Pattern:**

- A pattern is a rule that describes the character that can be grouped into tokens. It is expressed as a regular expression.
- Input stream of characters are matched with patterns and tokens are identified. A pattern is a set of strings described by a rule that the token may have.
- A rule that defines a set of input strings for which the same token is produced as output is known as pattern.
- Regular expressions play an important role for specifying patterns. Let us see an example, pattern/rule for id is the it should start with a letter followed by any number of letters of digits. This is given by the following regular expression:

```
[A - Za - z][A - Za - z 0 - 9]*
```

Using above pattern, the given input strings "xyz, abcd, a7b82" are recognized as token id.

3. **Lexeme:** [April 16, 18]

- A lexeme is a sequence of characters in the source program that matches the pattern for a token.
- Lexeme is an instance of a token. It is the actual text/character stream that matches with the pattern and is recognized as a token.
- For example, in the 'C' language the statement, `a = a + 1;` in which, token is identifier for lexeme a and pattern is rule for identifier.
- Take another example in 'C' language the statement, `printf("sum = %d\n", sum);` in which, sum is lexemes matching the pattern for token id and "sum = %d\n" is a lexeme matching literal. The printf() is lexeme matching the pattern keyword.
- Consider the following statements:

```
1. const pi = 3.14;
2. int x;
3. if a < b ? a : b;
4. a = "lexical analysis".
```

- Lexical analyzer scans the above statements and construct in the Table 2.1.

Table 2.1: Examples of tokens

Line No.	Lexeme	Pattern	Token
1.	const	const	keyword
	pi	[a-zA-z] [a-z-z0-9]*	identifier
	=	= | < | > | < = | > = | ! =	relational operator
	3.14	[0-9] [0-9]*	number

2.	int x	int [a-zA-z] [a-zA-Z0-9]*	keyword identifier
3.	if a, b <, ?, :	if [a-zA-z] [a-zA-Z0-9]* = | < | > | < = | > = | ! = ? | :	keyword (if) identifier operator
4.	a = "lexical analysis"	[a-zA-z] [a-zA-z0-9]* = |< | > | < = | > = | ! = "any characters between"	identifier operator literal

- A pattern is a rule describing the set of lexemes that can represent a particular token in source programs.
- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.
- For example, the pattern num matches both strings 0 and 1, but it is essential for the code generator to know what string was actually matched.

2.1.3 Lexical Errors

- During the lexical analysis phase Lexical error can be detected. Lexical error is a sequence of characters that does not match the pattern of any token.
- Scanner scans the input and finds lexeme. If the lexeme not matches with any pattern then lexical analyzer detects error.
- In simple words, a character sequence that cannot be scanned into any valid token is a lexical error.
- Misspelling of identifiers, keyword, or operators are considered as lexical errors. The error recovery technique is used by lexical analyzer.
- The following are the error recovery actions performed by lexical analyzer:
  1. Detecting one character from the remaining input.
  2. Inserting a missing character.
  3. Replacing incorrect character by a correct character.
  4. Transposing two adjacent characters.

2.1.4 Input Buffering

[April 16, Oct. 16]

- The lexical analyzer scans the input character by character and groups them into tokens.
- Moreover, it needs to look ahead several characters beyond the next token to determine the next token itself. So, an input buffer is needed by the lexical analyzer to read its input.

- In a case of large source program, significant amount of time is required to process the characters during the compilation. To reduce the amount of overhead needed to process a single character from input character stream, specialized buffering techniques have been developed.
- The lexical analyzer scans the characters of the source program one at a time to discover tokens; however, many characters beyond the next token may have to be examined before the next token itself can be determined.
- For this reason two pointers are used, one pointer to mark the beginning of the token being discovered, and the other, a look ahead pointer, to scan ahead of the beginning point, until the token is discovered.
- Fig. 2.4 shows the how to scan the input tokens using look ahead pointer.

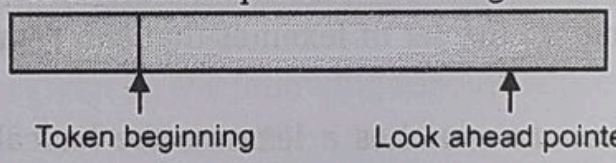


Fig. 2.4: Scanning the Input

- The two pointers to the input buffer i.e., token beginning pointer and look ahead pointer as shown in Fig. 2.4.
- Token Beginning Pointer points to the beginning of the string to be read. Look Ahead Pointer moves ahead to search for the end of the token.
- For example, the Fig. 2.5 shows the lexeme Begin pointer is at character t and forward pointer is at character a.
- The forward pointer is scanned until the lexeme total is found. Once, it is found, both these pointers point to *, which is the next lexeme to be discovered.

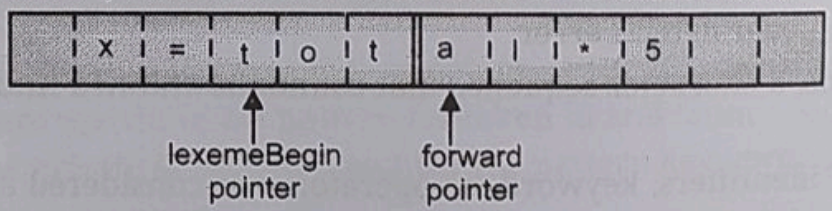


Fig. 2.5: Example of Input Buffer

- A buffer can be divided into two halves. If the look Ahead pointer moves toward halfway in First Half, the second half is filled with new characters to be read.
- If the look Ahead pointer moves towards the right end of the buffer of the second half the first half will be filled with new characters, and it goes on.
- Fig. 2.6 shows the buffer divided into two halves of n-characters, where n is number of characters on one disk block e.g., 1024.

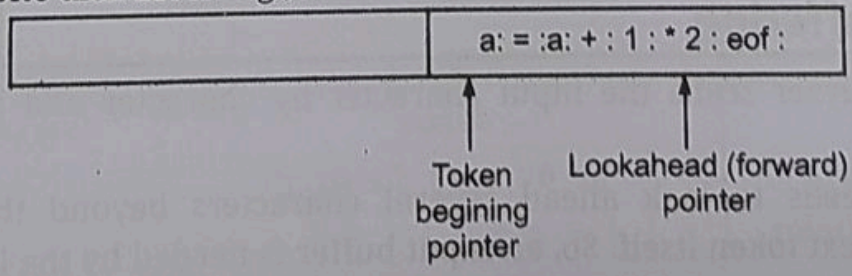


Fig. 2.6: Input Buffer with Two Halves

- The string of the characters between the two pointers is the current token. Initially both pointers points to the first character of the next token to be found.
- Once, the token is found, a look ahead pointer is set to the characters at its right end and beginning pointer is set to the first character of the next token. White spaces (blanks, tab, newline) are not tokens.
- If the look ahead pointer moves beyond the buffer halfway mark, then other half is filled with the next characters from the source file.
- Since, the look ahead pointer move from left half and then again to right half, it is possibility that we may loose characters that not yet been grouped into tokens.
- Every time when left half is full, right half is loaded. If forward pointer move at the end of right half again left half is loaded.
- We can make buffer larger if we choose another buffering scheme. We use sentinels buffering so that characters are not loose when we move left half to right half and vice versa.
- The code of input buffering is as follows:

```
if lookahead pointer is at end of left half
then
    load the right half
    and increment lookahead pointer
else
if lookahead pointer is at end of right half
then
    load the left half and move lookahead pointer to the beginning of left half
else
    increment lookahead pointer
```

- These buffering techniques, makes the reading process easy and also reduces the amount of overhead required to process.

2.1.5 Sentinels

[Oct. 16]

- In sentinels we use special character that is not the part of source program. This character is at the end of each half. So every time look ahead pointer checks this character and then the other half is loaded.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The advantage is that we will not loose characters for which token is not yet formed, while moving from one half to other half.

- Normally, eof is a special character used. But here additional test is required to che whether lookahead pointer points to an eof.
- The Fig. 2.7 shows the sentinels buffering scheme.

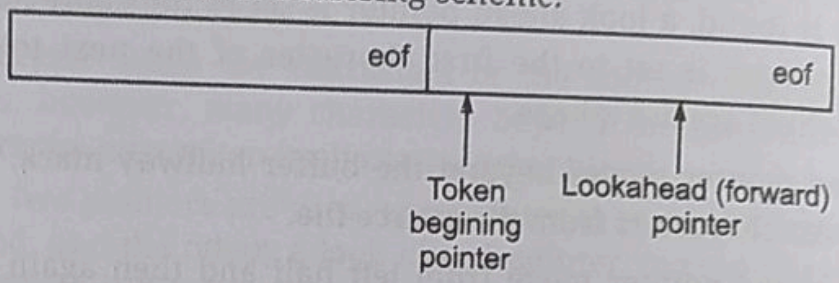


Fig. 2.7: Buffering with Sentinels

- In order to optimize the number of tests to one for each advance of forward point sentinels are used with buffer, (see Fig. 2.8).
- The idea is to extend each buffer half to hold a sentinel at the end. The 'eof' is usual preferred as it will also indicate end of source program.
- The sentinel is a special character eof (end of file) that cannot be a part of sour program.

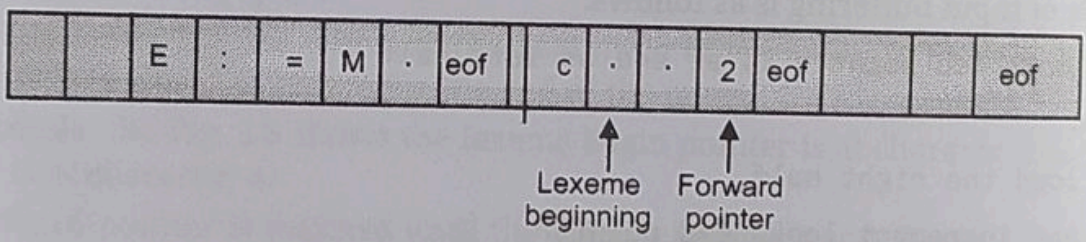


Fig. 2.8: Example of Sentinels

2.2 REVIEW OF FINITE AUTOMATA AS A LEXICAL ANALYZER

- A finite automation is formerly defined as a 5-tuple $(Q, \Sigma, \delta, q0, F)$ where,

  Q is a finite set of states which is non empty.

  Σ is input alphabet.

  q_0 is initial state.

  F is a set of final states and $F \subseteq Q$.

  δ is a transition function or mapping function $Q_x \Sigma \rightarrow Q$ using this the next state can be determined depending on the current input.
- Finite State Machines (FSMs) can be used to simplify lexical analysis. A finite automaton is a recognizer because it merely recognizes whether the input strings are in the language or not.
- A recognizer for a language is a program that takes as input a string x and answer is 'yes', if x is a sentence of the language and 'no' otherwise.
- We compile a regular expression into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

- Transition diagram show the actions that take place when a lexical analyzer is called by the parser to get the next token.
- We use a transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
- The design of lexical analyzers depends more or less on automata theory. In fact, a lexical analyzer is a finite automaton design.
- Lexical analysis is a part of compiler and it is designed using finite automaton, which is known as Lexical analyzer.
- Lexical analyzer is used to recognize the validity of input program, whether the input program is grammatically constructed or not.
- For example, suppose we wish to design a lexical analyzer for identifiers; an identifier is defined to be a letter followed by any number of letters or digits, as follows,

```
identifier = {{letter} {letter, digit}}
```

- It is easy to see that the DFA (Deterministic Finite Automata) in Fig. 2.9 will accept the above defined identifier. The corresponding transition table for the DFA is given as given below:

State/symbol	Letter	Digit
A	B	C
B	B	B
C	C	C

Initial state : A
Final state : B

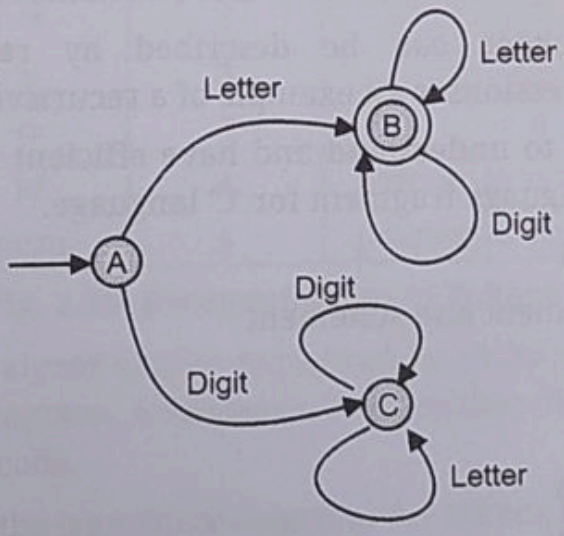


Fig. 2.9: DFA that Accepts Identifier

2.3 APPLICATIONS OF REGULAR EXPRESSIONS AND FINITE AUTOMATA (FA)

- An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

- There are three general approaches to design lexical analyzer:
  1. Write a lexical analyzer in a conventional system programming language (us regular expressions).
  2. Write a lexical analyzer in assembly language.
  3. Write a lexical analyzer using lexical analyzer generator (LEX utility on LINUX).
- Lexical analyzer represents each token in terms of regular expression and regu expression is represented by transition diagram.
- The method of design of lexical analysis is finite automata or transition diagram. W will discuss how lexical analyzer recognizes tokens and its type and value.
- Transition diagrams depict the actions that take place when a lexical analyzer is calle by the parser to get the next token.
- Transition diagrams are used to keep track of the information about characters tha are seen as the forward pointer scans the input.
- Finite Automata (FA) is a state machine that takes a string of symbols as input an changes its state accordingly. FA is a recognizer for regular expressions.
- Regular Expressions (REs) have the capability to express finite languages by defining pattern for finite strings of symbols.
- The grammar defined by regular expressions is known as regular grammar. Th language defined by regular grammar is known as regular language.
- Regular expression is an important notation for specifying patterns. Each patter matches a set of strings, so regular expressions serve as names for a set of strings.
- Programming language tokens can be described by regular languages. Th specification of regular expressions is an example of a recursive definition.
- Regular languages are easy to understand and have efficient implementation. Let u consider a programming language fragment for 'C' language.

```
Statement | if cond statement
          | if cond statement else statement
          | ε
Cond      | id relop id
          | id relop num
          | num relop id
          | id
```

- This is simple if - else statement in 'C' language where relop is any relational operator used for condition in if statement.
- Here, if - else statement is represented in CFG, where LHS symbols are not considered as tokens. Hence the tokens are if, else, relop, id and num.

- Now, for each token we represent regular expressions as follows:

  if → if

  else → else

  relop → < |<=| > |>=| = |< >|

  id → letter (letter | digit)*

  num → (digit)+

  where '*' is zero or more occurrences and '+' is one or more occurrences.

- We assume that tokens are separated by white spaces (blanks, tabs or newlines) so we can have regular expression.

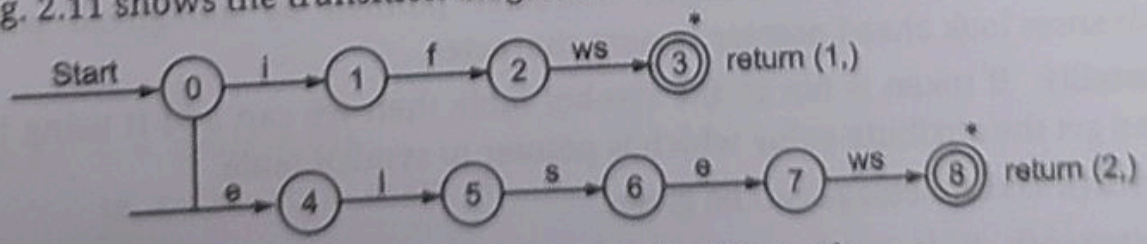
  ωs → (blank | tab | \n)+

- If the match for ws is found, the lexical analyzer does not return a token to the parser. A lexical analyzer recognizes tokens in the input buffer and gives the attribute values shown in Fig. 2.10.

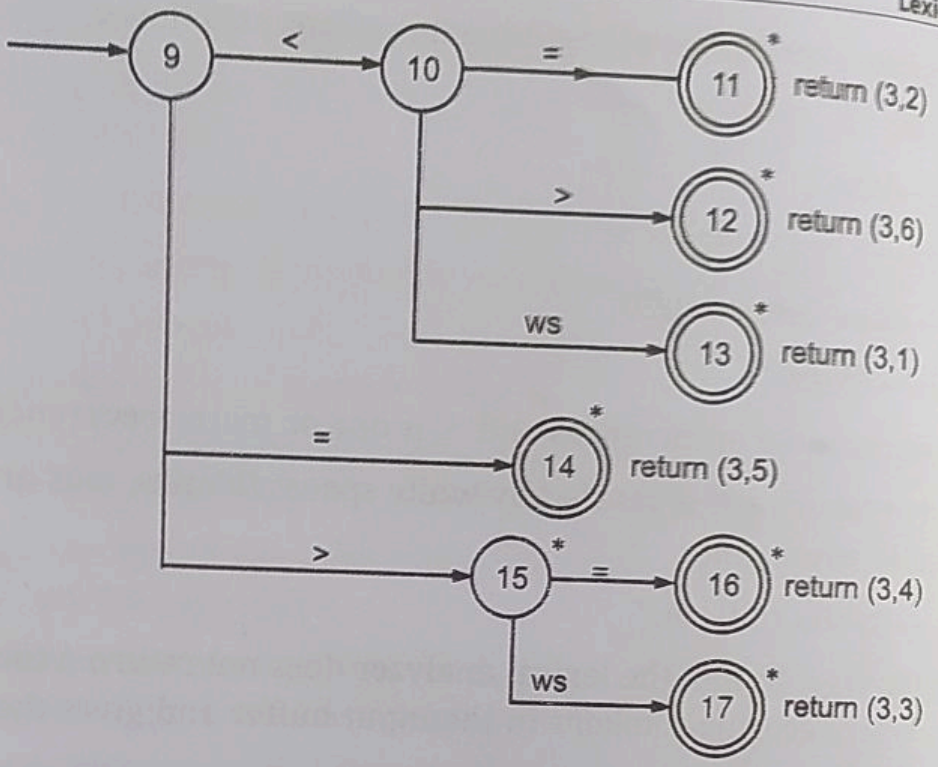
Token	Code	Attribute value
if	1	–
else	2	–
<	3	1
<=	3	2
>	3	3
>=	3	4
=	3	5
<>	3	6
id	4	pointer to symbol table
num	5	pointer to symbol table

Fig. 2.10: Reorganization of Tokens Analysis

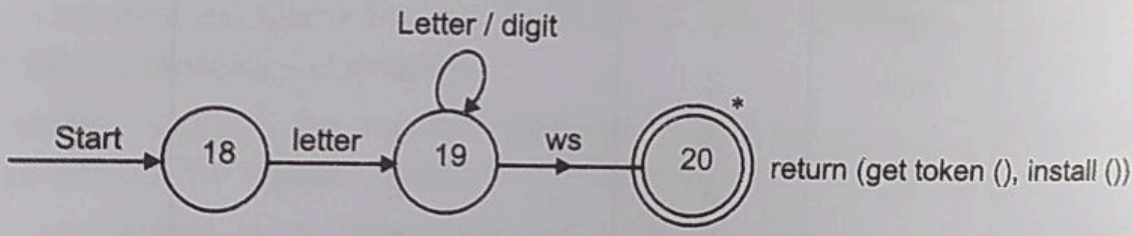
- Design of lexical analyzer can be explained by finite automation. For each token, we draw a transition diagram. A sequence of transition diagrams can be converted into a program or pseudo code.
- The Fig. 2.11 shows the transition diagrams for tokens shown in Fig. 2.10.



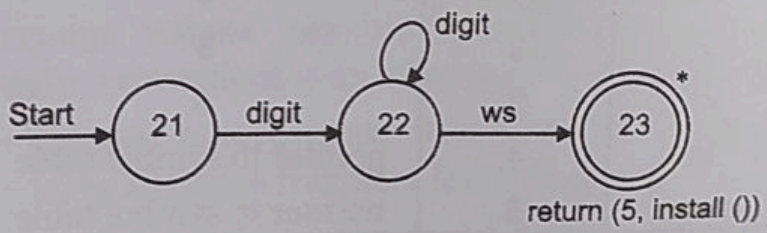
(a) Transition Diagram for Keywords



(b) Transition Diagram for Relational Operator



(c) Transition Diagram for Identifiers



(d) Transition Diagram for num

Fig. 2.11

For writing a code for each state in the transition diagram of Fig. 2.11, we use th following functions as standard functions:

1. **getchar()**: It returns the input character pointed by lookahead pointer an advances look ahead pointer to next character.
2. **install()**: If token is not in the symbol table then we can add it using install-id and get the attribute value which is pointer to symbol table.
3. **error()**: Error message can be given.
4. **gettoken()**: It is used to obtain the token. If the lexeme is a keyword, th corresponding token is returned; otherwise the token id is returned.

5. **retract:** Used to retract look ahead pointer one character, which token is found, next character is delimeter and since delimeter is not the part of token, we are retract procedure.

- To convert transition diagrams into a program, we can write a code for each state. To obtain the next character from input buffer we use getchar() function. Then finds the state which has no edge that is after blank or new line (ws).
- It is denoted by double circle, and here the token is found. If all transition diagrams have been tried without success, then error correction routine is called. We use '*' to indicate states on which retract is called.
- The pseudo code is as follows:

Pseudo-code for keyword:

```
state 0 : c = getchar();
          if c='i' goto state 1;
          else
          if c='e' goto state 4;
          else
          error();
state 1 : c = getchar();
          if c='f' goto state 2;
          else
          error();
state 2 : c = getchar()}
          if delimeter(c)
          goto state 3;
          else
          error();
state 3 : retract();
          return (if,install());
state 4 : c = getchar();
          if c='l' goto state 5
          else
          error( ;
state 5 : c = getchar();
          if c='s' goto state 6
          else
          error();
```

Pseudo-code:

```
state 0 : c = getchar();
          if letter (c) goto state 1
          else
          error();
state 1 : c = gertchar();
          if letter (c) or digit (c) goto state 1
          else
          error()
state 2 : retract (c);
          return (id, install ());
```

Example 2: Create psedo-code for number of C language.

Solution: Number is (digit) (digit)*

The finite automata is,

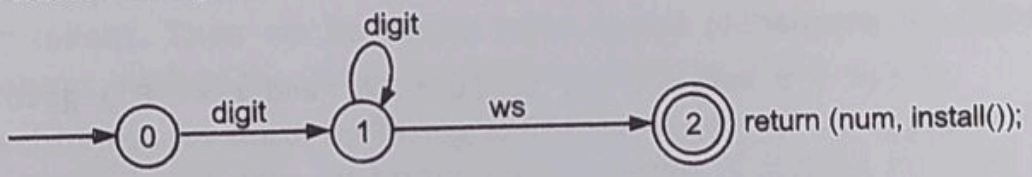


Fig. 2.13: The Finite Automata

Pseudo-code:

```
state 0 : c = getchar()
          if digit (c) goto state 1
          else
          error()
state 1 : c = getchar();
          if digit (c) goto state 1
          else
          error();
state 2 : retract();
          return (num, install ());
```

Example 3: Write pseudo-code for hex-decimal number of C language.

Solution: The hex number start with OX or Ox and it contains 0 – 9, A – F, a - f characters.

The pattern is `0 [xX] [0 - 9 A - F a - f]*`

```
state 6 : c = getchar();
          if c='e' goto state 7
          else
          error();
state 7 : c = getchar();
          if
          delimeter(c) goto state 8;
          else
          error();
state 8 : retract();
          return (else,install());
```

- Similarly we can write pseudo code for identifier as follows :

```
state 18 : c := getchar();
           if letter (c) goto state 19;
           else
           error();
state 19 : c := getchar();
           if letter (c) or digit (c) then goto state 19;
           else if delimiter (c) goto state 20;
           else
           error ();
state 20 : retract()
           return(id, install ());
```

- Hence, after reorganization of tokens the scanner represents by FA and it can be then implemented by writing code for each state. In UNIX, lex or flex is the utility available which is used to generate lexical analyzer automatically.

Example 1: Create pseudo-code for identifier of C language.

Solution: Find out regular expression for identifier.

$$id \rightarrow l\ (l \mid d)^*$$

i.e the pattern for id is

$$[a\text{-}\ zA\ -\ z]\ [a\ -\ zA\ -\ z0\ -\ 9]^*$$

We design the transition diagram for identifier as follows:

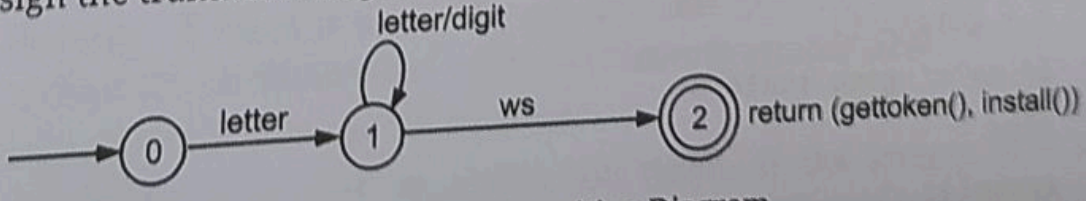


Fig. 2.12: Transition Diagram

The FA is,

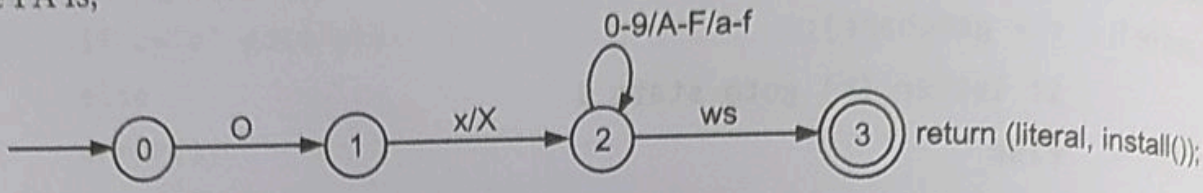


Fig. 2.14: Finite Automata Diagram

Pseudo-code :

```
state 0 : c = getchar();
          if c = '0' goto state 1
          else
          error();
state 1 : c = getchar();
          if c = 'x' or c = 'X' goto state 2
          else
          error();
state 2 : c = getchar();
          if ((c > 0 and c < 9)) or (c > 'A' and c < 'F') goto state 3
          else
          error();
state 3 : retract();
          return (literal, install ());
```

- Regular expressions play an important role in the study of finite automation and its application. The oldest applications of regular expressions were in specifying the component of a compiler called a lexical analyzer.
- Regular expressions are extensively used in the design of lexical analyzer. Regular expression is used to represent the language (lexeme) of finite automata (lexical analyzer).
- A regular expression is a compact notation that is used to represent the patterns corresponding to a token.
- Regular languages, which are defined by regular expressions are used extensively for matching patterns within text (as in Word processing or Internet searches) and for lexical analysis in computer language compilers.
- Regular expressions and finite automata are powerful tools for encoding text patterns and searching for these patterns in textual data.
- For example, hashtags in social media messages and posts (e.g. `#OccupyWall Street`) can be found using the notation `(^|\s) # ([A-Za-z0-9_]+)`, where `(^|\s) #` denotes

beginning of a line or a blank space followed by a # and ([A-Za-z0-9_]+) denotes a sequence of one or more alphanumeric characters and the underscore symbol.

2.3.1 Input Buffering in Lexical Analysis

- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- The lexical analyzer scans the characters of the source program one at a time to discover tokens.
- Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined.
- For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer.
- Input buffering is defined as a temporary area of memory into which each record of data is read when the Input statement executes. There are efficiency issues concerned with the buffering of input.
- A two-buffer input scheme that is useful when looking ahead on the input is necessary to identify tokens. Then we introduce some useful techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- The two techniques for input buffering are Buffer pairs and Sentinels. buffer pairs are used to hold the input data. In buffer pairs a buffer is divided into two N-character halves. Each buffer is of the same size N and N is usually the number of characters on one block (e..g., 1024 or 4096 bytes).
- The sentinel is a special character that cannot be part of the source program, (eof character is used as sentinel).
- For detail information of input buffering and sentinel refer to Sections 2.1.4 and 2.1.5.

2.3.2 Recognition of Tokens

- Tokens can be recognized by Finite Automata (FA). In this section, we will elaborate how to construct recognizers that can identify the tokens occurring in an input stream.
- These recognizers are most widely known as Finite Automata. As the name suggests, it is a machine with a finite number of states in which the machine can perform.
- Transition diagram is notations for representing FA. Transition diagram is a directed labeled graph in which it contains nodes and edges. Nodes represent the states and edges represent the transition of a state.
- Fig. 2.15 shows finite automaton that could be part of a lexical analyzer. The job of this automaton is to recognize the keyword then.
- It thus needs five states, each of which represents a different position in the word then that has been reached so far.

- These positions correspond to the prefixes of the word, ranging from the empty (i.e., nothing of the word has been seen so far) to the complete word.

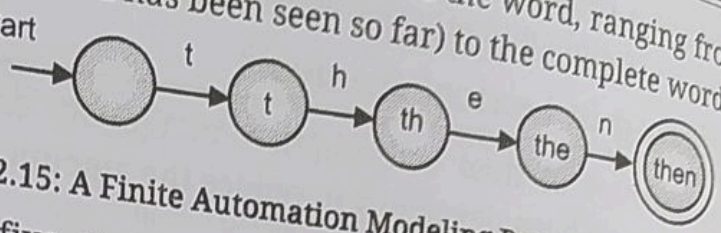


Fig. 2.15: A Finite Automation Modeling Recognition of Then

- In Fig. 2.15, the five states are named by the prefix of then seen so far. correspond to letters.
- We may imagine that the lexical analyzer examines one character of the progra it is compiling at a time and the next character to be examined is the input automaton.
- The start state corresponds to the empty string and each state has a transition o next letter of then to the state that corresponds to the next-larger prefix.
- The state named then is entered, when the input has spelled the word then. Since the job of this automaton to recognize when then has been seen, we could com that state the lone accepting state.
- Lexical analysis can be performed with pattern matching through the use of regu expressions. Therefore, a lexical analyzer can be defined and represented as a DFA.
- Recognition of tokens implies implementing a regular expression recognizer. T entails implementation of a DFA.
- Fig. 2.16 (a) shows steps in token recognition.
- The identified token is associated with a pattern which can be further specified usi regular expressions.
- From the regular expression, we construct an DFA. Then the DFA is turned into cod that is used to recognize the token.

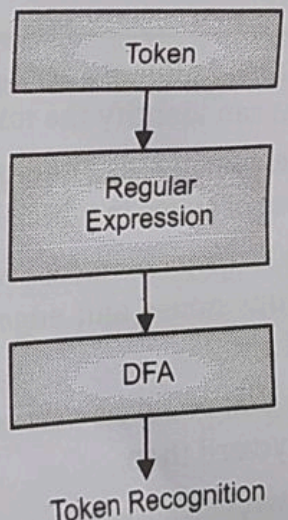


Fig. 2.16 (a): Steps in Token Recognition

- Fig. 2.16 (b) shows an example of transition diagram. Suppose that we want to build a lexical analyzer for the recognizing identifier, > =, >, integer const.
- The corresponding DFA that recognizes the tokens is shown in the Fig. 2.16 (b).

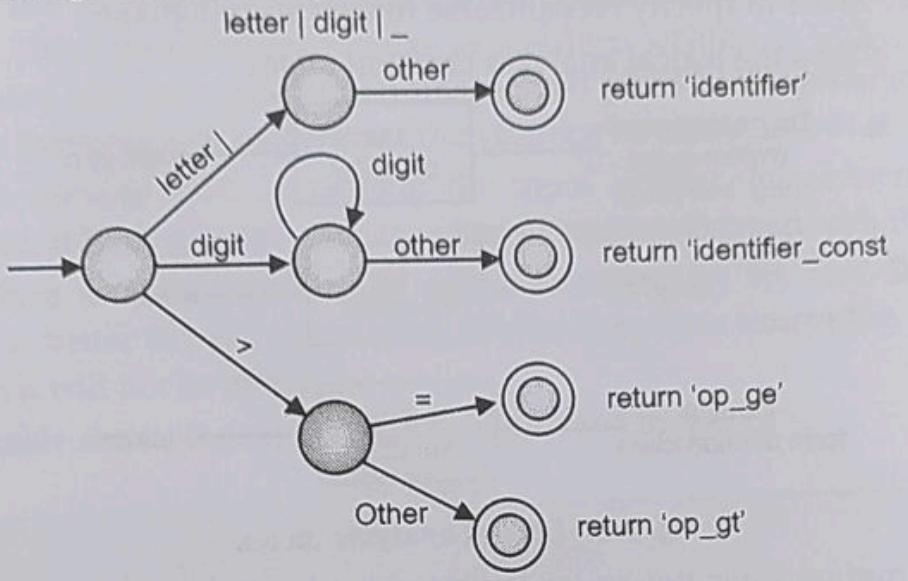


Fig. 2.16 (b): DFA that Recognizes Tokens id, integer_const, etc.

2.4 LEX: LEXICAL ANALYZER GENERATOR [Oct. 16, April 17, 18, 19]

- Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). The purpose of a lex program is to read an input stream and recognize tokens.
- Use a lexical-analyzer generator, such as the Lex compiler, to produce the lexical analyzer from a regular-expression based specification. In this case, the generator provides routines for reading and buffering the input.
- Lex is better known as Lexical Analyzer Generator in Unix OS. It is a command that generates lexical analysis program created from regular expressions and C language statements contained in specified source files.
- Lexical Analyzer Generator introduce a tool called Lex, which allows one to specify a lexical analyzer by specifying regular expressions to describe pattern for tokens. Lex tool itself is a lex compiler.
- A lex compiler or simply lex is a tool for automatically generating a lexical analyzer for a language. It is an integrated utility of the UNIX operating system. The input notation for the lex is referred to as the lex language.
- Lex is a program used to construct lexical analyzer for a variety of languages. The input of lex is in lex language.
- Lex is generally available on UNIX or LINUX. It generates a scanner written in 'C' language.
- Lex is one program of many that is used for generating lexical analyzers based on regular expressions.

- A lexical analyzer is a program that processes strings and returns tokens that recognized within the input string.
- The token identifies the recognized substring and associates attributes with it. Lex regular expressions to specify recognizable tokens of a language.
- The Fig. 2.17 shows the lexical analyzer creation on lex.

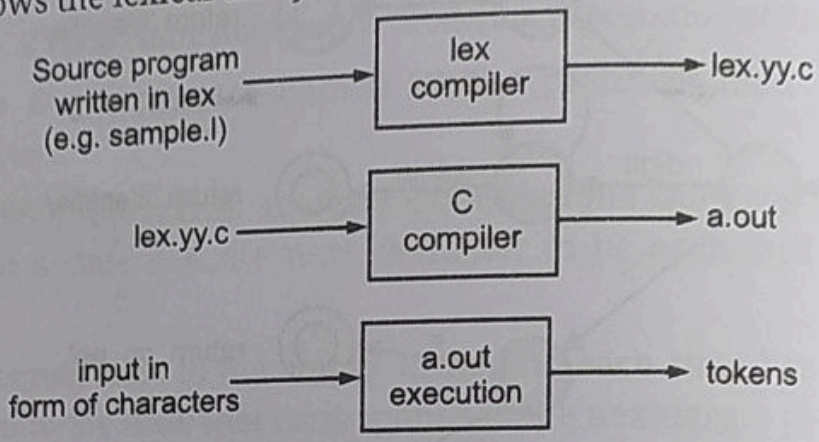


Fig. 2.17: Lexical analyzer on lex

- The source program is written in lex language having extension *·l*. Then this progra is run on lex compiler which always generate C program lex·yy·c which we link wi lex library - *ll*.
- This program is having representation of transition diagram for every regula expression present in the program. Then len·yy·c is compiled on 'C' compiler whic generate output file a·out.
- The about is used for transforming input stream into a sequence of tokens. Th compilation process on LINUX is as follows :

```
$ lex sample·l
$ cc lex·yy·c - ll    (where - ll is optional)
$ ·\a·out
```

Lex Program Specification: [Oct. 1

- A lex program consists of three sections and each section is separated by %%.

```
definition or declaration
%%
translation rules
%%
procedures written in 'C' language
```

- The three sections of lex program are explained below:
  1. **The declaration section** consists of declaration are header files, variables an constants. It also contains regular definitions. We surround the C code with specia delimeters "%{" & "%}", Lex copies the data between "%{" & "%}" directly to the file.

2. **The rule section** consists of the rules written in regular expression forms with corresponding action. This action part is written in 'C' language. In other words, each rule is made up of two parts: a **pattern** and an **action** separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are regular expressions written in UNIX-style. C code lines anywhere else are copied to an unspecified place in the generated 'C' file. If the C code is more than one statement then it must be enclosed within braces { }.

   When a lex scanner runs, it matches the input against the pattern in the rule section. When it finds a match then it execute C code associated with that pattern.

3. **The procedure section** is in C code and it is copied by lex to C file. For large program, it is better to have supporting code in a separate source file. If we change lex file then it will not be affected.

- The following table shows the regular expressions used in lex.

Table 2.2

Regular Expressions	Matches
• (dot)	any single character except the newline character "\n".
*	zero or more occurrences of expression
+	one or more occurrences of expression
?	zero or one occurrences of regular expression. e.g. (– (digit) ?)
\	Any escap character in 'C' language.
[a - z]	range of characters from 'a' to 'z'. range is indicated by '-' (hyphen).
[0 - 9]	range of digits from 0 to 9.
$	matches end of line, as it is last character of regular expression.
^	matches beginning of line, as it is first character of regular expression.
[012]	zero or one or two.
|	"or" e.g. a | b either a or b.
( )	regular expressions are grouped (a | b).

Example 4: A lex program to recognize verbs had, have and has.

Solution:

```
/* Lex program to recognize verbs */
% {
  #include<stdio.h>
% }
%%
```

```
(has/have/had)          { printf ("%s : is a verb \n", yytext);}
[a - zA - z]+           { printf ("%s : is not a verb \n", yytext);}
%%
main();
{
  yylex();
}
```

- Let us save this program as sample.l and compile, as follows :

```
$ lex sample.l
$ cc lex.yy.c - ll
$ ./a.out
  had
  had is a verb
$ ./a.out
  abc
  abc is not a verb
```

- Some Lex Library functions are discribed below: [Oct. 17, 18, Ap

1. **yylex()**: This function is used to start or resume scanning. The next call of program1 to yylex() will continue from the point where it left off. All codes in the section are copied into yylex().
2. **yytext()**: Whenever a lexer matches a token, the text of the token is stored in the null terminated string yytext. (work just like pointers in C). Whenever the next token is matched, the contents of yytext are replaced by new token.
3. **yywrap()**: The purpose of yywrap() function to additional processing in order to "wrap" things up before terminating.

```
yywrap()
{
  return (1);
}
```

When yylex() reaches the end of its input file, it calls yywrap( ), which returns a value of 0 or 1. If the value is 1, indicates that no further input is available. By default it always return 1.

4. **yyerror( )**: The yyerror( ) function is called which reports the error to the user.

```
yyerror()
{
  printf ("error (any)");
}
```

- This routine finds the error in the input. All versions of yacc also provide a simple error reporting routine.

Note:

- When we use a lex scanner and a yacc parser together, the parser is the high level routine.
- It calls the lexer yylex( ) whenever it needs a token from the input and lex returns a token value to the parser.
- In this lex-yacc communication, the lexer and parser have to agree what the token codes are. Hence, using a preprocessor #define we define integer for each token in lex program.
- For example: `#define id 257`

  `#define num 258`

  yacc can write a C header file containing all of the token definitions in y·tab·h.
- To compile and execute lex program on UNIX :

```
$ lex programme.l
$ cc lex.yy.c - ll
        (ll is lex library)
```

- We can also compile without - *ll* option.

```
$ ./a.out   (to run the program)
```

Example 5: Find out regular expression in LEX for language containing the strings starting with and ends with d over {a, d}.

Solution: a [ad]* d

Example 6: Find regular expression for hex-decimal number in C language.

Solution: [– +] ? 0 [xX] [0 – 9a – fA – F]$^{+}$

Example 7: Find regular expression for language ending with 1 over {0, 1}.

Solution: [0 1] [0 | 1]*1

Example 8: Write a regular expression for floating point number in C language.

Solution: [– +] ? [0 – 9]$^{+}$ (\ · [0 – 9]$^{+}$) ? (E (+ \–) ? [0 – 9]$^{+}$) ?

Example 9: A lex program to recognize the token of Example 5.

Solution:

```
/* Lex specification to recognize tokens if, else, <, >, <=, >=, =, <>, id,
num */
%{
#define if 255
#define else 256
```