Practical No. 1

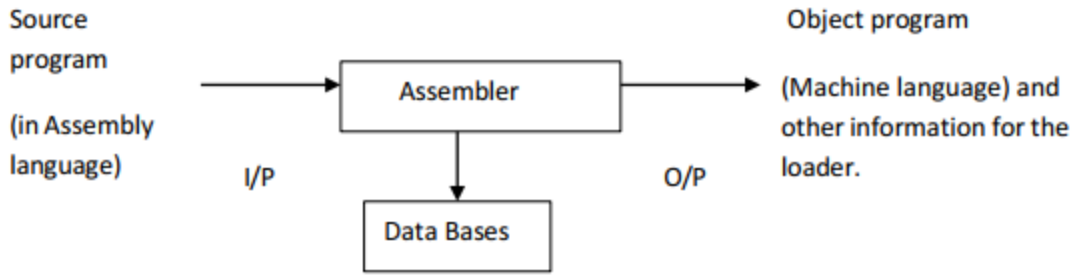
TITLE : To design a Pass-1 of two Pass assembler

PROBLEM STATEMENT : Design suitable data structures and implement pass-I of a two-pass assembler for pseudo- machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives.

OBJECTIVE : Design suitable data structures and develop a subset of an assembler
Subset should consist of a few instructions from each category and few assembler directives.

THEORY : An assembler is a computer program for translating assembly language essentially, a mnemonic representation of machine language — into object code.

To make writing and reading programs written in machine language more convenient, mnemonic(symbol) were used for each machine instruction. These mnemonics were then later translated into machine language. Such a mnemonic language is called **assembly language**. Assemblers are program to automatically translate assembly language into machine language.



Other than producing the machine language, the assembler must also produce the information for the loader to use eg. externally defined symbols must be noted and passed to the loader.

For translation purpose assembler must do the following:

1. Generate instructions:
    a. Evaluate the mnemonic in the operation field to produce its machine code.
    b. Evaluate the subfields – find the value of each symbol, process literals, and assign addresses.
2. Process pseudo ops.

These tasks can be grouped into two passes or sequential scans over the input; associated with each task are one or more assembler modules.

Pass I : Purpose – define symbols and literals.

1. Determine length of machine instructions.
2. Keep track of Location Counter (LC).

3. Remember values of symbols until pass 2.
4. Process some psuedo ops, e.g., EQU,DS
5. Remember literals.

Data structures:

Pass I data bases:

1. Input source program.
2. A Location Counter (LC), used to keep track of each instruction's location.
3. A machine operation table (MOT), that indicates the symbolic mnemonic for each and its length.
4. A pseudo – operation table (POT) , that indicates the symbolic mnemonic and action to be taken for each pseudo _ op in pass 1.
5. A symbol table (ST), that is used to store each literal encountered and its corresponding value.
6. A literal table (LT) that is used to store each literal encountered and its corresponding assigned location.
7. A copy of the input, to be passed to pass 2. This is to be stored in a secondary storage device.

Algorithm of Pass I:

1. loc_cntr:=0;(default value)
   pooltab_ptr:=1;POOLTAB[1]:=1;
   littab_ptr:=1;
2. While next statement is not an END statement
   a. If label is present then
      this_label:=symbol in label field;
      Enter(this_label, loc_cntr) in SYMTAB.
   b. If an LTORG statement then
      (i) Process literals LITTAB[POOLTAB[pooltab_ptr]]…LIITLAB [littab_ptr-1] to allocate memory and put the address in the address field.Update loc_cntr accordingly.
      (ii) Pooltab_ptr :=pooltab_ptr+1;
      (iii)POOLTAB[pooltab_ptr]:littab_ptr;
   c. If a STSRT or ORIGIN statement then
      loc_cntr :=value specified in operand field;
   d. If an EQU statement then
      (i) this_addr :=value of <address spec>;
      (ii) Correct the symtab entry for this_label to (this_label,this_addr).
   e. If a declaration statement then

(i) code :=code of the declaration statement;

(ii) size :=size of memory area required by DC/DS.

(iii) loc_cntr :=loc_cntr+size;

(iv) Generate IC '(DL,code)…'.

f. If an imperative statement then

(i) code:=machine opcode from OPTAB;

(ii) loc_cntr:=loc_cntr+instruction length from OPTAB;

(iii) If operand is a literal then

this_literal:=literal in operand field;

LITTAB[littab_ptr]:=this_literal;

littab_ptr:=littab_ptr+1;

Else(i.e. operand is a symbol)

this_entry:=SYMTAB entry number of operand;

Generate IC '(IS,code)(S,this_entry)';

3. (processing of END statement)

(a) Perform step 2(b).

(b) Generate IC '(AD,02)'.

(c) Go to Pass II.

CONCLUSION:

FAQs:

1) What are different jobs performed by pass1 and pass 2 of assembler?

2) What is LC? When value of LC is updated?

3) Explain general format of

a) intermediate code generation and

b) machine language instruction.

4) How DC statement are processed?

5) How information is entered in LTTAB and POOLTAB?

6) What are pros and cons of single pass assembler?

7) What are different jobs performed by pass 2 of assembler?

8) How DS statement are processed?

9) How information is entered in SYMBTAB?

10) Enlist the data structures used in design of assembler?

Practical No. 2

TITLE: To design a Pass-2 of two pass assembler.

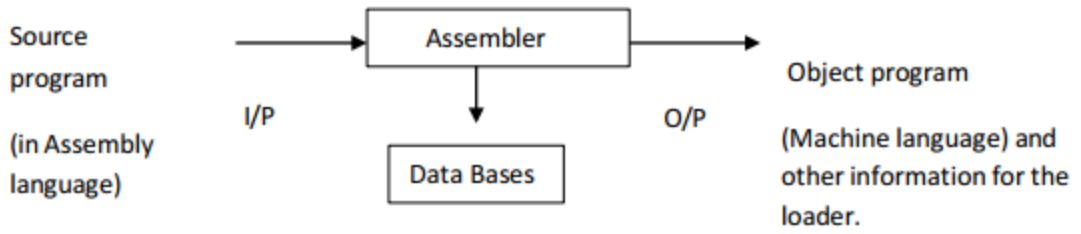
PROBLEM STATEMENT: Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment.

OBJECTIVE: Implementation of pass-II of a two-pass assembler. The output of assignment-1 should be input for this assignment.

THEORY:

An assembler is a computer program for translating assembly language essentially, a mnemonic representation of machine language into object code.

To make writing and reading programs written in machine language more convenient, mnemonic (symbol) were used for each machine instruction. These mnemonics were then later translated into machine language. Such a mnemonic language is called **assembly language**. Assemblers are program to automatically translate assembly language into machine language.



or sequent... ...ciated with each

These tasks can be grouped into two passes task are one or more assembler modules.

Pass II: Purpose – general object program.

1. Look up value of symbols.
2. Generate instructions.
3. Generate data (DS, DC and literals)
4. Process pseudo ops.

Data structures:

Pass II data bases:

1. Copy of source program input to Pass I.
2. Location Counter (LC).

3. MOT, that indicates for each instruction (a) symbolic mnemonic;(b) length;(c) binary machine opcode, and (d) format .
4. POT, that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in Pass II.
5. The ST prepared by Pass I, containing each label and its corresponding value.
6. A Base table (BT), that indicates which registers are currently specified as specified as base registers by USING pseudo-ops and what are the specified contents of these registers.
7. Work- space that is used to hold each instruction as its various parts are being assembled together.
8. Output of Pass II.

Instruction Format:

Algorithm of Pass II:

1. code_area_address:=address of code_area
   pooltab_ptr :=1;
   loc_cntr := 0;
2. While next statement is not an END statement
   (a) clear machine_code_buffer;
   (b) If an LTORG statement then
      i. Process literals LITTAB [POOLTAB [pooltab_ptr]]...LITTAB[lit_tab_ptr-1] to allocate memory and put address in the address Field. Update loc_cntr accordingly.
      ii. Pooltab_ptr:=pooltab_ptr+1;
      iii. POOLTAB[pooltab_ptr]:=littab_ptr;
   (c) If a STRAT or ORIGIN statement then
      i. Loc_cntr := value specifid in operand field;
      ii. Size:=0;
   (d) If a declaration statement then
      i. If a DC statement then assemble the constant in machine_code_buffer;
      ii. Size:=size of memory area required by DC/DS;
   (e) If an imperative statement then
      i. Get operand address from SYMTAB or LITTAB
      ii. Assemble instruction in machine_code_buffer.
      iii. Size:=size of instruction;
   (f) If size $\neq 0$ then

Move contents of machine_code_buffer to the address

code_area_address+loc_cntr;

Loc_cntr:=loc_cntr+size

3. (Processing of END statemnen)

a) Perform step 2(b) & 2(f).

b) Write Code_area into output file.

CONCLUSION:

FAQs:

1. What are different jobs performed by pass 2 of two assembler?
2. Explain general format of machine language instruction.
3. Enlist the data structures used in design of pass II assembler?
4. What is the difference between one pass and two pass assembler?
5. Why the task of assembler is dived into two passes?
6. How the forward reference is handled by one pass assembler?

Practical No. 3

TITLE : To design Pass I of Two Pass macro Processor

PROBLEM STATEMENT : Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java

OBJECTIVE : Design suitable data structures and develop a subset of a macro processor.

THEORY : A **macro** is a unit of specification for program generation through expansion. It consists of a name , a set of formal parameters and a body of code. The use of macro name with a set of actual parameters is replaced by some code generated from its body. This is called **macro expansion**.

A **macro definition** is enclosed between a macro header statement and a macro –end statement. These are typically located at the start of the program. A macro definition consists of

1. A macro prototype statement: it declares the name of macro and the names & kinds of its parameter.
2. One or more model statements: it is a statement from which an assembly language statement may be generated during macro expansion
3. Macro preprocessor statements: it is used to perform auxiliary functions during macro expansion.

The macro prototype statement has the following format.

<macro name>[<formal parameter spec.>]

<formal parameter spec.> has the syntax

&<parameter name>[<parameter kind>]

Types of Parameters:

Positional parameters:- a positional parameter is written as &<parameter name> e.g. &SAMPLE.

Keyword parameter:- < parameter name> is an ordinary string and <parameter kind> is the string '='

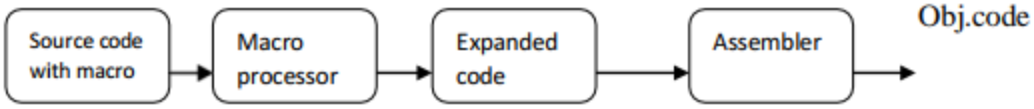
Example:

```
MACRO
INCR  &MEM_VAL, &INCR_VAL, &REG
MOVER       &REG, &MEM_VAL
ADD   &REG , &INCR_VAL
MOVEM       &REG, &MEM_VAL
MEND
```

- **Macro head & macro end** - MACRO & MEND
- **Prototype statement** - indicates that 3 parameter MEM_VAL, INCR_VAL, REG exists for the macro
- **Model statement-** statements with the operation code MOVER, ADD & MOVEM

Macro preprocessor:

The macro preprocessor accepts an assembly program containing definition and calls and translates it into an assembly program which does not contain any macro definitions or calls.



Data structures used:-

Macro name table (MNT) is designed to hold the names of all macros defined in a program. A macro name is entered in this table when a macro definition is processed along with its MDT counter.

Macro definition table(MDT) The body of macro is stored in a table called the macro definition table (MDT) for use during macro expansion.

Parameter name table(PNT) A list of actual parameters is designed to hold the formal parameters during the expansion of a macro call and macro definition.

Algorithm

1. mnt_counter =1
   mdt_counter=1
   mec = 1
2. while next statement is not End
   i. if word is MACRO
      write macroname in MNT along with mdt_counter
      mnt_counter++
      mdt_counter++
      make argument list array of parameters
      while next word is not endm
      if argument list array to find the positional value and store
      that in mdt preceding with #
      else
      write word in MDT

ii if word present in mnt (i.e macro call)
mdt_counter=1
while mec>0
if next word is 'ENDM'
if stack is not empty

pop the top of stack & set new value of mdt_counter

me--

else

if word present in mnt (i.e. nested call)

push argument list on stack & (mdt_counter+1)

make argument list array of new macro which is called

mdt_counter = mdt_counter in mnt stored in front of macroname.

mec++

else

if word is present in argument list array(is arg.), store positional value

else

3. Store the word in output file.

CONCLUSION:

FAQ:

1. Describe macro processor.
2. Compare MACRO with Subroutine?
3. Explain what is MNT and its content?
4. Explain what is MDT and its content?
5. Explain data structures used in design of macro processor?
6. What are positional and keyword parameters?
7. What is lexical substitution, explain its types?
8. What are pros and cons of single pass macro processor?
9. What is macro?

Practical No.4

TITLE : To design Pass II of Two Pass macro Processor.

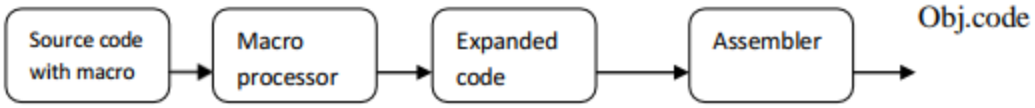
PROBLEM STATEMENT : Write a Java program for pass-II of a two-pass macro-processor. The output of assignment-3 (MNT, MDT and file without any macro definitions) should be input for this assignment.

OBJECTIVE : Design suitable data structures and develop a subset of a macro processor.

THEORY :

Macro preprocessor:

The macro preprocessor accepts an assembly program containing definition and calls and translates it into an assembly program which does not contain any macro definitions or calls.



A macro call leads to **macro expansion.** During macro expansion ,the macro call statement is replaced by a sequence of assembly statements.

Conditional Macro Expansion

Conditional Macro Expansion is a powerful tool to control which instructions are to be generated by the macro-processor to be the input to the assembler. Conditional assembly takes place during the code generation phase and only then it is assembled.

Conditional directives AIF and AGO are used for such purposes.

Syntax and semantics of the 2 are :-

LABEL: AIF (condition) action

Where

1. Label may be a “permanent” label which will also occur in the code passed to the assembler or it may be an expansion time label. In the latter case it is distinguished by putting a mark say “%” before the actual label.

2. The condition is a Boolean expression and can involve logical comparison among variables, constants, macro arguments etc. However, the truth value has to be evaluated during macro expansion time constants could be used. Again a distinction has to be made between normal variables (or constants) and expansion time variables or constants. Any expansion time variables can be preceeded by a special symbol say “&”.

3. The action may be branching to some label or execution of a single instruction if the condition is evaluated to be “true”, which can be interpreted by the conditional macro processor.

LABEL : AGO label of a branch target

Where, conditional macro processor unconditionally branches to the target label.

The conditional macro processor should have the capability of evaluating logic and arithmetic expressions involving expansion time variables and constants as well as assigning values to expansion time variables. Depending on such capabilities, these processors require a set of commands or verbs which are to be interpreted and evaluated.

Algorithm

3. mnt_counter =1
   mdt_counter=1
   mec = 1
4. while next statement is not End
   i. if word is MACRO
      write macroname in MNT along with mdt_counter
      mnt_counter++
      mdt_counter++
      make argument list array of parameters
      while next word is not endm
      if argument list array to find the positional value and store
      that in mdt preceding with #
      else
      write word in MDT

ii if word present in mnt (i.e macro call)
   mdt_counter=1
   while mec>0
   if next word is ‘ENDM’
   if stack is not empty
   pop the top of stack & set new value of mdt_counter
   me--
   else
   if word present in mnt (i.e. nested call)
   push argument list on stack & (mdt_counter+1)
   make argument list array of new macro which is called

mdt_counter = mdt_counter in mnt stored in front of macroname.

mec++

else

if word is present in argument list array(is arg.), store positional value

else

3. Store the word in output file.

CONCLUSION:

FAQ:

10. Explain what is macro expansion?
11. Explain the work of pass II in two pass macro processor?
12. Explain the work of pass I in two pass macro processor?
13. What are nested macro calls?
14. Enlist the tasks involved in macro expansion.

Practical No. 5

TITLE : Lexical analyzer for sample language using LEX. (Part I)

PROBLEM STATEMENT : Write a program using Lex specifications to implement lexical analysis phase of compiler to generate tokens of subset of 'Java' program.

OBJECTIVE :

- To understand first phase of compiler: Lexical Analysis.
- To learn and use compiler writing tools.
- Understand the importance and usage of LEX automated tool.

THEORY :

Introduction:

LEX stands for Lexical Analyzer.LEX is a UNIX utility which generates the lexical analyzer. LEX is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message. Lex and C are tightly coupled. A lex file (files in Lex have the .l extension eg: first.l) is passed through the lex utility, and produces output files in C (lex.yy.c). The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first.l These file is then compiled object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens as show in fig 1.1.

To generate a lexical analyzer two important things are needed. Firstly it will need a precise specification of the tokens of the language. Secondly it will need a specification of the action to be performed on identifying each token.

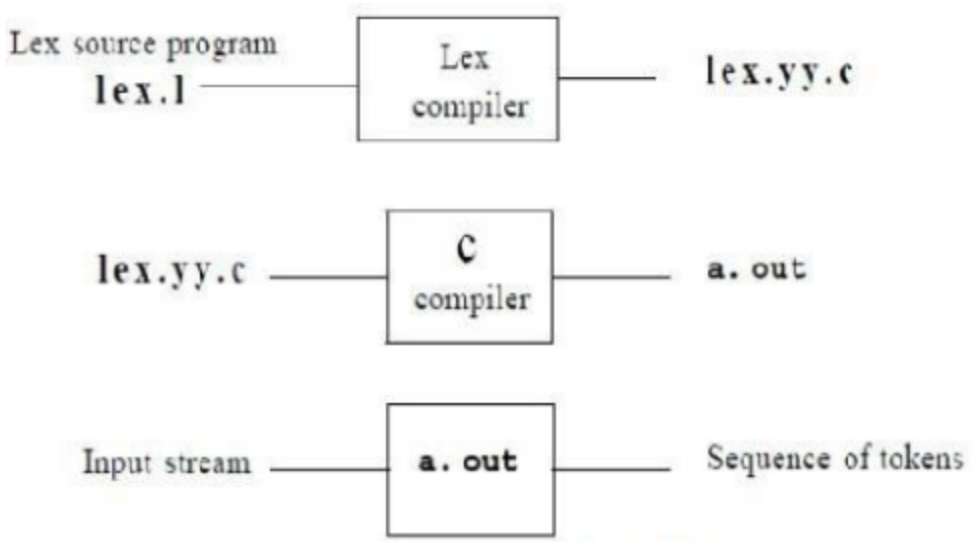


Fig 1.1: Creating a lexical analyzer with LEX

1. **LEX Specifications:**

The Structure of lex programs consists of three parts:

```
%{
        Definitions Section
%}
%%
        Rules Section
%%
        Auxiliary Function
```

➢ **Definition Section :**

The Definition Section includes declarations of variables, start conditions regular definitions, and manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14).

There are three things that can go in the definitions section:

- C code: Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.
- Definitions: A definition is very much like # define cpp directive. For example

letter [a-zA-Z]+

digit [0-9]+

These definitions can be used in the rules section: one could start a rule

{letter}{printf("n Wordis = %s",yytext);}

- State definitions: If a rule depends on context, it"s possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given.

➢ **Rule Section:**

Second section is for translation rules which consist of regular expression and action with respect to it. The translation rules of a Lex program are statements of the form:

p1 {action 1}
p2 {action 2}
p3 {action 3}
... ...
... ...
pn {action n}

Where, each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

➢ **Auxiliary Function(User Subroutines):**

Third section holds whatever auxiliary procedures are needed by the actions. If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main as follow:

```
int main()
{
    yylex();
    return 0;
}
```

In this section we can write a user subroutines its option to user e.g. yylex() is a unction automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

2. Built - in Functions:

No.	Function	Meaning
1	yylex()	The function that starts the analysis. It is automatically generated by Lex.
2	yywrap()	This function is called when end of file (or input) is encountered. If yywrap() returns 0, the scanner continues scanning, while if it returns 1 the scanner returns a zero token to report the end of file.
3	yyless(int n)	This function can be used to push back all but first „n" characters of the read Token.
4	yymore()	This function tells the lexer to append the next token to the current token.
5	yyerror()	This function is used for displaying any error message.

3. Built - in Variables:

No.	Variables	Meaning
1	yyin	Of the type FILE*. This point to the current file being parsed by the lexer. It is standard input file that stores input source program.
2	yyout	Of the type FILE*. This point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
3	yytext	The text of the matched pattern is stored in this variable (char*) i.e. When lexer matches or recognizes the token from input token the lexeme stored in null terminated string called yytext. OR This is global variable which stores current token
4	yyleng	Gives the length of the matched pattern. (yyleng stores the length or number of character in the input string)The value in yyleng is same as strlen() functions.
5	yylineno	Provides current line number information. (May or may not be supported by the lexer.)
6	yylval	This is a global variable used to store the value of any token.

1. Regular Expression:

No.	RE	Meaning
1	a	Matches a
2	abc	Matches abc
3	[abc]	Matches a or b or c
4	[a-f]	Matches a,b,c,d,e or f
5	[0-9]	Matches any digit
6	X^+	Matches one or more of x
7	X*	Matches zero or more of x
8	[0-9]+	Matches any integer
9	(…)	Grouping an expression into a single unit
10	|	Alteration (or)
11	(b|c)	Is euivalent to [a-c]*
12	X?	X is optional (0 or 1 occurrence)
13	If(def)?	Matches if or ifdef
14	[A-Za-z]	Matches any alphabetical character
15	.	Matches any character except new line
16	\.	Matches the . character
17	\n	Matches the new character
18	\t	Matches the tab character
19	\\	Matches the \ character
20	[\t]	Matches either a space or tab character
21	[^a-d]	Matches any character other than a,b,c and d
22	$	End of the line

2. **Steps to Execute the program:**

   $ lex filename.l (eg: first.l)

   $cc lex.yy.c–ll or gcc lex.yy.c–ll

   $./a .out

Algorithm:

1. Start the program.
2. Lex program consists of three parts.
   a. Declaration %%
   b. Translation rules %%
   c. Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form
   a. P1 {action}
   b. P2 {action}
   c. …
   d. …
   e. Pn {action}
5. Write a program in the vi editor and save it with .l extension.
6. Compile the lex program with lex compiler to produce output file as lex.yy.c. eg $ lex filename.l

   $ cc lex.yy.c -ll
7. Compile that file with C compiler and verify the output.

CONCLUSION:

_____

_____

_____

FAQ

1. What Is A Compiler?
2. Why Lexical And Syntax Analysers Are Separated Out?
3. What is LEX?
4. What are different phases of compilation?

Practical No. 6

TITLE : Lexical analyzer for sample language using LEX. (Part II)

PROBLEM STATEMENT : Write a program using Lex specifications to implement lexical analysis phase of compiler to count no. of words, lines and characters of given input file.

OBJECTIVE :

- To understand first phase of compiler: Lexical Analysis.
- To learn and use compiler writing tools.
- Understand the importance and usage of LEX automated tool.

THEORY :

Introduction:

Lex and yacc help you write programs that transform structured input. This includes an enormous range of application-anything from a simple text search program that looks for patterns in the input file to a C compiler that transform a source program into optimized object code.

In programs with structured input, two tasks that occur over and over are dividing the input into meaningful units, and then discovering the relationship among the units. For the text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that don't. For a C program the units are variable names, constants, string, operator, punctuation and so forth. This division into units (which are usually called tokens) is known as lexical analysis, or lexing for short. Lex helps you by taking a set of description of possible tokens and producing a C routing, which we call a lexical analyzer, or a lexer, or a scanner for short, that can identify those tokens. The set of descriptions you give to lex is called a lex specifications.

The token descriptions that lex uses are known as regular expressions, extended versions of the familiar patterns used by the grep and egrep commands. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A lex lexer is almost always faster than a lexer than you might write in C by hand.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statement, declaration, blocks, and procedures in the program. This task is known as parsing and the list of rules that define the relationships that the program understand is a grammar. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules. A yacc parser is generally not as fast as a parser you could write by hand, but the ease in writing

and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway.

When a task involves dividing the input into units and establishment some relationship among those units, you should think of lex and yacc. (A search program is so simple that it doesn't need to do any parsing so it uses lex but doesn't need yacc.)

a) The Simplest Lex Program

This lex program copies its standard input to its standard output:

```
%%
. /\n       ECHO;
%%
```

It acts very much like the UNIX cat command run with no arguments.

Lex automatically generates the actual C program code needed to handle reading the input file and sometimes, as in this case, writing the output as well.

Whenever you use lex and yacc to build departs of your program or to build tools to aid you in programming, once you master them they will prove their worth many times over by simplifying difficult input handling problems, providing easier "tinkering" to get the right semantics for your program.

b) Recognizing words with Lex

This first section, the definition section, introduces any initial C program code we want copied into the final program. This is especially important if, for example, we have header files that must be included for code later in the file to work. We surround the C code with the special delimiters "% {" and "%}". Lex copies the material between "% {" and "%}" directly to the generated C file, so you may write any valid C code here.

The next section is the rules section. 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 UNIX-style regular expression, a slightly extended version of the same expressions used by tools such as grep, sed and ed.

Example: -

```
[\t] +              /*ignore whitespace */;
```

The last two rules are:

```
[a-ZA-Z] + {printf("%s : is not a verb\n",yytext);}
.|\n    {ECHO; /* normal default anyway*/}
```

The two that make our lexer work are:

Lex patterns only match a given input character or string once.

Lex executes the action for the longest possible match for the current input.

The final section is the user subroutines section, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code. We have included a main () program.

```
%%
Main ()
{
 yylex();
}
```

The lexer produced by lex is a C routine called yylex (), so we call it. Unless the action explicit return statement, yylex () won't return until it has processed the entire input.

c) Parser-Lexer Communication

When you use a lex scanner and a yacc parser together, the parser is the higher-level routine. It calls the lexeryylex () whenever it needs a token from the input. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of yylex().

4. Steps to Execute the program:

$ lex filename.l (eg: first.l)

$cc lex.yy.c–ll or gcc lex.yy.c–ll

$./a .out

Algorithm:

1. Start the program.
2. Lex program consists of three parts.
   a. Declaration %%
   b. Translation rules %%
   c. Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form
   a. P1 {action}
   b. P2 {action}
   c. …
   d. …
   e. Pn {action}
5. Write a program in the vi editor and save it with .l extension.

6. Compile the lex program with lex compiler to produce output file as lex.yy.c. eg $ lex filename.l
   $ cc lex.yy.c -ll
7. Compile that file with C compiler and verify the output.

CONCLUSION:

_____

_____

_____

FAQ

1. What are Tokens in Compiler Design?
2. What is YACC?
3. What are different phases of compilation?
4. When expression sum=3+2 is tokenized then what is the token category of 3.

Practical No. 7

TITLE: Parser for sample language using YACC.

PROBLEM STATEMENT : Write a program using YACC specifications to implement syntax analysis phase of compiler to validate type and syntax of variable declaration in Java

OBJECTIVE:

- To understand Second phase of compiler: Syntax Analysis.
- To learn and use compiler writing tools.
- Understand the importance and usage of YACC automated tool

THEORY:

Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for "Yet Another Compiler Complier".

File in which parser generated is with .y extension. e.g. parser.y, which is containing YACC specification of the translator. After complete specification UNIX command. YACC transforms parser.y into a C program called y.tab.c using LR parser. The program y.tab.c is automatically generated. We can use command with –d option as

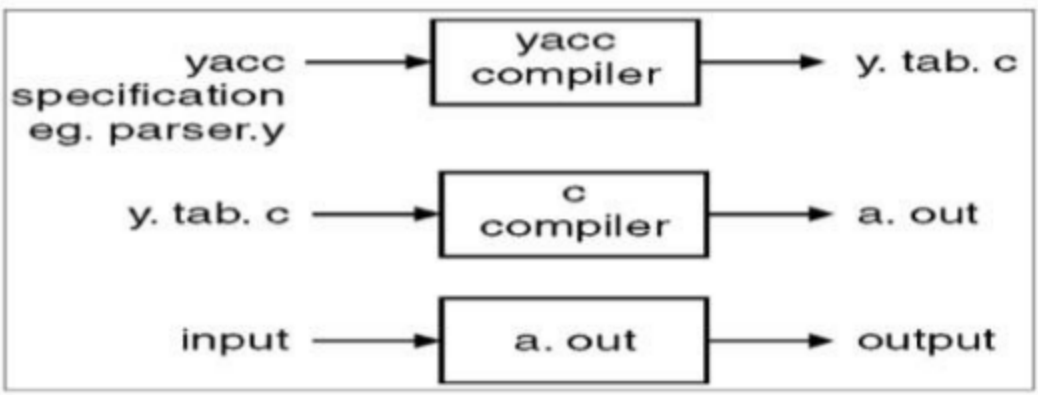
yacc –d parser.y

By using –d option two files will get generated namely y.tab.c and y.tab.h. The header file y.tab.h will store all the token information and so you need not have to create y.tab.h explicitly.

The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. By compiling y.tab.c with the ly library that contains the LR parsing program using the command.

cc y tab c – ly

we obtain the desired object program a out that perform the translation specified by the original program. If procedure is needed, they can be compiled or loaded with y.tab.c, just as with any C program.



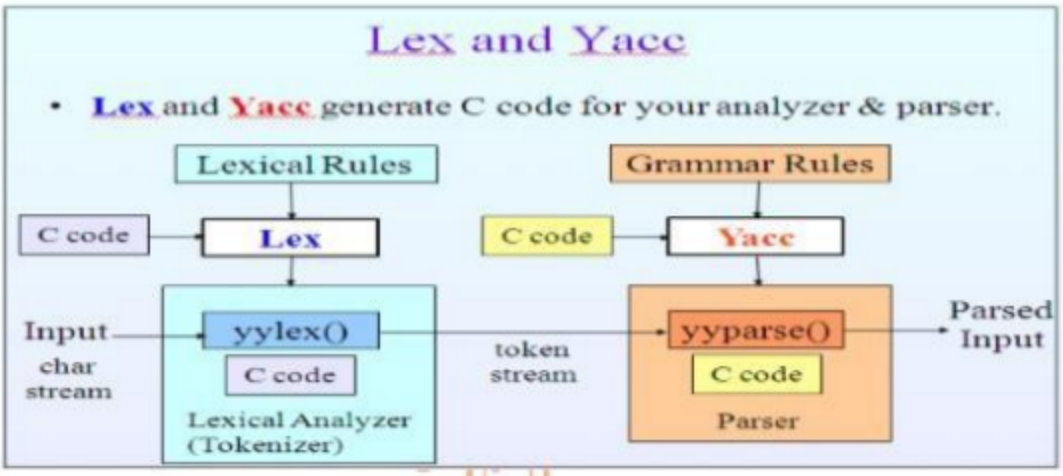


Fig: 3.1 YACC: Parser Generator Model

LEX recognizes regular expressions, whereas YACC recognizes entire grammar. LEX divides the input stream into tokens, while YACC uses these tokens and groups them together logically. LEX and YACC work together to analyze the program syntactically. The YACC can report conflicts or ambiguities (if at all) in the form of error messages.

1. YACC Specifications:

   The Structure of YACC programs consists of three parts:

```
%{
      Definitions Section
 %}
 %%
    Rules Section (Context Free Grammar)
 %%
      Auxiliary Function
```

- **Definition Section:**

  The definitions and programs section are optional. Definition section handles control information for the YACC-generated parser and generally set up the execution environment in which the parser will operate.

  Declaration part:

  In declaration section, %{ and %} symbol used for C declaration. This section is used for definition of token, union, type, start, associativity and precedence of operator. Token declared in this section can then be used in second and third parts of Yacc specification.

- **Translation Rule Section:**

In the part of the Yacc specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing:

<left side> → <alt 1> | <alt 2> | … <alt n>

Would be written in YACC as

```
<left side>  : <alt 1> {action 1}
             | <alt 2> {action 2}
             … … … … …
             | <alt n> {action n}
             ;
```

In a Yacc production, unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals. A quoted single character, e.g. 'c', is taken to be the terminal symbol c, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for ' c' to the parser, as an integer). Alternative bodies can be separated by a vertical bar, and a semicolon follows each head with its alternatives and their semantic actions. The first head is taken to be the start symbol.

A Yacc semantic action is a sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the nonterminal of the head, while $i refers to the value associated with the ith grammar symbol (terminal or nonterminal) of the body. The semantic action is performed whenever we reduce by the associated production, so normally the semantic action computes a value for $$ in terms of the $i's. In the Yacc specification, we have written the two E-productions.

E E + T/T

and their associated semantic action as:

```
exp  :  exp „+" term        {$$ = $1 + $3;}
     |  term
     ;
```

In above production exp is $1, „+" is $2 and term is $3. The semantic action associated with first production adds values of exp and term and result of addition copying in $$ (exp) left hand side. For above second number production, we have omitted the semantic action since it is just copying the value. In general {$$ = $1;} is the default semantic action.

- **Supporting C-Routines Section:**

The third part of a Yacc specification consists of supporting C-routines. YACC generates a single function called yyparse(). This function requires no parameters and returns either a 0 on success, or 1 on failure. If syntax error over its return 1.The special function yyerror() is called when YACC

encounters an invalid syntax. The yyerror() is passed a single string (char) argument. This function just prints user defined message like:

```
yyerror (char err)
{
    printf ("Divide by zero");
}
```

When LEX and YACC work together lexical analyzer using yylex () produce pairs consisting of a token and its associated attribute value. If a token such as DIGIT is returned, the token value associated with a token is communicated to the parser through a YACC defined variable yylval. We have to return tokens from LEX to YACC, where its declaration is in YACC. To link this LEX program include a y.tab.h file, which is generated after YACC compiler the program using –d option.

2. **Built-in Functions**:

Function	**Meaning**
yyparser ()	This is a standard parse routine used for calling syntax analyzer for given translation rules. When yyparse() is call, the parser attempts to parse an input stream.
yyerror()	This function is used for displaying any error message when a yacc detects a syntax error

3. **Built-in Types:**

Type	**Meaning**
%token	Used to declare the tokens used in the grammar. Eg.:- %token NUMBER
%start	Used to declare the start symbol of the grammar. Eg.:- %start S Where S is start symbol
%left	Used to assign the associatively to operators. Eg.: %left „+" „-„ -Assign left associatively to + & – with lowest precedence. %left „*" „/„ -Assign left associatively to * & / with highest precedence.
%right	Used to assign the associatively to operators. Eg.: %right „+" „-„ -Assign right associatively to + & – with lowest precedence %right „*" „/„ -Assign right associatively to * & / with highest precedence.
%nonassoc	Used to unary associate. Eg.:- %nonassoc UMINUS
%prec	Used to tell parser use the precedence of given code. Eg.:- %prec UMINUS
%type	Used to create the type of a variable. Eg.:- %type <name of any variable> exp
%union	Token data types are declared in YACC using the YACC declaration % union, like this : % union { char str ; int num ; }

4. Special Characters:

Characters	Meanings
%	A line with two percent signs separates the part of yacc grammar. All declarations in definition section start with %, including %{ %},%start, %token, %type, %left, %right, %nonassoc and %union.
$	In action, a dollar sign introduces a value references e.g: $3 value of the third symbol in the rule"s right-hand side.
' '	Literal tokens are enclosed in single quotes. Eg: „+" or „-„ or „*" or „\" etc.
< >	In value references in an action, you can override the value"s defaults type by enclosing the type name in angle brackets.
{ }	The C code in action is enclosed in curly braces
;	Each rule in rules section should end with semicolon, except those that are immediately followed by rule that starts a vertical bar.
|	When two consecutive rules have same left-hand side, the second rule is separated by vertical bar.
:	In rule section, colon is used to separate left-hand side and right-hand side.

5. Steps to Execute the program

$ lex filename.l (eg: cal.l)

$ yacc -d filename.y (eg: cal.y)

$cc lex.yy.c y.tab.c –ll –ly –lm

$./a .out

ALGORITHM:

Write a program to implement YACC for Subset of C (for loop) statement.

LEX program:

1. Declare header files y.tab.h which contains information of the tokens and also declare variable yylval within %{ and %}.
2. End of declaration section with %%
3. Write the Regular Expression for: FOR, OB, CB, SM, CON, EQ, ID, NUM, INC, DEC.
4. If match found for regular expression then write action that store token in yylval where p is pointer declared in YACC and return the valve of token.
5. End rule-action section by %%
6. Subroutines section is optional.

YACC program:

1. Declaration of header files and set flag=0;

2. Declare tokens FOR, OB, CB, SM, CON, EQ, ID, NUM, INC, DEC.
3. End of declaration section by %%
4. State Context Free Grammar for FOR loop in rule section and write appropriate action for same.

```
S   : FOR OPBR E1 SEMIC E2 SEMIC E3 CLBR   { printf("Accepted!");flag=1; }
    ;
E1 : ID EQ ID
   | ID EQ NUM
   ;
E2 : ID RELOP ID
   | ID RELOP NUM
   ;
E3 : ID INC
   | ID DEC
   ;
```

5. End the translation rule section by %%
6. Define main() function to call yyparse() function to parse an input stream
7. Define yyerror() function to displaying any error message when a yacc detects a syntax error.

```
yyerror(const char *msg) {    if(flag==0);    printf("\n\n Syntax is Wrong");  }
```

CONCLUSION:

_____

_____

_____

FAQ

1. What is YACC?
2. What are different phases of compilation?
3. What is Parser?
4. What are the different types of parsing in Compiler Design?