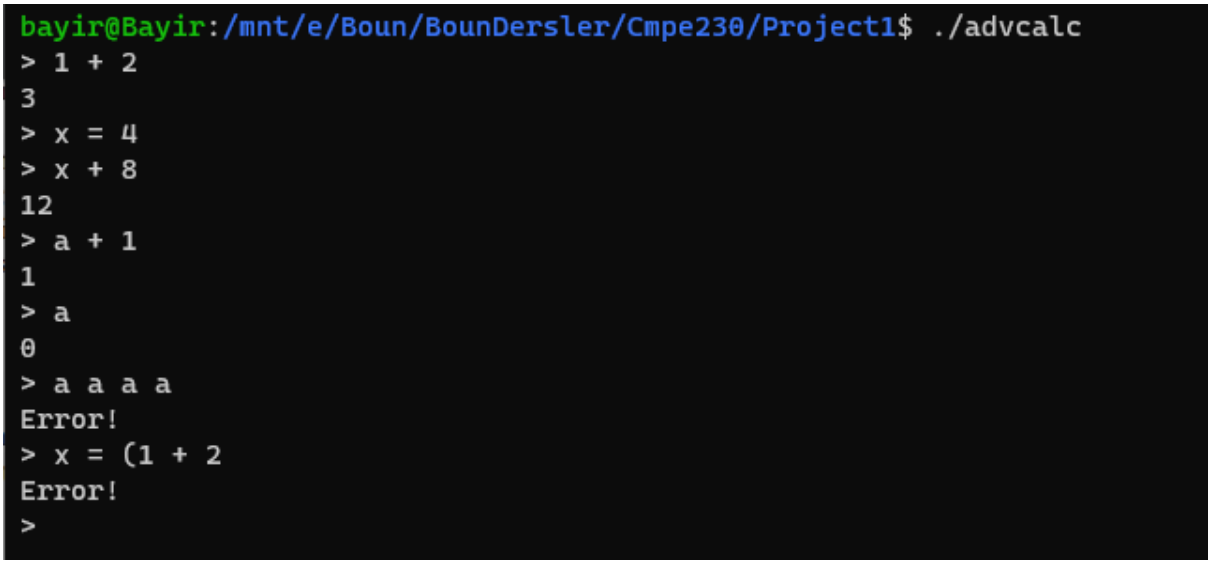
AdvCalc Documentation

Introduction

This is a command line program that evaluates arithmetic expressions. It can store a variable that can be used later.

Program Interface

Program can be started by typing ./advcalc in console. Program reads input from the command line. Users should type their expressions when they see the “> ” sign. When they type their expressions, the result will be printed in a new line. Any unassigned variable evaluates as 0. They can create a variable via typing “variable name” = “expressions”. Program will not print anything after creating the variable. The program can be terminated by typing ctrl+d. Any invalid expression will be printed as “Error!”. Simple usage shown below:



Program Structure

Program calculates the expressions via creating or editing structs. To process the expression, there are 5 main sections that handle the process. They are main, Lexer, Parser, Calculator, VariableController. It can be guessed that the project is designed as a basic compiler because the origin of processing arithmetic expression is the same as processing programming language.

Lexer

Lexer is a system that gets an input then creates meaningful tokens that are called lexemes by filtering unnecessary characters and gathering meaningful words.

Example:
input: (x + 57) / asd
outTokenArray: (“(“ , Left Parenthesis), (“x“ , Variable), (“+“ ,Plus), (“57“ , Number), (“)“ , Right Parenthesis), (“/“ ,Divide), (“asd“ ,Variable)
outSize: 7

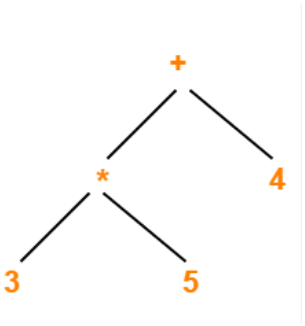
Parser

Parser is a system that reads lexemes then creates a parse tree for arithmetic expressions. To create a parse tree, the system looks at the BNF Grammar then applies the rules of the grammar. The tree will handle the precedence of the operators via looking at the BNF Grammar.

The BNF Grammar for this program:

<bitor> ::= <bitor> | <bitor> "|" <bitor>
<bitand> ::= <sum> | <bitand> "&" <sum>
<sum> ::= <fac> | <sum> "+" <fac> | <sum> "-" <fac>
<fac> ::= <par> | <fac> "*" <par> | <fac> "/" <par>
<par> ::= <func> | "(" <bitor> ")"
<func> ::= <number> | <var> | <funcname> "(" <bitor> "," <bitor> ")" | <funcname> "(" <bitor> ")"
<number> ::= <digit> | <digit> <number>
<digit> ::= [0-9]
<var> ::= <alpha> | <alpha> <var>
<alpha> ::= [a-z] | [A-Z]
<funcname> ::= "xor" | "ls" | "rs" | "lr" | "rr" | "not"

Example:
It can be visualized for this input: 3 * 5 + 4



Calculator

Calculator takes an input as root of the parse tree then recursively finds the integer result.

Variable Controller

While these 3 systems are running they will require variable value. This system will handle this request. It takes input as a variable name then makes reservations in RAM if necessary.

Main

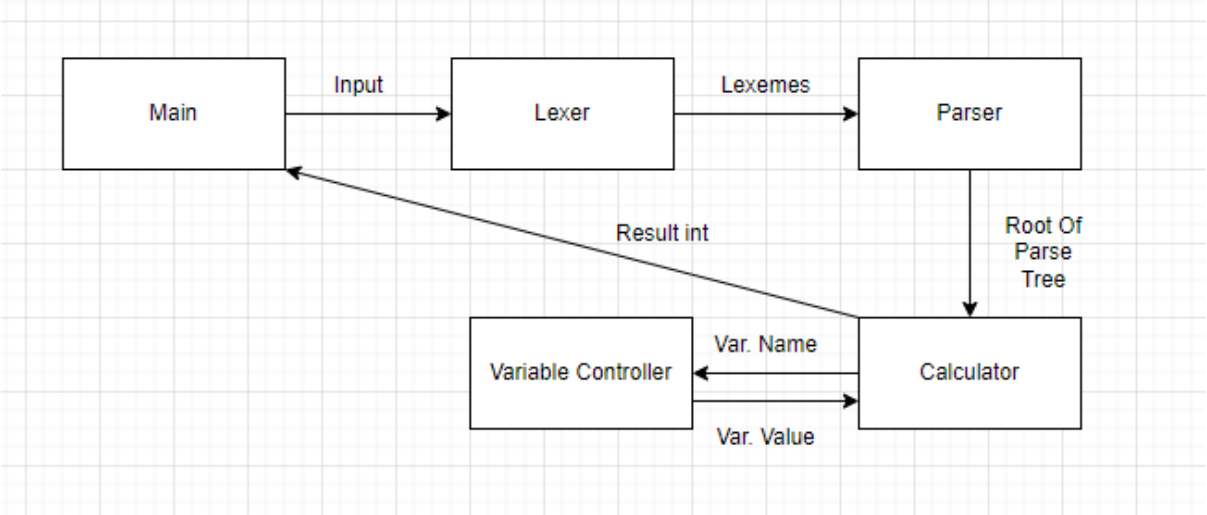
This is the command line handler part. It reads inputs from the console then sends it to systems then it will print the result. We can say that this is the client side of the code.

Structs and Enums

All structs and enums defined in Structures.h.

- **enum TokenType**: It will be used in Node struct that will give clues about what info Node carries.
- **struct Token**: It will help the creating Node struct instances.
- **struct Node**: Parse Tree element, if it is leaf node, it will carry data. Else have references of other nodes.
- **struct VariableStruct**: It will keep the variable data and name.

The high-level program structure diagram shown below:



Functions

Lexer:

Syntax:

void GetLexemes(const char line[], Token outTokenArray[], int* outSize)

Parameters:

- const char line[]: the expression
- Token outTokenArray[]: it will be filled with lexemes, it is a return variable from parameter
- int* outSize: it will be count of lexemes, it is a return variable from parameter

---

Parser:

Syntax:

```
void PrepareParser(int TokenCount, const Token* TokenPtr);
```

Description:

Before the parse processes, the system should be prepared. This function prepare the system

Parameters:

- int TokenCount: token amount
- const Token* TokenPtr: token array

Syntax:

```
void DisposeParser(struct Node* startPoint);
```

Description:

Free the parse tree

Parameters:

- struct Node* startPoint: root of the tree

Syntax:

```
bool anyErrorOccurred ();
```

Description:

Check is there any error

Return:

true if error occurred.

Syntax:

```
struct Node* ParseBitAnd();
```

Description:

Creates parse tree

Return:

Root of the tree

Calculator:

Syntax:

```
long long calculate(const struct Node* startPoint);
```

Description:

Evaluates the parse tree

Parameters:

- const struct Node* startPoint: root of the parse tree

Variable Controller:

Syntax:

```
void setVariableValue(const char* name, long long value);
```

Description:

Sets variable and save it

Parameters:

- const char* name: variable name
- long long value: variable value

Syntax:

```
long long getVariableValue(const char* name);
```

Description:

Gets variable value

Parameters:

- const char* name: variable name

Return:

Value of variable

Syntax:

```
void disposeVariables();
```

Description:

Free the saved variables
