UNIT - 1  INTRODUCTION TO COMPILING & LEXICAL ANALYSIS

1. Introduction to compilers -
   Compilers is a program which takes one language (source program) as input and translates it into an equivalent another language (target program)

   INPUT
   SOURCE PROGRAM    COMPILER    OUTPUT
   ↓        ↓                  ↓
   ERROR MESSAGES  TARGET PROGRAM

2. Major data structures in a compiler -
   Tokens - It represents basic program entities such as identifiers, literals, reserved words, operators, delimiters etc.
   Syntax Tree - It is generated by the parser. It is usually constructed as a Standard pointer-based structure that is dynamically allocated as parsing proceeds.
   Symbol Table - It keeps information associated with all kind of identifiers, e.g., constants, variables, functions, parameters, types, fields etc.
   Literal Table - It stores constant and strings used in a program.
   Quick insertion and lookup are essential. Deletion is not necessary.
   Temporary file - Used historically by old compilers due to memory constraints. Hold the data of various stages.

3. Bootstrapping and Porting -
   Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program. This complicated can further handle even more complicated program and so on.

T方言 -

A compiler written in language H (for host language) that translates language S (for host language) into language T (for target language).
Eq: \[ \begin{array}{ccc} A & B & B & C \\ H & H & H \\
\end{array} \Rightarrow \begin{array}{c} A \\ C \\ H \\
\end{array} \]

1st Step in Bootshap process:
\[ \begin{array}{c} A \\ H \\ \end{array} \Rightarrow \begin{array}{c} A \\ H \\ H \\
\end{array} \]

2nd Step in Bootshap process:
\[ \begin{array}{c} A \\ H \\ \end{array} \Rightarrow \begin{array}{c} A \\ H \\ H \\
\end{array} \]

Compiles written in its own language A

Running but inefficient

Quick & dirty compiles written in machine language

Compiles written in its own language A

Running but inefficient compilers

(from the 1st step)

\[ \rightarrow \text{Porting a compiler} - \]

\[ \text{STEP-1} - \begin{array}{c} A \\ K \\ \end{array} \Rightarrow \begin{array}{c} A \\ K \\ \end{array} \]

Compiler source

\[ \text{STEP-2} - \begin{array}{c} A \\ K \\ \end{array} \Rightarrow \begin{array}{c} A \\ K \\ \end{array} \]

Compiled source code retargeted to K

Cross compilers

COMPILER STRUCTURE -

1. Analysis-synthesis model of compilation

\[ \begin{array}{ccc} \text{Source Program} & \rightarrow & \text{Compiler} \\
& \rightarrow & \text{Analysis} \rightarrow \text{Synthesis} \\
& \rightarrow & \text{Target Program} \\
\end{array} \]
Analysis of the source program contains three steps: lexical analysis, syntax analysis, and semantic analysis. Intermediate code is generated from the input source program.

In synthesis, three steps are involved: intermediate code generation, code generation, and code optimization.

2. \[ a = b + c \times 60 \]

**Various Phases of a Compiler:**

- **Source Code**
  - Scanner
    - (Lexical Analyzer)
    - Symbo Tokens
      - Parser
        - (Syntax Analyzer)
        - Syntax Tree
          - Semantic Analyzer
            - Annotated Tree
              - Intermediate Code
                - Target Code Optimizer
                  - Target Code
                    - Code Generator
                      - Target Code

- **LITERAL TABLE**
- **SYMBOL TABLE**
- **ERROR HANDLER**
(1) **Lexical Analysis (Scanning)** -

It is a phase of compilation in which the complete source code is scanned and source program is broken up into groups of strings called *tokens*. A token is a sequence of characters having a collective meaning.

(2) **Syntax Analysis (Parsning)** -

In this phase, tokens generated by the lexical analyzer are grouped together to form a hierarchical structure called *parse tree* or *syntax tree*.

(3) **Semantic Analysis** -

It determines the meaning of the source string. It involves matching of parentheses, checking the scope of operation etc.

(4) **Intermediate Code Generation** -

It is a kind of code which is easy to generate and this code can be easily converted to target code. It can represented as 3-address code.

(5) **Code Optimization** -

It improves the intermediate code which reduces memory consumption and have a faster executing code, which improves the runtime of the target program.

(6) **Code Generation** -

Target code is generated as assembly code or machine code.

**LEXICAL ANALYSIS** -

1. **Input Bufferying** -

   The lexical analyzer scans the input string from left to right, one character at a time. It uses two pointers, *backward pointer* (`bptr`) and *forward pointer* (`fptr`) to keep track of the portion of the input string.

   ![Initial Configuration](image)

   As soon as the black space is encountered by `fptr`, it increments the companion pointer (comma or quotation).
Token bpfr

end of lexeme.

Two methods are used -

One buffer scheme

Two buffer scheme

buffer 1

buffer 2

If lexeme is short we use one buffer scheme but if lexeme is long we use two buffer scheme in which the 1st buffer and 2nd buffer are scanned alternately and when the end of current buffer is reached the other buffer is filled.

If the length of lexeme is longer than the length of the buffer then the input can not be scanned completely.

eof is introduced at the end of both the buffers i.e. called sentinel which is used to identify the end of buff.

Code for input buffering

if ( fptr == eof ( buff1) ) /* encounter end of first buffer */

/* Refill buffer 2 */

fptr ++;

else if fptr == eof( buff2) ) /* encounter end of second buffer */

/* Refill buffer 1 */

fptr ++;

else if ( fptr == eof( input) )

return;

/* terminate scanning */

else

fptr ++; /* still remaining input has to be scanned */
2. Specification of tokens -
To specify tokens, regular expressions are used. When a pattern is matched by some regular expression, then tokens can be recognized.

- Things and language -
  String is a collection of finite number of alphabet or letter. The things are synonymously called as words.
  Collection of string is called the language.
  \( |S| \rightarrow \text{length of a string} \quad E \rightarrow \text{empty string} \quad \emptyset \rightarrow \text{empty set of things} \)

- Operation on language -
  Union \((L_1 U L_2)\), Concatenation \(L_1 \cdot L_2\), Kleen Closure \(L^*\), Positive Closure \(L^+\).

A language denoted by regular expression is said to be a regular set or a regular language.

\[ \text{Eq} - \text{Regular Expression for identifier} = \text{letter} (\text{letter+digit})^* \]

*** Unspecified information cannot be represented by regular expression.

3. Recognition of Tokens -

Token Representation:

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Token value</th>
</tr>
</thead>
</table>

Token Type → category of token
Token value → Information regarding token.

The lexical analyzer reads the input program and generates a symbol table for tokens.

<table>
<thead>
<tr>
<th>TOKEN</th>
<th>CODE</th>
<th>VALUE</th>
<th>TOKEN</th>
<th>CODE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>1</td>
<td>-</td>
<td>! =</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>else</td>
<td>2</td>
<td>-</td>
<td>(</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>while</td>
<td>3</td>
<td>-</td>
<td>)</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>for</td>
<td>4</td>
<td>-</td>
<td>+</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>identifier</td>
<td>5</td>
<td>Prt to symbol table</td>
<td>-</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>constant</td>
<td>6</td>
<td>Prt to symbol table</td>
<td>=</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>&lt;</td>
<td>7</td>
<td>1</td>
<td>if ((a&lt;10))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>7</td>
<td>2</td>
<td>i=(i+2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>7</td>
<td>3</td>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>7</td>
<td>4</td>
<td>i=(i-2);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our lexical analyzer will generate the following token stream:
1, (8,4), (5,100), (7,1), (6,105), (8,2), (5,107), 1b, (5,107), (3,1), (6,110), 2, (5,107),
10, (5,107), (9,2), (6,110).

Corresponding symbol table for identifiers and constants will be:

<table>
<thead>
<tr>
<th>LOCATION COUNTER</th>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>IDENTIFIER</td>
<td>a</td>
</tr>
<tr>
<td>105</td>
<td>CONSTANT</td>
<td>10</td>
</tr>
<tr>
<td>107</td>
<td>IDENTIFIER</td>
<td>l</td>
</tr>
<tr>
<td>110</td>
<td>CONSTANT</td>
<td>2</td>
</tr>
</tbody>
</table>

4. Block schematic of lexical analyzer -

Input buffer

LEXEME

Lexical Analyzer

FINITE STATE MACHINE

FINITE AUTOMATA SIMULATOR

Patterns

PATTERN MATCHING ALGORITHM

TOKENS

5. Transition diagram for Identifier - R.E. = letter (letter + digit)*

start

Transition diagram for constant -

R.E = digit* + (digit* (.) digit*) + (digit* (.) digit* E (+/-) digit*)

start
LEX (lexical Analyzer Generator) -

It is a UNIX utility. It is used to design the regular

expressions for corresponding tokens in a lexical task.

LEX is used to create specification file with .l (dot l) extension.

LEX converted .l to .c where .c is a C program which is actually

a lexical analyzer program.

\[
\text{LEX specification} \rightarrow \text{LEX} \rightarrow \text{lex.yy.c}
\]

\[
\text{lex.yy.c} \rightarrow \text{C compiler} \rightarrow \text{executable program}
\]

\[
\text{Input String} \rightarrow \text{.out} \rightarrow \text{Stream of tokens}
\]

- (x.1) Specification file stores the regular expressions for the tokens
- lex.yy.c consists of tabular representation of the transition diagram.

\[
\text{LEX program consists of three parts -}
\]

\[
\%\%
\text{DECLARATION SECTION}
\]

\[
\%\%
\]

\[
\%\%
\text{RULE SECTION}
\]

\[
\%\%
\text{AUXILIARY PROCEDURE SECTION}
\]

- Declaration section -> declaration of variable contents
- Rule section -> consists of regular expression with associated action
- \[
\text{eg.} \quad R_1 : \text{action}
\]

Auxiliary procedure section -> Required procedures are defined.
```
Eq 6:

```C

```C

"Rama"
"Sita"
"Geeta"
"Neeta"

printf("In Noun\n");

"Sings"
"dances"

printf("In Verb\n");

main()

yylex(); \rightarrow Routiner, defined in yy_lex.yy.c program.

\n
\n
int yywrap() \rightarrow called when scanner encounters end of file.

\n
\n
\n
\n
return 1;

Following commands are used to run the lex program %I in UNIX:

```C

$ lex %I
$ cc len yy.c
$ a.out

After entering this commands a blank space for entering input
gets available. Thus we can give some valid input:

Rama eats
Noun
Verb

Then press either control+c or control+d to come out of the output

mycompanion
LEX Actions:

1. BEGIN - It indicates the start state.
2. ECHO - It emits the input as it is.
3. yytext - When lexer matches or recognizes the token from input.
   tokens then the lexeme is stored in a null terminated string called yytext.
4. yyin - Standard input file that stores input source program.
5. yylen - Stores the length of the input string.
6. yylen()
7. yywrap()

Design of Lexical Analyzer Generator -
- Two approaches:
  - Pattern matching using NFA
  - Using DFA for lexical analyzers

Pattern matching using NFA:

Advantages of lexical analyzers:

1. Lexical analysis and syntax analysis are separated out which reduces the burden on parsing phase.
2. Compile efficiency get increased due to separation.

Issues of lexical analyzers:

1. Ambiguity of words (same word can have different meaning)
2. Hookahind (identifying limits for sentence boundaries, e.g., Dr., Mr.)