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

****

**Compiler Construction || COMP-451**

**Section A**

**Project**

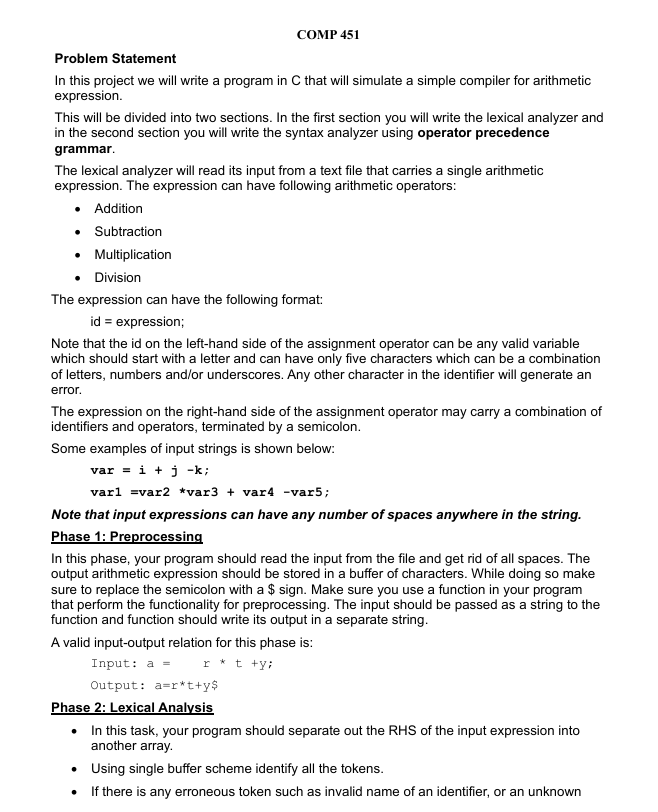
**Group Members:**

**Muhammad Nabeel Mahmood || 241607845**

**Nasir Khan Kakar || 251687559**

**Submitted to:**

**Sir Rauf Butt**

****

**Introduction**

In this project, we build a simple compiler for arithmetic expressions supporting addition, subtraction, multiplication, and division. The compiler processes expressions in three phases: preprocessing, lexical analysis, and syntax analysis. Preprocessing removes spaces and replaces the semicolon with a '$' sentinel. Lexical analysis identifies valid tokens, including identifiers and operators, and flags invalid elements. If any invalid tokens are found, the program terminates with an error message. Next, the syntax analyzer implements an operator precedence parser. It requests user input for each identifier's numeric value, then systematically evaluates the expression. A detailed stack-based table is displayed to show step-by-step parsing and evaluation, with each operator applied according to precedence rules. Error handling includes division by zero and incorrect expressions. Upon successful parsing, the compiler outputs the result or an error message if the input is invalid. This project exemplifies key concepts in compiler design. Consolidates learning in tokenization and parsing.

In this project, we aim to simulate a simple compiler for arithmetic expressions. Specifically, we focus on three major phases:

**Preprocessing (Phase 1)**

(as per lab 3 understanding)

Removing spaces and replacing ; with $.

**Lexical Analysis (Phase 2)**

(as per lab 10 for lexical analyser creating token)

Identifying valid identifiers and operators from the preprocessed string.

Handling erroneous tokens if they exist (e.g., invalid characters or identifiers).

**Syntax Analysis (Phase 3)**

Implementing an operator-precedence-based parser for the four arithmetic operators ( +, -, \*, / ).

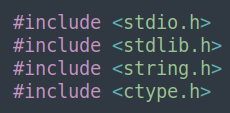
Asking the user for numeric values of identifiers.

Displaying a step-by-step “stack implementation table” to show how the expression is parsed.

Finally, displaying the computed result if the expression is valid.

We are working in a single .c file that contains all these phases. The code deals with reading a single expression of the format: **<id> = <expression>**

**Header Files**



stdio.h is used for standard input/output functions like printf, scanf, fgets.

stdlib.h is used for functions like exit, atoi, and others.

string.h is used for string manipulation (strcpy, strcat, strlen, etc.).

ctype.h is used for functions like isalpha, isalnum, isspace, etc.

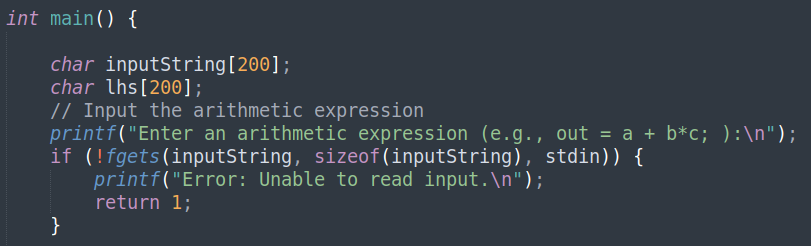
**Algorithm and Logic in Simple English**

1. **Read** a line containing your expression (like out = a + b\*c;).
2. **Preprocess**:
   * Remove spaces.
   * Replace ; with $.
3. **Separate** the left-hand side (LHS) of = from the right-hand side (RHS).
4. **Lexical Analysis** on the RHS:
   * Identify tokens: if it starts with a letter or \_, read it as an IDENTIFIER.
   * If it’s +, -, \*, or /, read it as an OPERATOR.
   * If it’s $, read it as SENTINEL.
   * If something else, error out.
5. **Syntax Analysis**:
   * We parse from left to right. The grammar ensures we do division first, then multiplication, then addition and subtraction.
   * When we see an IDENTIFIER, we ask the user for its numeric value.
   * We push or pop from the stack as we see operators, building partial results step by step.
   * If at the end, we have exactly one item on the stack (and we see the $ sentinel in the correct place), we say “Accepted!” and print that final item as the result. Otherwise, it’s a syntax error.

* **Phase 1**: Preprocess string (remove spaces, replace ; with $).
* **Phase 2**: Tokenize the RHS (operators, identifiers, or $).
* **Phase 3**:
  + Request user values for each identifier.
  + Use a top-down approach to parse with functions: ADD\_or\_SUB -> MUL\_or\_DIV -> DIV -> factor().
  + Evaluate as you parse.
  + If successful, accept and print result.
  + If not, reject and print an error message.

**Code Snippets and Explanations**

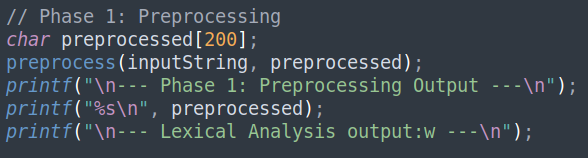
Main() function:



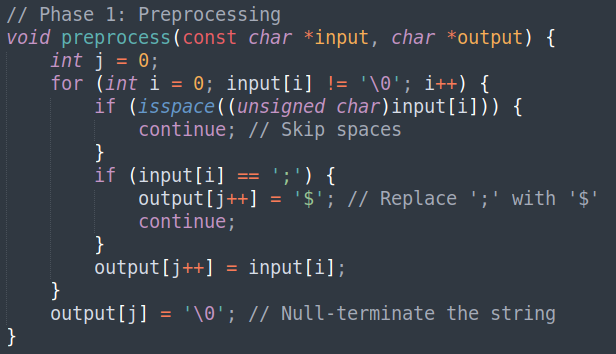
1. **Variables**:
   * inputString[200]: A character array (C-string) used to temporarily store the entire user input (up to 199 characters + null terminator).
   * lhs[200]: A separate buffer used to store the left-hand side (LHS) of the assignment expression (everything up to =).
2. **Reading User Input**:
   * The program prompts the user with printf("Enter an arithmetic expression ...").
   * fgets(inputString, sizeof(inputString), stdin) attempts to read one line from the user (from stdin).
   * If fgets fails (returning NULL), the program prints an error and returns 1 (exiting).

This ensures the expression (e.g., "out = a + b\*c;") is captured in inputString.

**Preprocessing Phase**

****

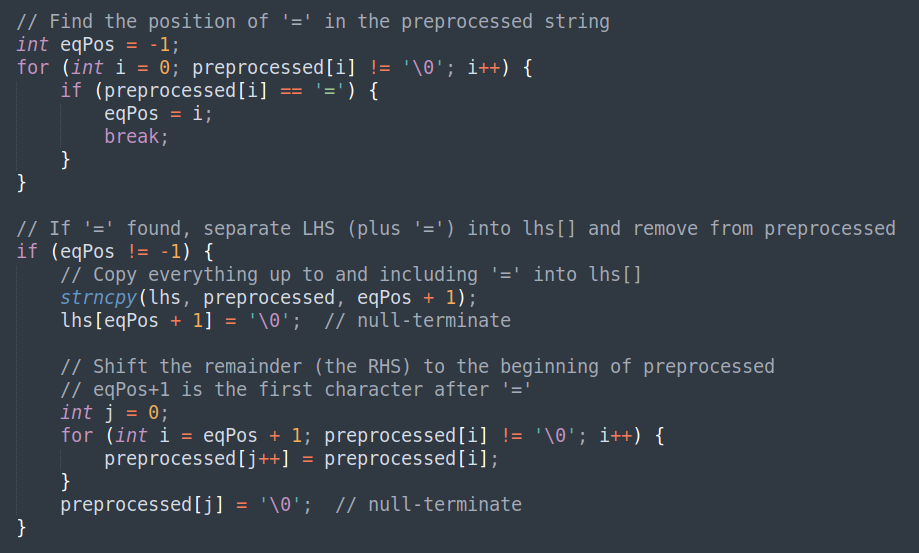
Preprocess() function



1. **Preprocessing**:
   * A separate function preprocess(...) is called with the raw inputString.
   * preprocess removes all spaces and replaces the semicolon (;) with a $.
   * The results are stored in preprocessed[200].
2. **Diagnostic Prints**:
   * The program prints the "**Phase 1: Preprocessing Output**" header, then shows the resulting string after preprocessing.
   * It then prints a header indicating that we’re about to see the Lexical Analysis output.

*(Example: If the user typed out = a + b\*c;, the preprocessed output might be out=a+b\*c$.)*

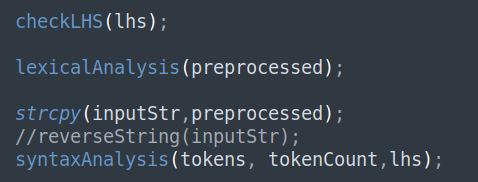
**Lexical Analyzer Phase**



1. **Finding** “=”:
   * We iterate through preprocessed to locate the index of the first '=' character.
   * eqPos is set to that index or remains -1 if not found.
2. **Separating the LHS**:
   * If eqPos != -1, we know = is present. We then copy everything **up to and including** = into lhs.
     + strncpy(lhs, preprocessed, eqPos + 1) copies from preprocessed[0] to preprocessed[eqPos].
     + Then lhs[eqPos + 1] = '\0' ensures the result is null-terminated.
     + Example: if preprocessed is "out=a+b\*c$", then lhs becomes "out=".
   * Next, we shift everything **after** the = to the front of preprocessed. Essentially, this strips out the LHS (and the =) from preprocessed, leaving only the **RHS**.
     + For out=a+b\*c$, we end up with lhs = "out=" and preprocessed = "a+b\*c$".

This step effectively isolates the **variable/identifier** on the left side of the assignment from the rest of the expression.

**Remaining main() function executes the syntax analyser phases**

****

checkLHS(lhs); function

A computer code with text

Description automatically generated with medium confidence

**checkLHS(lhs);**

* This function (not shown in your snippet, but presumably in the code) performs any checks to ensure the LHS is a valid identifier (or valid form of assignment). For instance, it might confirm that lhs ends with = and that the variable name adheres to certain rules (e.g., first character must be alpha or underscore, length constraints, etc.).
* If invalid, it can print an error or exit.

**lexicalAnalysis(preprocessed);**

* Next, we call the **Lexical Analysis** function on the **RHS** (preprocessed), which now only contains something like "a+b\*c$".
* The lexical analyzer will tokenize this string: identifying identifiers, operators, and the sentinel $.
* Valid tokens get stored in the global tokens[] array, and tokenCount is updated accordingly.

**Copying into inputStr**

* strcpy(inputStr, preprocessed); copies the final RHS into the global variable inputStr. This is used later during syntax analysis for reference or printing.

**syntaxAnalysis(tokens, tokenCount, lhs);**

* Finally, the code calls the syntax analysis phase, supplying:
  + The array of tokens (tokens),
  + The count of tokens (tokenCount),
  + The lhs string (e.g., "out=").
* Inside syntaxAnalysis, the code will:
  + Ask the user for numerical values for any identifiers in the RHS,
  + Parse/evaluate the expression according to the operator precedence rules,
  + Print a stack-based parsing table,
  + Provide a final result or display an error if something is invalid.

**Data structures to token**

A computer screen shot of a program

Description automatically generated

1. **typedef enum { ... } TokenType;**

* This defines a new type called TokenType that can be IDENTIFIER, OPERATOR, SENTINEL, or INVALID.
* **IDENTIFIER**: valid variable names like x, var, abc123.
* **OPERATOR**: symbols like +, -, \*, /.
* **SENTINEL**: the $ that we use to replace the semicolon at the end of the expression.
* **INVALID**: for any invalid or unknown token.

2. **typedef struct { char lexeme[64]; TokenType type; } Token;**

* A Token holds a string called lexeme (up to 64 characters) and the TokenType.
* Example: Token t; t.lexeme = "abc"; t.type = IDENTIFIER;

3. **Global arrays and variables**:

* Token tokens[32]: We can store up to 32 tokens (identifiers, operators, etc.).
* int tokenCount: How many valid tokens we have found so far.
* int currentTokenIndex: Points to which token we are currently processing in the syntax analysis.
* char stack[128][64]: We treat this as a stack of up to 128 elements, each of which can be up to 64 characters.
* char stac\_chr[64]: A temporary buffer used for printing/pushing/popping from the stack in a user-friendly manner.
* int top = -1;: The top index of the stack is initially -1, meaning the stack is empty.
* char inputStr[64]: The input expression (after some processing).
* int identifierValues[32]: An array to store numeric values for each identifier (e.g., if x is 3, then identifierValues[...] = 3).

**Stack Operations (will use in stack table iterations for operator precedence)**

**A computer screen shot of a program code

Description automatically generated**

1. **push(char \*value)**
   * Increments the top index by 1 and then copies (strcpy) the value onto the stack at stack[top].
   * If the stack is full (top >= 127), we print Error: Stack overflow and exit.
2. **pop()**
   * Returns the item at stack[top] and then decrements top.
   * If the stack is empty (top < 0), prints Error: Stack underflow and exits.
3. **popBottom()** (not used heavily in this code, but provided for illustration)
   * Returns the “bottom” (first) element of the stack (stack[0]).
   * Then shifts every element in stack up by one position to fill the gap.
   * Decrements top by 1.

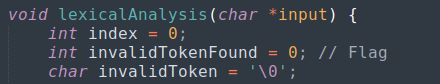
All these functions ensure LIFO (Last In, First Out) behavior for the stack.

**Lexical Analysis and Syntax Analyzer Function**

**Explanation:**

1. **Function Signature:** A blue and white text

   Description automatically generated
   * **input**: the output of the preprocess() which the RHS (expression).
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.

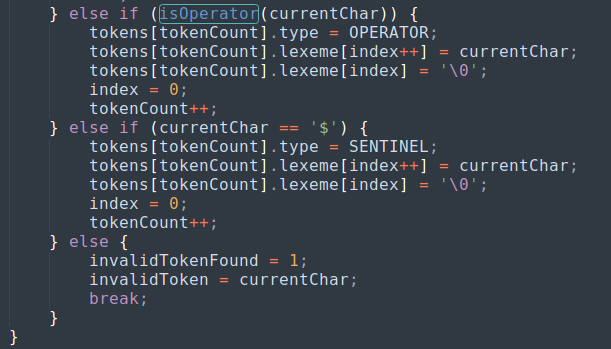
1. **Main Loop:** 
   * Iterates through each character in the input string until the null terminator is reached.
2. **Identifier Handling:**

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:**



* + 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:**
   * Checks if the current character is the sentinel value (**$**).
   * Stores the sentinel in the **tokens** array and updates the **tokenCount**.
2. **Invalid Token Handling:**
   * If the current character is not a valid identifier, operator, or sentinel, it sets the **invalidTokenFound** flag and stores the invalid token character.
3. **Token Output Loop:**

A screen shot of a computer program

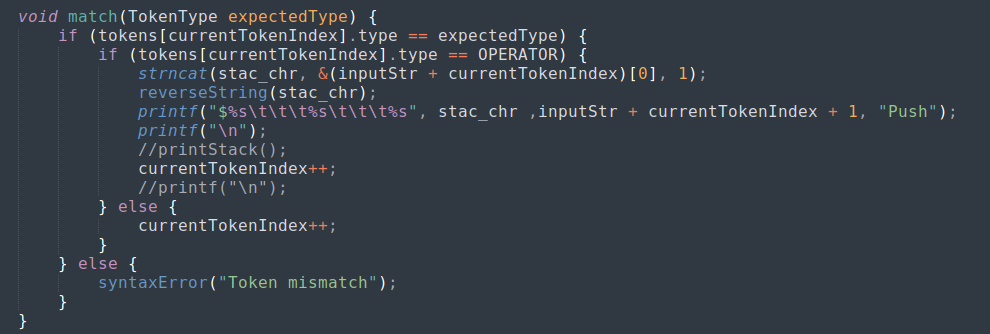
Description automatically generated

* + 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.

**Match(); function**

****

* **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.

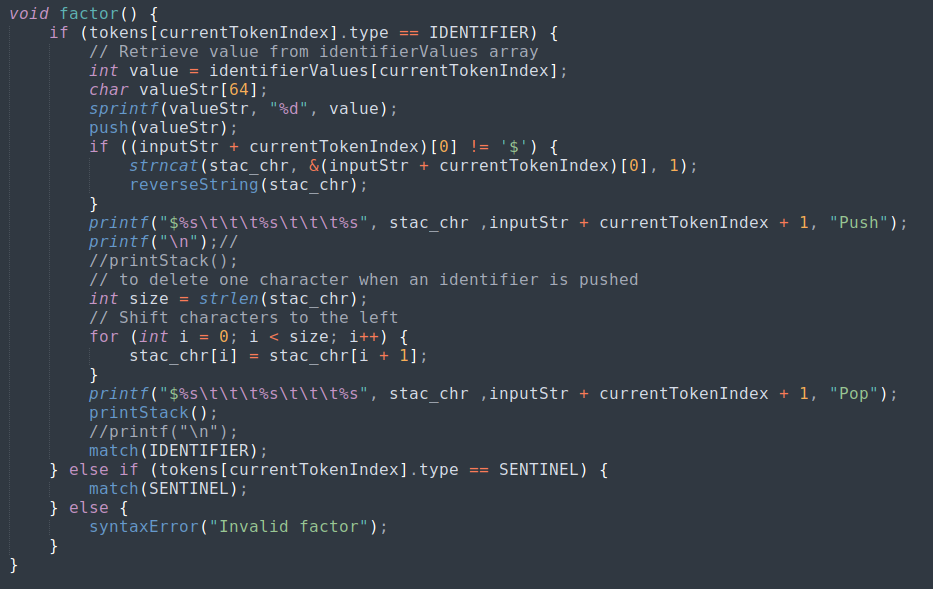
**Condition: Token is a Sentinel**

* **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**

* **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.

**factor(); function**



1. A **factor** in arithmetic grammar is usually the smallest piece like an **identifier** or a **number** (if we had literal numbers). Here, we only have identifiers.

2. If tokens[currentTokenIndex].type == IDENTIFIER, we:

1. Retrieve the integer value the user provided for that identifier (from identifierValues[currentTokenIndex]).
2. Convert that value to a string (sprintf(valueStr, "%d", value)).
3. **Push** that numeric string onto the stack.
4. Print some debugging lines to show what’s happening (the “stack implementation table”).
5. Finally, we call match(IDENTIFIER); to confirm we consumed an IDENTIFIER.

3. If it’s a SENTINEL (meaning $), we do match(SENTINEL).

4. Otherwise, we call syntaxError("Invalid factor");.

**DIV(); function (it is called in every operator function as it has the highest precedence)**



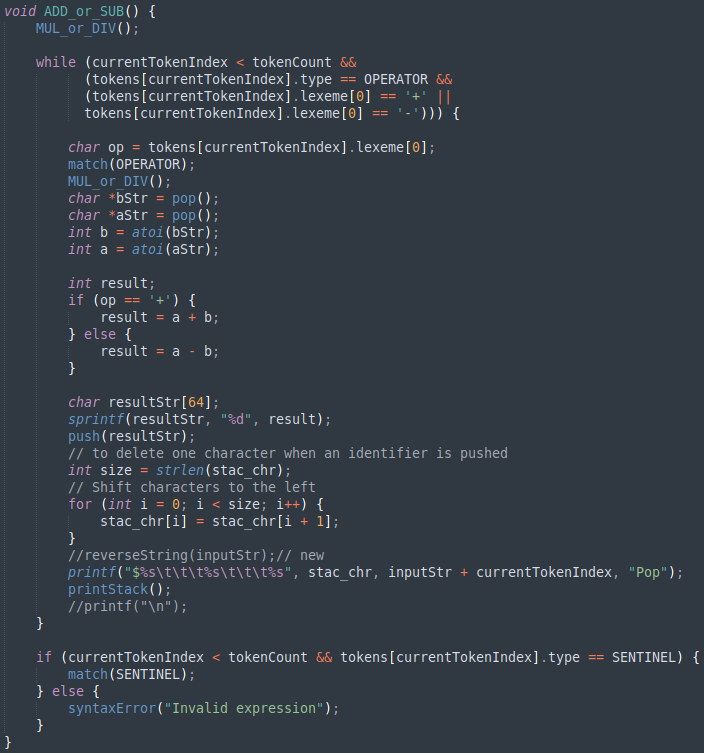
* We **first call** factor(), which might push an identifier’s value or match $.
* Then, **while** the current token is an OPERATOR specifically '/', we do the following:
  1. match(OPERATOR); to acknowledge we see the /.
  2. We call factor() again (this places the next operand on the stack).
  3. **Pop** the top two values: bStr and aStr (these are strings, so we convert them to integers using atoi).
  4. Check for b == 0. If so, print Error: Division by zero. and exit.
  5. Otherwise, do a / b, store it in result.
  6. Push result (converted to string) back onto the stack.
  7. Print a line indicating the pop operation.

(In simpler terms: we parse the immediate factor, then if the next operator is /, we parse another factor, pop the stack (two values), perform the division, push the result back, and keep doing this while we see / as the next operator. If we see anything else, we break out of the loop.)



* This function builds on DIV().
* First, it calls DIV(). This ensures all possible divisions are handled.
* Then it **loops** for as long as the current token is \*.
  + Matches the OPERATOR, calls factor().
  + Pops two values, multiplies them, pushes the result.
  + Prints the intermediate step.

By doing DIV() first, we ensure that / is evaluated before \* if they happen to appear. If your grammar says multiplication and division have the same precedence but left associativity, you can handle them together in one function or separately. This code chooses to handle DIV() first in the function, then handle \*.



* First we call MUL\_or\_DIV(). That handles all / and \* first (since they have higher precedence).
* Then we **loop** as long as the next operator is + or -. For each:
  1. We do match(OPERATOR).
  2. Call MUL\_or\_DIV() again for the next factor.
  3. Pop two values, do a + b or a - b according to the operator.
  4. Push the result back on the stack.
  5. Print the resulting table row.
* Once we exit the loop, we check if the next token is SENTINEL. If it is, we match(SENTINEL). If not, we call syntaxError("Invalid expression");.

Thus, the expression ends with a $ to confirm correctness.

(In normal arithmetic, \* and / have higher precedence, so you must do those before you do + or -. This is exactly what our code is ensuring.)

**Syntax Analyzer();**

**A screen shot of a computer

Description automatically generated**

A computer screen shot of a program code

Description automatically generated

1. **Asking for identifier values**:

* We loop through all tokens; whenever we see an IDENTIFIER, we prompt the user to input a numerical value.
* We store that value in identifierValues[i].

2. **Printing the heading** for our table:

* We show columns labeled Stack, Stack2, Action, and Stack3.
* The initial line prints something like $\t\t\t<inputStr>\t\t\tNone.

3. **Call ADD\_or\_SUB()** to start parsing from the highest-level grammar rule.

4. If top == 0 after finishing, we say it’s “Accepted.” Because if the expression is valid, we expect exactly one final result on the stack.

5. We then print the final result:

* printf("%s %s", lhs, stack[top]); means “identifier = result.”

6. Otherwise, we print an error and reject the string.

**Detailed explanation of SyntaxAnalyzer**

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**

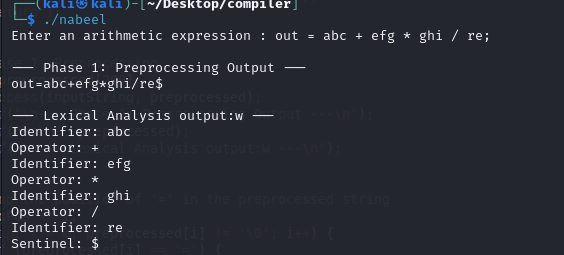
* **Explanation**: Checks if the parsing was successful and prints the result.
  + **Condition (if)**: Checks if the stack is empty, indicating successful parsing.
    - **top == 0**: Evaluates whether the stack is empty.
    - **Success**: If true, prints the accepted message and the result.
      * **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)**.

**OUTPUT screen shots**

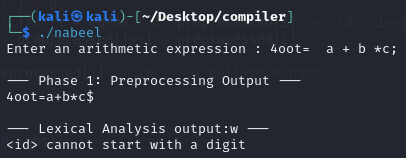
**A black background with white text

Description automatically generated**

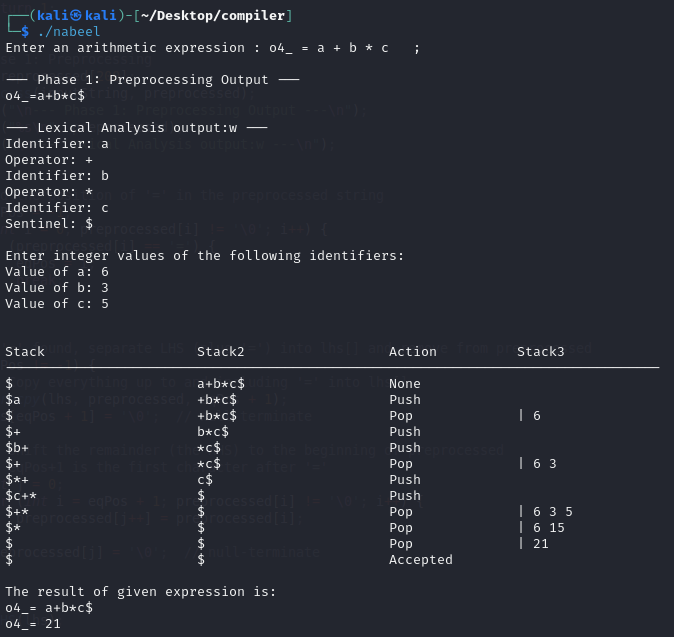
**Comment:** Preprocessing removing all spaces and swapping “;” with “$”.



**Comment:** lexical analyser separating LHS RHS and operators to identifiers.



**Comment:** lexical analyser error detecting

****

**Comment:** syntax analyser providing correct result.

(per my performance in lab 11 and 10 i was unable to reverse itrate the stack2. The logic in lab 11 was also having the stack2 ireverseable.)

A screenshot of a computer program

Description automatically generated

**Comment:** “/” has greater precedence than “\*” and “\*” is greater than “+”,”-”.

