**EX NO: 01 SYMBOL TABLE USING LEX TOOL**

**DATE:**

**AIM:**

To develop a lexical analyzer to recognize a few patterns in C like identifiers, constants, comments, operators etc. and to create a symbol table, while recognizing them using LEX tool.

**ALGORITHM:**

1. Import the required C standard input/output library (#include <stdio.h>).
2. Specify the %option noyywrap directive to disable the yywrap function.
3. Define regular expressions to match various types of tokens. The regular expressions include:
   1. Operators: The code defines a regular expression to match various operators such as ==, <=, >=, <, >, !=, ++, --, &&, ||, !, &, |, <<, >>, ~, ^, =, +=, -=, \*=, %=, /=, +, -, \*, /, and %. When an operator is matched, it prints the matched operator and categorizes it as an "Operator."
   2. Comments: The code defines a regular expression to match C-style single-line comments, starting with // and extending to the end of the line. When a comment is matched, it prints the entire comment string and categorizes it as a "Comment."
   3. Constants: The code defines a regular expression to match numeric constants consisting of one or more digits. When a constant is matched, it prints the matched numeric constant and categorizes it as a "Constant."
   4. Identifiers: The code defines a regular expression to match C-style identifiers, which can start with a letter or underscore and can be followed by letters, digits, or underscores. When an identifier is matched, it prints the matched identifier and categorizes it as an "Identifier."
4. The code includes a section that matches and ignores characters that are not explicitly recognized as tokens. For example, parentheses and curly braces are matched and ignore
5. The code matches newline characters (\n) and ignores them.
6. In the main function, the code sets up the structure of the symbol table and prepares to print token information. It prints a header for the symbol table.
7. The code invokes the yylex() function, which performs tokenization according to the defined regular expressions and actions.
8. As tokens are matched and classified, the code prints each token and its corresponding type (e.g., "Operator," "Comment," "Constant," "Identifier") in a formatted manner.
9. The code prints a separator line to indicate the end of the symbol table and returns 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

%}

%option noyywrap

%%

"=="|"<="|">="|"<"|">"|"!="|"++"|"--"|"&&"|"||"|"!"|"&"|"|"|"<<"|">>"|"~"|"^"|"="|"+="|"-="|"\*="|"%="|"/="|"+"|"-"|"\*"|"/"|"%" {

printf("%s\t\t|\tOperator\n", yytext);

}

"//".\* {

printf("%s\t|\tComment\n", yytext);

}

[0-9]+ {

printf("%s\t\t|\tConstant\n", yytext);

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

printf("%s\t\t|\tIdentifier\n", yytext);

}

[(){}]+ {

}

\n {

}

. {

}

%%

int main() {

printf("\nSYMBOL TABLE\n");

printf("--------------------------------------\n");

printf("Symbol\t\t|\tType\n");

printf("--------------------------------------\n");

yylex();

printf("--------------------------------------\n");

return 0;

}

**prog.c**

int main()

{

//comment

int a = 5,b = 10;

if (a > b){

printf("a>b");

}

else

{

printf("b>a");

}

return 0;

}

**OUTPUT:**

D:\CD\_LAB>flex ex1.l

D:\CD\_LAB>gcc lex.yy.c -o ex1

D:\CD\_LAB>ex1 < prog.c

SYMBOL TABLE

--------------------------------------

Symbol | Type

--------------------------------------

int | Identifier

main | Identifier

//comment | Comment

int | Identifier

a | Identifier

= | Operator

5 | Constant

b | Identifier

= | Operator

10 | Constant

if | Identifier

a | Identifier

> | Operator

b | Identifier

printf | Identifier

a | Identifier

> | Operator

b | Identifier

else | Identifier

printf | Identifier

b | Identifier

> | Operator

a | Identifier

return | Identifier

0 | Constant

-------------------------------------

**RESULT:**

Hence successfully implemented a Lexical analyzer to recognize a few patterns in C like identifiers, constants, comments, operators etc. and to create a symbol table, while recognizing them using LEX tool.

**EX NO: 02 SIMPLE PROGRAMS USING LEX TOOL**

**DATE:**

**a) To recognize floating point numbers**

**AIM:**

To develop a lex program to recognize floating point numbers.

**ALGORITHM:**

1. Initialize the Lexical Analyzer with the %{ ... %} section, which includes necessary header files and definitions.
2. Set the option %option noyywrap to disable the yywrap function.
3. Define the lexical rules within the %% section. These rules match different patterns in the input.
   1. [-+]?[0-9]+"."[0-9]\*: This rule matches floating-point numbers with an optional sign, such as -76.5. If a match is found, print "Floating-Point Number" along with the matched text and exit the program.
   2. [-+]?[0-9]+"E"[-+]?[0-9]+: This rule matches floating-point numbers in scientific notation, such as "3.14E+10". If a match is found, print "Floating-Point Number" along with the matched text and exit the program.
   3. [-+]?[0-9]+"e"[-+]?[0-9]+: This rule is similar to the previous one but matches floating-point numbers with "e" notation, such as "1.5e-3". If a match is found, print "Floating-Point Number" along with the matched text and exit the program.
   4. [-+]?[0-9]+: This rule matches integers with an optional sign, such as "-42". If a match is found, print "Not a Floating-Point Number" along with the matched text and exit the program.
   5. .: This rule is a catch-all for any other characters. It is configured to ignore them.
4. In the int main() function:
   1. Display a prompt message asking the user to enter a number.
   2. Call the yylex() function to start the lexical analysis process.
   3. Return 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%option noyywrap

%%

[-+]?[0-9]+"."[0-9]\* { printf("Floating-Point Number: %s\n", yytext); exit(0); }

[-+]?[0-9]+"E"[-+]?[0-9]+ { printf("Floating-Point Number: %s\n", yytext); exit(0); }

[-+]?[0-9]+"e"[-+]?[0-9]+ { printf("Floating-Point Number: %s\n", yytext); exit(0); }

[-+]?[0-9]+ { printf("Not a Floating-Point Number: %s\n", yytext); exit(0); }

. { //Ignore other characters }

%%

int main() {

printf("Enter a number: ");

yylex();

return 0;

}

**OUTPUT:**

D:\CD\_LAB>flex ex2a.l

D:\CD\_LAB>gcc lex.yy.c -o ex2a

D:\CD\_LAB>ex2a

Enter a number: -8.3

Floating-Point Number: -8.3

D:\CD\_LAB>ex2a

Enter a number: 897

Not a Floating-Point Number: 897

D:\CD\_LAB>ex2a

Enter a number: 6.5

Floating-Point Number: 6.5

**RESULT:**

Hence, a lex program to recognize floating point numbers is developed and executed successfully.

**b) To identify and count the number of identifiers**

**AIM:**

To develop a lex program to identify and count the number of identifiers.

**ALGORITHM:**

1. Initialize variables, including count to 0, which will be used to count the identifiers.
2. Define lexical rules:
   1. [a-zA-Z\_][a-zA-Z0-9\_]\* {count++;}: This rule matches any sequence of characters that starts with an alphabetic character (a-z, A-Z) or an underscore followed by any combination of alphanumeric characters (letters or digits). When such a sequence is encountered, increment the count variable to count the identifier.
   2. .: This rule is a catch-all rule that matches any character not matched by the previous rule. This rule ensures that any characters other than identifiers are ignored and not processed further.
3. In the main function:
   1. Initialize the lexical analysis process using yylex().
   2. After lexical analysis, print the count of identifiers using printf.
4. The program exits with a return status of 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

int count=0;

%}

%option noyywrap

%%

[a-zA-Z\_][a-zA-Z0-9\_]\* {count++;}

. {

}

%%

int main() {

yylex();

printf("No. of identifiers: %d",count);

return 0;

}

**sample.txt**

a=b+c

**OUTPUT:**

D:\CD\_LAB>flex ex2b.l

D:\CD\_LAB>gcc lex.yy.c -o ex2b

D:\CD\_LAB>ex2b < sample.txt

No. of identifiers: 3

**RESULT:**

Hence, a lex program to recognize floating point numbers is developed and executed successfully.

**c) To count number of words, lines and characters**

**AIM:**

To develop a lex program to count the number of words, lines and characters.

**ALGORITHM:**

1. Include the standard input-output header.
2. Declare four integer variables: wordCount, charCount, lineCount, and inWord. Initialize wordCount, charCount, and lineCount to 0, and inWord to 0.
3. Define the Lex rule section:
   1. [a-zA-Z]+ matches one or more alphabetic characters (words).
      1. Action: Increment the wordCount by 1.
      2. Update charCount by adding the length of the matched word (yyleng).
      3. Set inWord to 1 to indicate that a word was found.
   2. \n matches newline characters.
      1. Action: If inWord is true, increment wordCount to count the last word on the line.
      2. Increment lineCount to count the newline.
      3. Set inWord to 0 to indicate the start of a new line.
   3. . matches any character (excluding newline).

Action: If inWord is true, increment charCount by 1 to count characters in words.

1. Call the yylex() function to initiate the Lexical Analysis process.
2. Display the results:
   1. Print "No. of words:" followed by the value of wordCount.
   2. Print "No. of characters:" followed by the value of charCount.
   3. Print "No. of lines:" followed by the value of lineCount.
3. Return 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

int wordCount = 0;

int charCount = 0;

int lineCount = 0;

int inWord = 0;

%}

%option noyywrap

%%

[a-zA-Z]+ { wordCount++; charCount += yyleng; inWord = 1; }

\n { if (inWord) { wordCount++; } lineCount++; inWord = 0; }

. { if (inWord) charCount++; }

%%

int main()

{

yylex();

printf("No. of words: %d\n", wordCount);

printf("No. of characters: %d\n", charCount);

printf("No. of lines: %d\n", lineCount);

return 0;

}

**sample.txt**

Hi Hello

This is the sample program

**OUTPUT:**

D:\CD\_LAB>flex ex2c.l

D:\CD\_LAB>gcc lex.yy.c -o ex2c

D:\CD\_LAB>ex2c < sample.txt

No. of words: 9

No. of characters: 34

No. of lines: 2

**RESULT:**

Hence, a lex program to count the number of words, lines and characters is developed and executed successfully.

**d) To count number of consonants and vowels**

**AIM:**

To develop a lex program to count the number of consonants and vowels.

**ALGORITHM:**

1. Include the standard input-output header.
2. Declare two integer variables, vowelCount and consonantCount, and initialize them to 0.
3. Define the Lex rule section:
   1. [aeiouAEIOU] matches any vowel (both lowercase and uppercase).

Action: Increment the vowelCount by 1.

* 1. [a-zA-Z] matches any alphabet character (both lowercase and uppercase).

Action: Increment the consonantCount by 1.

* 1. \n matches newline characters and is ignored.
  2. . matches any other character (excluding newlines) and is ignored.

1. Call the yylex() function to initiate the Lexical Analysis process.
2. Display the results:
   1. Print "No. of vowels:" followed by the value of vowelCount.
   2. Print "No. of consonants:" followed by the value of consonantCount.
3. Return 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

int vow = 0, cons = 0;

%}

%option noyywrap

%%

[aeiouAEIOU] { vow++; }

[a-zA-Z] { cons++; }

\n { /\* Ignore newlines \*/ }

. { /\* Ignore other characters \*/ }

%%

int main() {

yylex();

printf("No. of vowels: %d\n", vow);

printf("No. of consonants: %d\n", cons);

return 0;

}

**sample.txt**

Hi Hello

This is the sample program

**OUTPUT:**

D:\CD\_LAB>flex ex2d.l

D:\CD\_LAB>gcc lex.yy.c -o ex2d

D:\CD\_LAB>ex2d < sample.txt

No. of vowels: 10

No. of consonants: 19

**RESULT:**

Hence, a lex program to count the number of vowels and consonants is developed and executed successfully.

**e) To insert line numbers for statements**

**AIM:**

To develop a lex program to insert line numbers for statements.

**ALGORITHM:**

1. Include the standard input-output header.
2. Declare an integer variable **nl** (line number) and initialize it to 0.
3. Define the Lex rule section:
   1. **(.\*)\n** matches any sequence of characters followed by a newline character.
      1. Action: When a line is matched, increment the **nl** (line number) by 1.
      2. Print the current **nl** value and the matched line (**yytext**) with a space in between.
4. In the main function:
   1. Set the input source (**yyin**) to read from the file specified as a command-line argument (**argv[1]**).
   2. Initiate the Lexical Analysis process using **yylex()**.
   3. Close the input file (**yyin**).
   4. Return 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

int nl=0;

%}

%option noyywrap

%%

(.\*)\n { printf("%d %s ", ++nl,yytext); }

%%

int main(int argc, char \*argv[]){

yyin = fopen(argv[1],"r");

yylex();

fclose(yyin);

return 0;

}

**prog.c**

int main()

{

//comment

int a = 5,b = 10;

if (a > b){

printf("a>b");

}

else

{

printf("b>a");

}

return 0;

}

**OUTPUT:**

D:\CD\_LAB>flex ex2e.l

D:\CD\_LAB>gcc lex.yy.c -o ex2e

D:\CD\_LAB>ex2e < prog.c

1 int main()

2 {

3 //comment

4 int a = 5,b = 10;

5 if (a > b){

6 printf("a>b");

7 }

8 else

9 {

10 printf("b>a");

11 }

12 return 0;

13 }

14

**RESULT:**

Hence, a lex program to count the number of vowels and consonants is developed and executed successfully.

**EX NO: 03 LEXICAL ANALYZER**

**DATE:**

**AIM:**

To implement a Lexical Analyzer using LEX Tool.

**ALGORITHM:**

1. Include the standard input-output and string manipulation (<string.h>) headers.
2. Define a section for Lex specifications:
   1. Use %{ and %} to enclose the C code section where you declare global variables and include headers.
3. Define the Lex rules section:
   1. For preprocessor directives (e.g., #include<stdio.h>), use a pattern to match these lines, but you don't print them. This effectively skips preprocessor directives.
   2. Define patterns for various token types, such as keywords (e.g., "auto", "break"), relational operators (e.g., "==", "<="), unary operators (e.g., "++", "--"), logical operators (e.g., "&&", "||"), bitwise operators (e.g., "&", "|", "<<"), assignment operators (e.g., "=", "+="), arithmetic operators (e.g., "+", "-"), end of statement (e.g., ";"), string literals (e.g., "\".\*\""), and identifiers (e.g., [a-zA-Z\_][a-zA-Z\_0-9]\*).
   3. When a token matches any of these patterns, print the corresponding type and the matched token using printf. For example, for keywords, you would use printf("Keyword: %s\n", yytext);.
   4. You can also specify patterns to skip certain characters or character sequences like parentheses, curly braces, whitespace (including tabs and newlines), and integers.
4. Define a %} section to close the Lex specifications.
5. Implement a yywrap function to indicate the end of input. This function returns 1 to indicate the end.
6. Define the main function:
   1. Use yylex() to initiate the Lexical Analysis process.
7. In the main function, return 0 to indicate successful execution.

**PROGRAM:**

%{

#include <stdio.h>

#include <string.h>

%}

%%

"#include<stdio.h>" ; //Skips preprocessor directives

"auto" { printf("Keyword: auto\n"); }

"break" { printf("Keyword: break\n"); }

"case" { printf("Keyword: case\n"); }

"char" { printf("Keyword: char\n"); }

"const" { printf("Keyword: const\n"); }

"continue" { printf("Keyword: continue\n"); }

"default" { printf("Keyword: default\n"); }

"do" { printf("Keyword: do\n"); }

"double" { printf("Keyword: double\n"); }

"else" { printf("Keyword: else\n"); }

"enum" { printf("Keyword: enum\n"); }

"extern" { printf("Keyword: extern\n"); }

"float" { printf("Keyword: float\n"); }

"for" { printf("Keyword: for\n"); }

"goto" { printf("Keyword: goto\n"); }

"if" { printf("Keyword: if\n"); }

"int" { printf("Keyword: int\n"); }

"long" { printf("Keyword: long\n"); }

"register" { printf("Keyword: register\n"); }

"return" { printf("Keyword: return\n"); }

"short" { printf("Keyword: short\n"); }

"signed" { printf("Keyword: signed\n"); }

"sizeof" { printf("Keyword: sizeof\n"); }

"static" { printf("Keyword: static\n"); }

"struct" { printf("Keyword: struct\n"); }

"switch" { printf("Keyword: switch\n"); }

"typedef" { printf("Keyword: typedef\n"); }

"union" { printf("Keyword: union\n"); }

"unsigned" { printf("Keyword: unsigned\n"); }

"void" { printf("Keyword: void\n"); }

"volatile" { printf("Keyword: volatile\n"); }

"while" { printf("Keyword: while\n"); }

[(){}] ; //Skips parentheses and curly braces

"=="|"<="|">="|"<"|">"|"!=" { printf("Relational Operator: %s\n",yytext); }

"++"|"--" { printf("Unary Operator: %s\n", yytext); }

"&&"|"||"|"!" { printf("Logical Operator: %s\n",yytext); }

"&"|"|"|"<<"|">>"|"~"|"^" { printf("Bitwise Operator: %s\n",yytext); }

"="|"+="|"-="|"\*="|"%="|"/=" { printf("Assignment Operator: %s\n",yytext); }

"+"|"-"|"\*"|"/"|"%" { printf("Arithmetic Operator: %s\n", yytext); }

";" { printf("End of Statement: %s\n", yytext); }

"\""([^\"\\]|\\.)\*"\"" { printf("String Literal: %s\n", yytext); }

[ \t\n] ; // Skip whitespace

"printf" { printf("Predefined Function: %s\n", yytext); }

"main()" { printf("Predefined Function: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z\_0-9]\* { printf("Identifier: %s\n", yytext); }

[0-9]+ { printf("Integer Literal: %s\n", yytext); }

"." { printf("Unknown: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

int main()

{

yylex();

return 0;

}

**INPUT C PROGRAM: (prog.c)**

#include<stdio.h>

int main()

{

int myNum = 10;

if (myNum > 0) {

printf("The value is a positive number.");

}

else if (myNum < 0) {

printf("The value is a negative number.");

}

else {

printf("The value is 0.");

}

return 0;

}

**OUTPUT:**

D:\>flex condition.l

D:\>gcc lex.yy.c -o condi

D:\>condition <prog.c

Keyword: int

Predefined Function: main()

Keyword: int

Identifier: myNum

Assignment Operator: =

Integer Literal: 10

End of Statement: ;

Keyword: if

Identifier: myNum

Relational Operator: >

Integer Literal: 0

Predefined Function: printf

String Literal: "The value is a positive number"

End of Statement: ;

Keyword: else

Keyword: if

Identifier: myNum

Relational Operator: <

Integer Literal: 0

Predefined Function: printf

String Literal: "The value is a negative number"

End of Statement: ;

Keyword: else

Predefined Function: printf

String Literal: "The value is 0"

End of Statement: ;

Keyword: return

Integer Literal: 0

End of Statement: ;

**RESULT:**

Hence successfully implemented a Lexical Analyzer using LEX Tool.

**EX NO: 04 a) VALIDATE ARITHMETIC EXPRESSION**

**DATE:**

**AIM:**

To develop a program to recognize a valid arithmetic expression that uses operator +, -, \* and / using LEX and YACC.

**ALGORITHM:**

**calc.l**

1. Include necessary header files.
2. Declare global variables like **yylval** and **iden**.
3. Define rules to recognize identifiers (**[a-zA-Z]+**) and numbers (**[0-9]+**).
4. Ignore tabs and newlines.
5. For any other character, return it as is.

**calc.y**

1. Include necessary header files, including the generated **y.tab.h**.
2. Declare the **NUMBER** and **ID** tokens.
3. Define the grammar rules:
   1. A **program** consists of an expression **E**.
   2. **E** can be an **E + T** or just **T**.
   3. **T** can be a **T \* F** or just **F**.
   4. **F** can be an **ID** or a **NUMBER**.
4. In the **main** function:
   1. Print a prompt to enter an arithmetic expression.
   2. Call **yyparse** to start parsing.
   3. Return 0.

**PROGRAM:**

**ex4a.l**

%{

#include <stdio.h>

#include "y.tab.h"

int yylval;

char iden[20];

%}

%option noyywrap

%%

[a-zA-Z]+ { strcpy(iden, yytext); yylval = 1; return ID; }

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[\t] ;

[\n] return 0;

. return yytext[0];

%%

**ex4a.y**

%{

#include <stdio.h>

#include "y.tab.h"

int yylval;

char iden[20];

%}

%token NUMBER ID

%%

program: E {

printf("Valid expression\n");

}

;

E: E '+' T

| T

;

T: T '\*' F

| F

;

F: ID

| NUMBER

;

%%

int main() {

printf("Enter an arithmetic expression: ");

yyparse();

return 0;

}

int yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

return 1;

}

**OUTPUT:**

D:\CD\_LAB>flex ex4a.l

D:\CD\_LAB>bison -d -y ex4a.y

D:\CD\_LAB>gcc lex.yy.c y.tab.c -w -o ex4a

D:\CD\_LAB>ex4a

Enter an arithmetic expression: a+b

Valid expression

D:\CD\_LAB>ex4a

Enter an arithmetic expression: 2+4\*f

Valid expression

**RESULT:**

Hence, a program to recognize a valid arithmetic expression that uses operator +, -, \* and / is developed and executed successfully using LEX and YACC.

**EX NO: 04 b) RECOGNIZE IDENTIFIERS**

**DATE:**

**AIM:**

To develop a program to recognize a valid variable which starts with a letter followed by any number of letters or digits using LEX and YACC.

**ALGORITHM:**

**ex4b.l**

1. Include necessary header files, like <stdio.h> and "y.tab.h" (if needed).
2. Declare the IDENTIFIER token using %token IDENTIFIER.
3. Define a rule to recognize valid identifiers using the regular expression [a-zA-Z][a-zA-Z0-9]\*.
4. For any other character, return it as is.
5. Implement the yywrap function to indicate the end of input (return 1).

**ex4b.y**

1. Include necessary header files, such as <stdio.h> and <stdlib.h>.
2. Declare the IDENTIFIER token using %token IDENTIFIER.
3. Specify the %start symbol and the %type for the identifier.
4. Define the grammar rules:
   1. A program consists of an IDENTIFIER.
5. In the main function:
   1. Print a prompt to enter a string.
   2. Call yyparse to start parsing.
   3. Return 0.

**PROGRAM:**

**ex4b.l**

%{

#include "y.tab.h"

%}

%%

[a-zA-Z][a-zA-Z0-9]\* { yylval = strdup(yytext); return IDENTIFIER; }

. ;

%%

int yywrap() {

return 1;

}

**ex4b.y**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token IDENTIFIER

%%

program: IDENTIFIER {

printf("Valid Identifier: %s\n", $1);

exit(1);

}

;

%%

int main() {

printf("Enter a String: ");

yyparse();

return 0;

}

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

exit(1);

}

**OUTPUT:**

D:\CD\_LAB>flex ex4b.l

D:\CD\_LAB>bison -d -y ex4b.y

D:\CD\_LAB>gcc lex.yy.c y.tab.c -w -o ex4b

D:\CD\_LAB>ex4b

Enter a string: sum

Valid identifier

D:\CD\_LAB>ex4b

Enter a string: student12

Valid identifier

**RESULT:**

Hence, a program to recognize a valid variable which starts with a letter followed by any number of letters or digits is developed and executed successfully using LEX and YACC.

**EX NO: 04 d) CALCULATOR**

**DATE:**

**AIM:**

To implement of calculator using LEX and YACC.

**ALGORITHM:**

**calc.l**

1. Initialize the lexical analysis process.
2. Include the necessary header files and declare an external integer variable **yylval**.
3. Define regular expressions to match specific patterns in the input text:
   1. **[0-9]+** matches integers and sets **yylval** with the integer value. Return the **NUMBER** token.
   2. **[\t]** matches tab characters and is ignored.
   3. **[\n]** matches newline characters and returns a special **0** token.
   4. **.** matches any other character and returns the character itself.
4. Define the **yywrap()** function to return **1** to signal the end of the input stream.

**calc.y**

1. Start the arithmetic expression parsing process.
2. Include the necessary header file **<stdio.h>**.
3. Declare an integer variable **flag** and set it to 0.
4. Declare the **NUMBER** token and define operator precedence rules.
5. Define the top-level rule **ArithmeticExpression**:
   1. Print the result of the expression if parsing is successful.
   2. Return 0 to indicate successful parsing.
6. Define grammar rules for parsing arithmetic expressions:
   1. Expressions can involve addition, subtraction, multiplication, division, modulo, and parentheses.
   2. Implement semantic actions to calculate and assign values accordingly.
7. In the **main** function:
   1. Prompt the user to enter an arithmetic expression.
   2. Invoke the parser using **yyparse()**.
8. In the **yyerror** function:
   1. Print a message indicating that the entered arithmetic expression is invalid.
   2. Set the **flag** to 1 to mark the expression as invalid.
9. After parsing, if **flag** remains 0, print a message indicating that the entered arithmetic expression is valid.

**PROGRAM:**

**calc.l**

%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

%%

[0-9]+ { yylval=atoi(yytext);

return NUMBER;

}

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

**calc.y**

%{

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression : E {

printf("\nResult=%d\n",$$);

return 0;

};

E : E'+'E {$$=$1+$3;}

| E'-'E {$$=$1-$3;}

| E'\*'E {$$=$1\*$3;}

| E'/'E {$$=$1/$3;}

| E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

|NUMBER {$$=$1;}

;

%%

void main()

{

printf("\nEnter any arithmetic expression which can have operations Addition,Subtraction, multiplication, division or modulo:\t");

yyparse();

if(flag==0)

printf("\nEntered arithmetic expression is valid\n");

}

int yyerror(const char \*s)

{

printf("\nEntered arithmetic expression is invalid\n\n");

flag=1;

}

**OUTPUT:**

C:\CD>flex calc.l

C:\CD>bison -d -y calc.y

C:\CD>gcc  lex.yy.c y.tab.c -o  calc -w

C:\CD>calc

Enter any Arithmetic Expression which can have operations Addition, Subtraction, multiplication, division or modulo:

3+8-2

Result=9

Entered Arithmetic Expression is valid

**RESULT:**

Hence successfully implemented calculator using LEX and YACC.

**EX NO: 05 THREE ADDRESS CODE**

**DATE:**

**AIM:**

To generate three address code for a simple program using LEX and YACC.

**ALGORITHM:**

**threeaddresscode.l**

1. Start the lexical analysis process.
2. Include the necessary header files, such as **<stdio.h>** and **"y.tab.h"**.
   1. Declare an external integer variable **yylval**.
   2. Declare a character array **iden** of size 20 for storing identifiers.
3. Define regular expressions to match specific patterns in the input text:
   1. **[0-9]+** matches sequences of digits and converts them to an integer value using **atoi()**. Set **yylval** with the integer value and return the **digit** token.
   2. **[a-zA-Z]+** matches sequences of letters (identifiers) and copies them into the **iden** array. Set **yylval** to 1 to indicate an identifier and return the **id** token.
   3. **[\t]** matches tab characters and is ignored.
   4. **[\n]** matches newline characters and returns a special **0** token.
   5. **.** matches any other character and returns the character itself.
4. Define the **yywrap()** function, which returns **1** to signal the end of the input stream.

**threeaddresscode.y**

1. Start the process of parsing arithmetic expressions.
2. Include the necessary header files, such as **<math.h>**, **<ctype.h>**, and **<stdio.h>**.
   1. Declare an integer variable **var\_cnt** and a character array **iden** of size 20 for storing identifiers.
3. Declare tokens **digit** and **id**.
4. Define grammar rules for parsing arithmetic expressions using Yacc/Bison syntax.
5. Define the **S** rule, which handles assignment statements:
   1. When an assignment statement is recognized, print the assignment operation along with the identifier.
6. Define rules for parsing expressions:
   1. Implement semantic actions to generate intermediate code.
   2. Print intermediate code for addition, subtraction, multiplication, division, exponentiation, and parentheses.
7. Define rules for parsing expressions with parentheses and numeric constants.
8. In the **main** function:
   1. Initialize the variable **var\_cnt** to 0.
   2. Prompt the user to enter an expression.
9. Execute the parser using **yyparse()**.
10. If a parsing error occurs, the **yyerror** function is called, and an error message is printed.

**PROGRAM:**

**threeaddresscode.l**

%{

#include<stdio.h>

#include<string.h>

#include "y.tab.h"

extern int yylval;

extern char iden[20];

%}

%%

[0-9]+ { yylval=atoi(yytext);

return digit;

}

[a-zA-Z]+ { strcpy(iden,yytext);

yylval=1;

return id;

}

[\t] {;}

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

**threeaddresscode.y**

%{

#include <math.h>

#include<ctype.h>

#include<stdio.h>

int var\_cnt=0;

char iden[20];

%}

%token digit

%token id

%%

S:id '=' E { printf("%s = t%d\n",iden, var\_cnt-1); }

E:E '+' T { $$=var\_cnt; var\_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }

|E '-' T { $$=var\_cnt; var\_cnt++; printf("t%d = t%d - t%d;\n", $$, $1, $3 ); }

|T { $$=$1; }

;

T:T '\*' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d \* t%d;\n", $$, $1, $3 ); }

|T '/' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d / t%d;\n", $$, $1, $3 ); }

|F {$$=$1 ; }

;

F:P '^' F { $$=var\_cnt; var\_cnt++; printf("t%d = t%d ^ t%d;\n", $$, $1, $3 );}

| P { $$ = $1;}

;

P: '(' E ')' { $$=$2; }

|digit { $$=var\_cnt; var\_cnt++; printf("t%d = %d;\n",$$,$1); }

;

%%

int main()

{

var\_cnt=0;

printf("Enter an expression : \n");

yyparse();

return 0;

}

yyerror(){

printf("Error\n");}

**OUTPUT:**

C:\CD>flex threeaddress.lex

C:\CD>bison -d -y threeaddresscode.y

C:\CD>gcc lex.yy.c y.tab.c -o threeaddress -w C:\CD>threeaddress

Enter an expression:

a=2+4-3

t0 = 2;

t1 = 4;

t2 = t0 + t1;

t3 = 3;

t4 = t2 - t3;

a = t4

**RESULT:**

Hence, generated three address code for a simple program using LEX and YACC.

**EX NO: 06 TYPE CHECKING**

**DATE:**

**AIM:**

To implement type checking using LEX and YACC.

**ALGORITHM:**

**Type.l**

1. Define necessary header files, including "y.tab.h" for token definitions.
2. Specify regular expressions for integer, float, addition, subtraction, space, and newline tokens.
3. Implement actions for each regular expression:
   1. For integers, convert the string to an integer and return the INT token.
   2. For floats, convert the string to a float and return the FLOAT token.
   3. For addition and subtraction, return the corresponding ADD or SUB token.
   4. Ignore spaces.
   5. Return the EOL token for newline.
   6. Print an error message for any invalid characters.
4. Implement the **yywrap** function returning 1.

**Type.y**

1. Include necessary header files like stdio, stdbool, and stdlib.
2. Define a union for semantic values containing int\_val and float\_val.
3. Declare tokens and their corresponding types using **%token**.
4. Specify the start symbol and types for non-terminals using **%start** and **%type**.
5. Define the grammar rules:
   1. A program consists of lines separated by EOL.
   2. A line is an expression, and its type is printed based on whether it's INT or FLOAT.
   3. An expression can be the sum or difference of two terms, and the resulting type is determined based on the types of the operands.
   4. A term is either an INT or a FLOAT.
6. Implement the **main** function calling **yyparse**.
7. Implement the **yyerror** function to print error messages.

**PROGRAM:**

**Type.l**

%{

#include "y.tab.h"

%}

%%

[0-9]+ {

yylval.int\_val = atoi(yytext);

return INT;

}

[0-9]+"."[0-9]+ {

yylval.float\_val = atof(yytext);

return FLOAT;

}

"+" { return ADD; }

"-" { return SUB; }

" " ; // Ignore spaces

\n { return EOL; }

. { printf("Invalid character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

**Type.y**

%{

#include <stdio.h>

#include <stdbool.h>

#include<stdlib.h>

%}

%union {

int int\_val;

float float\_val;

}

%token <int\_val> INT

%token <float\_val> FLOAT

%token ADD SUB EOL

%start program

%type <int\_val> line

%type <int\_val> expression

%type <int\_val> term

%%

program: /\* empty \*/

| program line EOL

;

line: expression {

if ($1 == INT) {

printf("Result: %d (integer)\n", $$);

} else if ($1 == FLOAT) {

printf("Result: %f (float)\n", $$);

} else {

printf("Type error: Incompatible types\n");

}

}

;

expression: expression ADD term {

if ($1 == INT && $3 == INT) {

$$ = INT;

} else {

$$ = FLOAT;

}

}

| expression SUB term {

if ($1 == INT && $3 == INT) {

$$ = INT;

} else {

$$ = FLOAT;

}

}

| term

;

term: INT { $$ = INT; }

| FLOAT { $$ = FLOAT; }

;

%%

int main() {

yyparse();

return 0;

}

void yyerror(char \*msg) {

fprintf(stderr, "%s\n", msg);

exit(1);

}

**OUTPUT:**

a12-29@a1229-ThinkStation-P330:~$ gedit type.l  
a12-29@a1229-ThinkStation-P330:~$ gedit type.y  
a12-29@a1229-ThinkStation-P330:~$ flex type.l  
a12-29@a1229-ThinkStation-P330:~$ yacc -d type.y  
a12-29@a1229-ThinkStation-P330:~$ gcc -o type lex.yy.c y.tab.c -w  
a12-29@a1229-ThinkStation-P330:~$ ./type  
1  
Result: 257 (integer)  
2.5  
Result: 0.000000 (float)

**RESULT:**

Hence, type checking is implemented successfully using LEX and YACC.

**EX NO: 07 CODE GENERATION**

**DATE:**

**AIM:**

To implement a back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

**ALGORITHM:**

1. Include the necessary header file **<stdio.h>** for standard input and output operations and **<string.h>** for string manipulation.
2. Declare an array **icode[20][30]** to store up to 20 intermediate code instructions, **str[20]** for the current instruction, and **opr[10]** for the operator being used.
3. Initialize an integer variable **i** to 0.
4. Print a message to instruct the user to enter a set of intermediate code instructions.
5. Use a **do-while** loop to read and store the intermediate code instructions in the **icode** array until the instruction "exit" is encountered. The loop increments the index **i** after each input.
6. Reset the value of **i** to 0 to start processing the instructions.
7. Use another **do-while** loop to process each intermediate code instruction:
   1. Copy the content of the current instruction **icode[i]** to the **str** variable.
   2. Check the operator in the instruction by examining the character at the 4th position (**str[3]**).
   3. Depending on the operator, set the **opr** variable to the corresponding mnemonic (e.g., "+" becomes "ADD").
   4. Print assembly-like instructions to perform the operation:
      1. Load the value of the 5th character (**str[4]**) into a register (e.g., **MOV R0, A**).
      2. Perform the operation using the mnemonic stored in **opr** (e.g., **ADD A, R0**).
      3. Store the result back to the location specified in the 1st character (**str[0]**).
8. Continue processing instructions as long as "exit" is not encountered in the **icode** array.
9. Once all instructions have been processed, the program exits and returns 0.

**PROGRAM:**

#include<stdio.h>  
#include<string.h>  
int main()  
{  
char icode[20][30];  
char str[20],opr[10];  
int i =0;  
printf("\n enter the set of intermediate code instructions:\n");  
  
do  
{  
scanf("%s",icode[i]);  
}while(strcmp(icode[i++],"exit") != 0);  
i = 0;  
do{  
strcpy(str, icode[i]);  
switch(str[3])  
{  
case '+':  
strcpy(opr,"ADD");  
break;  
case '-':  
strcpy(opr,"SUB");  
break;  
case '\*':  
strcpy(opr,"MUL");  
break;  
case '/':  
strcpy(opr,"DIV");  
break;  
}  
printf("\n MOV %c,R%d",str[2],i);  
printf("\n %s %c,R%d",opr,str[4],i);  
printf("\n MOV R%d,%c",i,str[0]);  
}while(strcmp(icode[++i],"exit") !=0);  
  
return 0;  
}

**OUTPUT:**

C:\CD>gcc -o codegeneration codegeneration.c

C:\CD>codegeneration

enter the set of intermediate code instructions:

a=b+c

b=c-d

c=b+c

d=c-d

exit

MOV b,R0

ADD c,R0

MOV R0,a

MOV c,R1

SUB d,R1

MOV R1,b

MOV b,R2

ADD c,R2

MOV R2,c

MOV c,R3

SUB d,R3

MOV R3,d

**RESULT:**

Hence, a program is developed to implement a back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

**Additional Programs**

**1)**

**AIM:**

To design a parser for language anbn with help of productions:

S -> aSb | b

**ALGORITHM:**

**parser.l**

1. Include the header file "y.tab.h".
2. Define regular expressions and actions in the Lex file:
   1. **[aA]**: Recognize 'A' or 'a' and return the token **A**.
   2. **[bB]**: Recognize 'B' or 'b' and return the token **B**.
   3. **[\n]**: Recognize newline characters and return the token **NL**.
   4. **.**: Recognize any other character and return it as a single-character token.
3. Define the **yywrap** function to signal the end of input.

**parser.y**

1. Include the necessary header files **<stdio.h>** and **<stdlib.h>**.
2. Declare tokens **A**, **B**, and **NL** to be used in the grammar.
3. Define the grammar rules for the parsing process:
   1. **stmt**: Represents a statement and checks if it is of the form "S NL."
      * When a valid statement is recognized, print "Valid String" and exit the program.
   2. **S**: Represents the start symbol of the grammar.
      * It can generate productions of the form **A S B**.
4. Implement the **yyerror** function, which handles parsing errors:
   1. When a parsing error occurs, print "Invalid String" and exit the program.
5. In the **main** function:
   1. Prompt the user to enter an input string.
   2. Start the parsing process using **yyparse()**.

**PROGRAM:**

**parser.l**

%{

#include "y.tab.h"

%}

%%

[aA] {return A;}

[bB] {return B;}

\n {return NL;}

. {return yytext[0];}

%%

int yywrap(){

return 1;

}

**parser.y**

%{

#include<stdio.h>

#include<stdlib.h>

%}

%token A B NL

%%

stmt: S NL {printf("Valid String\n");exit(0);}

;

S: A S B |

;

%%

int yyerror(char \*msg){

printf("Invalid String\n");

exit(0);

}

void main(){

printf("Enter the input string: ");

yyparse();

}

**OUTPUT:**

$lex parser.l

$yacc -d -y parser.y

$gcc lex.yy.c y.tab.c -w -o parser

$./parser

Enter the input string: aabb

Valid String

**RESULT:**

Hence, successfully designed a parser for language L = {anbn , n≥0} with help of productions: S -> aSb | b

**2)**

**AIM:**

To design a parser for language anb | n≥5 with help of productions:

S -> aaaaaAb

A -> aA | ε

**ALGORITHM:**

**parser2.l**

1. Include the header file "y.tab.h".
2. Define regular expressions and actions in the Lex file:
   1. **[aA]** Recognize 'A' or 'a' and return the token **A**.
   2. **[bB]** Recognize 'B' or 'b' and return the token **B**.
   3. **[\n]** Recognize newline characters and return the token **NL**.
   4. **.** Recognize any other character and return it as a single-character token.
3. Define the **yywrap** function to signal the end of input.

**parser2.y**

1. Include the necessary header files **<stdio.h>** and **<stdlib.h>**.
2. Declare tokens **A**, **B**, and **NL**.
3. Define the grammar rules:
   1. **stmt**: Check if it is of the form "S NL." Print "Valid String" and exit the program if valid.
   2. **S**: Generate sequences of **A** and **B** tokens, with variable lengths.
   3. **T**: Generate sequences of **A** tokens, allowing for recursive expansion.
4. Implement the **yyerror** function to handle parsing errors.
5. In the **main** function:
   1. Prompt the user to enter an input string.
   2. Start parsing using **yyparse()**.

**PROGRAM:**

**parser2.l**

%{

#include "y.tab.h"

%}

%%

[aA] {return A;}

[bB] {return B;}

\n {return NL;}

. {return yytext[0];}

%%

int yywrap(){

return 1;

}

**parser2.y**

%{

#include<stdio.h>

#include<stdlib.h>

%}

%token A B NL

%%

stmt : S NL {printf("Valid String\n");exit(0);}

;

S : A A A A A T B

;

T : A T |

;

%%

int yyerror(char \*msg){

printf("Invalid String\n");

exit(0);

}

void main(){

printf("Enter the input string: ");

yyparse();

}

**OUTPUT:**

$lex parser2.l

$yacc -d -y parser2.y

$gcc lex.yy.c y.tab.c -w -o parser2

$./parser2

Enter the input string: aaaaaaab

Valid String

**RESULT:**

Hence, successfully designed a parser for language anb | n≥5 with help of productions:

S -> aaaaaAb

A -> aA | ε