**FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)**

****

## Compiler Construction - COMP 451 B

**2023SP**

**Project**

**Junaid Imran – 241567697**

**Hassan Ali - 241564087**

## Logic Of Project Task 1

This is a C program that processes an arithmetic expression provided as a command line argument. Let's break down the logic of this code:

### 1. Preprocessor Directives:

- The code starts with preprocessor directives. `#include <stdio.h>` includes the standard input/output library for using functions like `printf`.

- Two macros are defined using `#define`. `SENTINEL` is defined as '$', and `MAX\_OPERANDS` is defined as 4.

### 2. Function Definitions:

- There are two helper functions defined in this code:

- `Operator(char token)`: This function checks if a given token is an arithmetic operator (+, -, \*, /). It returns 1 if true, and 0 otherwise.

- `operand(char token)`: This function checks if a given token is a valid operand (lowercase alphabets). It returns 1 if true, and 0 otherwise.

### 3. `main` Function:

- The `main` function is the entry point of the program.

- It first checks the number of command line arguments. If there are not exactly 2 arguments, it prints an error message and returns with an error code.

- The expression is obtained from the second command line argument (`argv[1]`).

- The expression is printed to the console.

- A loop is used to process each character in the expression until the sentinel symbol ('$') is encountered.

- Inside the loop:

- The current character is stored in the variable `token`.

- It checks if `token` is an arithmetic operator using the `Operator` function. If true, it prints the operator.

- If `token` is a valid operand using the `operand` function, it checks if the operand count exceeds `MAX\_OPERANDS`. If so, it prints an error message and returns with an error code. Otherwise, it prints the operand and increments the operand count.

- If `token` is neither an operator nor a valid operand, it prints an error message and returns with an error code.

### 4. Program Termination:

- If the program reaches the end of the loop without encountering any errors, it returns with a success code (0).

Overall, this program parses an arithmetic expression and prints the operators and operands encountered. It also performs basic validation on the number of operands and checks for invalid tokens in the expression.

## Logic Of Project Task 2

In task 2, a C program was designed to imitate an operator precedence parser, which prints out the stack implementation table onto the console. It evaluates an arithmetic expression and displays the calculated value onto the console. The logic of this task is explained below:

1. The code begins with some preprocessor directives, including header files and defining constants for maximum values and stack sizes.

2. The precedence table (`prec\_table`) is defined to determine the precedence of different tokens in the arithmetic expression. It is a 2-dimensional array where each cell represents the precedence relation between two tokens. There are a total of 6 tokens, 1 identifier, 4 arithmetic operators, and the sentinel value ‘$’.

3. The `operand` function checks if a character is an operand (a lowercase letter).

4. Two stack structures (`IntStack` and `CharStack`) are defined using arrays. These stacks will be used to store integers and characters, respectively.

5. Several stack-related functions are defined, such as initializing a stack, checking if a stack is empty or full, pushing elements onto a stack, peeking at the top element, popping an element from a stack, and getting the size of a stack.

6. The `getPrecedence` function determines the precedence relation between two tokens based on the `prec\_table`. It takes two characters as input and returns an integer value representing the precedence relation (1 for higher precedence, 0 for equal precedence, -1 for lower precedence).

7. The `calculate` function performs the calculation for a given operator and two operands. It pops the operands from the integer stack, performs the operation, and pushes the result back onto the stack.

8. The `operatorPrecedenceParser` function implements the operator precedence parsing algorithm. It takes the arithmetic expression, an array of values for the operands, the array of operand characters, and the number of values as input.

9. Inside the `operatorPrecedenceParser` function, two stacks (`stack1` and `stack2`) are initialized. `stack1` is the output stack, and `stack2` is used for calculations.

10. The sentinel character (`$`) is pushed onto `stack1`.

11. The code enters a loop that continues until the current character is not the sentinel character (`$`) and the top of `stack1` is not the sentinel character.

12. In each iteration, the code checks the precedence relation between the top of `stack1` and the current character.

13. If the precedence relation is 1, it means that the top of `stack1` has higher precedence. In this case, the top of `stack1` is popped, and if it is an operand, its corresponding value is retrieved from the `values` array and pushed onto `stack2`. If it is an operator and there are at least two operands on `stack2`, the `calculate` function is called to perform the calculation.

14. If the precedence relation is 0, it means that the current character has higher precedence. In this case, the current character is pushed onto `stack1`, and the loop moves to the next character.

15. If the precedence relation is -1, it means that the expression is invalid, and an error is printed, and the program exits.

16. After the loop, the final state of `stack1` is printed, and it is checked if the input string has been exhausted and the sentinel character has been reached. If so, the expression is accepted; otherwise, it is rejected.

17. Finally, the top element of `stack2` is printed, which represents the final result of the arithmetic expression.

18. In the `main` function, command-line arguments are checked, and if an argument is provided, the arithmetic expression is stored in the `arithmetic\_expr` variable.

19. The code then prompts for values of the operands and stores them in the `values` array, along with the corresponding operand characters in the `operands` array.

20. The `operatorPrecedenceParser` function is called with the arithmetic expression, operand values, operand characters, and the number of values.