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

****

**COMP-451**

**Session**

**Project**

**Annas Ahmad 231-520655**

Introduction:

The provided C code is a simple lexical analyzer designed to tokenize arithmetic expressions. Lexical analysis is an essential step in the compilation process, where the input source code is broken down into a sequence of tokens for further processing. Tokens are the fundamental building blocks representing syntactic elements in a programming language.

The code defines a set of libraries, including <stdio.h>, <ctype.h>, and <string.h>, to leverage standard functions for input/output, character handling, and string manipulation, respectively. These libraries contribute to the overall functionality of the program.

The core of the code revolves around the identification and categorization of different types of tokens within an arithmetic expression. It utilizes an enumeration called TokenType to represent distinct token types, such as IDENTIFIER, OPERATOR, SEMICOLON, and ERROR. The structure Token is introduced to store information about each identified token, including its type and value.

To preprocess the input expression, the code employs the preprocess function, which eliminates spaces and replaces semicolons with the dollar sign ($). The preprocessed expression is then passed to the tokenize function, responsible for scanning and categorizing individual tokens. The identified tokens, along with their types and values, are printed within the tokenize function.

While the code successfully tokenizes basic arithmetic expressions, there are considerations for improvement. For instance, the current implementation prints tokens directly within the tokenization loop; a more modular design might involve storing tokens in an array or another data structure for subsequent analysis. Additionally, the assumption of a maximum identifier length of 5 characters might limit its applicability in scenarios where longer identifiers are expected.

In summary, this C code provides a foundational understanding of lexical analysis, showcasing the identification and categorization of tokens in an arithmetic expression, serving as a fundamental step towards the compilation of programming languages.

Libraries:

The code includes three standard C libraries:

1. <stdio.h>: Standard Input and Output functions, used for input/output operations.
2. <ctype.h>: Character handling functions, such as isalnum for checking if a character is alphanumeric and isspace for checking if a character is a space.
3. <string.h>: String manipulation functions, used for operations like string comparison and copying.

Explanation

1. Token Types:

* The code defines an enumeration TokenType to represent different types of tokens. The supported token types are IDENTIFIER, OPERATOR, SEMICOLON, and ERROR.

1. Token Structure:

* A Token structure is defined to store information about a token. It includes a TokenType and a character array value to store the token value.

1. isValidIdentifierChar Function:

* The function isValidIdentifierChar checks whether a given character is a valid character for an identifier. It returns a non-zero value if the character is alphanumeric or an underscore (\_).

1. preprocess Function:

* The preprocess function takes an input expression and removes spaces while replacing semicolons (;) with the dollar sign ($). The result is stored in the output buffer.

1. tokenize Function:

* The tokenize function processes the preprocessed input and identifies tokens.
* It uses a loop to iterate through each character in the input.
* It checks for identifiers, operators (+, -, \*, /), and the dollar sign ($) representing semicolons. Invalid characters are categorized as errors.
* For identifiers, it reads characters until it encounters a non-identifier character or reaches the maximum length (5 characters).
* The identified tokens are printed with their type and value.

1. main Function:

* The main function:
* Declares character arrays input and preprocessed to store the input expression and its preprocessed version.
* Reads the arithmetic expression from the user using fgets.
* Calls the preprocess function to clean the input.
* Calls the tokenize function to tokenize the preprocessed input and print the resulting tokens.

Code-1:

#include <stdio.h>

#include <ctype.h>

#include <string.h>

typedef enum {

    IDENTIFIER,

    OPERATOR,

    SEMICOLON,

    ERROR

} TokenType;

typedef struct {

    TokenType type;

    char value[6];

} Token;

int isValidIdentifierChar(char ch) {

    return isalnum(ch) || ch == '\_';

}

void preprocess(char \*input, char \*output) {

    int i = 0, j = 0;

    while (input[i] != '\0') {

        if (!isspace(input[i])) {

            if (input[i] == ';') {

                output[j++] = '$';

            } else {

                output[j++] = input[i];

            }

        }

        i++;

    }

    output[j] = '\0';

}

void tokenize(char \*input) {

    Token token;

    int i = 0;

    while (input[i] != '\0') {

        if (isalpha(input[i]) && isValidIdentifierChar(input[i])) {

            int j = 0;

            while (isValidIdentifierChar(input[i]) && j < 5) {

                token.value[j] = input[i];

                i++;

                j++;

            }

            token.value[j] = '\0';

            token.type = IDENTIFIER;

        }

        else if (strchr("+-\*/", input[i]) != NULL) {

            token.type = OPERATOR;

            token.value[0] = input[i];

            token.value[1] = '\0';

            i++;

        }

        else if (input[i] == '$') {

            token.type = SEMICOLON;

            token.value[0] = '$';

            token.value[1] = '\0';

            i++;

        }

        else {

            token.type = ERROR;

            token.value[0] = input[i];

            token.value[1] = '\0';

            i++;

        }

        printf("Token: Type=%d, Value=%s\n", token.type, token.value);

    }

}

int main() {

    char input[100];

    char preprocessed[100];

    printf("Enter the arithmetic expression: ");

    fgets(input, sizeof(input), stdin);

    preprocess(input, preprocessed);

    tokenize(preprocessed);

    return 0;

}

Inputs & Outputs

A screen shot of a computer screen

Description automatically generated