UNIT – I
LEXICAL ANALYSIS

Translator:

It is a program that translates one language to another Language.

1. INTRODUCTION TO LANGUAGE PROCESSING

The Language Processing System can be represented as shown below figure.

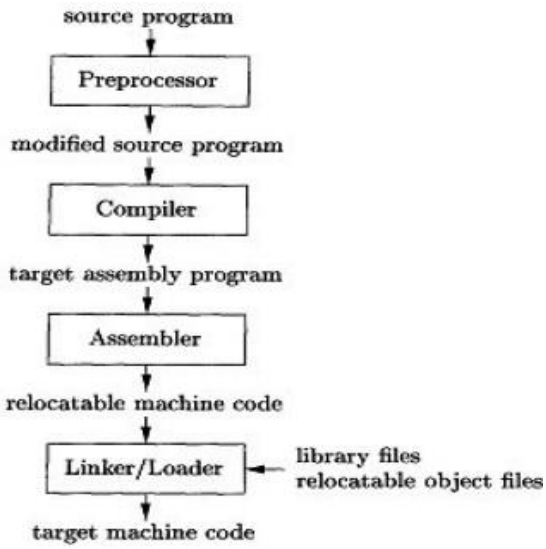
PRE-PROCESSOR: A pre-processor produce input to compilers. They may perform the following functions.

Macro processing: A pre-processor may allow a user to define macros that are short hands for longer constructs.

File inclusion: A pre-processor may include header files into the program text.

Rational pre-processor: these pre-processors augment older languages with more modern flow-of-control and data structuring facilities.

Language Extensions: These pre-processor attempts to add capabilities to the language by certain amounts to build-in macros.



COMPILER:

Compiler is a translator, program that translates a program written in (HLL) - the source program and translate it into an equivalent program in (MLL) - the target program. As an important part of the compiler is error showing to the programmer.

ASSEMBLER:

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language.

Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program)

INTERPRETER:

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

During execution, it checks the line by line for error(s).

Difference between Compiler and Interpreter as

No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is I **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement** : **More**(Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

LOADER AND LINK-EDITOR:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could "relocate" directly behind the user's program. The task of adjusting programs they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

Examples of Compilers:

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. Fortran compilers
10 .Java compilers
11.Pascal compilers
12. PL/I compilers
13. Python compilers

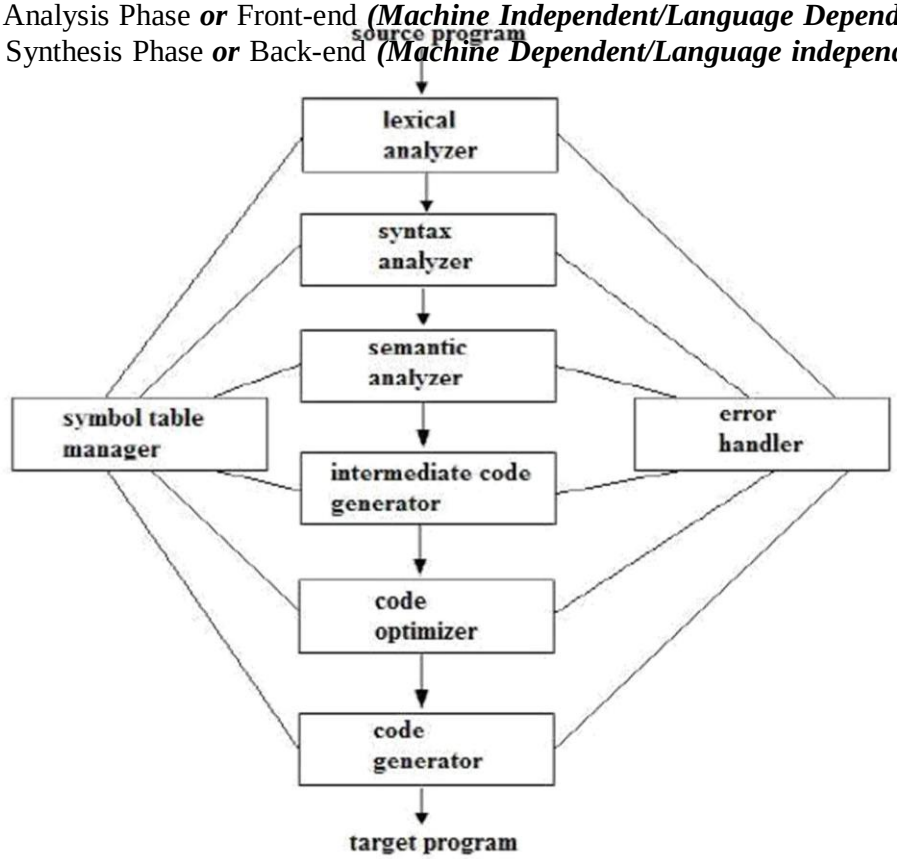
2. STRUCTURE OF THE COMPILER DESIGN *OR* PHASES OF A COMPILER

A compiler operates in phases. A phase is a *logically* interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

Analysis Phase ***or*** Front-end ***(Machine Independent/Language Dependent)***

Synthesis Phase ***or*** Back-end ***(Machine Dependent/Language independent)***



Compilation process is partitioned into no. of sub processes called ***'phases***'.

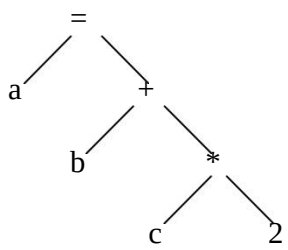
Lexical Analysis:-

- It is also called as Scanner, because it reads the program.
- 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



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 Generations:-

- 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.

Table Management (or) Book-keeping:-

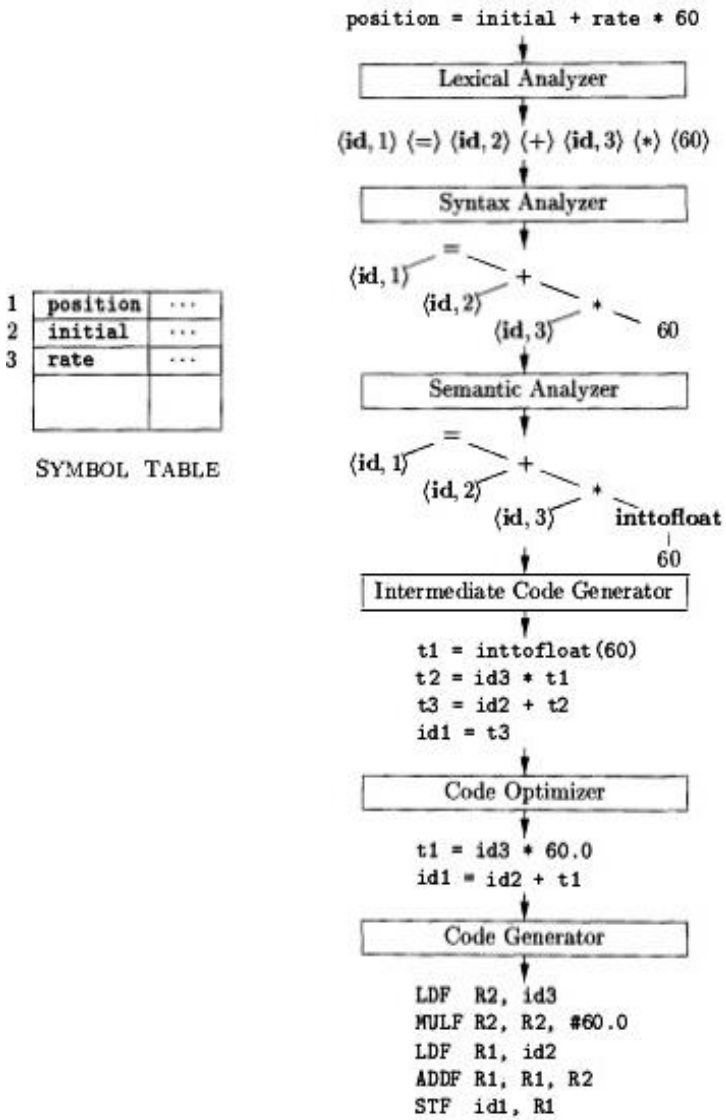
- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from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

- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Error Handlers:-

- 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 during type 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.

Example: To illustrate the translation of source code through each phase, consider the statement ***position := initial + rate * 60***. The figure shows the representation of this statement after each phase.



Compiler can be grouped into front and back ends.

Front end: analysis

These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

Back end: synthesis

It includes code optimization phase and code generation along with the necessary error handling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- ***Single*** pass: usually requires everything to be defined before being used in source program.
- ***Multi*** pass: compiler may have to keep entire program representation in memory.

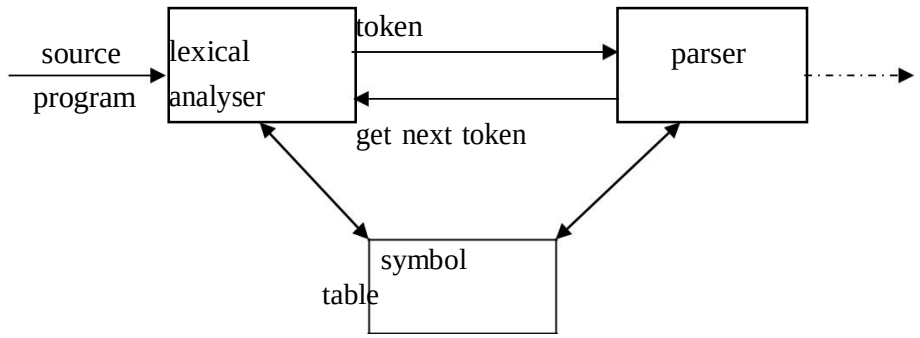
Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

3) 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.

3.1) Role of The Lexical Analyzer:

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



- Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Issues of Lexical Analyzer

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

Tokens

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, and COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2;

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

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.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

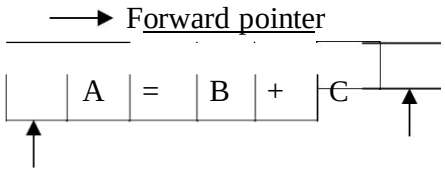
Error Recovery Strategies in Lexical Analysis:

The following are the error-recovery actions in lexical analysis:

1) Deleting an extraneous character.
2) Inserting a missing character.
3) Replacing an incorrect character by a correct character.
4) Transforming two adjacent characters.
5) **Panic mode recovery**: Deletion of successive characters from the token until **error** is resolved.

INPUT BUFFERING

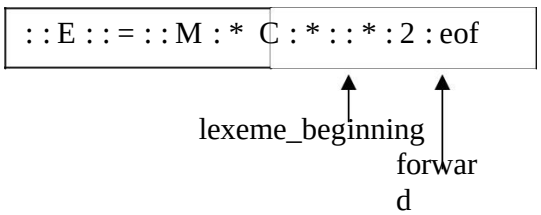
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



We introduce a two-buffer scheme that handles large look ahead safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

- A buffer is divided into two N-character halves, as shown below



- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- Two pointers to the input are maintained:
  1. Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer **forward** scans ahead until a pattern match is found.

Once the next lexeme is determined, forward is set to the character at its right end.

- The string of characters between the two pointers is the current lexeme.
  After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found.

Advancing forward pointer:

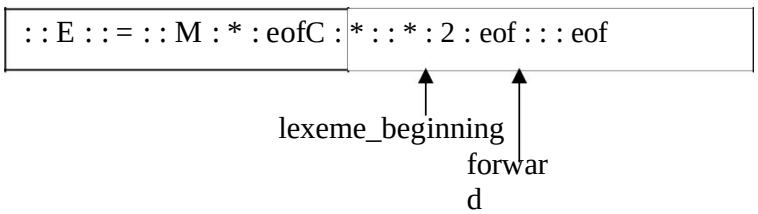
Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

Code to advance forward pointer:

```
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 second half;
    move forward to beginning of first half
end
else forward := forward + 1;
```

Sentinels

- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. The sentinel arrangement is as shown below:



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.

Code to advance forward pointer:

```
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
```

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1) Strings

2) Languageand 3) Regular expression

Strings and Languages

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

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

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.
2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.
   For example, nana is a suffix of banana.
3. A **substring** of s is obtained by deleting any prefix and any suffix from s.
   For example, nan is a substring of banana.
4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s.
   For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1.Union

2.Concatenation

3.Kleene closure

4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}
2. Concatenation : LS={0a,1a,0b,1b,0c,1c}
3. Kleene closure : L^* ={ ε,0,1,00….}
4. Positive closure : L^+ ={0,1,00….}

Regular Expressions

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If ‘a’ is a symbol in Σ, 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.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

   a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
   b) (r)(s) is a regular expression denoting the language L(r)L(s).
   c) (r)* is a regular expression denoting (L(r))*.
   d) (r) is a regular expression denoting L(r).
4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

Regular set

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.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

……

$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter (letter | digit) *

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances (+)*:

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular xpression that denotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator + has the same precedence and associativity as the operator *.

2. *Zero or one instance (?)*:

- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. *Character Classes*:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

RECOGNITION OF TOKENS

Consider the following grammar fragment:

stmt → if expr then stmt
| if expr then stmt else stmt | ε
expr → term relop term | term
term → id | num

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter(letter|digit)*
num → digit+ (.digit+)?(E(+|-)?digit+)?

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.**Transition diagrams**

It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

Transition diagram for relational operators

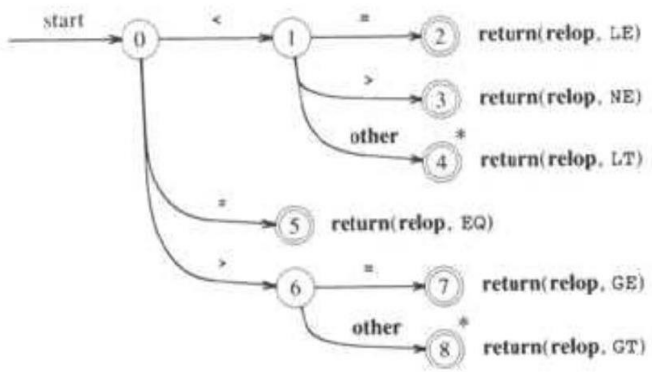
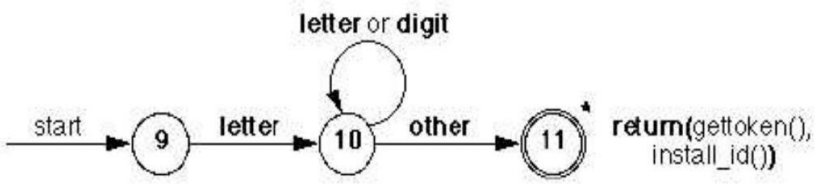


Fig. 3.12. Transition diagram for relational operators.

Transition diagram for identifiers and keywords



A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

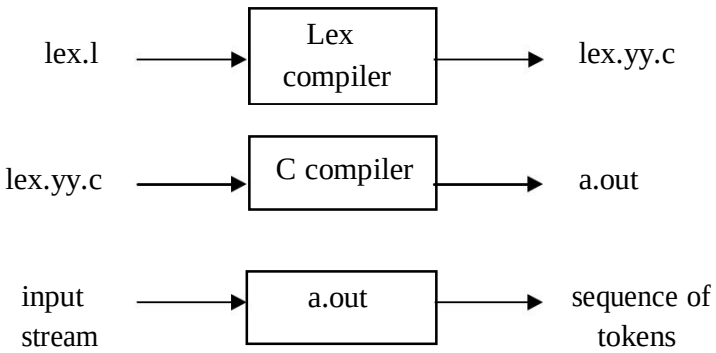
- Lex
- YACC

LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form

  p_1 {$action_1$}
  p_2 {$action_2$}
  …
  p_n {$action_n$}

  where p_i is regular expression and $action_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

YACC- Yet Another Compiler-Compiler

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

FINITE AUTOMATA

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

Types of Finite Automata

There are tow types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by
$M = \{Q_n, \Sigma, \delta, q_0, f_n\}$

Q_n – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q_0 – starting state
f_n – final state

Deterministic Finite Automata

DFA is a special case of a NFA in which
i) no state has an ε-transition.
ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by
$M = \{Q_d, \Sigma, \delta, q_0, f_d\}$

Q_d – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q_0 – starting state
f_d – final state

Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:
i) Convert RE to NFA using Thomson's rules
ii) Convert NFA to DFA
iii) Construct minimized DFA