Homework 7

Name: 陳偉俊

Student ID: 61347090S

Code

Lex Code

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+  {
    yylval = atoi(yytext);
    printf("LEX: recognized NUMBER: %s\n", yytext);
    return NUMBER;
}
[ \t]   ;       /* ignore white space */
\n  {
    printf("LEX: recognized NEWLINE\n");
    return 0;   /* logical EOF */
}
.   {
    printf("LEX: recognized SYMBOL: %c\n", yytext[0]);
    return yytext[0];
}
%%
```

Yacc Code

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
%}
```

```
%token NAME NUMBER
%%
statement:  NAME '=' expression     { printf("YACC (rule 1): reduced to s
    |   expression     {
            printf("YACC (rule 2): reduced to statement (expression)\n");
            printf("= %d\n", $1);
        }
    ;

expression: expression '+' NUMBER   {
            $$ = $1 + $3;
            printf("YACC (rule 3): reduced expression + NUMBER to %d\n",
        }
    |   expression '-' NUMBER   {
            $$ = $1 - $3;
            printf("YACC (rule 4): reduced expression - NUMBER to %d\n",
        }
    |   NUMBER          {
            $$ = $1;
            printf("YACC (rule 5): reduced NUMBER to expression: %d\n", $
        }
    ;
```

Overview

This exercise is about implementing arithmetic (plus(+), minus(-)) expression parser using Lex and Yacc. The implementation consists of two main components: a lexical analyzer (written in Lex) that recognizes numbers and arithmetic operators, and a parser (written in Yacc) that processes these tokens according to grammar rules for arithmetic expressions. To understand the parsing process, I added print messages that show both token recognition and reduction steps.

Lexical Analysis Implementation

The lexical analyzer prints debug messages whenever it recognizes a token. It handles three types of input: numbers ([0-9]+), arithmetic operators (+, -), and whitespace. Each recognition triggers a message showing exactly what was found. This way, we can see the token stream being fed to the parser.

Parser Implementation

The Yacc implementation defines grammar rules for expressions and statements, with support for addition and subtraction operations. We added debug messages to each reduction rule (1~5), allowing us to observe how the parser builds complex expressions from simpler components. The grammar ensures left-associative evaluation of expressions, meaning operations are performed from left to right.

Test Cases And Observations

Basic Operation

Input: `1-3`

This simple case demonstrates the fundamental parsing process. The lexer first identifies individual tokens (numbers and operator), then the parser reduces them following the grammar rules. The final result (-2) shows correct operator application.

Output:

```
hw7 % ./a.out
1-3
LEX: recognized NUMBER: 1
YACC (rule 5): reduced NUMBER to expression: 1
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 3
YACC (rule 4): reduced expression - NUMBER to -2
LEX: recognized NEWLINE
YACC (rule 2): reduced to statement (expression)
= -2
```

Multiple Operations

Input: `4-3-2-1`

This case reveals the left-associative nature of our parser. Operations are performed sequentially from left to right, with each intermediate result becoming the left operand for the next operation. The progression (`4→1→-1→-2`) clearly shows this behavior.

Output:

```
hw7 % ./a.out
4-3-2-1
```

```
LEX: recognized NUMBER: 4
YACC (rule 5): reduced NUMBER to expression: 4
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 3
YACC (rule 4): reduced expression - NUMBER to 1
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 2
YACC (rule 4): reduced expression - NUMBER to -1
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 1
YACC (rule 4): reduced expression - NUMBER to -2
LEX: recognized NEWLINE
YACC (rule 2): reduced to statement (expression)
= -2
```

Mixed Operations

Input: `5+3-2`

This would verify proper handling of different operators and confirm operator precedence.

Output:

```
hw7 % ./a.out
5+3-2
LEX: recognized NUMBER: 5
YACC (rule 5): reduced NUMBER to expression: 5
LEX: recognized SYMBOL: +
LEX: recognized NUMBER: 3
YACC (rule 3): reduced expression + NUMBER to 8
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 2
YACC (rule 4): reduced expression - NUMBER to 6
LEX: recognized NEWLINE
YACC (rule 2): reduced to statement (expression)
= 6
```

Possible Edge Cases

These would test boundary conditions with zero and large numbers.

Input: `0-0`

Output:

```
0-0
LEX: recognized NUMBER: 0
YACC (rule 5): reduced NUMBER to expression: 0
LEX: recognized SYMBOL: -
LEX: recognized NUMBER: 0
YACC (rule 4): reduced expression - NUMBER to 0
LEX: recognized NEWLINE
YACC (rule 2): reduced to statement (expression)
= 0
```

Input: `999999+1`

```
hw7 % ./a.out
999999+1
LEX: recognized NUMBER: 999999
YACC (rule 5): reduced NUMBER to expression: 999999
LEX: recognized SYMBOL: +
LEX: recognized NUMBER: 1
YACC (rule 3): reduced expression + NUMBER to 1000000
LEX: recognized NEWLINE
YACC (rule 2): reduced to statement (expression)
= 1000000
```

Error Cases

These invalid inputs test error handling in the parser.

Input: `+3`

Output:

```
hw7 % ./a.out
+3
LEX: recognized SYMBOL: +
Error: syntax error
```

Input: `5+`

Output:

```
hw7 % ./a.out
5+
LEX: recognized NUMBER: 5
YACC (rule 5): reduced NUMBER to expression: 5
LEX: recognized SYMBOL: +
LEX: recognized NEWLINE
Error: syntax error
```