Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

name.lex:

```
name .*[pP][rR][aA][dD].*
%%
{name} printf("Invalid\n");
exit exit(0);
.* printf("Valid\n");
%%
int main()
{
   yylex();
}

int yywrap() {
   return 1;
}
```

OUTPUT:

```
qwerty
Valid

Pradyumn
Invalid

asdfkjlfbpradkaslfsd
Invalid

lfsadnh_2132/.
Valid
```

Date: 07/08/2024

Experiment 2.1

AIM

To write a lex program to recognize all strings which do not contain the first four characters of our name as a substring.

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Define regex name as .*[pP][rR][aA][dD].* to recognize strings containing the first four letters of my name as a substring
    2. If input matches the above substring, print “Invalid”
    3. Otherwise, if input matches the string”exit”, terminate the program
    4. Otherwise, print “Valid”
    5. Define main function in suer code section to call yylex()
3. Use lex command to generate C program
4. Run the C program
5. Stop

RESULT

Successfully compiled and ran the lex program.

Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

program_2.lex:

```
%{
#include <stdlib.h>
#include "y.tab.h"
#include <string.h>
%}
%option noyywrap
%%
[a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return IDENTIFIER; }
\n { return '\n'; }
.* { yylval.str = strdup(yytext); return INVALID; }
%%
```

program_2.y:

```
%{
#include <stdio.h>
#define YYSTYPE char*
%}
%union {
   char* str;
}
%token IDENTIFIER
%token INVALID
%%
program: line
   | line program;

line: IDENTIFIER "\n" {printf("Valid\n");}
   | INVALID "\n" {printf("Invalid\n");}
%%
```

Date: 07/08/2024

Experiment 2.2

AIM

To write a YACC program to identify valid variable names

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input starts with a letter followed by a combination of letters and digits:
        1. Copy value to yylval.
        2. Return token IDENTIFIER.
    3. If input is a newline character, return newline character.
    4. For any other string:
        1. Copy string to yylval.
        2. Return token INVALID.
3. Create YACC to parse input as follows:
    1. The input consists of multiple lines.
    2. Each line can be either:
        1. An identifier followed by a newline character. In this case, display “Valid”.
        2. An invalid token followed by a newline character. In this case, display “Invalid”.
    3. In user code section:
        1. Define error handling.
        2. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program
7. Stop

```
void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

OUTPUT:

```
abcd123
Valid
456gef
Invalid
a_b_1
Invalid
a123@
Invalid
pqr987
Valid
```

RESULT

Successfully compiled and ran the YACC and LEX program.

Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

calc.lex:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
#include <string.h>
extern int yylval;
%}
%option noyywrap
%%
[0-9]+ { yylval = atoi(yytext); return IDENTIFIER; }
[+\-*/()\n] { return yytext[0]; }
. { printf("Invalid character: %s\n", yytext); return INVALID; }
%%
```

calc.y:

```
%{
#include <stdio.h>
#define YYSTYPE int
extern YYSTYPE yylval;
%}
%token IDENTIFIER
%token INVALID
%%
program: line
    | program line;

line: exp "\n" {printf("Result: %d\n",$1);}

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

Date: 07/08/2024

Experiment 2.3

AIM

To implement a calculator using LEX and YACC

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input is a sequence of digits:
        1. Copy value to yylval.
        2. Return token IDENTIFIER.
    3. If input is an arithmetic operator or a newline character, return the first character in the input.
3. Create YACC to parse input as follows:
    1. The input consists of multiple lines.
    2. Each line contains an expression followed by a newline character.
    3. Each expression can be expanded as one of the following:
        1. expression + term

            In this case, the value of resultant expression is the sum of values of the first expression and the term.

        2. expression – term

            In this case, the value of resultant expression is the difference of values of the first expression and the term.

        3. term

            In this case, the value of the expression is that of the term.
    4. Each term can be expanded as one of the following:
        1. term * factor

            In this case, the value of resultant term is the product of values of the first term and the factor.

        2. term / factor

```
    | exp "-" term {$$ = $1-$3;}
    | term {$$ = $1;};

term: term "*" factor {$$ = $1*$3;};
    | term "/" factor {$$ = $1/$3;};
    | factor {$$ = $1;};

factor: "(" exp ")" {$$ = $2;}
    | IDENTIFIER {$$ = $1;};
%%
void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

OUTPUT:

15-3*4

Result: 3

20/4+7

Result: 12

1-1-1

Result: -1

10+20/(5-3)

Result: 20

In this case, the value of resultant term is the difference of values of the first term and the factor.

3. factor

   In this case, the value of the term is that of the factor.

5. A factor can be expanded as follows:
   1. A factor can be an IDENTIFIER token. In this case, the value of the factor is the numerical value of the identifier.
   2. (expression)

      In this case, the value of the factor is that of the expression within the brackets.
6. In user code section:
   1. Define error handling.
   2. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program
7. Stop

RESULT

Successfully implemented a calculator with YACC and LEX.

Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

ast.lex:

```
%{
#include "y.tab.h"
#include <string.h>
%}
%option noyywrap
%%
[0-9]+ { yylval.intval = atoi(yytext); return INTEGER; }
[a-zA-Z_][a-zA-Z0-9_]* {yylval.strval = strdup(yytext); return IDENTIFIER;}
[+\-*/();=] { return yytext[0]; }
. { /*Skip remaining characters */ }
%%
```

ast.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}

%type <nodeType> program
%type <nodeType> stmt
%type <nodeType> exp
%type <nodeType> term
%type <nodeType> factor
%token <strval> IDENTIFIER
%token <intval> INTEGER

%{
enum ASTKind {
```

Date: 14/08/2024

Experiment 2.4

AIM

To generate abstract syntax tree of an expression

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input is a sequence of digits:
        1. Copy value to yylval
        2. Return token INTEGER
    3. If input is a sequence of alphabets, digits, or ‘_’ character sucha that it doesn’t start with a digit:
        1. Copy value to yylval
        2. Return token IDENTIFIER
    4. If input is an arithmetic operator, parenthesis, semicolon, ‘=’ symbol, or a newline character, return the first character in the input.
    5. For any other symbol, do nothing.
3. Create YACC to parse input as follows:
    1. Create structure AST to represent node of abstract syntax tree with following attributes:
        1. Union val storing value of the node. This can be an integer value, the name of the identifier, or a binary operator.
        2. Variable kind to identify the type of value stored.
        3. Left and right subtree pointers.
    2. Declare following functions:
        1. make_int takes a numerical value as input and returns an AST leaf node with that value.
        2. make_id takes string identifier as input and returns an AST leaf node with that value.
        3. make_binop takes an operator and two AST nodes as input and returns a tree with a node representing the operator as root node with the input nodes as children.

```
    AST_INT,
    AST_VAR,
    AST_BINOP,
    AST_ASSIGN
};
struct AST {
    union ASTType{
        int intval;
        char *idname;
        char binop;
    } val;
    enum ASTKind kind;
    struct AST* left;
    struct AST* right;
};
%}

%union {
    int intval;
    char *strval;
    struct AST* nodeType;
}

%{
void addASTList(struct AST* n);
struct AST *make_int(int val);
struct AST *make_id(const char *s);
struct AST *make_binop(char op, struct AST *l, struct AST *r);
struct AST *make_assign(const char *name, struct AST *r);
%}

%%
```

4. make_assign takes a string value and an AST node as input and makes node assigning the string as value and the AST node as rightr subtree.
5. print_ast recursively prints a given tree with appropriate indentation.

3. Define parser grammar as follows:
    1. The input consists of multiple statements. Each statement returns an AST which should be printed and deleted after parsing.
    2. Each statement contains one of the following:
        1. IDENTIFIER = expression ;

            In this case, value of the statement can be obtained as make_assign(identifier,expression).

        2. expression ;

            In this case, the value of statement is that of the expression.
    3. Each expression can be expanded as one of the following:
        1. expression + term

            In this case, the value of resultant expression can be obtained as make_binop(‘+’,expression, term).

        2. expression – term

            In this case, the value of resultant expression can be obtained as make_binop(‘-’,expression, term).

        3. term

            In this case, the value of the expression is that of the term.
    4. Each term can be expanded as one of the following:
        1. term * factor

            In this case, the value of resultant term can be obtained as make_binop(‘*’,term, factor).

        2. term / factor

            In this case, the value of resultant term can be obtained as make_binop(‘/’,term, factor).

        3. factor

            In this case, the value of the term is that of the factor.
    5. A factor can be expanded as follows:
        1. A factor can be an IDENTIFIER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_id function.

```
program: /* empty*/ { $$ = NULL; }
    | program stmt; { $$ = NULL; print_ast($2,0);free_ast($2);}

stmt: exp ";" {$$ = $1;}
    | IDENTIFIER "=" exp ";" { $$ = make_assign($1, $3); };

exp: exp "+" term {$$ = make_binop('+',$1,$3);};
    | exp "-" term {$$ = make_binop('-',$1,$3);};
    | term {$$ = $1;};

term: term "*" factor {$$ = make_binop('*',$1,$3);};
    | term "/" factor {$$ = make_binop('/',$1,$3);};
    | factor {$$ = $1;};

factor: "(" exp ")" {$$ = $2;}
    | INTEGER {$$ = make_int($1);}
    | IDENTIFIER { $$ = make_id($1); };
%%
struct AST *make_int(int val){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = NULL;
    n->right = NULL;
    n->val.intval = val;
    n->kind = AST_INT;
    return n;
}
struct AST *make_id(const char *s){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = NULL;
    n->right = NULL;
    n->val.idname = strdup(s);
    n->kind = AST_VAR;
```

2. A factor can be an INTEGER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_int function.
3. (expression)

   In this case, the value of the factor is that of the expression within the brackets.

4. In user code section:
   1. Define error handling.
   2. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program
7. Stop

```
    return n;
}
struct AST *make_binop(char op, struct AST *l, struct AST *r){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = l;
    n->right = r;
    n->val.binop = op;
    n->kind = AST_BINOP;
    return n;
}
struct AST *make_assign(const char *name, struct AST *r){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = NULL;
    n->right = r;
    n->val.idname = name;
    n->kind = AST_ASSIGN;
    return n;
}

void print_ast(struct AST *a, int indent){
    if (!a) return;
    for (int i=0;i<indent;++i){
        printf("-");
    }
    switch(a->kind){
        case AST_BINOP:
            printf("Binop(%c)\n",a->val.binop);
            break;
        case AST_INT:
            printf("Int(%d)\n",a->val.intval);
            break;
        case AST_VAR:
```

```
            printf("Id(%s)\n",a->val.idname);
            break;
        case AST_ASSIGN:
            printf("Assign(%s)\n",a->val.idname);
    }
    print_ast(a->left,indent+2);
    print_ast(a->right,indent+2);
}

void free_ast(struct AST* a){
    if (!a) return;
    free_ast(a->left);
    free_ast(a->right);
    free(a);
}

void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

OUTPUT:

x=5;y=10;z=x+y*3/5;

Assign(x)

--Int(5)

Assign(y)

--Int(10)

Assign(z)

--Binop(+)

----Id(x)

----Binop(/)

------Binop(*)

--------Id(y)

--------Int(3)

------Int(5)

a = 1+(2*3+(6/2)-1)*9;

Assign(a)

--Binop(+)

----Int(1)

----Binop(*)

------Binop(-)

--------Binop(+)

----------Binop(*)

----------Int(2)

----------Int(3)

----------Binop(/)

----------Int(6)

----------Int(2)

--------Int(1)

------Int(9)

RESULT

Successfully generated abstract syntax tree with YACC and LEX.

Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

for.lex:

```
%{
#include "y.tab.h"
#include <string.h>
%}
%option noyywrap
%%
[0-9]+ { return INTEGER; }
for { return FOR; }
int|float|char { return TYPE; }
[a-zA-Z_][a-zA-Z0-9_]* {return IDENTIFIER;}
(==) { return RELATIONAL_OPERATOR; }
=|(\+=)|(-=)|(\*=)|(\/=)|(\|\|=)|(&&=)|(<<=)|(>>=)|(\^=)|(%=) { return ASSIGN; }
[+/%*\-] { return ARITHMETIC_OPERATOR; }
(<=)|(>=)|(!=)|[<>] { return RELATIONAL_OPERATOR; }
(\|\|)|(&&) { return LOGICAL_OPERATOR; }
(<<)|(>>)|[&|^] { return BITWISE_OPERATOR;}
\( { return LPAREN; }
\) { return RPAREN; }
\{ { return LCURLY; }
\} { return RCURLY; }
; { return SEMICOLON; }
, { return COMMA; }
! { return NOT; }
. { /*Skip remaining characters */ }
%%
```

Date: 14/08/2024

Experiment 2.5

AIM

To check syntax of for loop in C

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by the YACC program.
    2. If input is a sequence of digits:
        1. Return token INTEGER.
    3. If input is the keyword for:
        1. Return token FOR.
    4. If input is a type keyword (int, float, char):
        1. Return token TYPE.
    5. If input is a valid identifier (starts with a letter or _, followed by letters, digits, or _):
        1. Return token IDENTIFIER.
    6. If input matches relational operators (==, <=, >=, !=, <, >):
        1. Return token RELATIONAL_OPERATOR.
    7. If input matches assignment operators (=, +=, -=, *=, /=, etc.):
        1. Return token ASSIGN.
    8. If input matches arithmetic operators (+, -, *, /, %):
        1. Return token ARITHMETIC_OPERATOR.
    9. If input matches logical operators (||, &&):
        1. Return token LOGICAL_OPERATOR.
    10. If input matches bitwise operators (<<, >>, &, |, ^):
        1. Return token BITWISE_OPERATOR.
    11. If input matches parentheses, curly braces, semicolon, comma, or !:
        1. Return the corresponding token (LPAREN, RPAREN, LCURLY, RCURLY, SEMICOLON, COMMA, NOT).
    12. For any other symbol, skip.

for.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}

%token IDENTIFIER
%token INTEGER
%token FOR
%token TYPE
%token LOGICAL_OPERATOR
%token RELATIONAL_OPERATOR
%token ARITHMETIC_OPERATOR
%token BITWISE_OPERATOR
%token LPAREN
%token RPAREN
%token LCURLY
%token RCURLY
%token SEMICOLON
%token COMMA
%token NOT
%token ASSIGN

%left LOGICAL_OPERATOR
%left RELATIONAL_OPERATOR
%left ARITHMETIC_OPERATOR
%left BITWISE_OPERATOR
%right NOT
%left LPAREN RPAREN
```

3. Create YACC to parse input as follows:
    1. Define tokens for all operators, keywords, and symbols as per the LEX file.
    2. Define grammar rules to recognize a valid C-style for loop:
        1. The input consists of zero or more for loops.
        2. Each for loop must match the pattern:

            for (initialization ; condition ; update) statement_block

            On successful recognition of a for loop, print Valid.
        3. Initialization can be a comma seperated series of variable declarations or assignments. Initialization can be NULL as well.
        4. Condition can be any logical or relational expression.
        5. Update can be assignments or arithmetic expressions.
        6. Statement block can be a single statement or a block enclosed in {}.
    3. In the user code section:
        1. Define error handling function.
        2. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program using gcc.
7. Stop

```
%%
//Grammar
program: /* empty */
   | program for_loop {printf("Valid\n");}
for_loop: FOR LPAREN init_statements SEMICOLON logic_opt SEMICOLON update_stmts
RPAREN stmt_block;
init_statements:  /* empty */
   | init_stmt_list;
logic_opt: /* empty */
   | logic_exp;
init_stmt_list: init_stmt
   | init_stmt_list COMMA init_stmt;
init_stmt: assignment_stmt
   | TYPE id_def;
update_stmts:  /* empty */
   | update_stmt_list;
update_stmt_list: update_stmt
   | update_stmt_list COMMA update_stmt;
update_stmt: assignment_stmt
   | arithmetic_exp;
stmt_block:LCURLY stmt_list RCURLY
   | stmt;
stmt_list: stmt
   | stmt_list stmt;
stmt: assignment_stmt SEMICOLON
   | for_loop
   | declaration_stmt SEMICOLON
   | arithmetic_exp SEMICOLON;
assignment_stmt: IDENTIFIER assignment_operator arithmetic_exp;
assignment_operator: ASSIGN
   ;
declaration_stmt : TYPE id_list
id_list : id_list COMMA id_def
```

```
    | id_def;
id_def: IDENTIFIER
    | assignment_stmt;
arithmetic_exp: arithmetic_exp ARITHMETIC_OPERATOR arithmetic_exp
    | arithmetic_exp BITWISE_OPERATOR arithmetic_exp
    | LPAREN arithmetic_exp RPAREN
    | IDENTIFIER
    | INTEGER;
logic_exp: arithmetic_exp RELATIONAL_OPERATOR arithmetic_exp
    | logic_exp LOGICAL_OPERATOR logic_exp
    | NOT logic_exp
    | LPAREN logic_exp RPAREN;

%%
void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

OUTPUT:

```
for (int i=0;i<n;i+=1) {
    a = i*3;
    b = 6;
}
for (a=0,b=1,c=2;c<10;a=a+3,b=b-2,c=c+5) x = a*b*c;
for (int i=0;i<n;i=i+1){
    for (int j=0;j<m;j=j+2){
        x = x + i-j;
    }
}
for (;;) ;
for (;;;)
```

output:

Valid

Valid

Valid

Error: syntax error

RESULT

Successfully verified for loop syntax