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

****

**Compiler Construction || COMP-451**

**Section B**

**Project**

**Expression Parser**

**Group Members:**

**Abdullah Mehtab || 241607845**

**Umm-e-Abiha || 231522019**

**Submitted to:**

**Sir Rauf Butt**

**A white background with black text

Description automatically generated**

**A screenshot of a task

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

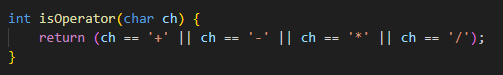
**Task-1**

The task at hand is to write a C program that performs lexical analysis on an arithmetic expression provided as a command-line argument. The expression can include arithmetic operators (**+**, **-**, **\***, **/**), identifiers (single lowercase letters), and must end with a sentinel value (**$**). The program should identify and categorize these tokens, handle invalid tokens, and output the results accordingly.

**Code Breakdown and Detailed Explanation**

**Helper Functions**

**isOperator**

****

* **Function Signature:** **int isOperator(char ch)**
* **Parameters:** **ch** - the character to be checked.
* **Returns:** **1** if **ch** is an operator, **0** otherwise.
* **Description:** This function checks if the character **ch** is one of the arithmetic operators.

**Data Structures**

A screen shot of a computer code

Description automatically generated**Token**

* **TokenType Enum:** Defines the possible types of tokens.
* **Token Struct:** Represents a token with a **lexeme** and a **type**.
* **tokens Array:** Stores up to 32 tokens.
* **tokenCount:** Tracks the number of tokens.

**Main Function**

**Explanation:**

1. **Function Signature:**
   * **argc**: Stands for "argument count". It represents the number of command-line arguments passed to the program, including the program's name.
   * **argv[]**: Stands for "argument vector". It is an array of strings representing the command-line arguments. **argv[0]** is the program's name, and **argv[1]** onwards are the additional arguments.
2. **Argument Check:**

A black background with white text

Description automatically generated

* + This condition checks if fewer than 2 arguments are provided. If true, it prints an error message and exits the program with a status code of 1, indicating an error.

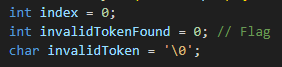
1. A black background with white text

   Description automatically generated**Copying Input:**
   * **input** is a character array to hold the input expression. **strcpy** copies the input string from **argv[1]** to **input**.
2. **Printing Initial Message:**
   * Prints the input expression received.
3. **Calling Lexical Analysis:**
   * Calls the **lexicalAnalysis** function to analyze the input expression.

**Lexical Analysis Function**

**Explanation:**

1. **Function Signature:**
   * **input**: A character pointer to the input expression string.
2. **Variable Declarations:**



* + **index**: Keeps track of the current position in the token's lexeme.
  + **invalidTokenFound**: A flag to indicate if an invalid token is found.
  + **invalidToken**: Stores the invalid token character.

1. A black background with white text

   Description automatically generated**Main Loop:**
   * Iterates through each character in the input string until the null terminator is reached.
2. **Identifier Handling:**

A screen shot of a computer code

Description automatically generated

* + Checks if the current character is an alphabet or underscore (valid identifier characters).
  + Enters a loop to gather the full identifier by checking if the characters are alphanumeric or underscores.
  + Stores the identifier in the **tokens** array and updates the **tokenCount**.

1. **Operator Handling:**

A screen shot of a computer code

Description automatically generated

* + Checks if the current character is an operator using the **isOperator** function.
  + Stores the operator in the **tokens** array and updates the **tokenCount**.

1. **Sentinel Handling:**

A screen shot of a computer code

Description automatically generated

* + Checks if the current character is the sentinel value (**$**).
  + Stores the sentinel in the **tokens** array and updates the **tokenCount**.

1. **Invalid Token Handling:**

A computer screen with white text

Description automatically generated

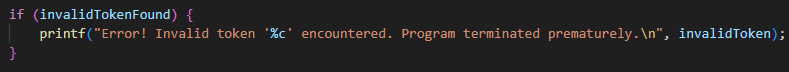
* + If the current character is not a valid identifier, operator, or sentinel, it sets the **invalidTokenFound** flag and stores the invalid token character.

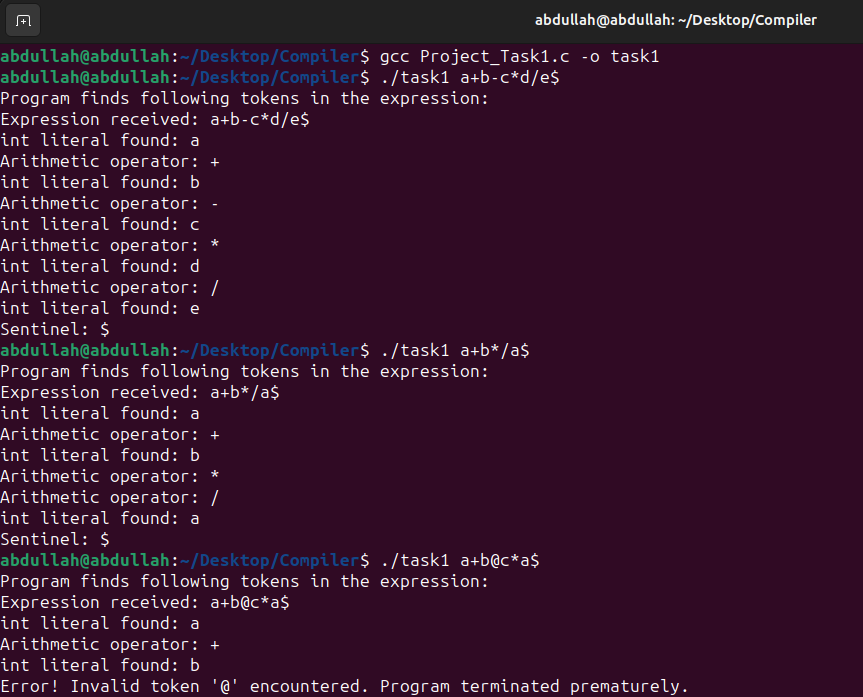
1. **Token Output Loop:**

A computer screen shot of code

Description automatically generated

* + Iterates through the **tokens** array and prints the type and lexeme of each token.

1. **Invalid Token Message:**
   * If an invalid token was found, it prints an error message and terminates the program.

**Task 1 Output**

**Task-2**

**Introduction**

In this task, we aim to build upon the lexical analysis program developed in Task-1. Now, we will extend the functionality to include parsing based on a context-free grammar (CFG) using an operator precedence parser. We'll implement a parsing algorithm that recognizes valid expressions based on the provided CFG and outputs the result of the expression while displaying the stack implementation table.

**Code Overview**

We have a program written in C that performs lexical analysis on an arithmetic expression provided as a command-line argument. The program identifies and categorizes tokens such as identifiers, operators, and a sentinel value. Now, we will enhance this program to include parsing based on a CFG and implement an operator precedence parser to recognize valid expressions.

For the main function, everything is exactly similar to Task-1, but only one new line was added right after the lexical analysis. The syntax analyser.



Which is the call to the syntax analyser, which will take in the tokens and the token count derived from the lexical analysis, and parse through the input using the given CFG, and make a parsing table.

**Removing Left Recursion from given CFG**

*E*→*E*+*E* ∣ *E*−*E* ∣ *E*∗*E* ∣ *E*/*E* ∣ id

As given, this grammar is ambiguous, and has left recursion, hence, to remove the left recursion from the given grammar, we use the classic formula, let's go through the steps systematically.

We apply the transformation rule:   
If 𝐴→𝐴𝑎 ∣ *b*, then it becomes:   
𝐴→𝑏𝐴′  
𝐴′→𝑎𝐴′ ∣ 𝜖

For the given grammar, we can see that the productions involve operations (+, -, \*, /) and the identifier idid. We'll convert this into a form without left recursion.

1. **Identify and Separate Recursive and Non-Recursive Productions**:
   * Recursive: 𝐸→𝐸+𝐸 ∣ 𝐸−𝐸 ∣ 𝐸∗𝐸 ∣ 𝐸/𝐸
   * Non-Recursive: 𝐸→id

**Step-by-Step Conversion**

1. **Start with *E*→id*E*′**:
   * 𝐸*E* can derive an identifier followed by *E*′, which will handle the operations.
2. **Define 𝐸′*E*′ for Handling Operations**:
   * 𝐸′→+𝐸𝐸′ ∣ −𝐸𝐸′ ∣ ∗𝐸𝐸′ ∣ /𝐸𝐸′ ∣ 𝜖
   * 𝐸′ represents the part of the expression that comes after the first identifier and can include operations followed by further expressions or terminate with 𝜖*ϵ*.
3. **Recombine to form the Full Grammar**:
   * The non-recursive part is directly included in 𝐸*.*
   * The recursive part is managed by 𝐸′.

**Final Grammar without Left Recursion:**

𝐸→id𝐸′𝐸′→+id𝐸′ ∣ −id𝐸′ ∣ ∗id𝐸′ ∣ /id𝐸′ ∣ 𝜖

This grammar eliminates the left recursion by ensuring that 𝐸produces an identifier first, and then *E*′ handles any following operations, potentially recursively, but without direct left recursion.

The grammar is still ambiguous though.

**Removing ambiguity from the CFG**

The given grammar is ambiguous because it does not enforce the correct precedence and associativity of the operators. To use this grammar in an operator precedence parser, we need to ensure that the grammar reflects the precedence and associativity rules correctly.

To rewrite the grammar to make it unambiguous by considering operator precedence and associativity. The precedence and associativity are as follows:

* Multiplication (∗) and division (/) have higher precedence than addition (+) and subtraction (−).
* Addition (+) and subtraction (−) have the same precedence and are left-associative.
* Multiplication (∗) and division (/) have the same precedence and are left-associative.

**Step-by-Step Construction of Unambiguous Grammar**

1. **Separate Different Levels of Precedence**:
   * We need different non-terminals for expressions involving different precedence levels.
2. **Define the Grammar for Each Precedence Level**:
   * Let's define three levels:
     + *E* for addition and subtraction (lowest precedence)
     + *T* for multiplication and division (higher precedence)
     + *F* for factors (identifiers or nested expressions)

**Construct the Grammar**

**Highest Precedence (Factors)**

* 𝐹→id

**Multiplication and Division (Higher Precedence)**

* 𝑇→𝐹𝑇′
* 𝑇′→∗𝐹𝑇′ ∣ /𝐹𝑇′ ∣ 𝜖

**Addition and Subtraction (Lowest Precedence)**

* 𝐸→𝑇𝐸′
* 𝐸′→+𝑇𝐸′ ∣ −𝑇𝐸′ ∣ 𝜖*E*′→+*TE*′ ∣ −*TE*′ ∣ *ϵ*

**Final Unambiguous Grammar**

* 𝐸→𝑇𝐸′
* 𝐸′→+𝑇𝐸′ ∣ −𝑇𝐸′ ∣ 𝜖
* E′→+TE′ ∣ −TE′ ∣ ϵ
* 𝑇→𝐹𝑇′
* T→FT′ 𝑇′→∗𝐹𝑇′ ∣ /𝐹𝑇′ ∣ 𝜖
* F→id

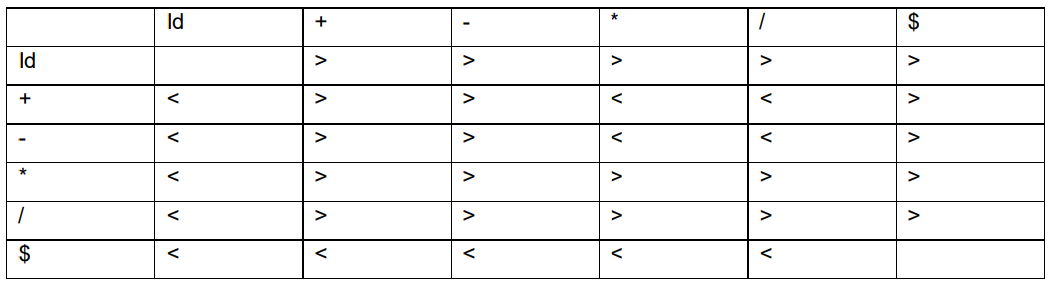
**Explanation**

* The non-terminal *E* handles addition and subtraction.
* The non-terminal *T* handles multiplication and division.
* The non-terminal *F* handles factors, which are identifiers.

This grammar ensures that:

* Multiplication and division are performed before addition and subtraction due to the structure of the grammar.
* Left-associativity is maintained by recursive structures of 𝐸′ and 𝑇′*.*

**Operator Precedence Relation Table**

We can use the operator precedence relation table given to implement the parser. This unambiguous grammar works with the given precedence relations and is suitable for an operator precedence parser.

**Variable Declarations and Structures:**

* **TokenType:** An enumeration defining different types of tokens in the input expression, such as identifiers, operators, numeric literals, sentinel, and invalid tokens.
* **Token:** A structure representing a token, consisting of a lexeme (string representation) and its type (from **TokenType**).
* **tokens:** An array of **Token** structures to store the identified tokens from the input expression.
* **tokenCount:** An integer variable to keep track of the number of tokens identified in the input expression.
* **currentTokenIndex:** An integer variable to keep track of the current token being processed while parsing.
* **stack:** A 2D character array to implement a stack data structure for storing lexemes during parsing.
* **stac\_chr:** A character array to store the characters being pushed to the stack (for displaying stack implementation table).
* **top:** An integer variable indicating the top index of the stack, used for stack operations.
* **inputStr:** A character array to store the input expression provided via the command-line argument.
* **identifierValues:** An array to store the values of identifiers identified in the expression.

**Helper Functions**

A computer code on a black background

Description automatically generated**1. push()**

**Explanation:**

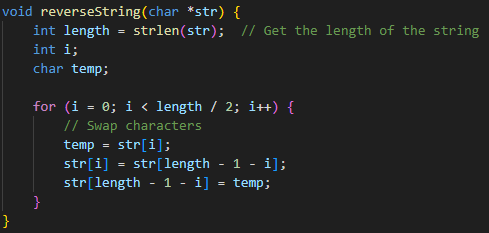
* **Purpose:** The **push** function adds a new value onto the stack.
* **Parameters:** **value** - A string representing the value to be pushed onto the stack.
* **Variables:**
  + **top** - An integer tracking the top index of the stack. Initially set to -1.
  + **stack** - A 2D array of characters representing the stack, where each element is a string.
* **Logic:**
  + **Condition:** **if (top < 128 - 1)** checks if there is space in the stack (maximum size 128).
    - **True:** The value is copied into the stack at the position **++top** (increment **top** first, then use it as the index).
    - **False:** An error message "Error: Stack overflow." is printed, and the program exits with **exit(1)**.

**2. pop()**

A computer screen shot of a program code

Description automatically generated

**Explanation:**

* **Purpose:** The **pop** function removes and returns the top value from the stack.
* **Returns:** A pointer to the string that was at the top of the stack.
* **Logic:**
  + **Condition:** **if (top >= 0)** checks if there are elements in the stack.
    - **True:** Returns the top value from the stack and decrements **top**.
    - **False:** An error message "Error: Stack underflow." is printed, and the program exits with **exit(1)**.

**3. reverseString()**

**Explanation:**

* **Purpose:** The **reverseString** function reverses the given string in place.
* **Parameters:** **str** - A pointer to the string to be reversed.
* **Variables:**
  + **length** - An integer storing the length of the string.
  + **i** - Loop counter.
  + **temp** - Temporary variable to facilitate swapping.
* **Logic:**
  + **Loop:** **for (i = 0; i < length / 2; i++)** iterates over the first half of the string.
    - **Inside Loop:** Characters at positions **i** and **length - 1 - i** are swapped using **temp**.

**4. syntaxError()**

**A computer screen with text

Description automatically generated**

**Explanation:**

* **Purpose:** The **syntaxError** function displays an error message and exits the program.
* **Parameters:** **msg** - A string containing the error message.
* **Logic:**
  + **Print Statements:** Prints "Syntax Error: " followed by the provided message.
  + **Exit:** Exits the program with **exit(1)**.

A computer code with many colorful text

Description automatically generated with medium confidence**5. printStack()**

**Explanation:**

* **Purpose:** The **printStack** function prints the current contents of the stack.
* **Logic:**
  + **Print Statement:** Begins with **\t\t|**.
  + **Loop:** **for (int i = 0; i <= top; ++i)** iterates through the stack elements.
    - **Inside Loop:** Prints each element of the stack.
  + **New Line:** Ends the print statement with a newline character.

**6. isValidIdentifier()**

A computer code on a black background

Description automatically generated

**Explanation:**

* **Purpose:** The **isValidIdentifier** function checks if the given string is a valid identifier.
* **Parameters:** **id** - A string representing the identifier to be checked.
* **Returns:** 1 (true) if the identifier is valid, 0 (false) otherwise.
* **Logic:**
  + **First Character Check:** **if (!isalpha(id[0]) && id[0] != '\_')** checks if the first character is a letter or underscore.
    - **False:** Returns 0 (invalid identifier).
  + **Loop:** **for (int i = 1; i < len; ++i)** iterates through the remaining characters.
    - **Inside Loop:** Checks if each character is alphanumeric or an underscore. If not, returns 0 (invalid identifier).
  + **Return 1:** If all checks pass, returns 1 (valid identifier).

**Main functions**

**‘match’ function**

The **match** function is used to verify that the current token in the token stream matches the expected token type. If the token matches, the function proceeds to the next token; if it doesn't, a syntax error is triggered.

**Parameters**

* **expectedType**: The token type that is expected at the current position in the token stream. It is of the enum type **TokenType**.

**Variables**

* **currentTokenIndex**: An integer representing the current position in the token stream.
* **tokens**: An array of token structures that holds the entire token stream.
* **inputStr**: The input string being parsed.
* **stac\_chr**: A character array used to store a string representation of the stack.

**Detailed Breakdown**

**Condition: Token Type Matching**



* **Explanation**: This condition checks if the type of the current token matches the expected type.
  + **True**: The current token is of the expected type, and the function will proceed to process the token.
  + **False**: The current token does not match the expected type, triggering a syntax error.

**Inner Condition: Token is an Operator**

****

* **Explanation**: This inner condition checks if the current token is an operator.
  + **True**: Additional processing is required for operator tokens.
  + **False**: The function simply advances to the next token.

**Processing Operator Tokens**

****

* **Explanation**: This line concatenates the current operator character to **stac\_chr**.
  + **strncat**: A C standard library function that appends a specified number of characters from one string to another. Here, it appends one character from the position **inputStr + currentTokenIndex** (which is basically the remaining input from the stack implementation table) to **stac\_chr (**the stack).



* **Explanation**: This function call reverses the string stored in **stac\_chr**.
  + **reverseString**: A user-defined function that reverses the characters in the given string.



* **Explanation**: This line prints the current state of the stack, the remaining input string, and the action taken ("Push").



* **Explanation**: Calls the **printStack** function to display the current stack contents.
  + **printStack**: A user-defined function that prints all elements currently in the stack.

A black screen with white text

Description automatically generated

* **Explanation**: Advances to the next token by incrementing **currentTokenIndex**.
* **Otherwise Else Clause for Non-Operator Tokens**
* **Explanation**: For tokens that are not operators, simply advance to the next token without additional processing.

**Syntax Error Handling**

A black background with white text

Description automatically generated

* **Explanation**: If the current token does not match the expected type, trigger a syntax error.
  + **syntaxError**: A user-defined function that prints an error message and exits the program.
  + **Parameters**: **"Token mismatch"** - The error message indicating that the token type did not match the expected type.

**‘factor’ function**

The **factor** function is responsible for handling factors in arithmetic expressions. A factor can be an identifier, a sentinel, or an expression enclosed in parentheses. This function ensures that the factor is correctly processed and pushed onto the stack or appropriately handled if it is a sentinel.

**Variables**

* **currentTokenIndex**: An integer representing the current position in the token stream.
* **tokens**: An array of token structures that holds the entire token stream.
* **inputStr**: The input string being parsed.
* **identifierValues**: An array holding the values of identifiers.
* **stac\_chr**: A character array used to store a string representation of the stack.

**Detailed Breakdown**

**Condition: Token is an Identifier**



* **Explanation**: This condition checks if the current token is an identifier.
  + **True**: The function processes the identifier.
  + **False**: The function checks the next condition.

**A screen shot of a computer code

Description automatically generatedProcessing Identifier Tokens**

**Explanation**: This block retrieves the value associated with the identifier and pushes it onto the stack.

* + **value**: Stores the value of the current identifier from the **identifierValues** array.
  + **valueStr**: A character array to hold the string representation of the identifier value.
  + **sprintf**: Converts the integer value to a string and stores it in **valueStr**.
  + **push**: A function that pushes the string onto the stack.

A screen shot of a computer code

Description automatically generated

* **Explanation**: This block appends the current character to **stac\_chr** and reverses the string if the character is not a dollar sign.
  + **Condition**: Checks if the current character is not a dollar sign.
  + **strncat**: Appends the current character to **stac\_chr**.
  + **reverseString**: Reverses the string stored in **stac\_chr**.



* **Explanation**: Prints the current state of the stack, the remaining input string, and the action taken ("Push").
  + **printStack**: A user-defined function that prints all elements currently in the stack.

**Deleting One Character when an Identifier is Pushed**

A computer screen shot of text

Description automatically generated

* **Explanation**: This block shifts characters to the left in **stac\_chr** to delete one character.
  + **size**: Stores the length of **stac\_chr**.
  + **Loop**: Iterates over each character in **stac\_chr** and shifts it to the left.



* **Explanation**: Prints the current state of the stack, the remaining input string, and the action taken ("Pop").
  + **printStack**: A user-defined function that prints all elements currently in the stack.



* **Explanation**: Calls the **match** function to ensure the current token matches the expected **IDENTIFIER** type and advances to the next token.
  + **match**: A user-defined function that verifies token types and handles errors.

**Condition: Token is a Sentinel**



* **Explanation**: This condition checks if the current token is a sentinel.
  + **True**: The function calls **match** to handle the sentinel token.
  + **False**: The function triggers a syntax error.

**Syntax Error Handling**

A black background with white text

Description automatically generated

* **Explanation**: If the current token is neither an identifier nor a sentinel, trigger a syntax error.
  + **syntaxError**: A user-defined function that prints an error message and exits the program.
  + **Parameters**: **"Invalid factor"** - The error message indicating that the factor is invalid.

**‘MUL\_or\_DIV’ function**

The **MUL\_or\_DIV** function processes multiplication and division operations in an arithmetic expression. It ensures that factors involved in these operations are correctly evaluated, and the results are pushed back onto the stack.

**Variables**

* **currentTokenIndex**: An integer representing the current position in the token stream.
* **tokenCount**: The total number of tokens.
* **tokens**: An array of token structures that holds the entire token stream.
* **inputStr**: The input string being parsed.
* **stac\_chr**: A character array used to store a string representation of the stack.

**Detailed Breakdown**

**Initial Call to factor**



* **Explanation**: The function begins by calling **factor()** to process the first factor in the multiplication or division operation.
  + **factor**: A user-defined function that processes factors in an arithmetic expression.

**While Loop: Processing Operators**

A screenshot of a computer code

Description automatically generated

* **Explanation**: This while loop continues as long as there are more tokens to process and the current token is either a multiplication (**\***) or division (**/**) operator.
  + **Condition**: Checks if the current token is an operator and if it is either **\*** or **/**.

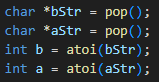
**Extracting and Matching the Operator, then processing the next factor**

A black screen with white text

Description automatically generated

* Stores the operator (**\*** or **/**) in the variable **op**.
  + **op**: A character variable that holds the current operator.
* Calls the **match** function to ensure the current token matches the expected **OPERATOR** type and advances to the next token.
  + **match**: A user-defined function that verifies token types and handles errors.

**Popping Operands from the Stack**



**Explanation**: Pops two operands from the stack and converts them from strings to integers.

* + **bStr and aStr**: Strings representing the operands.
  + **pop**: A user-defined function that removes and returns the top element from the stack.
  + **b and a**: Integers obtained by converting **bStr** and **aStr** using **atoi**.

**Performing the Operation**

**A computer code on a black background

Description automatically generated**

* **Explanation**: Performs the multiplication or division operation based on the operator.
  + **Multiplication**: If **op** is **\***, multiply **a** and **b**.
  + **Division**: If **op** is **/**, check for division by zero and perform the division.
  + **Error Handling**: Prints an error message and exits the program if division by zero is attempted.

**Pushing the Result onto the Stack**

A black background with white text

Description automatically generated

* **Explanation**: Converts the result to a string and pushes it onto the stack.
  + **resultStr**: A character array to hold the string representation of the result.
  + **sprintf**: Converts the integer result to a string and stores it in **resultStr**.
  + **push**: A user-defined function that pushes the string onto the stack.

**Updating and Printing the Stack**

A computer screen with text

Description automatically generated

* **Explanation**: Updates and prints the current state of the stack.
  + **size**: Stores the length of **stac\_chr**.
  + **Loop**: Iterates over each character in **stac\_chr** and shifts it to the left.
  + **printStack**: A user-defined function that prints all elements currently in the stack.

**‘ADD\_or\_SUB’ function**

The **ADD\_or\_SUB** function processes addition and subtraction operations in an arithmetic expression. It ensures that terms involved in these operations are correctly evaluated, and the results are pushed back onto the stack.

**Variables**

* **currentTokenIndex**: An integer representing the current position in the token stream.
* **tokenCount**: The total number of tokens.
* **tokens**: An array of token structures that holds the entire token stream.
* **inputStr**: The input string being parsed.
* **stac\_chr**: A character array used to store a string representation of the stack.

**Detailed Breakdown**

**Initial Call to MUL\_or\_DIV**



**Explanation**: The function begins by calling **MUL\_or\_DIV()** to process the first term in the addition or subtraction operation.

* + **MUL\_or\_DIV**: A user-defined function that processes multiplication and division operations within the expression.

**While Loop: Processing Operators**

A screenshot of a computer code

Description automatically generated

* **Explanation**: This while loop continues as long as there are more tokens to process and the current token is either an addition (**+**) or subtraction (**-**) operator.
  + **Condition**: Checks if the current token is an operator and if it is either **+** or **-**.

**Extracting and matching the Operator**

* Stores the operator (**+** or **-**) in the variable **op**.
  + **op**: A character variable that holds the current operator.
* **Explanation**: Calls the **match** function to ensure the current token matches the expected **OPERATOR** type and advances to the next token.
  + **match**: A user-defined function that verifies token types and handles errors.

**Processing the Next Term**

****

* **Explanation**: Processes the next term in the expression.
  + **MUL\_or\_DIV**: A user-defined function that processes multiplication and division operations within the expression.

**Popping Operands from the Stack**

A black background with white text

Description automatically generated

* **Explanation**: Pops two operands from the stack and converts them from strings to integers.
  + **bStr and aStr**: Strings representing the operands.
  + **pop**: A user-defined function that removes and returns the top element from the stack.
  + **b and a**: Integers obtained by converting **bStr** and **aStr** using **atoi**.

**A computer screen shot of a math equation

Description automatically generatedPerforming the Operation**

**Explanation**: Performs the addition or subtraction operation based on the operator.

* + **Addition**: If **op** is **+**, adds **a** and **b**.
  + **Subtraction**: If **op** is **-**, subtracts **b** from **a**.

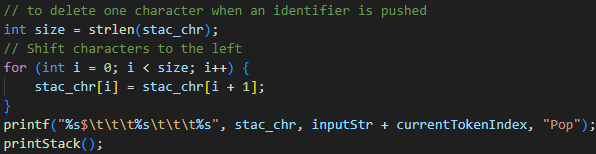
**Pushing the Result onto the Stack**

A black background with white text

Description automatically generated

* **Explanation**: Converts the result to a string and pushes it onto the stack.
  + **resultStr**: A character array to hold the string representation of the result.
  + **sprintf**: Converts the integer result to a string and stores it in **resultStr**.
  + **push**: A user-defined function that pushes the string onto the stack.

**Updating and Printing the Stack**

****

* **Explanation**: Updates and prints the current state of the stack.
  + **size**: Stores the length of **stac\_chr**.
  + **Loop**: Iterates over each character in **stac\_chr** and shifts it to the left.
  + **printf**: A standard C library function for formatted output.
  + **printStack**: A user-defined function that prints all elements currently in the stack.

**Final Check for SENTINEL Token**

A black screen with white text

Description automatically generated

* **Explanation**: After processing all addition and subtraction operations, the function checks if the next token is a SENTINEL to mark the end of the expression.
  + **Condition**: Checks if the current token is a SENTINEL.
  + **match(SENTINEL)**: Ensures that the SENTINEL token is present and matches the expected type.
  + **Error Handling**: If the SENTINEL is not found, a syntax error is reported, and the function exits.

**‘syntaxAnalysis’ function**

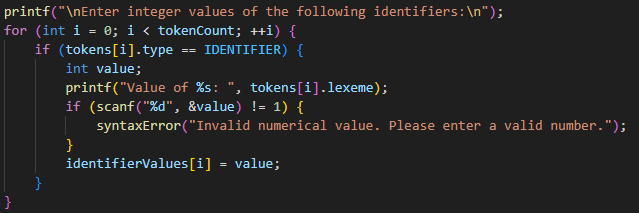
The **syntaxAnalysis** function is responsible for the overall control flow of the syntax analysis phase in parsing an arithmetic expression. It interacts with the user to retrieve values for identifiers, manages the stack operations, and ensures the correctness of the parsed expression.

**Variables**

* **tokens**: An array of token structures representing the token stream.
* **tokenCount**: The total number of tokens.
* **identifierValues**: An array storing the integer values of identifiers.
* **inputStr**: The input string being parsed.
* **top**: An integer representing the top of the stack.
* **stack**: An array representing the evaluation stack.

**Detailed Breakdown**

**User Input for Identifiers**



* **Explanation**: This section prompts the user to input integer values for each identifier found in the token stream.
  + **printf**: Outputs a prompt message.
  + **Loop (for)**: Iterates over each token in the token stream.
  + **Condition (if)**: Checks if the current token is an identifier.
    - **tokens[i].type == IDENTIFIER**: Evaluates whether the token type is an identifier.
  + **User Input**: Prompts the user to enter a value for the identifier.
    - **scanf**: Reads an integer value from the user.
    - **Error Handling**: If the input is not a valid integer, **syntaxError** is called.
      * **syntaxError**: A user-defined function that handles syntax errors by displaying an error message and terminating the program.

**Display Initial Stack and Input**

* **Explanation**: Prints the initial state of the stack and input string.
  + **printf**: Outputs the headers and initial state of the stack and input string.
  + **$**: Represents the initial stack containing only the end marker.
  + **inputStr**: The input string being parsed.

**Initialize Parsing the Expression**

* **Explanation**: Calls the **ADD\_or\_SUB** function to parse the expression.
  + **ADD\_or\_SUB**: A user-defined function that handles addition and subtraction operations in the expression.

**Final Check and Output**

**A screen shot of a computer code

Description automatically generated**

* **Explanation**: Checks if the parsing was successful and prints the result.
  + **Condition (if)**: Checks if the stack is empty, indicating successful parsing.
    - **top == 0**: Evaluates whether the stack is empty.
    - **Success**: If true, prints the accepted message and the result.
      * **stack[top]**: Contains the final result of the parsed expression.
    - **Failure**: If false, prints an error message and exits.
      * **Error Handling**: Uses **fprintf** to print an error to **stderr** and terminates the program with **exit(1)**.

**Error Handling**

* **Division by Zero**: Managed by checking if the divisor is zero before performing division. An error message is printed, and the program terminates if division by zero is detected.
* **Stack Overflow and Underflow**: Managed by the **push** and **pop** functions respectively, with appropriate error messages and program termination.
* **Invalid Factor**: Triggered when the token is neither an identifier nor a sentinel, ensuring that only valid factors are processed.
* **Token Mismatch**: Handled by the **syntaxError** function, which provides a clear error message and exits the program to prevent further incorrect processing.
* **Invalid Numerical Value**: Managed by **scanf**, which ensures that the input is a valid integer. If not, **syntaxError** is called.
* **Invalid Expression**: Checked after parsing with the condition **if (top == 0)**. If the stack is not empty, an error message is displayed, and the program exits.

**A screenshot of a computer

Description automatically generatedTask-2 Output**

**Task-3**

In Task 3, the goal is to enhance the existing parser to not only print the stack implementation table but also generate assembly code for valid input strings. This involves implementing a Syntax-Directed Translation (SDT) to produce assembly instructions based on the parsed expression.

The algorithm for this was roughly mentioned in the question paper, it was followed to complete this task.

First we make two new global variables, for the assembly file, and the register count. Whenever a lexeme (variable identifier) was detected, it was written into the file along with its specific register (starting from R0), and whenever any operation was performed, a new register was made aswell. Since we were given an unlimited amount of registers to work with, we could smartly utilize the register counts to ensure each new operation register is re-using the result values from the previous registers. How? For each operation, the code calculates the result using values stored in the previous two registers (by subtracting), ensuring reuse of the intermediate values. So, when a new operation is performed, it uses the result from the two previous registers to generate the correct value. This smartly avoids unnecessary memory usage and ensures efficient computation.

**Syntax-Directed Translation (SDT)**

**Algorithm Overview:**

1. Traverse the parsed expression and generate assembly instructions for each operation.
2. Use registers to store intermediate values and perform arithmetic operations.

**Implementation Details**

**New Global Variables**

**A black background with white text

Description automatically generated**

**A black screen with white text

Description automatically generatedIntegration into factor() Function**

* **Explanation**: When an identifier (variable) is encountered, its value is loaded into a register, and assembly instruction **LI** (Load Immediate) is written to the assembly file.

**A computer screen with text

Description automatically generatedIntegration into MUL\_or\_DIV() Function**

* **Explanation**: Generates assembly instructions for multiplication and division operations. Intermediate results are stored in registers, and the corresponding assembly code is written to the file.

**A computer screen shot of a black background

Description automatically generatedIntegration into ADD\_or\_SUB() Function**

* **Explanation**: Generates assembly instructions for addition and subtraction operations. Intermediate results are stored in registers, and the corresponding assembly code is written to the file.

**A black screen with white text

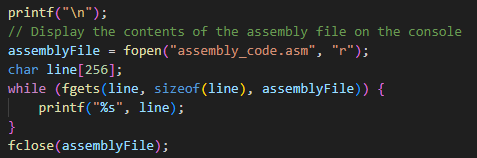
Description automatically generatedIntegration into syntaxAnalysis() Function**

* **Explanation**: Opens the assembly code file for writing and writes a header indicating the generated assembly code for the input expression.

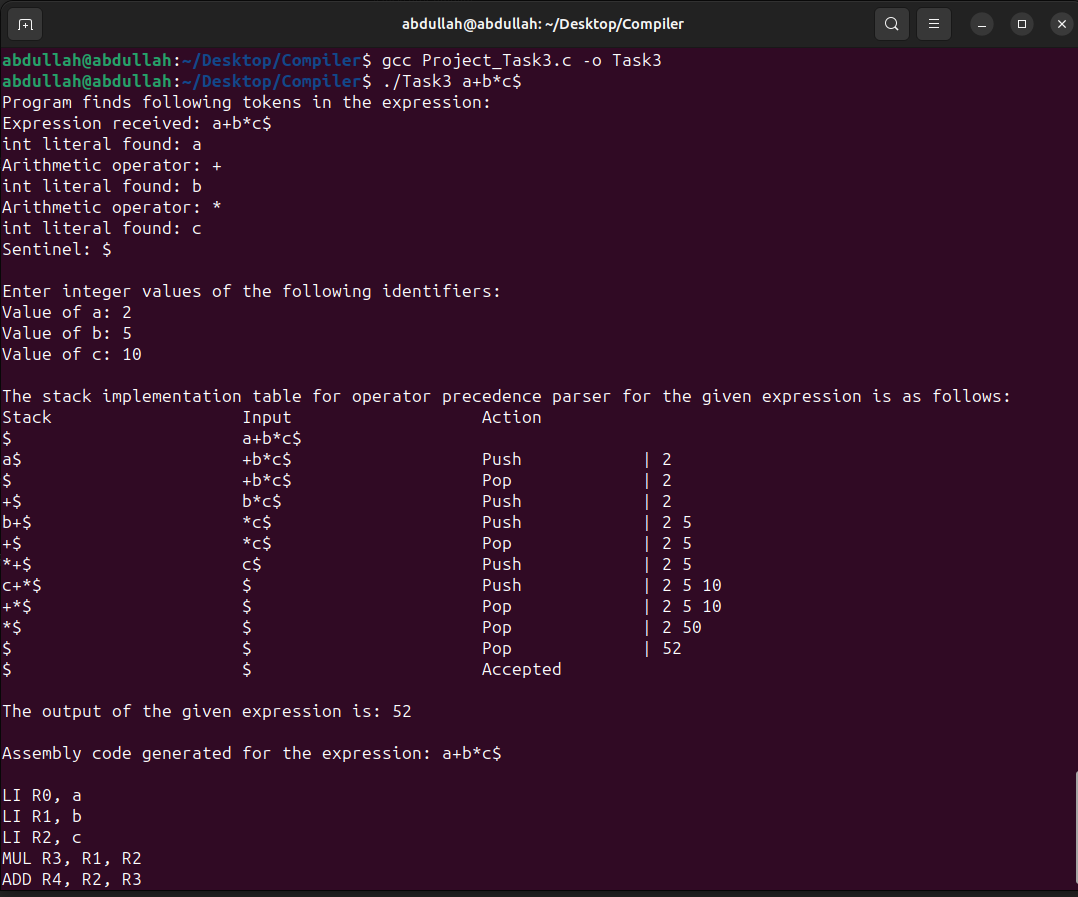


* **Explanation**: Closes the assembly code file after writing the assembly instructions.

**Integration into main() Function**

****

* **Explanation**: Opens the assembly code file for reading and prints its contents (assembly instructions) on the console.

**Task-3 Output**

**Screenshots of random Input’s Output**

**A screenshot of a computer

Description automatically generatedSample-1**

**A black background with white text

Description automatically generated**

A black and white screen with white text

Description automatically generatedCalculator Proof

**A screenshot of a computer

Description automatically generatedSample-2**

**A computer screen with a purple background

Description automatically generated**

Calculator Proof

**A black background with white text

Description automatically generated**

**A screenshot of a computer program

Description automatically generatedSample-3**

**A computer screen shot

Description automatically generated**

**A screenshot of a computer

Description automatically generated**Calculator Proof