Bogazici University

ADV CALC
DOCUMENTATION

Ahmet Fırat Gamsız
2020400180
Department of Computer Engineering

Ali Tarık Şahin
2020400207
Department of Computer Engineering

CmpE 230 Systems Programming
Prof Dr Can Özturan
April 1, 2023

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 all the calculations are made with respect to precedence. It takes the head of the token list and iterates through it as many as the number of precedence levels. Starting from the most significant operation or function, it does the computation and replaces the result with the components of operations. As doing so, it gets token list shrunk. At the end, the remaining number in the token list will be the result according to this algorithm. Mostly, this algorithm is a loop-based algorithm, but the only exception is parentheses cases. Considering expressions inside parentheses as a distinct problem leads the algorithm to use recursion to handle parentheses.

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 to 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 the same way but as character array. For global variables that AdvCalc stores a lookup table has been used. This lookup table consists of a string array for variable

* An array of characters is referenced as string throughout the text.

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 non-space char. 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 less than 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 the 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). Also, every time a comma is matched; current parenthesis count should be one bigger than saved parenthesis count to ensure the comma is in the function's scope. 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 is 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

* The reason of this increment is that function count is actually saved after increasing it for the current function that has been seen.

value. The query returns 0 if a variable was not found and conversion is completed using this value.

Calculator: This function handles the operational and functional calculations. It is a recursive function, and it is called only when an opening parenthesis is seen. Because its main functionality is to calculate operations in an expression and since each parenthesis represents an expression to be solved, this function can be called for each one of them. On the other hand, other operations are made through several while loops in the following format:

```
//Loop for operation

while (temp_head->token_type != CLOSE_P  &&  temp_head->token_type != EOL )
{

    if (temp_head->token_type  ==  OPERATION_NAME) {

       calculate_opr(temp_head, OPERATION_NAME); //function for binary
operator calculations

    }

    temp_head = temp_head->next;

}
```

The order of these loops matter. Operations in the earlier loops will be done earlier, which handles operation precedence. All the operations and functions except for the “not operation” are binary, hence one function called calculate_opr is used to handle them all, checking the type with switch case structure in the following form.

```
switch (type) {

    case MULTI:

        opr_result = left_value * right_value;

        break;
```

At the end of the calculate function closing parenthesis and not function is handled. Fundamentally, the algorithm behind this function is simple. The function loops through the expression for each precedence level. For instance, in the first loop it does all the multiplication operations if any, then in the second loop, it does all the summation and subtraction operations. Afterwards binary and, and binary or operations. This goes on with respect to decreasing operation precedence. Also, being recursive regarding parentheses, it gives highest precedence to parentheses and functions.

How to Use

• Type *make* in the command line and then type *./advcalc* to initialize program.

• This program supports the following operations and functions: +, -, *, &, |, xor, ls, rs, lr , rr, not.

• Variable use is also supported, once a variable is initialized, it can be used throughout the program. <var> = <expression> syntax is used to assign a value to a variable.

• Every value and every calculation should be 64-bit integer-valued.

• The language does not support the unary minus (-) operator (i.e x = -5 or a = -b is not valid). However, subtraction operation is allowed.

• The variable names should consist of lowercase and uppercase Latin characters in the English alphabet [a-zA-Z].

• Expressions or assignments should consist of 256 characters at most.

• Undefined variables will have a value of 0.

• Use ‘%’ characters for comments.

• <Ctrl-D> for Ubuntu and Linux, <Ctrl-C> for Windows ends the program.

Input Output Example

```
> a = 1 - 16
> a
-15
> rr(a,2)
9223372036854775804
> lr(a,2)
-57
> not(a + 2 * 3)
8
> (a)!
Error!
> Error!
Error!
> %Let me do it for you!
> Goodbye
0
> <Ctrl-D>
%
```

Difficulties Encountered

- Forming the full list of valid bigrams requires special attention of the designer since missing out a valid bigram causes unexpected behavior.
- Functions and parentheses have particular validity conditions that are different from the rest of the expression. Since this syntax cannot be checked by a list of valid bigrams, an algorithmic approach was followed.

- After doing each operation, the result has to take its place and the operation token, and its components have to be removed. (e.g. (3 + 4 * 5) => (3 + 20) => (23) => 23) Therefore, implementing an array-based token structure would make such replacements difficult. Hence, linked list-based token structure is a necessity.
- Functions require a special check and a particular implementation belonging to themselves due to their having somehow different structure. (e.g. xor(a, b)) However, replacing "," with function name makes each function somewhat similar to regular operations. (e.g. xor(a,b) => (axorb))
- Not function is a special one which accepts one parameter; therefore, it is difficult to fit it into another implementation of functions or operations. It needed special care and got checked for every parentheses situation.

Discussion

Why did we use a bottom-up approach?

We wanted to adopt an approach that is similar to how a human being would approach such calculations. The underlying reason is that it would be a bug free method, if any, easy to debug. On the other hand, the implementation is also easy because the algorithm needed is already the algorithm human beings use; therefore, what remains is to code how a human being solves this calculation.

Why did we use linked list implementation?

We needed some dynamic data structure to work with both when processing input and when handling calculations. Linked list structure allowed us to make flexible moves around tokens, which made our implementation easy and elegant.

What is the complexity of the program?

All the functionality of the program can be reduced to O(N) time complexity. Functional costs are just a matter of a couple of loops through the input size, which is roughly 8-9 loops. Considering O(9*N) complexity, it can be regarded as linear time complexity.

Conclusion

Using the advantages of the C programming language, we were able to produce a fast, easy to use program and a maintainable, easy to read code repository behind it.

Appendix

Token types and their definitions

`VAR` -> Variable

`INT` -> Integer

`OPEN_P` -> Open parenthesis

`CLOSE_P` -> Closed parenthesis

`SUM` -> + sign

`MULTI` -> * sign

`MINUS` -> - sign

`EQUAL` -> = sign

`B_AND` -> & sign

`B_OR` -> | sign

`B_XOR` -> xor function

`LS` -> ls function

`RS` -> rs function

`LR` -> lr function

`RR` -> rr function

`NOT` -> not function

`COMMA` -> , sign

`COMMENT` -> % sign

`EOL` -> End of line