Lab Assignment – 4

Experiment 12:

Algorithm:

- **Lexical analysis (Calc.l)**
    - Read input character by character.
    - If digits - convert to integer - return token `NUMBER`.
    - If whitespace - ignore.
    - If newline - return `\n`.
    - Otherwise - return the character itself (operator or parenthesis).
- **Parser initialization (Calc.y)**
    - Define grammar rules for arithmetic expressions.
    - Handle operator precedence: `* /` higher than `+ -`.
    - Use recursive rules for expressions.
- **Program rule**
    - Repeatedly read expressions followed by newline.
    - For each valid expression, print result.
- **Expression evaluation**
    - If token is `NUMBER` - value is that integer.
    - If `expr + expr` - compute sum.
    - If `expr - expr` - compute difference.
    - If `expr * expr` - compute product.
    - If `expr / expr` -
        - If divisor = 0 - print error "Division by zero", result = 0.
        - Else - compute quotient.
    - If `( expr )` - result = inner expression value.

Code:

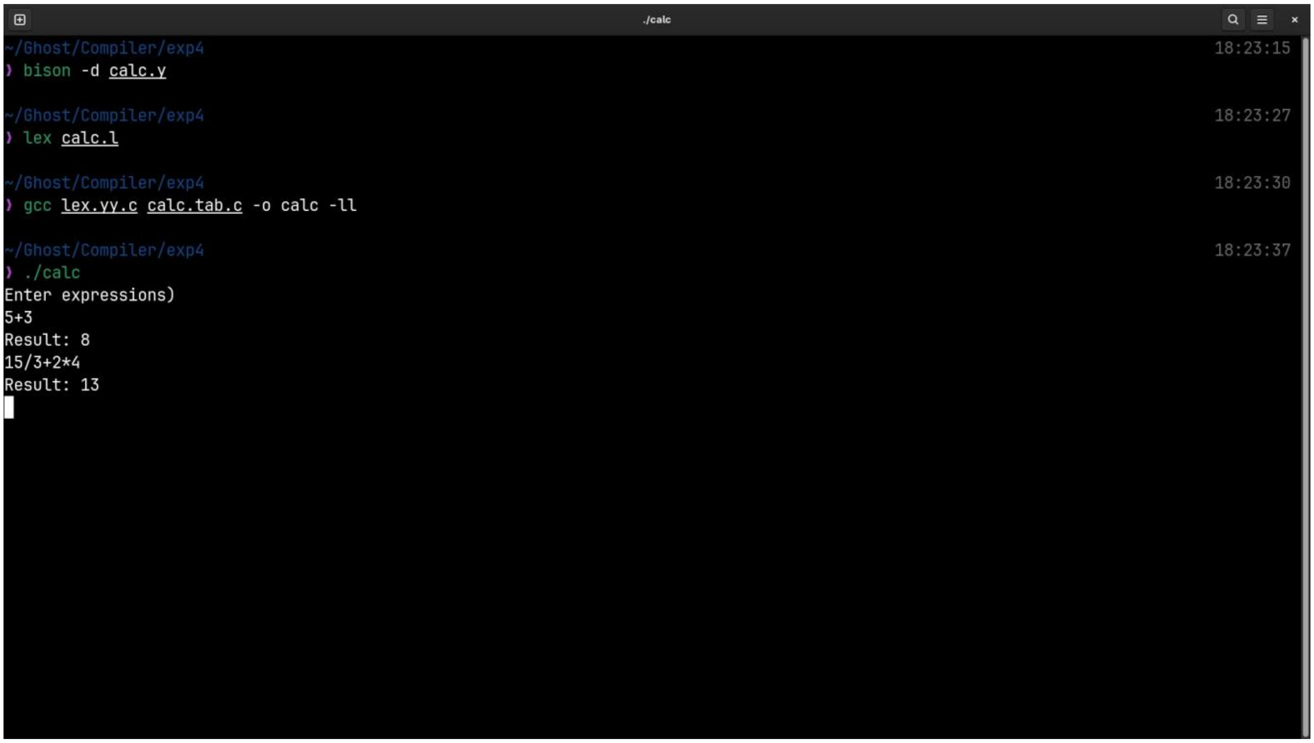
Calc.l:

```
%{
#include "calc.tab.h"
#include <stdlib.h>
%}

%%
[0-9]+         { yylval = atoi(yytext); return NUMBER; }
[ \t]          ; // ignore whitespace
\n             { return '\n'; }
.              { return yytext[0]; }
%%

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

Calc.y:

```
%{
#include  <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
program:
    program expr '\n'   { printf("Result: %d\n", $2); }
    | /* empty */
    ;

expr:
    NUMBER             { $$ = $1; }
    | expr '+' expr     { $$ = $1 + $3; }
    | expr '-' expr     { $$ = $1 - $3; }

    | expr '*' expr     { $$ = $1 * $3; }

    | expr '/' expr    {

        if ($3 == 0) {

            yyerror("Division by zero");

            $$ = 0;

        } else {

            $$ = $1 / $3;

        }

    }

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

    ;


%%


void yyerror(const char *s) {

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

}


int main() {

    printf("Enter expressions)\n");

    yyparse();

    return 0;

}
```

Output:



```
~/Ghost/Compiler/exp4                                                                              18:23:15
❯ bison -d calc.y

~/Ghost/Compiler/exp4                                                                              18:23:27
❯ lex calc.l

~/Ghost/Compiler/exp4                                                                              18:23:30
❯ gcc lex.yy.c calc.tab.c -o calc -ll

~/Ghost/Compiler/exp4                                                                              18:23:37
❯ ./calc
Enter expressions)
5+3
Result: 8
15/3+2*4
Result: 13
```

Experiment 13:

Algorithm:

- **Lexical analysis (Infix.l)**
    - If input is a letter - return token `ID`.
    - If whitespace - ignore.
    - If newline - return `\n`.
    - Otherwise - return character itself (operators, parentheses).
- **Parser initialization (Infix.y)**
    - Define grammar for expressions with precedence rules:
        - `* /` higher than `+ -`.
    - Union type stores character values.
- **Program rule**
    - Repeatedly read expressions followed by newline.
    - After each expression, print newline.
- **Expression rules (convert to postfix)**
    - If token is `ID` - print the variable name.
    - If `expr + expr` - print operands first, then print `"+"`.
    - If `expr - expr` - print operands first, then print `"-"`.
    - If `expr * expr` - print operands first, then print `"*"`.
    - If `expr / expr` - print operands first, then print `"/"`.
    - If `( expr )` - evaluate inside parentheses.

Code:

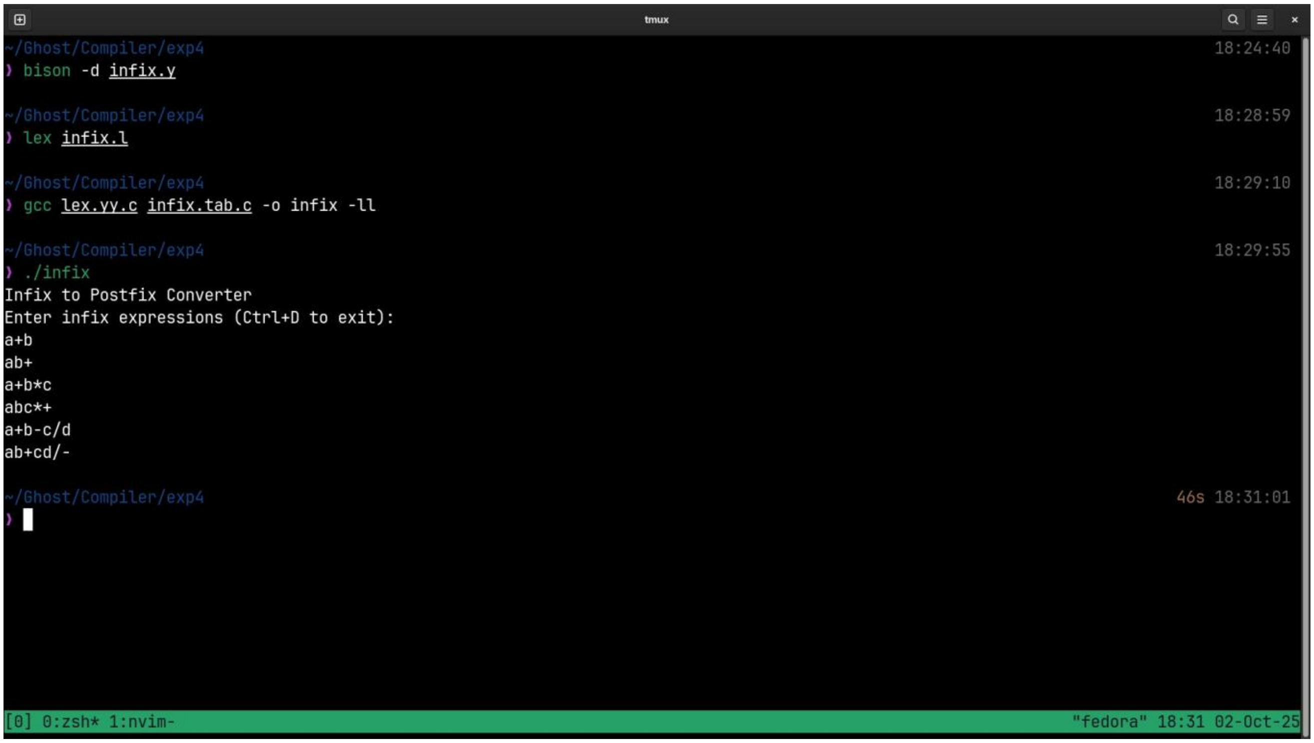
Infix.l:

```
%{
#include "infix.tab.h"
%}

%%
[a-zA-Z]        { yylval.c = yytext[0]; return ID; }
[ \t]           ; // ignore whitespace
\n              { return '\n'; }
.               { return yytext[0]; }
%%

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

Infix.y:

```
%{
#include  <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}

%union {
    char c;
}

%token <c> ID
%type <c> expr
%left '+' '-'
%left '*' '/'

%%
program:
    program expr '\n'   { printf("\n"); }
    | /* empty */
    ;

expr:
    ID                    { printf("%c", $1); $$ = $1; }
    | expr '+' expr      { printf("+"); $$ = '+'; }
    | expr '-' expr      { printf("-"); $$ = '-'; }
    | expr '*' expr      { printf("*"); $$ = '*'; }
    | expr '/' expr      { printf("/"); $$ = '/'; }
    | '(' expr ')'       { $$ = $2; }
    ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Infix to Postfix Converter\n");
    printf("Enter infix expressions (Ctrl+D to exit):\n");
    yyparse();
    return 0;
}
```

Output:



```
~/Ghost/Compiler/exp4                                                                                  18:24:40
❯ bison -d infix.y

~/Ghost/Compiler/exp4                                                                                  18:28:59
❯ lex infix.l

~/Ghost/Compiler/exp4                                                                                  18:29:10
❯ gcc lex.yy.c infix.tab.c -o infix -ll

~/Ghost/Compiler/exp4                                                                                  18:29:55
❯ ./infix
Infix to Postfix Converter
Enter infix expressions (Ctrl+D to exit):
a+b
ab+
a+b*c
abc*+
a+b-c/d
ab+cd/-

~/Ghost/Compiler/exp4                                                                              46s 18:31:01
❯ █

[0] 0:zsh* 1:nvim-                                                               "fedora" 18:31 02-Oct-25
```

ADITYA
23BCE1344

Experiment 14a:

Algorithm:

- **Lexical analysis (Lang1.l)**
    - If input is `a` - return token `A`.
    - If input is `b` - return token `B`.
    - If newline - return `\n`.
    - Otherwise - return the character itself.
- **Parser initialization (Lang1.y)**
    - Define tokens `A`, `B`.
    - Maintain counters: `a_count = 0`, `b_count = 0`.
- **Program rule**
    - Program consists of multiple `line`s.
- **Line rule**
    - If line matches `S \n`:
        - If `a_count != b_count` and both counts > 0 - print `"ACCEPTED: a^n b^m (m != n)"`.
        - Else - print `"REJECTED"`.
        - Reset `a_count` and `b_count`.
    - If only `\n` - reset counts.
    - If error in line - print `"REJECTED"`, reset counts, continue.
- **S rule**
    - String must be `alist` followed by `blist`.
- **alist rule**
    - Count consecutive `A` tokens (increment `a_count`).
- **blist rule**
    - Count consecutive `B` tokens (increment `b_count`).

Code:

Lang1.l:

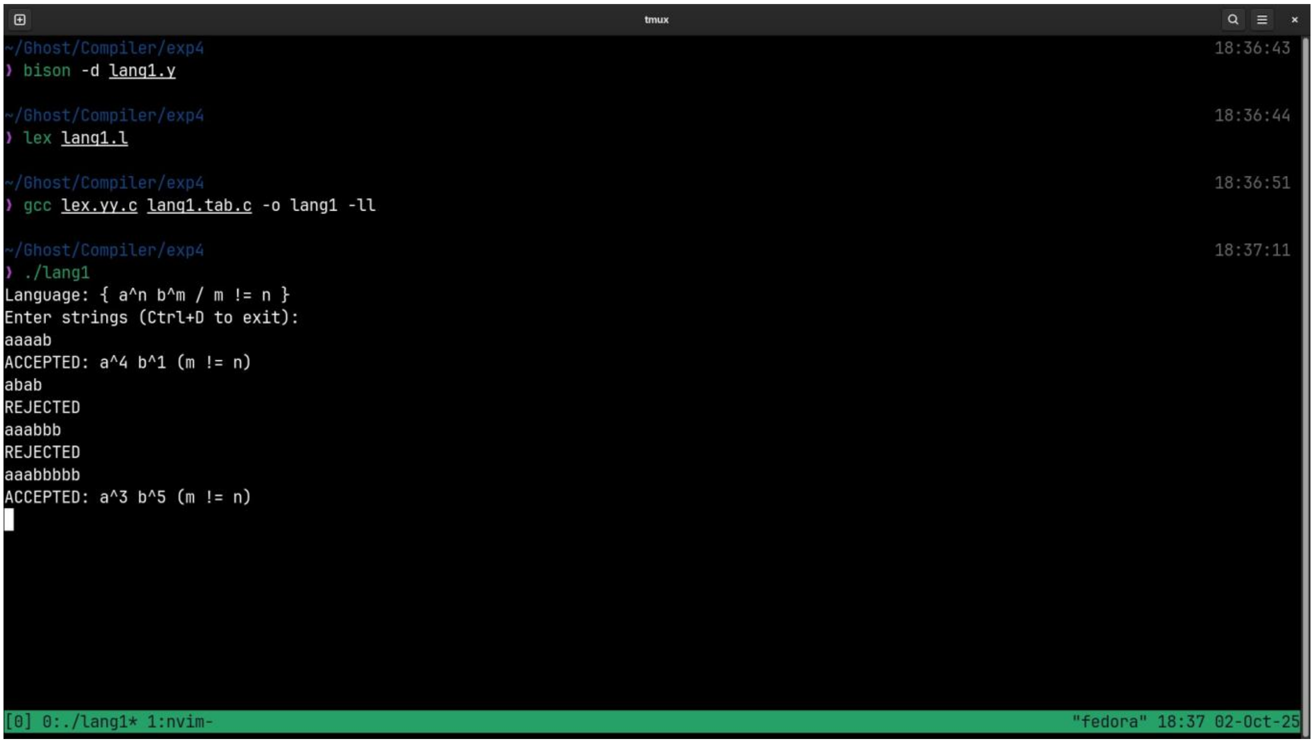
```
%{
#include "lang1.tab.h"
%}

%%
a           { return A; }
b           { return B; }
\n          { return '\n'; }
.           { return yytext[0]; }
%%

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

Lang1.y:

```
%{
#include  <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
int a_count = 0, b_count = 0;
%}

%token A B

%%
program:
   program line
   | /* empty */
   ;

line:
   S '\n'  {
      if (a_count != b_count && a_count > 0 && b_count > 0) {
         printf("ACCEPTED: a^%d b^%d (m != n)\n", a_count, b_count);
```

```
        } else {
            printf("REJECTED\n");
        }
        a_count = 0; b_count = 0;
    }
    | '\n'  { a_count = 0; b_count = 0; }
    | error '\n' {
        printf("REJECTED\n");
        a_count = 0; b_count = 0;
        yyerrok;
    }
    ;

S:
    alist blist
    ;

alist:
    alist A     { a_count++; }
    | A         { a_count++; }
    ;

blist:
    blist B     { b_count++; }
    | B         { b_count++; }
    ;

%%

void yyerror(const char *s) {
    /* Error handled in line rule */
}

int main() {
    printf("Language: { a^n b^m / m != n }\n");
    printf("Enter strings (Ctrl+D to exit):\n");
    yyparse();
    return 0;
}
```

Output:



```
~/Ghost/Compiler/exp4                                                    18:36:43
❯ bison -d lang1.y

~/Ghost/Compiler/exp4                                                    18:36:44
❯ lex lang1.l

~/Ghost/Compiler/exp4                                                    18:36:51
❯ gcc lex.yy.c lang1.tab.c -o lang1 -ll

~/Ghost/Compiler/exp4                                                    18:37:11
❯ ./lang1
Language: { a^n b^m / m != n }
Enter strings (Ctrl+D to exit):
aaaab
ACCEPTED: a^4 b^1 (m != n)
abab
REJECTED
aaabbb
REJECTED
aaabbbbb
ACCEPTED: a^3 b^5 (m != n)

[0] 0:./lang1* 1:nvim-                                "fedora" 18:37 02-Oct-25
```

Experiment 14b:

Algorithm:

- **Lexical analysis (Lang2.l)**
    - If input is `a` - return token `A`.
    - If input is `b` - return token `B`.
    - If newline - return `\n`.
    - Otherwise - return character itself.
- **Parser initialization (Lang2.y)**
    - Define tokens `A`, `B`.
    - Maintain counter `n_value = 0` for tracking number of `(bbaa)` repetitions and balancing with `(ba)` repetitions.
- **Program rule**
    - Program consists of multiple `line`s.
- **Line rule**
    - If line matches grammar `S \n`:
        - Print "`ACCEPTED (n = <value>)`".
        - Reset `n_value = 0`.
    - If line is just `\n`: reset counter.
    - If invalid - print "`REJECTED (syntax error)`", reset counter, continue.
- **S rule**
    - Input must begin with sequence `A B` (i.e., "`ab`").
    - Then followed by `X` and `Y`.
- **X rule (left side repetitions)**
    - Either `B B` (base case - "`bba`" starts here).
    - Or recursively `X B B A A`: each repetition of "`bbaa`" increases `n_value` by 1.
- **Y rule (right side balancing)**
    - Either single `A` (base case - final "`a`").
    - Or recursively `Y B A`: each "`ba`" decreases `n_value` by 1.

- If n_value drops below 0 - error "Too many BA pairs".

1. **Acceptance condition**
   - Grammar ensures correct order.
   - n_value increments for each (bbaa) and decrements for each (ba).
   - Valid string ends with n_value = 0.

Code:

Lang2.l:

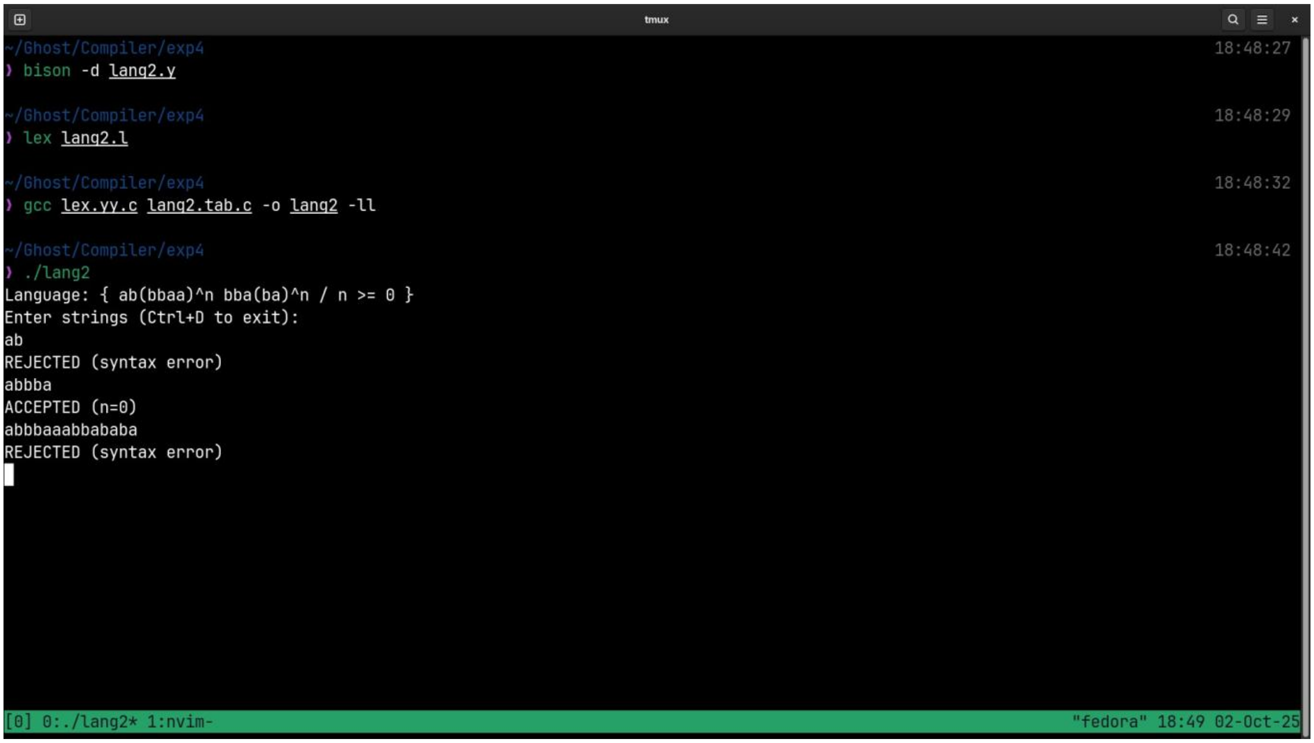
```
%{
#include "lang2.tab.h"
%}

%%
a           { return A; }
b           { return B; }
\n          { return '\n'; }
.           { return yytext[0]; }
%%

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

Lang2.y:

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

int yylex();
void yyerror(const char *s);
int n_value = 0;
%}

%token A B

%%
```

```
program:
    program line
    | /* empty */
    ;

line:
    S '\n'     {
        printf("ACCEPTED (n=%d)\n", n_value);
        n_value = 0;
    }
    | '\n'    {
        n_value = 0;
    }
    | error '\n' {
        printf("REJECTED (syntax error)\n");
        n_value = 0;
        yyerrok;
    }
    ;

S:
    A B X Y
    ;

X:
    X B B A A  { n_value++; }
    | B B      /* base case: bba part */
    ;

Y:
    Y B A      {
        n_value--;
        if (n_value < 0) {
            yyerror("Too many BA pairs");
            YYERROR;
        }
    }
    | A        /* base case: final 'a' */
    ;

%%

void yyerror(const char *s) {
    /* Error handled in line rule */
}

int main() {
    printf("Language: { ab(bbaa)^n bba(ba)^n / n >= 0 }\n");
    printf("Enter strings (Ctrl+D to exit):\n");
    yyparse();
    return 0;
}
```

Output:



```
~/Ghost/Compiler/exp4                                                                    18:48:27
❱ bison -d lang2.y

~/Ghost/Compiler/exp4                                                                    18:48:29
❱ lex lang2.l

~/Ghost/Compiler/exp4                                                                    18:48:32
❱ gcc lex.yy.c lang2.tab.c -o lang2 -ll

~/Ghost/Compiler/exp4                                                                    18:48:42
❱ ./lang2
Language: { ab(bbaa)^n bba(ba)^n / n >= 0 }
Enter strings (Ctrl+D to exit):
ab
REJECTED (syntax error)
abbba
ACCEPTED (n=0)
abbbaaabbababa
REJECTED (syntax error)

[0] 0:./lang2* 1:nvim-                                              "fedora" 18:49 02-Oct-25
```