**Ex.No:1a**

**Date:**

**LEXICAL ANALYZER -** IDENTIFIERS

**AIM:**

To use the LEX tool and Develop a lexical analyzer to recognize a Identifiers in C.

**ALGORITHM:**

* Include the necessary header file (stdio.h).
* Define the Lex rules using regular expressions. In this case, it recognizes identifiers, which start with a letter (a-zA-Z or underscore) and can be followed by letters, digits, or underscores.
* When an identifier is matched, print it using printf, with the "Identifier: " prefix and the identifier text (yytext).
* Define a rule that matches any other character (.) and ignore it. This means the program will skip characters that do not match the identifier rule.
* In the main function, call the yylex() function to start the Lexical analysis.
* Return 0 to indicate successful program execution.

**PROGRAM:**

%{

#include <stdio.h>

%%

%}

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

printf("Identifier: %s\n", yytext);

}

. {

// Ignore all other characters.

}

%%

int main() {

yylex();

return 0;

}

**OUTPUT**

a=b+5

Identifier: a

Identifier: b

**RESULT**

Hence we used the LEX tool and a program is Develop a lexical analyzer to recognize a Identifiers in C is executed successfully and the output is verified.

**Ex.No:1b**

**Date:**

**LEXICAL ANALYZER -** CONSTANTS

**AIM**

To use the LEX tool and Develop a lexical analyzer to recognize a Constants in C.

**ALGORITHM**

* Define regular expressions for different token types (INT\_CONSTANT, FLOAT\_CONSTANT, CHAR\_CONSTANT, NEWLINE).
* Use Flex's rules to match these regular expressions and perform actions when a match is found.
  + When an INT\_CONSTANT is found, print "Integer Constant" along with the matched text.
  + When a FLOAT\_CONSTANT is found, print "Float Constant" along with the matched text.
  + When a CHAR\_CONSTANT is found, print "Character Constant" along with the matched text.
  + Ignore and skip NEWLINE characters.
  + Ignore all other characters.
* In the main function, open the input file specified in the command-line arguments for reading.
* Set yyin (Flex's input file) to the opened file.
* Call yylex to start the lexical analysis process.
* Close the input file when the lexical analysis is complete.
* Return 0 to indicate successful execution.

**PROGRAM**

%{

#include <stdio.h>

%}

DIGIT [0-9]

INT\_CONSTANT {DIGIT}+

FLOAT\_CONSTANT {DIGIT}\*\.{DIGIT}+|\.{DIGIT}+|\.{DIGIT}\*

CHAR\_CONSTANT '\''[^'\n]+'\''

SPACE [ \t]+

NEWLINE \n

%%

{INT\_CONSTANT} {

printf("Integer Constant: %s\n", yytext);

}

{FLOAT\_CONSTANT} {

printf("Float Constant: %s\n", yytext);

}

{CHAR\_CONSTANT} {

printf("Character Constant: %s\n", yytext);

}

{SPACE} {

// Ignore spaces and tabs

}

{NEWLINE} {

// Ignore newlines

}

%%

int yywrap(){

}

int main() {

yylex();

return 0;

}

**OUTPUT**

23.56

Float Constant: 23.56

23

Integer Constant: 23

**RESULT**

Hence we used the LEX tool and developed a lexical analyzer to recognize a Constants in C.

**Ex.No:1c**

**Date:**

**LEXICAL ANALYZER -** COMMENTS

**AIM**

To use the LEX tool and Develop a lexical analyzer to recognize a Comments in C.

**ALGORITHM**

* Include the necessary header files.
* Use %option noyywrap to skip the yywrap function.
* Define Lexical Rules and Actions:
  + Match single-line comments (e.g., // comment) and print them as "Single-line Comment: comment text."
  + Match multi-line comments (e.g., /\* comment \*/) and print them as "Multi-line Comment: comment text."
  + Ignore all other characters.
* In the main function:
  + Set the input stream to the file specified as a command-line argument.
  + Call yylex for lexical analysis.
  + Close the input stream after processing the file.
* Return 0 to indicate successful execution.

**PROGRAM**

%{

#include <stdio.h>

%}

%option noyywrap

%%

"//"(.\*) {

printf("Single-line Comment: %s\n", yytext);

}

"/\*"([^\*]|\\*+[^\*/])\*"\*/" {

printf("Multi-line Comment: %s\n", yytext);

}

. {

// Ignore other characters

}

%%

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

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

yylex();

}

fclose(yyin);

return 0;

**C Program file:**

#include <stdio.h>

// This is a single-line comment

int main() {

/\*

This is a

multi-line

comment

\*/

printf("Hello, World!\n");

return 0;

}

**OUTPUT:**

//

Single-line Comment: //

/\*

\*/

Multi-line Comment: /\*

\*/

**RESULT**

Hence LEX tool is developed a lexical analyzer to recognize a Comments in C.

**EX.No:1d**

**Date:**

**LEXICAL ANALYZER -** OPERATORS

**AIM**

To use the LEX tool and Develop a lexical analyzer to recognize a Operatorsin C.

**ALGORITHM**

Include the necessary header file(s).

Define the patterns for various operators using regular expressions. For each operator, specify the action to be taken when a match is found. In this case, it prints the operator description.

Define a rule to ignore whitespace characters (space, tab, newline).

Define a rule to ignore any other characters that don't match the specified operator patterns.

In the main function:

a. Check if the program was given an input file as a command-line argument. If not, print a usage message and return with an error code.

b. Open the specified input file for reading. If the file cannot be opened, print an error message and return with an error code.

c. Set the input source for the lexer (yyin) to the opened file.

d. Call the lexer (yylex) to start tokenizing the input file.

e. Close the input file when tokenization is complete.

**PROGRAMS**

%{

#include <stdio.h>

%}

%%

"++" { printf("Operator: Increment (++)\n"); }

"--" { printf("Operator: Decrement (--)\n"); }

"==" { printf("Operator: Equal (==)\n"); }

"!=" { printf("Operator: Not Equal (!=)\n"); }

"<=" { printf("Operator: Less than or Equal (<=)\n"); }

">=" { printf("Operator: Greater than or Equal (>=)\n"); }

"<" { printf("Operator: Less than (<)\n"); }

">" { printf("Operator: Greater than (>)\n"); }

"&&" { printf("Operator: Logical AND (&&)\n"); }

"||" { printf("Operator: Logical OR (||)\n"); }

"+" { printf("Operator: Addition (+)\n"); }

"-" { printf("Operator: Subtraction (-)\n"); }

"\*" { printf("Operator: Multiplication (\*)\n"); }

"/" { printf("Operator: Division (/)\n"); }

"%" { printf("Operator: Modulus (%)\n"); }

"=" { printf("Operator: Assignment (=)\n"); }

"+=" { printf("Operator: Add and Assign (+=)\n"); }

"-=" { printf("Operator: Subtract and Assign (-=)\n"); }

"\*=" { printf("Operator: Multiply and Assign (\*=)\n"); }

"/=" { printf("Operator: Divide and Assign (/=)\n"); }

"%=" { printf("Operator: Modulus and Assign (%=)\n"); }

"!" { printf("Operator: Logical NOT (!)\n"); }

"&" { printf("Operator: Bitwise AND (&)\n"); }

"|" { printf("Operator: Bitwise OR (|)\n"); }

"^" { printf("Operator: Bitwise XOR (^)\n"); }

"~" { printf("Operator: Bitwise NOT (~)\n"); }

"<<=" { printf("Operator: Left Shift and Assign (<<=)\n"); }

">>=" { printf("Operator: Right Shift and Assign (>>=)\n"); }

"<<" { printf("Operator: Left Shift (<<)\n"); }

">>" { printf("Operator: Right Shift (>>)\n"); }

"&=" { printf("Operator: Bitwise AND and Assign (&=)\n"); }

"|=" { printf("Operator: Bitwise OR and Assign (|=)\n"); }

"^=" { printf("Operator: Bitwise XOR and Assign (^=)\n"); }

"sizeof" { printf("Operator: Sizeof (sizeof)\n"); }

"->" { printf("Operator: Structure Pointer Access (->)\n"); }

"." { printf("Operator: Member Access (.)\n"); }

"(" { printf("Operator: Left Parenthesis (()\n"); }

")" { printf("Operator: Right Parenthesis ())\n"); }

"[" { printf("Operator: Left Square Bracket ([)\n"); }

"]" { printf("Operator: Right Square Bracket (])\n"); }

"{" { printf("Operator: Left Curly Brace ({)\n"); }

"}" { printf("Operator: Right Curly Brace (})\n"); }

";" { printf("Operator: Semicolon (;)\n"); }

"," { printf("Operator: Comma (,)\n"); }

":" { printf("Operator: Colon (:)\n"); }

"?" { printf("Operator: Question Mark (?)\n"); }

"..." { printf("Operator: Ellipsis (...)\n"); }

"->\*" { printf("Operator: Structure Pointer Access and Dereference (->\*)\n"); }

"<<=" { printf("Operator: Left Shift and Assign (<<=)\n"); }

">>=" { printf("Operator: Right Shift and Assign (>>=)\n"); }

"&=" { printf("Operator: Bitwise AND and Assign (&=)\n"); }

"|=" { printf("Operator: Bitwise OR and Assign (|=)\n"); }

"^=" { printf("Operator: Bitwise XOR and Assign (^=)\n"); }

"<<=" { printf("Operator: Left Shift and Assign (<<=)\n"); }

">>=" { printf("Operator: Right Shift and Assign (>>=)\n"); }

"," { printf("Operator: Comma (,)\n"); }

";" { printf("Operator: Semicolon (;)\n"); }

":" { printf("Operator: Colon (:)\n"); }

"?" { printf("Operator: Question Mark (?)\n"); }

[ \t\n] ; // Ignore whitespace and tabs

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

%%

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

if (argc != 2) {

printf("Usage: %s <input\_file>\n", argv[0]);

return 1;

}

FILE\* input\_file = fopen(argv[1], "r");

if (!input\_file) {

printf("Error: Unable to open input file.\n");

return 1;

}

yyin = input\_file;

yylex();

fclose(input\_file);

return 0;

}

**OUTPUT:**

Operator: Include (<)

Operator: Include (>)

Operator: Include (<stdio.h>)

Operator: Main (main)

Operator: Assignment (=)

Operator: Addition (+)

Operator: Assignment (=)

Operator: Addition (+)

Operator: If (if)

Operator: Greater than (>)

Operator: Number (15)

Operator: Left Curly Brace ({)

Operator: Result (Result)

Operator: is (is)

Operator: not (not)

Operator: Greater than (>)

Operator: Number (15)

Operator: Right Curly Brace (})

**RESULT**

Hence LEX tool is developed a lexical analyzer to recognize a Operators in C.

**Ex.No:2**

**Date:**

**LEXICAL ANALYZER**

**AIM**

To Implement a Lexical Analyzer using LEX TooL.

**ALGORITHM**

* Include necessary header files: #include <stdio.h> and #include <stdlib.h>.
* Define regular expressions for various tokens using Lex notation. These definitions include DIGIT, LETTER, ID, INT\_LITERAL, STRING, and WS (whitespace).
* Use the %% delimiter to separate the lexical rules section from the C code section.
* Define lexical rules for token recognition. These rules are written in the format {PATTERN} {ACTION}, where PATTERN is a regular expression to match, and ACTION is the code to execute when a match is found.
* Skip whitespace and newline characters using the rule {WS}.
* Recognize and print INTEGER\_LITERAL tokens using the rule {INT\_LITERAL}.
* Recognize and print STRING\_LITERAL tokens using the rule {STRING}.
* Recognize and print keywords (e.g., "if," "else," "while") and other reserved words using rules like "if", "else", etc.
* Recognize and print special symbols (e.g., "{" or "(") using rules like "{", "(", etc.
* Recognize and print the ASSIGN token using the rule "=".
* Recognize and print IDENTIFIER tokens using the rule {ID}.
* Recognize and print operators (e.g., "==" or "+") using rules like "==", "+", etc.
* Ignore newlines using the rule \n.
* For any unrecognized characters, print an error message.
* Define the main function, which opens the input file, sets it as the input for Lex (yyin), and calls the lexical analyzer (yylex) to tokenize the input.
* Close the input file and return 0 on successful execution

**PROGRAM**

%{

%}

delim [\n\t]

white\_space {delim}+

letter [a-zA-Z]

digit [0-9]

identifier [a-zA-Z\_][a-zA-Z\_0-9]\*

arith [-+\*/^]

%%

int|float|double|long|short|signed|unsigned {printf("\n\t %s is a type indentifier",yytext);}

main {printf("\n\t %s is a function",yytext);}

printf {printf("\n\t %s is a function",yytext);}

scanf {printf("\n\t %s is a function",yytext);}

if|else|return {printf("\n\t %s is a keyword",yytext);}

for|while {printf("\n\t %s is Control statement",yytext);}

{identifier} {printf("\n\t %s is a identifier",yytext);}

{arith} {printf("\n\t %s is arithmetic operator",yytext);}

"#include<stdio.h>" {printf("\n\t %s is a header file",yytext);}

"=" {printf("\n\t %s is a Assignment operator",yytext);}

"==" {printf("\n\t %s is a Equal operator",yytext);}

"!=" {printf("\n\t %s is a Not Equal operator",yytext);}

";" {printf("\n\t %s is a Statement Spliter",yytext);}

"<" {printf("\n\t %s is a Less than operator",yytext);}

">" {printf("\n\t %s is a greater than operator",yytext);}

"<=" {printf("\n\t %s is a Less than or eqaul to operator",yytext);}

"{" {printf("\n\t %s is curly braces",yytext);}

"}" {printf("\n\t %s is a curly braces",yytext);}

">=" {printf("\n\t %s is a greater than or equal to operator",yytext);}

\".\*\" {printf("\n\t %s ",yytext);}

%%

int yywrap(void){}

int main(int argc, char \*\* argv)

{

if(argc>1)

{

FILE \*file;

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

if(!file)

{

printf("Could not open file");

exit(0);

}

yyin=file;

}

yylex();

return 0;

}

**INPUT FILE:**

#include<stdio.h>

int main(){

int x=10;

if(x<15){

printf("%d is lesser than 15",x);

}

else{

printf("%d is greater than 15",x);

}

}

**OUTPUT**

a12-58@a1258-ThinkStation-P330:~/4058\_CD\_LAB$ ./ex2 <ex2\_Cf.c

#include<stdio.h> is a header file

int is a type indentifier

main is a function()

{ is curly braces

int is a type indentifier

x is a identifier

= is a Assignment operator10

; is a Statement Spliter

if is a keyword(

x is a identifier

< is a Less than operator15)

{ is curly braces

printf is a function(

"%d is lesser than 15" ,

x is a identifier)

; is a Statement Spliter

} is a curly braces

else is a keyword

{ is curly braces

printf is a function(

"%d is greater than 15" ,

x is a identifier)

; is a Statement Spliter

} is a curly braces

} is a curly braces

**RESULT**

Hence a Lexical Analyzer is implemented using LEX TooL.

**Ex.No:3a**

**Date:**

**YACC SPECIFICATION - ARITHMATIC OPERATORS**

**AIM**

To implement a program to recognize a valid arithmetic expression that uses operator +, -, \* and /.

**ALGORITHM**

* Import the necessary header file stdio.h.
* Define a token NUMBER and specify the precedence and associativity of operators using %left for + and - and %left for \* and /.
* Define the grammar for mathematical expressions using Lex and Yacc syntax.
* The grammar includes rules for addition, subtraction, multiplication, division, parentheses, and numeric literals.
* In the yylex function, read characters from the input and check if they are numeric digits (0-9). If a digit is encountered, set yylval to the numeric value and return the NUMBER token.
* In the yyparse function, invoke the parser to start processing the input.
* In the yyerror function, handle and report any parsing errors.
* The main function calls yyparse to start the parsing process and then returns 0 to indicate successful execution.
* The program uses Lex and Yacc to parse and evaluate mathematical expressions entered by the user.
* The user can input expressions like "2 + 3 \* (4 - 1)" and the program will parse and evaluate them according to the defined rules.

**PROGRAM**

%{

#include <stdio.h>

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%%

expression: expression '+' expression

| expression '-' expression

| expression '\*' expression

| expression '/' expression

| '(' expression ')'

| NUMBER

;

%%

int yylex() {

int token = getchar();

if (token >= '0' && token <= '9') {

yylval = token - '0';

return NUMBER;

}

return token;

}

int main() {

yyparse();

return 0;

}

void yyerror(const char \*msg) {

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

}

**OUTPUT:**

-: Operator '-'

+: Operator '+'

NUMBER: Value '2'

\*: Operator '\*'

NUMBER: Value '3'

NUMBER: Value '4'

/: Operator '/'

NUMBER: Value '5'

NUMBER: Value '2'

**RESULT:**

Hence, implemented a program to recognize a valid arithmetic expression that uses operator.

**Ex.no:3b**

**Date:**

**YACC SPECIFICATION - VALID VARIABLE**

**AIM**

To implement a program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

**ALGORITHM**

* Define the tokens and operator precedence in the "%%" section.
* Define the grammar rules for parsing expressions.
* Implement the yylex() function to scan and return tokens.
  + Read characters from standard input.
  + If a digit is encountered, set yylval with the numeric value and return NUMBER.
  + Otherwise, return the character code itself.
* Implement the main() function.
  + Call yyparse() to start parsing the input.
  + Return 0 to indicate successful execution.
* Implement the yyerror() function to handle parsing errors.
  + Print an error message to the standard error stream.

**PROGRAM**

**LEX PROGRAM:**

%{

#include <stdio.h>

#include "y.tab.h"

%}

%option noyywrap

%%

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

. { return yytext[0]; }

%%

**YACC PROGRAM:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int yylex();

void yyerror(const char \*msg);

extern char\* yytext;

extern FILE\* yyin;

extern char\* yyval;

%}

%token IDENTIFIER

%%

program: identifier

| /\* empty \*/

;

identifier: IDENTIFIER {

if (is\_valid\_variable($1)) {

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

} else {

printf("Invalid variable: %s\n", $1);

}

free($1);

};

%%

int is\_valid\_variable(char\* var) {

// Check if the variable starts with a letter and contains only letters and digits.

if (isalpha(var[0])) {

for (int i = 1; i < strlen(var); i++) {

if (!isalnum(var[i])) {

return 0; // Invalid variable

}

}

return 1; // Valid variable

}

return 0; // Invalid variable

}

int main() {

yyin = stdin; // Set input to standard input

yyparse();

return 0;

}

void yyerror(const char \*msg) {

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

}

**OUTPUT:**

Valid variable: variable1

Invalid variable: invalid-variable$

Valid variable: anotherVariable2

**RESULT:**

Hence,implemented a program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

**Ex.No:3c**

**Date:**

**YACC SPECIFICATION - CONTROL STRUCTURES**

**AIM**

To implement a program to recognize a valid control structures syntax of C language (For loop, while loop, if-else, if-else-if, switch-case, etc.).

**ALGORITHM**

* Define token types:
  + IF, ELSE, WHILE, FOR, IDENTIFIER, and NUMBER are defined as integer constants representing different token types.
* Declare global variables:
  + yylval to store integer values associated with tokens.
  + yytext to store text associated with identifiers.
* Define function prototypes:
  + int yylex(): The lexer function that scans and returns tokens.
  + void yyerror(const char\* msg): A function to print error messages.
  + int yyparse(): The parser function responsible for implementing the parsing logic.
* Define grammar rules:
  + program: Represents the root of the grammar. It can be either a control\_structure or empty.
  + control\_structure: Represents high-level control structures like if\_else, for\_loop, or while\_loop.
  + if\_else: Defines the syntax for an if statement with or without an else clause.
  + for\_loop: Defines the syntax for a for loop.
  + while\_loop: Defines the syntax for a while loop.
  + expression: Represents mathematical expressions with operators (+, -, \*, /) and operands (IDENTIFIER or NUMBER).
  + for\_expr: Represents expressions used in the initialization and update parts of a for loop.
  + statement: Defines the syntax for statements, including compound statements in curly braces, control structures, and expressions followed by a semicolon.
* Implement the main function:
  + Calls yyparse to start the parsing process.
* Implement the yylex function:
  + Reads characters from the input and tokenizes them.
  + Recognizes keywords (if, else, while, for) and identifiers.
  + Handles numerical values and returns the appropriate token type.
* Implement the yyerror function:
  + Outputs an error message to stderr with the provided message.
* Implement the yyparse function:
  + This function should implement the parsing logic according to the defined grammar rules.
  + It returns 0 if parsing is successful and -1 if there's an error.

**PROGRAM**

**LEX PROGRAM:**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "y.tab.h"

%}

%%

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

char buffer[64];

strncpy(buffer, yytext, sizeof(buffer)-1);

buffer[sizeof(buffer)-1] = '\0';

if (strcmp(buffer, "if") == 0) {

return IF;

} else if (strcmp(buffer, "else") == 0) {

return ELSE;

} else if (strcmp(buffer, "while") == 0) {

return WHILE;

} else if (strcmp(buffer, "for") == 0) {

return FOR;

} else {

yylval = strdup(buffer);

return IDENTIFIER;

}

}

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

. { return yytext[0]; }

%%

**YACC PROGRAM:**

%{

#include <stdio.h>

#include <stdlib.h>

#include "y.tab.h"

extern int yylex();

extern int yylval;

extern char\* yytext;

%}

%token IF ELSE WHILE FOR IDENTIFIER NUMBER

%%

program: control\_structure | /\* empty \*/ ;

control\_structure: if\_else | for\_loop | while\_loop ;

if\_else: IF '(' expression ')' statement | IF '(' expression ')' statement ELSE statement ;

for\_loop: FOR '(' for\_expr ';' expression ';' for\_expr ')' statement ;

while\_loop: WHILE '(' expression ')' statement ;

expression: IDENTIFIER | NUMBER | expression '+' expression | expression '-' expression | expression '\*' expression | expression '/' expression ;

for\_expr: expression | /\* empty \*/ ;

statement: '{' program '}' | control\_structure | expression ';' ;

%%

**C PROGRAM FILE:**

if (x > 0) {

y = x + 1;

} else {

y = x - 1;

}

for (i = 0; i < 5; i++) {

printf("%d\n", i);

}

**OUTPUT:**

IF '(' expression ')' statement

'{' program '}'

ELSE statement

FOR '(' for\_expr ';' expression ';' for\_expr ')' statement

printf '(' expression ')' ';'

**RESULT:**

Hence, implemented a program to recognize a valid control structures syntax of C language

**Ex.No:4**

**Date:**

**CALCULATOR**

**AIM**

To implement Calculator using LEX and YACC

**ALGORITHM**

* Lexical Analysis (Lexer):
  + Skip whitespace characters like spaces, tabs, and newlines.
  + Match and return tokens for numeric literals (NUMBERS), parentheses ('(' and ')'), and arithmetic operators ('+', '-', '\*', and '/').
  + For numeric literals, store the integer value in the variable yylval.
* Syntax Analysis (Parser):
  + Define the grammar rules for the calculator expressions.
  + Parse the input to create a syntax tree for expressions.
  + Evaluate expressions and print the result using the %token declarations for NUMBER and OPERATOR.
* yywrap function:
  + Indicate the end of input by returning 1.
* main function:
  + Call yyparse to start the parsing process.
* yyerror function:
  + Handle and print parsing errors.

**PROGRAM**

**LEX PROGRAM:**

%{

#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;

}

**YACC PROGRAM:**

%{

#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 have operations:\n");

yyparse();

if(flag==0)

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

}

void yyerror()

{

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

flag=1;

}

**OUTPUT:**

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ ./a.out

Enter Any Arithmetic Expression which have operations:

9\*3

Result=27

Entered arithmetic expression is Valid

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ ./a.out

Enter Any Arithmetic Expression which have operations:

2\*3+1

Result=7

Entered arithmetic expression is Valid

**RESULT:**

Hence, Successfully implemented Calculator using LEX and YACC.

**Ex.No:5**

**Date:**

**THREE ADDRESS CODE**

**AIM**

To implement Three Address Code using LEX and YACC

**ALGORITHM**

1. Define necessary header files and global variables:

- Include necessary header files: `stdio.h`, `string.h`, `y.tab.h`.

- Declare the global variables `iden` (an array of characters) and `yylval` (an integer).

2. Define the Lex specification:

- Specify regular expressions and corresponding actions.

- Recognize integers (digits) and identifiers (letters).

- Ignore tab characters and return a newline character.

- For any other character, return the character itself.

3. Implement the `yywrap` function:

- This function returns 1 to signal the end of input.

4. Define additional header files and global variables:

- Include additional header files: `math.h`, `ctype.h`, `string.h`, and `stdio.h`.

- Declare a global variable `var\_cnt` to count temporary variables.

- Re-declare the `iden` variable, which is used for identifier storage.

5. Define token definitions:

- Define two tokens: `digit` and `id`.

6. Write the grammar rules for the parser:

- Define the S, E, T, F, and P rules using the tokens and operations.

- Print the corresponding code for each rule to generate a three-address code.

7. Implement the `main` function:

- Initialize `var\_cnt` to 0.

- Prompt the user to enter an expression.

- Call the `yyparse` function to parse the expression.

8. Implement the `yyerror` function:

- Display an "Error" message when a parsing error occurs.

**PROGRAM**

**LEX PROGRAM:**

%{

#include<stdio.h>

#include<string.h>

#include"y.tab.h"

extern char iden[20];

extern int yylval;

%}

%%

[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;

}

**YACC PROGRAM:**

%{

#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:**

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ yacc -d 3address.y

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ lex 3address.l

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ gcc lex.yy.c y.tab.c -w

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ ./a.out

Enter an expression :

a=2\*6-8

t0 = 2;

t1 = 6;

t2 = t0 \* t1;

t3 = 8;

t4 = t2 - t3;

a = t4

**RESULT:**

Hence,Successfully implemented Three Address Code using LEX and YACC.

**Ex.No:6**

**Date:**

**CODE GENERATION**

**AIM**

To implement Code Generation using LEX and YAAC

**ALGORITHM**

Declare an array icode to store intermediate code instructions and variables str and opr to process them.

Prompt the user to enter a set of intermediate code instructions and read them into the icode array. The input is terminated when the user enters "exit."

Initialize a loop variable i to 0.

Enter a loop to process each intermediate code instruction:

a. Copy the current instruction from icode[i] to the str variable.

b. Examine the operator (e.g., '+', '-', '\*', '/') at the 4th character of str.

c. Based on the operator, set the opr variable to the corresponding assembly-like operation ('ADD', 'SUB', 'MUL', or 'DIV').

d. Print assembly-like instructions to perform the operation:

"MOV %c,R%d" to move the value at the 2nd character of str to register i.

"%s %c,R%d" to perform the operation with the value at the 4th character of str and register i.

"MOV R%d,%c" to move the result back to the 0th character of str.

e. Increment i to process the next instruction.

Continue the loop until the input instruction is "exit."

Finally, return 0 to indicate successful execution.

**PROGRAM**

#include<stdio.h>

#include<string.h>

int main()

{

char icode[20][30];

char str[20],opr[10];

fflush(stdin);

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**:

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ gedit code.c

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ gcc code.c -o code

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ gcc code.c -o code

a12-32@a1232-ThinkStation-P330:~/4058\_CD$ ./code

Enter the set of intermediate code instructions:

a=b+c

c=d\*e

exit

MOV b,R0

ADD c,R0

MOV R0,a

MOV d,R1

MUL e,R1

MOV R1,c

MOV i,R2

MUL ,R2

MOV R2,e

**RESULT:**

Hence, Successfully implemented Code Generation using LEX and YAAC

**Ex.No:7**

**Date:**

**TYPE CHECKER**

**AIM:**

To implement a type checker program using lex and yacc.

**LEX PROGRAM:**

%{

#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;

}

**YACC PROGRAM:**

%{

#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:**

a12-21@a1221-ThinkStation-P330:~$ lex sam.l

a12-21@a1221-ThinkStation-P330:~$

a12-21@a1221-ThinkStation-P330:~$ bison -d -y sam.y

a12-21@a1221-ThinkStation-P330:~$

a12-21@a1221-ThinkStation-P330:~$ gcc -o sam lex.yy.c y.tab.c -w

a12-21@a1221-ThinkStation-P330:~$

a12-21@a1221-ThinkStation-P330:~$ ./sam

3

Result: 258 (integer)

7.9

Result: 0.000000 (float)

j

Invalid character: j

syntax error

**RESULT:**

Hence, Successfully implemented a type checker program using lex and yacc.

**Ex.no :**

**Date :**

**CODE OPTIMIZATION**

**AIM**

To implement Simple code Optimization techniques using C

**ALGORITHM:**

Algorithm:

Constant Folding:

Initialize a function constantFolding that returns an integer.

Inside the function, compute and return the result of the constant expression 2 + 3 \* 4.

Strength Reduction:

Initialize a function strengthReduction that takes an integer parameter x and returns an integer.

Inside the function, replace the multiplication by 2 with a left shift operation (x \* 2 becomes x << 1).

Return the result.

Algebraic Transformation:

Initialize a function algebraicTransformation that takes an integer parameter x and returns an integer.

Inside the function, perform the algebraic transformation on the expression x \* 5 - x \* 3 and return the result.

Main Function:

Inside the main function:

Call the constantFolding function and store the result in resultConstantFolding.

Print the result of constant folding using printf.

Define an integer variable inputValue with an example value (e.g., 7).

Call the strengthReduction function with inputValue and store the result in resultStrengthReduction.

Print the result of strength reduction using printf.

Call the algebraicTransformation function with inputValue and store the result in resultAlgebraicTransformation.

Print the result of algebraic transformation using printf.

Return 0 to indicate successful program execution.

**PROGRAM:**

#include <stdio.h>

// Function to demonstrate constant folding

int constantFolding() {

return 2 + 3 \* 4; // Constant expression

}

// Function to demonstrate strength reduction

int strengthReduction(int x) {

return x \* 2; // Strength reduction: multiplication by 2 can be replaced with a left shift

}

// Function to demonstrate algebraic transformation

int algebraicTransformation(int x) {

return x \* 5 - x \* 3; // Algebraic transformation: x \* 5 - x \* 3 can be simplified to x \* 2

}

int main() {

// Constant Folding

int resultConstantFolding = constantFolding();

printf("Constant Folding Result: %d\n", resultConstantFolding);

// Strength Reduction

int inputValue = 7;

int resultStrengthReduction = strengthReduction(inputValue);

printf("Strength Reduction Result: %d\n", resultStrengthReduction);

// Algebraic Transformation

int resultAlgebraicTransformation = algebraicTransformation(inputValue);

printf("Algebraic Transformation Result: %d\n", resultAlgebraicTransformation);

return 0;

}

**OUTPUT:**

Constant Folding Result: 14

Strength Reduction Result: 14

Algebraic Transformation Result: 14

**RESULT:**

Hence the program to implement simple code optimization techniques is writen.