**Introduction**

AdvCalc is an interpreter for an advanced calculator written in C programming language. The calculator accepts expressions and assignment statements.

**Purpose**

This project aims to implement a fully functional version of AdvCalc. AdvCalc must be able to evaluate the given set of operations and functions, assign return values to variables and retrieve stored values from variables. Also it must be able to detect and log errors.

**Design**

The design of the calculator consists of three parts: Analyzer, Computer, Main.

Analyzer is the part where user input is turned into a computable form. Analyzer consists of a lexer, syntax checker and reformatter. Lexer’s aim is to tokenize the user input into a list of tokens. Every token must carry its value as a string\* and its type as enumerable. Then the generated list of tokens is passed to syntax checker. As its name suggests syntax checker checks whether the given list of tokens follow the rules of AdvCalc’s syntax without modifying the given list. If syntax is violated an error will be raised. Finally, the list is passed to reformatter. Reformatter converts binary functions to binary operators and replaces variable with actual values. On any occurrence of an error, execution of analyzer is immediately stopped, and the error is passed to Main.

Computer is the part where …

Main is the conductor of the program. Taking input, initializing necessary variables, and handling errors is provided by Main. Also, Main performs as a data pipeline for Analyzer and Computer. Functions’ inputs and outputs are connected via Main.

**Implementation**

*Data types*

This project uses various data types for different tasks. An enumerated type called token\_type is created to identify different types of tokens. A full list of token types and corresponding definitions are given in the Appendix. A struct called token is defined carry lexemes and their related information. These tokens are connected to each other to form a doubly linked list. Token struct holds type of the token as token\_type, token value as string and pointer to next and previous token. Reserved keywords are held in globally initialized string array. Valid signs are held in same way but as character array. For global variables that AdvCalc stores a lookup table has been used. This lookup table consist of a string array for variable keys, long long array of actual values and integer index that points to the next empty position in lookup table. String positions in the array correspond to the actual values and access is provided by this correspondence. The index is manually kept up to date each time a new variable is defined. Lookup tables elements are global variables.

*Functions*

*Lexer:* This function converts user’s string input to a linked list of tokens. Its main principle is looping over every char by itself or its subfunctions and extracting tokens from every lexeme. Its subfunctions are a set of parsers that loop until an irrelevant character’s been encountered. Lexer detects the possible type of token from the first char after an empty space. Then it passes the current pointer to the appropriate parser. Then parser creates the token, assigns its token type and token value, and returns it. Then lexer connects the token to linked list and continues from the end of the lexeme until the new line or comment character is detected. These characters are tokenized as end of line token and denote the end of expression. Loop is terminated after EOL token. Also, if an equal sign is found, its first occurrence’s pointer is saved for later use. On other occurrences lexer returns -1. On encounter of any unknown token lexer returns -1. Errors are reported to main by returning -1.

*Syntax checker:* This function checks the correctness of syntax of given linked list. If an equal sign is found, the linked list is scanned until the sign’s pointer to check the rules for left hand side. Then the right-hand side (or full expression if no equal sign was found) is passed to expression checker function. This function iterates every token to check the whole expression. Each token is checked by a list of predefined rules. The rule list branches into two sets according to the whether inspected token is the first token of the expression. Then the function check token’s type and the next token’s type and if they follow the syntax iteration passes to next token. However, a different mechanism is used to check matching brackets and positioning of the commas. This mechanism makes two checks: Whether every parenthesis is matched and whether every function (except not) has a comma in it. The first check is done by increasing the parenthesis counter for every open parenthesis and decreasing it for closed ones. Invariant is that this counter can never be zero. Also, it should be zero at the end of the function. For the second check the same matching mechanism is also implemented to match functions with commas. However, every time a function’s been seen, current parenthesis and function counts are pushed into a stack. Invariant is checked when the stack is not empty, a closing parenthesis was seen, and parenthesis count is equal to saved count + 1 (which means the function will be closed in this loop). The invariant for this rule is that saved function count should be one bigger\* than the current function count (which means the current function is matched with a comma and the current function count returns to its old value before this function was seen). Violation of these invariants causes checker to return -1 to main.

*Reformatter:* This function replaces commas with their corresponding functions and replaces variables with their actual values. This function loops over the list and when it sees a function it removes the function from list and pushes its token value to a stack. When a comma seen its value is replaced by the top of the stack and its token type is converted to function. By this way binary functions are converted into binary operators. Also, when a variable is seen lookup table is queried and if a value is found token is converted to an integer token with that value. The query returns 0 if a variable was not found and conversion is completed using this value.

*Calculate fonksiyonlari cart curt*

*Main* burada ayrica fonksiyonlarin pass by pointer oldugundan ve pointerlarla guncel tutuldugundan bahsedebiliriz

**How to Use**

**[COMPILATION DETAILS???]. burayi istersen tamamen degistirebilirsin** After that user can start the calculator by typing “./advcalc”. User can use the given list of infix expressions and functions. To make assignments user must follow the template “<var> = <expression>”. Users should be careful about list of rules:

* Every number and intermediate result will have 64-bit limit.
* Variable names can include only lower and uppercase Latin characters in the English excluding the reserved function names.
* Any undefined variable will return 0.