Strathmore
UNIVERSITY

GROUP 3 - COMPILER CONSTRUCTION (CAT 2)

MINI PROJECT

Names	Admission No.
Njuguna Samuel Mbugua	134152
Abera Lelo Michael	136829
Faith Jelel Kimokiy	115064
Kendi Njeru	131590
Nathan Nderitu	137218
Gitonga Matilda Mwendwa	103720
Julie Anne Enola	134159

Below is the link to the flex and bison files on github:

https://github.com/mbugua-s/compiler-construction/tree/main/CAT%202

BASIC ARITHMETIC EXPRESSION EVALUATOR

Lexer (Lexical Analysis)

The lexer is implemented using Flex. Two regular expressions are used to obtain the numbers and the operators from the input. These are then passed into the parser.

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

This links the parser to the lexer using the header file.

```
%%

[0-9]+        { yylval = atoi(yytext); printf("Number token found: %d\n", yylval); return NUMBER; }
[-+*/()]      { printf("Operator token found: %c\n", *yytext); return *yytext; }
[ \t]         ; /* Skip whitespace */
[\n]          { return 0; }  /* Indicates end of input */
.             { return *yytext; } /* Return any other character */
```

This section contains the regular expressions. The first one obtains the numbers from the input, and the second one obtains the operators from the input.

```
%%

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

Concepts used:

- Scanning using Regular Expressions

Parser (Syntax Analysis)

The parser is implemented using Bison. Bison is a parser generator that generates an LR parser using LALR(1) techniques. A CFG is used to check for the operator, which is then used to select the appropriate operation.

Concepts used:

- Parsing using Context-free Grammar

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

This section includes the libraries and headers used for input, output, tokenization and error handling.

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

This section declares the token types and the precedence for the operators. The NUMBER token was generated by the lexer.

%%

```
input: input statement '\n'
    |
    ;

statement: expr { printf("Result: %d\n", $1); }
        ;
```

```
expr: 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; }
    | NUMBER          { $$ = $1; }
    ;
```

The grammar rules that are used for the parsing are defined here. For division, a check is used to prevent division by zero.

```
%%

int main() {
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}
```

Concepts used in this solution:

Lexer (Tokenization):

The tokenization process in the solution mirrors the role of a lexer in a compiler. The input expression is broken down into fundamental units, such as numbers, operators, and parentheses. Each of these units becomes a token.

Parser (Syntactic Analysis):

The process of converting the tokenized infix expression into postfix notation represents the syntactic analysis phase performed by a parser. This phase involves understanding the structure of the expression and converting it into a format that facilitates evaluation.

Semantic Analysis (Expression Evaluation):

The evaluation of the postfix expression can be viewed as a form of semantic analysis. The parser, having understood the syntactic structure, interprets the meaning of the expression and computes the result.