Programmıng languages project ı

sarı, emotikon, gülenyüz, yaratıcılık içeren bir resim

Açıklama otomatik olarak oluşturuldu

Ferdi Can 05220000957

Samet Kılıç 05230000202

LEXICAL ANALYZER PROJECT

Contents

[Project Purpose and Overview 2](#_Toc165901061)

[Method strToInt; 2](#_Toc165901062)

[Method isSpecial; 3](#_Toc165901064)

[Method Reset; 4](#_Toc165901066)

[isInKeywords Method; 4](#_Toc165901068)

[isInOperators Method; 5](#_Toc165901069)

[identifierControl Method; 6](#_Toc165901070)

[integerControl Method; 7](#_Toc165901071)

[operatorControl Method; 9](#_Toc165901072)

[bracketControl Method; 10](#_Toc165901073)

[stringControl Method; 11](#_Toc165901074)

[keywordControl Method; 12](#_Toc165901075)

[endOfLineControl Method; 13](#_Toc165901076)

[commaControl Method; 14](#_Toc165901077)

[Method fileReading; 14](#_Toc165901078)

[Algorithm: Lexical Analysis and Tokenization of Source Code 18](#_Toc165901079)

[Method creatingFile; 20](#_Toc165901080)

[Code Explanation; 20](#_Toc165901081)

[Main Method; 22](#_Toc165901082)

[Test Cases; 22](#_Toc165901083)

[Case 1 Input File: 22](#_Toc165901084)

[Output File; 23](#_Toc165901085)

[Case 2 Input File; 23](#_Toc165901086)

[Output File; 24](#_Toc165901087)

[Case 3 Input File; 24](#_Toc165901088)

[Output File; 24](#_Toc165901089)

[Case 4 Input File; 25](#_Toc165901090)

[Output File; 25](#_Toc165901091)

[Case 5 Input File; 26](#_Toc165901092)

[Output File; 27](#_Toc165901093)

[Case 6 Input File; 27](#_Toc165901094)

[Output File; 28](#_Toc165901095)

# Project Purpose and Overview

**Project Title:** Lexical Analyzer for STAR Programming Language

**Objective:** The primary objective of this project is to develop a lexical analyzer for the STAR programming language. This lexical analyzer will parse a source file extended with “.sta” written in STAR language and produce a corresponding tokenized representation of the code. The tokenized representation will be saved in a file extended with “.lex” for further processing by other components of a compiler or interpreter.

**Background:** STAR is a small scripting language designed for basic arithmetical operations. It supports two data types: integers and strings, and has a limited set of features compared to more complex programming languages. The lexical analyzer is the first phase of a compiler or interpreter, responsible for breaking down the source code into individual tokens, which are the smallest units of meaning in the language.

**Functionality:** The lexical analyzer will tokenize the source code according to the lexical rules defined for the STAR language. It will identify and categorize identifiers, integer constants, operators, brackets, string constants, keywords, end of line markers, commas, and comments. Each token will be represented by its type and value, and the tokenized representation will closely follow the syntax and structure of the source code.



# Method strToInt;

int strToInt(char \*intStr) {

    int number = atoi(intStr);

    return number;

}

## Code Explanation;

* **Purpose**: This function converts a string representing an integer to an actual integer value.
* **Parameters**:
  + **intStr**: Pointer to the string containing the integer representation.
* **Return Value**: The integer value parsed from the string.
* **Algorithm**:
  + It uses the **atoi** function from the standard library, which converts the initial portion of the string pointed to by **intStr** to an integer representation.
  + The resulting integer value is then returned.

# Method isSpecial;

int isSpecial(char indexCharacter) {

    int index = 0;

    while (specialChars[index] != '\0') {

        if (indexCharacter == specialChars[index])

            return 1;

        index++;

    }

    return 0;

}

## Code Explanation;

* **Purpose**: This function checks whether a given character is a special character.
* **Parameters**:
  + **indexCharacter**: The character to be checked.
* **Return Value**: Returns 1 if the character is special, 0 otherwise.
* **Algorithm**:
  + It iterates through the array **specialChars** until it encounters the null terminator **'\0'**.
  + If **indexCharacter** matches any character in the **specialChars** array, it returns 1, indicating that the character is special. Otherwise, it returns 0.

# Method Reset;

void reset(char \*str) {

    int uzunluk = strlen(str);

    for (int i = 0; i < uzunluk; i++) {

        str[i] = '\0';

    }

}

## Code Explanation;

* **Purpose**: This function resets a string by setting all its characters to null terminators **'\0'**.
* **Parameters**:
  + **str**: Pointer to the string to be reset.
* **Return Value**: Void (no return value).
* **Algorithm**:
  + It calculates the length of the string using **strlen**.
  + Then, it iterates through each character of the string and sets it to **'\0'**, effectively resetting the string to an empty state.

# isInKeywords Method;

**Purpose**: This function checks if a given string str is present in an array of keywords called keyWords. It returns 1 if the string is found in the array, indicating it's a keyword; otherwise, it returns 0, indicating it's not a keyword.

**Parameters**:

**str**: Pointer to the string being checked for being a keyword.

Return Value:

1: If the string is found in the array keyWords, indicating it's a keyword.

0: If the string is not found in the array keyWords, indicating it's not a keyword.

**Algorithm**:

It iterates through the keyWords array, which presumably contains 8 keywords (as per the loop's termination condition).

For each iteration, it uses strcmp function to compare the string str with the keyword at the current index of the keyWords array.

If a match is found (i.e., strcmp returns 0), it means the string str is equal to the current keyword being checked. In this case, the function returns 1, indicating that the string is a keyword.

If the loop finishes without finding a match, it means the string str is not present in the keyWords array, and hence the function returns 0, indicating that the string is not a keyword.

int isInKeywords(char \*str)

{

    for (int i = 0; i < 8; i++)

    {

        if (strcmp(str, keyWords[i]) == 0)

        {

            return 1; // Keyword found.

        }

    }

    return 0; // Keyword not found.

}

# isInOperators Method;

**Purpose**: This function checks if a given character str is present in a 2D array operators. It returns 1 if the character is found in the first row of the array, indicating it's an operator; otherwise, it returns 0.

**Parameters**:

**str**: The character being checked for being an operator.

**Return Value:**

1: If the character is found in the first row of the operators array, indicating it's an operator.

0: If the character is not found in the first row of the operators array, indicating it's not an operator.

**Algorithm**:

It iterates through the first row of the operators array, which presumably contains 4 operators (as per the loop's termination condition).

For each iteration, it checks if the character str is equal to the character at the current index of the first row of the operators array.

If a match is found, it means the character str is equal to one of the operators in the first row of the operators array. In this case, the function returns 1, indicating that the character is an operator.

If the loop finishes without finding a match, it means the character str is not present in the first row of the operators array, and hence the function returns 0, indicating that the character is not an operatör

int isInOperators(char str)

{

    for (int i = 0; i < 4; i++)

    {

        if (str == operators[0][i])

        {

            return 1;

        }

    }

    return 0;

}

# identifierControl Method;

If the string does not meet the identifier criteria or encounters an error.

**Algorithm**:

Check if the length of the string str is greater than 10 characters. If so, print an error message indicating that the identifier is too long, and return 0.

Check if the first character of the string str is not an alphabetical character (i.e., a letter). If not, print an error message indicating that the identifier must start with a letter, and return 0.

If both conditions pass, print the identifier and return 0.

int identifierControl(char \*str, FILE \*ptr)

{

    if (strlen(str) > 10)

    {

        fprintf(ptr, "Error: Identifier is too long.\n");

        return 0;

    }

    else if (!isalpha(str[0]))

    {

        fprintf(ptr, "Error: Identifier must start with a letter.\n");

        return 0;

    }

    else

    {

        fprintf(ptr, "Identifier(%s)\n", str);

        return 0;

    }

}

# integerControl Method;

**Purpose**: This function validates an integer num to ensure it meets certain criteria.

**Parameters**:

**num**: The integer being validated.

**ptr**: Pointer to a file stream where error messages or integer information will be printed.

**Return Value:**

1: If the integer meets the criteria and is successfully processed.

0: If an error is encountered or if the integer does not meet the criteria.

**Algorithm**:

**Negative Number Check:**

If the integer num is negative, it prints an error message indicating that the integer is negative.

Then, it makes a recursive call to the function with 0 as the argument. This recursive call is likely intended to handle the case of negative numbers, but it's problematic because it doesn't have a condition to break out of the recursion for negative numbers. This can lead to an infinite loop or stack overflow error.

**Integer Length Check:**

If the integer num is greater than 99999999, it prints an error message indicating that the integer is too long.

**Integer Validity Check:**

If the integer num is within the valid range and not negative, it prints the integer with a message indicating that it's valid.

If the integer meets the criteria, it returns 1 to indicate success.

**Issues**:

The recursive call when num is negative doesn't have a condition to stop the recursion. This can lead to infinite recursion and potential stack overflow.

There's a redundant check for num < 0 inside the else block. Since it's already checked at the beginning of the function, it's unnecessary here.

int integerControl(int num, FILE \*ptr)

{

    if (num < 0)

    {

        fprintf(ptr, "Error: Integer is negative.\n");

        integerControl(0, ptr); // Recursive call to negative nums.

    }

    else

    {

        if (num > 99999999)

        {

            fprintf(ptr, "Error: Integer is too long.\n");

            return 0;

        }

        else if (num < 0)

        {

            fprintf(ptr, "Error: Integer is negative.\n");

            return 0;

        }

        else

        {

            fprintf(ptr, "IntConts(%d)\n", num);

            return 1;

        }

    }

}

# operatorControl Method;

**Purpose**: This function verifies if a given string str represents a valid operator. It checks if the string has a length of 1 character and if that character is present in an array of operators.

**Parameters**:

str: Pointer to the string being checked for being an operator.

ptr: Pointer to a file stream where information or error messages will be printed.

Return Value:

1: If the string represents a valid operator.

0: If the string does not represent a valid operator or if its length is not exactly 1.

**Algorithm**:

**Length Check:**

If the length of the string str is not 1 character, it means it cannot represent a valid operator, so the function returns 0.

**Operator Check:**

If the length is 1, it checks if the character at index 0 of the string is present in an array of operators (presumably stored elsewhere).

It uses the isInOperators function to check if the character is in the array of operators.

If the character is found in the array, it prints a message indicating that it's an operator and returns 1.

**Default Return:**

If the string length is 1 but the character is not found in the array of operators, the function returns 0, indicating that it's not a valid operator.

int operatorControl(char \*str, FILE \*ptr)

{

    if (strlen(str) != 1)

    {

        return 0;

    }

    else if (isInOperators(str[0]))

    {

        fprintf(ptr, "Operator(%s)\n", str);

        return 1;

    }

    else

    {

        return 0;

    }

}

# bracketControl Method;

**Purpose**: This function checks if the first character of the given string str represents a left curly bracket { or a right curly bracket }.

**Parameters**:

**str**: Pointer to the string being checked for being a curly bracket.

**ptr**: Pointer to a file stream where information or error messages will be printed.

Return Value:

1: If the first character of the string represents a curly bracket.

0: If the first character of the string does not represent a curly bracket.

**Algorithm**:

**Left Curly Bracket Check:**

If the first character of the string str is {, it prints a message indicating it's a left curly bracket and returns 1.

**Right Curly Bracket Check:**

If the first character of the string str is }, it prints a message indicating it's a right curly bracket and returns 1.

**Default Return:**

If the first character of the string is neither { nor }, the function returns 0, indicating that it's not a curly bracket.

int bracketControl(char \*str, FILE \*ptr)

{

    if (str[0] == '{')

    {

        fprintf(ptr, "LeftCurlyBracket\n");

        return 1;

    }

    else if (str[0] == '}')

    {

        fprintf(ptr, "RightCurlyBracket\n");

        return 1;

    }

    else

    {

        return 0;

    }

}

# stringControl Method;

**Purpose**: This function checks if a given string str represents a valid string constant. It verifies if the length of the string is within a specified limit (256 characters in this case).

**Parameters**:

str: Pointer to the string being checked for being a string constant.

ptr: Pointer to a file stream where information or error messages will be printed.

**Return Value:**

1: If the string represents a valid string constant.

0: If the length of the string exceeds the specified limit.

**Algorithm**:

**Length Check:**

If the length of the string str is greater than 256 characters, it prints an error message indicating that the string constant exceeds the limit.

**Valid String Print:**

If the length of the string is within the limit, it prints a message indicating that it's a string constant, enclosing the string within double quotes ".

**Return**:

It returns 1 if the string is within the limit and successfully processed. Otherwise, it returns 0.

int stringControl(char \*str, FILE \*ptr)

{

    if (strlen(str) > 256)

    {

        fprintf(ptr, "String constant exceeds 256 characters!");

        return 0;

    }

    else

    {

        fprintf(ptr, "String(\"%s\")\n", str);

        return 1;

    }

}

# keywordControl Method;

**Purpose**: This function checks if a given string str is a keyword by checking if it exists in a predefined list of keywords.

**Parameters**:

str: Pointer to the string being checked for being a keyword.

ptr: Pointer to a file stream where information or error messages will be printed.

**Return Value:**

1: If the string is a keyword.

0: If the string is not a keyword.

**Algorithm**:

**Keyword** **Check**:

It checks if the string str is present in a list of keywords using the isInKeywords function.

If the string is found in the list of keywords, it prints a message indicating that it's a keyword and returns 1.

**Default Return:**

If the string is not found in the list of keywords, it returns 0, indicating that it's not a keyword.

int keywordControl(char \*str, FILE \*ptr)

{

    if (isInKeywords(str))

    {

        fprintf(ptr, "Keyword(%s)\n", str);

        return 1;

    }

    else

    {

        return 0;

    }

}

# endOfLineControl Method;

**Purpose**: This function checks if the given string str represents an end of line marker, which is defined as a string consisting of a single period ..

**Parameters:**

str: Pointer to the string being checked for being an end of line marker.

ptr: Pointer to a file stream where information or error messages will be printed.

**Return Value:**

1: If the string represents an end of line marker.

0: If the string does not represent an end of line marker.

**Algorithm**:

**End of Line Check:**

It checks if the first character of the string str is a period ..

If it is, it prints a message indicating that it's the end of line marker and returns 1.

**Default Return:**

If the first character of the string is not a period . then it returns 0, indicating that it's not the end of line marker.

int endOfLineControl(char \*str, FILE \*ptr)

{

    if (str[0] == '.')

    {

        fprintf(ptr, "EndOfLine\n");

        return 1;

    }

    else

    {

        return 0;

    }

}

# commaControl Method;

**Purpose**: This function checks if the first character of the given string str is a comma ,, indicating the presence of a comma.

**Parameters**:

str: Array of characters representing the string being checked for a comma.

ptr: Pointer to a file stream where information or error messages will be printed.

**Return Value:**

1: If the first character of the string is a comma, indicating the presence of a comma.

0: If the first character of the string is not a comma.

**Algorithm**:

**Comma Check:**

It checks if the first character of the string str is a comma ,.

If it is, it prints a message indicating that it's a comma and returns 1.

**Default Return:**

If the first character of the string is not a comma, it returns 0, indicating that there's no comma present.

int commaControl(char str[100], FILE \*ptr)

{

    if (str[0] == ',')

    {

        fprintf(ptr, "Comma\n");

        return 1;

    }

    else

    {

        return 0;

    }

}

# Method fileReading;

void fileReading(char \*filePath) {

    FILE \*fptr;

    fptr = fopen(filePath, "r");

    FILE \*newPtr;

    newPtr = createFile(filePath);

    char lineString[100];

    char tokenString[257] = {'\0'}; // for tokenStrings to identify which token it is.

    int commentControl = 0;

    int strconControl = 0;

    int tokenIndex = 0;

    // Read the line content and print it

    while (fgets(lineString, 100, fptr)) { // Beginning of everyline of the file.

        int index = 0;

        // indexes of the string which will be parameterized.

        int startIndex = 0;

        while (lineString[index] != '\0') { // Beginning of everyindex of the line.

            char charOfIndex = lineString[index];

            char charOfStart = lineString[startIndex];

            char commentStrings[] = {charOfIndex, lineString[index + 1], '\0'};

            int commentCondition = strcmp("/\*", commentStrings) == 0 && tokenString[0] == '\0';

            // Comment finding

            if (commentCondition && !commentControl) { // If they are the same:

                index = index + 2;

                startIndex = index;

                commentControl = 1;

                continue;

            } else if (commentControl && tokenString[0] == '\0') { // If commentControl is 1, it means searching for "\*/".

                if (strcmp("\*/", commentStrings) == 0) {

                    index = index + 2;

                    startIndex = index;

                    commentControl = 0;

                } else {

                    index++;

                }

                continue;

            }

            // String tokenizer.

            if (charOfIndex == '"' && strconControl == 0 && tokenString[0] == '\0') { // If strconControl is 1, it means searching for "\*/".

                index++;

                startIndex = index;

                strconControl = 1;

                continue;

            } else if (strconControl == 1 && charOfIndex == '"') {

                stringControl(tokenString, newPtr);

                strconControl = 0;

                index++;

                startIndex++;

                reset(tokenString);

                tokenIndex = 0;

                continue;

            } else if (strconControl) {

                tokenString[tokenIndex++] = charOfIndex;

                tokenString[tokenIndex] = '\0';

                index++;

                startIndex++;

                continue;

            }

            // Integer tokenizer.

            // Finds integer and tokenizes it.

            int firstCond = isdigit(charOfStart);

            int secondCond = (charOfStart == '-' && isdigit(lineString[startIndex + 1]));

            int condition = isdigit(charOfIndex) || (charOfIndex == '-' && index == startIndex);

            if (firstCond || secondCond) {

                if (condition) {

                    tokenString[tokenIndex++] = lineString[index++];

                    continue;

                }

                // If enters else block, it means end of the integer token and it is time to tokenize the value.

                else {

                    startIndex = index;

                    tokenString[tokenIndex] = '\0';

                    int returnedInt = strToInt(tokenString);

                    integerControl(returnedInt, newPtr);

                    reset(tokenString);

                    tokenIndex = 0;

                    continue;

                }

                // nonInteger tokenizer.

                // Finds nonIntegers and tokenizes it.

            } else {

                int condition = charOfIndex != ' ' && charOfIndex != '\n';

                int condition2 = !isSpecial(charOfIndex) && charOfIndex != '"';

                if (condition && condition2) {

                    tokenString[tokenIndex++] = lineString[index++];

                    continue;

                }

                // If enters else block, it means end of the nonInteger token and it is time to tokenize the value.

                else {

                    startIndex = index;

                    tokenString[tokenIndex] = '\0';

                    if (keywordControl(tokenString, newPtr)) {

                    } else if (isalpha(charOfStart)) {

                        identifierControl(tokenString, newPtr);

                    }

                    if (charOfIndex == '/' && lineString[index + 1] == '\*') {

                        if (commentControl) {

                            continue;

                        } else {

                            reset(tokenString);

                            tokenIndex = 0;

                            continue;

                        }

                    }

                    if (isSpecial(charOfIndex)) {

                        tokenString[0] = charOfIndex;

                        tokenString[1] = '\0';

                    }

                    if (isInOperators(charOfIndex)) {

                        operatorControl(tokenString, newPtr);

                    }

                    if (charOfIndex == '{' || charOfIndex == '}') {

                        bracketControl(tokenString, newPtr);

                    }

                    if (charOfIndex == '.') {

                        endOfLineControl(tokenString, newPtr);

                    }

                    if (charOfIndex = ',') {

                        commaControl(tokenString, newPtr);

                    }

                    reset(tokenString);

                    tokenIndex = 0;

                    if (lineString[index] == '"')

                        continue;

                }

            }

            startIndex++;

            index++;

            // Every end of the index, we check the next index.

        }

    }

    if (commentControl) {

        fprintf(newPtr, "Error! Comment part is not completed!");

    }

    if (strconControl) {

        fprintf(newPtr, "Error! String constant part is not completed!");

    }

    fclose(fptr);

    fclose(newPtr);

}

## Algorithm: Lexical Analysis and Tokenization of Source Code

1. **File Handling**:
   * Open the input file specified by **filePath** in read mode (**"r"**) using **fopen**. This operation creates a file pointer **fptr** to access the contents of the source file.
   * Utilize the **createFile** function to create a new file for writing tokenized information. Pass **filePath** to **createFile** to ensure the new file has the appropriate name and extension. Store the file pointer returned by **createFile** in **newPtr**.
2. **Initialization**:
   * Initialize necessary variables:
     + **lineString**: A character array to hold each line read from the input file. Its size is set to 100, indicating a maximum line length of 99 characters plus the null terminator.
     + **tokenString**: Another character array used to store tokenized strings. Its size is set to 257, with one extra space to accommodate the null terminator. This array is cleared by filling it with null terminators initially.
     + **commentControl**: A flag to track whether the function is currently inside a comment block (**1** for inside comment, **0** otherwise).
     + **strconControl**: A flag to track whether the function is currently inside a string constant (**1** for inside string constant, **0** otherwise).
     + **tokenIndex**: An integer variable to keep track of the current position in the **tokenString** buffer.
3. **Read and Tokenize Lines**:
   * Use a **while** loop to read each line of the input file using **fgets**. This loop iterates until the end of the file is reached.
   * For each character in the line:
     + Check for comments, string constants, integers, and non-integer tokens based on specific conditions.
     + Tokenize the content accordingly:
       - If it's a comment or string constant, handle it appropriately.
       - If it's an integer, convert it to an integer value using the **strToInt** function and write it to the output file.
       - If it's a non-integer token (such as keywords, identifiers, operators, brackets, end-of-line markers, or commas), process it accordingly using helper functions and write it to the output file.
4. **Error Handling**:
   * After processing each line, check if there are any incomplete comment or string constant blocks.
   * If **commentControl** is still active at the end of file processing, it indicates an incomplete comment block. Write an error message to the output file indicating the issue.
   * Similarly, if **strconControl** is still active, it indicates an incomplete string constant. Write an error message to the output file accordingly.
5. **File Closure**:
   * Once the processing of the input file is complete, close both the input file pointer (**fptr**) and the output file pointer (**newPtr**) using the **fclose** function.

**Summary:**

The **fileReading** function is responsible for thoroughly analyzing the contents of the input source file, identifying different types of tokens, and writing the tokenized information to a new file. It meticulously handles various scenarios such as comments, string constants, integers, and non-integer tokens, ensuring accurate tokenization. Additionally, it includes error handling mechanisms to detect and report incomplete comment or string constant blocks. Finally, it concludes by closing both the input and output files, completing the lexical analysis process.

# Method creatingFile;

FILE \*createFile(char \*filePath) {

    // Creates a file with correct extension.

    // Takes the file name and combine it with file extension ".lex".

    int len = strlen(filePath);

    char newPath[len + 1];

    int i = 0;

    if (!strstr(filePath, ".sta")) {

        printf("Only acceptable for .sta file extension type!");

        return NULL;

    }

    while (filePath[i] != '.') {

        newPath[i] = filePath[i];

        i++;

    }

    newPath[i] = '\0';

    char sendPath[len + 1];

    sprintf(sendPath, "%s.lex", newPath);

    FILE \*ptr;

    ptr = fopen(sendPath, "w");

    return ptr;

}

## Code Explanation;

* **Purpose**: This function creates a new file with the correct extension and returns a file pointer to the newly created file.
* **Parameters**:
  + **filePath**: A pointer to a string containing the path to the original input file.
* **Return Value**:
  + **FILE\***: A file pointer to the newly created file. If the file creation fails, it returns **NULL**.
* **Detailed Explanation**:
  + **File Extension Verification**:
    - The method first checks if the input **filePath** contains the ".sta" file extension. It does so by using **strstr** function to search for the substring ".sta" within the **filePath**.
    - If the ".sta" extension is not found, it prints an error message indicating that the input file extension is not acceptable for the expected file type (".sta"). Then it returns **NULL** to signify the failure in file creation.
  + **Extracting File Name**:
    - If the file extension is verified, the method proceeds to extract the file name without the extension. It does so by copying characters from **filePath** to a new string **newPath** until it encounters the '.' character, indicating the beginning of the file extension. It then terminates **newPath** with a null character to ensure proper termination.
  + **Creating New File Path**:
    - After extracting the file name, the method concatenates the ".lex" extension to the file name to form the new file path. This new path is stored in the **sendPath** array using **sprintf**.
  + **File Creation**:
    - Using the new file path (**sendPath**), the method attempts to create a new file for writing using **fopen** with mode "w" (write mode). If successful, it returns a file pointer (**ptr**) to the newly created file.
    - If the file creation fails for any reason, such as insufficient permissions or disk space, **fopen** will return **NULL**, indicating the failure.
* **Summary**:
  + **createFile** is a method responsible for creating a new file with the correct extension (".lex") based on the provided input file path.
  + It verifies that the input file has the expected ".sta" extension and extracts the file name without the extension.
  + Then it concatenates the ".lex" extension to the file name to form the new file path.
  + Finally, it attempts to create the new file for writing and returns a file pointer to the newly created file. If the file creation fails, it returns **NULL**.

# Main Method;

int main() {

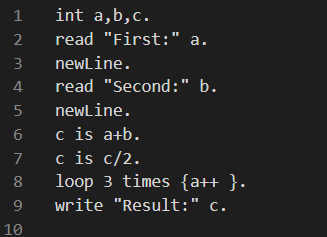
    fileReading("aa.sta"); //aa.sta is the file we are going to use.

    return 0;

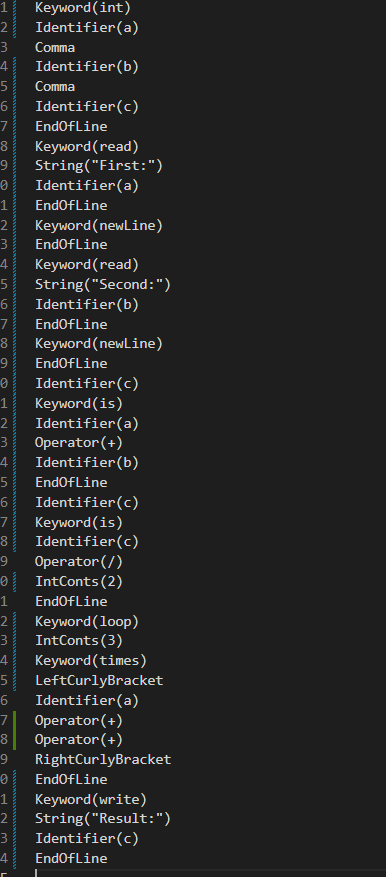
}

# Test Cases;

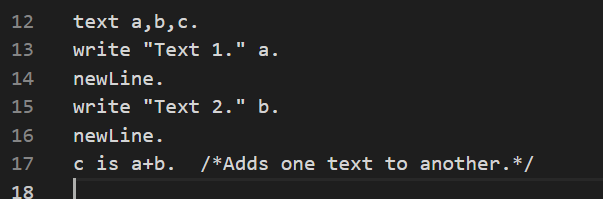
## Case 1 Input File:



### Output File;

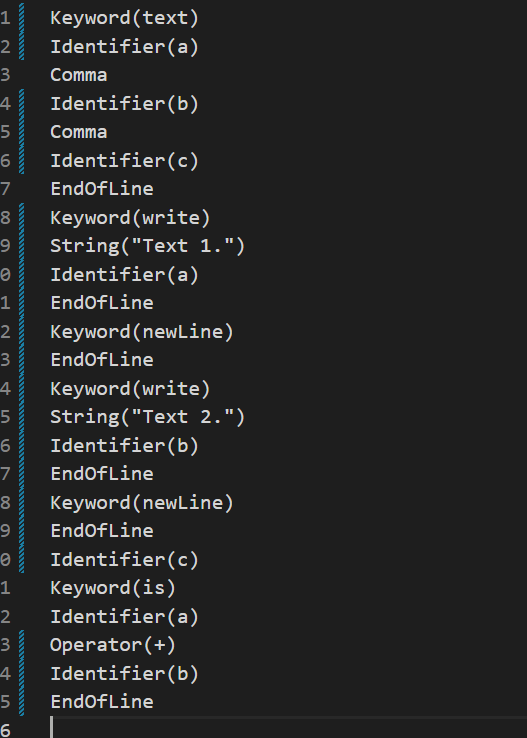


## Case 2 Input File;

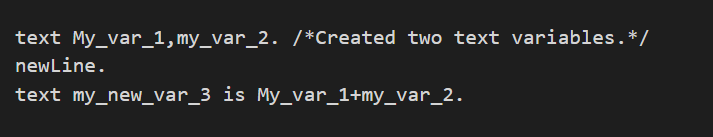


### Output File;

We see that comment line did not be tokenize.

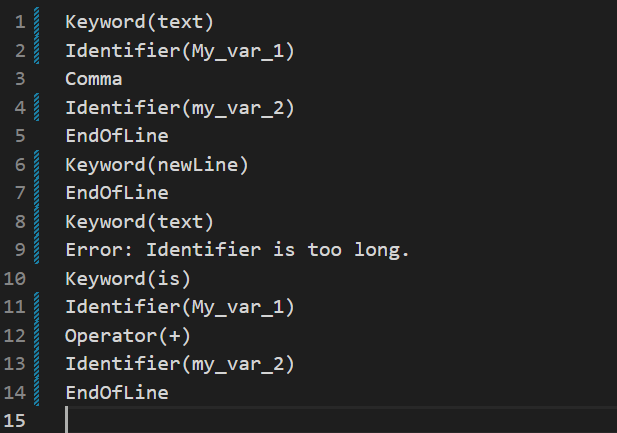


## Case 3 Input File;



### Output File;

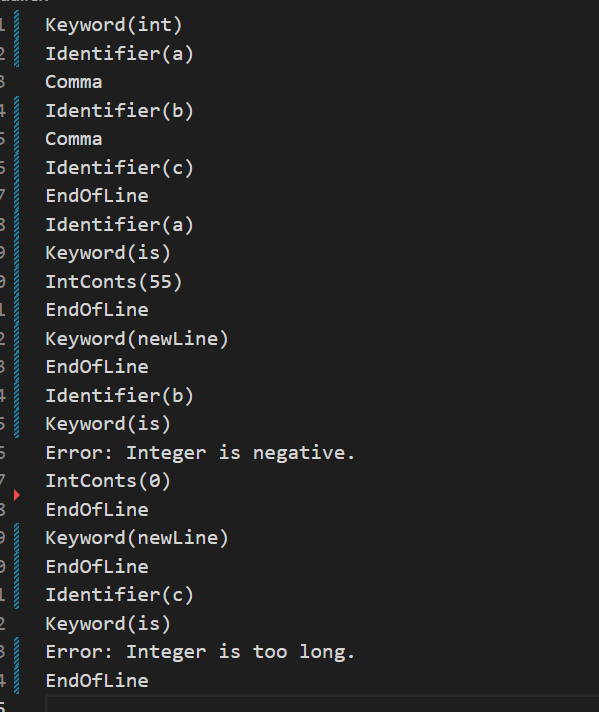
We see the error that says identifier lenght is exceeded.



## Case 4 Input File;

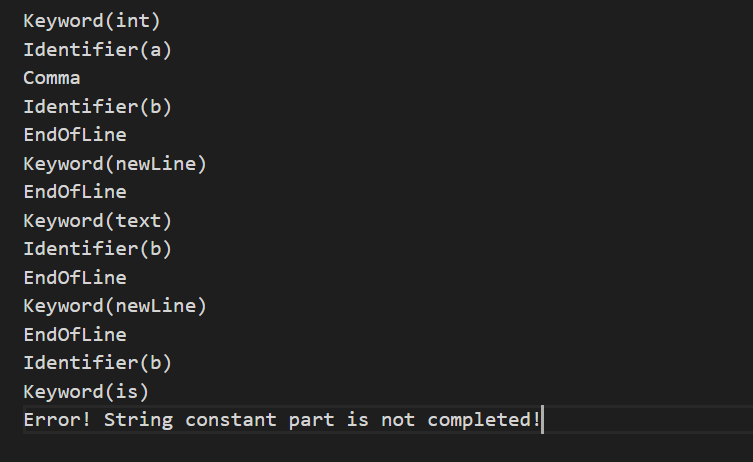
### Output File;

We see that Integer exceeding 8 digit gives an error.



## Case 5 Input File;

### Output File;

We see that unfinished string part gives an error.

## Case 6 Input File;

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

### Output File;

We see that unfinished comment part gives an error.