PROJECT REPORT

CMPE230

Programming Project

Nazire Ata 2020400120 – Elif Nur Deniz 2020400045

01/04/2023

Introduction

This C program works as an interpreter for AdvCalc, a calculator that accepts expressions and assignment statements, and prints out the calculated value, or states an error if the operation is faulty.

Program Interface

Users can simply access the calculator by opening the file in terminal and running 'make' and './advcalc' commands. The 'make' command will activate Makefile. The content of Makefile is a simple compile code that produces and names executables accordingly. The second command runs the executable, then the calculator is ready to use. Users can write their operations line by line then press enter to see the result. To exit the program, users can use <CTRL+d> which indicates the end of characters that you are entering on the command line.

Program Execution

Input and Output

The program requires the user to input some mathematical expressions or assignments in text form. The provided text may contain whitespaces between characters or "%" to denote comments. If the provided input is faulty in any way, the program will output the message "Error!".

Some other criteria for the input format are explained in the project description file thoroughly.

Functionality

This program outputs the result of the mathematical operation provided by the user.

Program Structure

Custom Methods

Mathematical Operations:

plus: Takes two integers and returns an integer, the sum of the numbers.

times: Takes two integers and returns an integer, the multiplication of the numbers.

minus: Takes two integers and returns an integer, the difference of the numbers.

and: Takes two integers and returns an integer, the result of binary and operated on the numbers.

or: Takes two integers and returns an integer, the result of binary ors operated on the numbers.

xor: Takes two integers and returns an integer, the result of binary xor operated on the numbers.

ls: Takes two integers ap, ip and returns an integer, the result of left shift applied to ap ip times.

rs: Takes two integers ap, ip and returns an integer, the result of right shift applied to ap ip times.

lr: Takes two integers ap, ip and returns an integer, the result of left rotation applied to ap ip times.

rr: Takes two integers ap, ip and returns an integer, the result of right rotation applied to ap ip times.

not: Takes one integer and returns an integer, the result of binary not operated on the number.

poww: Takes one integer and returns an integer, the result of 31 to the power of parameter ‘a’.

search_char: Takes a string str and char to_find, returns an integer, the index of the first occurrence of to_find in str.

comments: Takes a string s, returns another string that starts from the beginning of s and ends at the first occurrence of %.

remove_whitespaces: Takes a string data, returns another string that starts from the first non-whitespace character of data and ends at the last non-whitespace character of data.

remove_parentheses: Takes a string data, returns another string that starts from the first non-whitespace character of data and ends at the last non-whitespace character of data.

is_variable: Takes a string data, returns false if data contains nothing but digits i.e. if the data is a number.

is_valid_variable: Takes a string data, returns if the given string consists only of letters i.e. if data is a variable and there are no operations left to do on it.

divide: This function consists of many steps.

Step 1: In the first for loop, first thing we do is to ignore everything enclosed in parentheses, any operation that contain parentheses or is enclosed by them have higher precedence than operations like +, - etc. If the parentheses are balanced until index i, we check the character there. If it is any of the “|, &, +, -, *” (we check in this order for the correct precedence) we set isTerminal to false, since data still has some operations to obtain a value. We record the first time we see any of the “|, &, +, -, *”, each at a separate int. After traversing data, an error message is displayed if data has unbalanced parentheses.

Step 2: We check if we will do any of the operations “|, &, +, -, *”, in that order, by checking if they appeared anywhere in the previous step. If one of them did, we divide the string in two as the part before and after the operation character, the current node will now have two children and an operation: left child whose value will be representing the first input of the operation, and the right child for the second input of the binary operation. Currently, we are only forming the parse tree, not calculating anything.

Step 3: If “|, &, +, -, *” do not appear in data, we check for higher precedence operations. We check the first three letters of data, as the expressions consist of two or three letters. The approach is the same for xor, ls, rs, lr, and rr. First locate the comma that separates the inputs for the operation, string one is the left side and string two is the right side of the comma. Then remove any whitespaces that might be at the beginning or end and initialize the left child of the current node with the output string for one as well as the right child with two. These nodes will be representing the first and second inputs of the operation. The operation is somewhat different for not. We only create the left child of the current node by stripping the input string for not.

Step 4: If an operation is found in data, then our data is not a terminal, and we have to divide further.

execute:

Step 1: If root does not have an operation, it is either a number or a variable, then the value of the node will be calculated and returned. If it is a number, we set the value equal to the numeric value of data. Else, we check if the variable name is valid, i.e. if it only consists of English letters. If the name is valid, we check the hash table for the previously calculated value for that variable and set the value equal to the returned integer. If the name is invalid an error message is displayed.

Step 2: If the current node has an operation, the corresponding function is called and the returned integer is set to be the value of the current node. Binary operations are called with inputs from left and right children, as the only unary function (not) is called with its left child.

Data Structures

The program uses a binary tree to parse the given string. Every node in the tree has an integer value, a string data to be parsed further, a string operation to determine what operation will be done with the left and right children, a node left, and a node right as usual.

The program also uses a hash table made up of a custom type "var". Each var has a string key to hold the variable name, an integer value, and a var pointer next to form a linked list-like structure. There are also some custom methods to complement the hash table structure:

insert:

First checks if the given key is defined in the hash table, if it is not, a new variable is created and stored at the calculated index for the given key of the hash table. If the index for the key is already in use, a while loop is used to find the key throughout the linked list. If the key is found its value is updated, if the key does not exist in the table var is added to the back of the list.

get: First use hash function to calculate the index for the given key. Then at the specified index, iterate through the linked list to find the key and return its value.

Improvements and Extension

Initially, the plan was to return integer pointers in some custom functions like "plus", "xor", etc. After some discussion it was decided that returning int would be much applicable. The project also could be improved by adding more operations like division, exponentiation, logarithm, etc. to further "advance" the calculator.

Difficulties Encountered

Writing in a new language and using a relatively new topic, pointers, made it harder to handle the project overall. However, it was a good practice to complete this project to get familiar with pointers and the language C.

Conclusion

In conclusion, this project helped us comprehend how interpreters work by parsing and tokenizing.