STAR LANGUAGE INTERPRETER PROJECT

sarı, emotikon, gülenyüz, yaratıcılık içeren bir resim

Açıklama otomatik olarak oluşturuldu

**SAMET KILIÇ, 05230000202**

**MUHAMMED SAİD GOLOTOĞLU, 0521000195**

**MUSTAFA HAKAN GEZGİNCİ, 05230000196**

**FERDİ CAN, 05220000957**

**Contents**

[Project Purpose and Overview 4](#_Toc167023139)

[Short Important Explanation 5](#_Toc167023140)

[Data Variables 5](#_Toc167023141)

[Methods; 6](#_Toc167023142)

[lexLinesCreate; 6](#_Toc167023143)

[isSameInt; 7](#_Toc167023144)

[isSameTxt; 8](#_Toc167023145)

[toIdent; 9](#_Toc167023146)

[endIndex; 10](#_Toc167023147)

[checkIntIndex; 11](#_Toc167023148)

[checkStrIndex; 12](#_Toc167023149)

[checkLineIndex; 13](#_Toc167023150)

[toInt; 14](#_Toc167023151)

[toStr; 15](#_Toc167023152)

[substractTxt; 16](#_Toc167023153)

[txtOperator 17](#_Toc167023154)

[intOperator; 19](#_Toc167023155)

[txtOrInt; 21](#_Toc167023156)

[oneRev; 22](#_Toc167023157)

[numberEnd; 23](#_Toc167023158)

[findRightCurly; 24](#_Toc167023159)

[curlyLoop; 25](#_Toc167023160)

[normalLoop; 27](#_Toc167023161)

[loopControl; 28](#_Toc167023162)

[ifNonPrompt 29](#_Toc167023163)

[ifString; 31](#_Toc167023164)

[readControl; 33](#_Toc167023165)

[writeControl; 34](#_Toc167023166)

[isSpecialLine; 36](#_Toc167023167)

[typeControl; 37](#_Toc167023168)

[bothInt; 38](#_Toc167023169)

[intControl; 38](#_Toc167023170)

[bothStr; 40](#_Toc167023171)

[strControl; 41](#_Toc167023172)

[interpreter; 43](#_Toc167023173)

[main; 44](#_Toc167023174)

[*Test Cases* 44](#_Toc167023175)

[Defining Variables and Exceeding Maximum Int Value; 44](#_Toc167023176)

[Maximum Text Limit; 45](#_Toc167023177)

[String Operations and Lines of Code; 46](#_Toc167023178)

[Loops; 47](#_Toc167023179)

[Read Operations; 48](#_Toc167023180)

# Project Purpose and Overview

**Project Title:** Interpreter for the STAR Programming Language

**Objective:** The primary objective of this project is to develop an interpreter for the STAR programming language using the C programming language. First, lexical analyzer which we did in previous Project reads “.sta” extended file and is forming “.lex” extended file, then interpreter interpretes star codes from lexical file. The interpreter will handle all aspects of STAR language execution, including variable declarations, arithmetic operations, control structures, and input/output operations.

**Background:** STAR is a small, simple scripting language designed for basic arithmetic operations. It supports two data types: integers and strings (referred to as "text"). Integers in STAR are non-negative and limited to a maximum value of 99,999,999. Strings are limited to 256 characters. The language has a straightforward syntax and semantics, making it suitable for demonstrating the fundamental concepts of language interpretation.

**Functionality:** The interpreter will execute STAR source code by parsing and interpreting each line of code according to the syntax and semantics of the language.

# Short Important Explanation

Details of the lexical analysis part are not given in this project because it was covered in the previous project. This project will focus only on the interpreting part of the lexical analysis part.

# Data Variables

char keyWords[][10] = {"int", "text", "is", "loop", "times", "read", "write", "newLine"};

char operators[][1] = {"+", "-", "\*", "/"};

char specialChars[] = {'+', '-', '\*', '/', '.', '\n', ',', '{', '}'};

char intIdentifiers[10][100];

int lastInt; // last empty integer identifier index.

int intVar[10]; // integer values.

char strIdentifiers[10][270];

int lastStr; // last empty text identifier index.

char strNames[10][100]; //text names.

int leftBraces[20];//for right left curly operations.

int leftIndex=0;

int rightBraces[20]; //for right left curly operations.

int rightIndex=0;

char lexLines[150][300]; // holds lexical analyzer outputs.

int currentIndex; // holds the lex lines index where interpreter operates.

# Methods;

## lexLinesCreate;

void lexLinesCreate(char \*filePath) { // takes the strings in aa.lex and writes every elements into lexLines strings.

    FILE \*fptr;

    fptr = fopen(filePath, "r");

    char lineString[300];

    int i = 0;

    while (fgets(lineString, 300, fptr)) {

        strcpy(lexLines[i], lineString);

        i++;

    }

}

* **Purpose:** This function reads strings from a file specified by **filePath** and writes each string into the **lexLines** array.
* **Parameters:**
  + **filePath**: Pointer to the path of the file to be read.
* **Algorithm:**
  + Open the file specified by **filePath** for reading using **fopen**.
  + Initialize an array **lineString** to hold each line read from the file.
  + Initialize a counter **i** to keep track of the index in the **lexLines** array.
  + Use a **while** loop to read each line from the file using **fgets** until the end of the file is reached.
  + Copy each line read from the file into the **lineString** array using **fgets**.
  + Copy the content of **lineString** into the **lexLines** array at index **i** using **strcpy**.
  + Increment the counter **i** to move to the next index in the **lexLines** array.
  + Repeat steps 4-7 until all lines in the file have been read and copied into the **lexLines** array.
  + Close the file using **fclose**.

## isSameInt;

int isSameInt(char \*str) {  // checks if the int value in the parameter is the same as in intIdentifiers array.

    int i = 0;

    while (intIdentifiers[i][0] != '\0') {

        if (strcmp(str, intIdentifiers[i]) == 0) {

            return 1;

        }

        i++;

    }

    return 0;

}

* **Purpose:** This function checks if the integer value represented by the string **str** is the same as any integer value stored in the **intIdentifiers** array.
* **Parameters:**
  + **str**: Pointer to the string representing the integer value to be checked.
* **Return Value:**
  + **1** if the integer value represented by **str** is found in the **intIdentifiers** array.
  + **0** if the integer value represented by **str** is not found in the **intIdentifiers** array.
* **Algorithm:**
  + Initialize a counter **i** to traverse the **intIdentifiers** array.
  + Use a **while** loop to iterate through the **intIdentifiers** array until an empty string (**'\0'**) is encountered.
  + Inside the loop, use **strcmp** to compare the string **str** with each string in the **intIdentifiers** array.
  + If a match is found, return **1** to indicate that the integer value represented by **str** is found in the **intIdentifiers** array.
  + If no match is found after iterating through the entire **intIdentifiers** array, return **0** to indicate that the integer value represented by **str** is not found in the **intIdentifiers** array.

## isSameTxt;

int isSameTxt(char \*str) { // checks if the string value in the parameter is the same as in strIdentifiers array.

    int i = 0;

    while (strIdentifiers[i][0] != '\0') {

        if (strcmp(str, strIdentifiers[i]) == 0) {

            return 1;

        }

        i++;

    }

    return 0;

}

* **Purpose:** This function checks if the string value represented by the string **str** is the same as any string value stored in the **strIdentifiers** array.
* **Parameters:**
  + **str**: Pointer to the string representing the value to be checked.
* **Return Value:**
  + **1** if the string value represented by **str** is found in the **strIdentifiers** array.
  + **0** if the string value represented by **str** is not found in the **strIdentifiers** array.
* **Algorithm:**
  + Initialize a counter **i** to traverse the **strIdentifiers** array.
  + Use a **while** loop to iterate through the **strIdentifiers** array until an empty string (**'\0'**) is encountered.
  + Inside the loop, use **strcmp** to compare the string **str** with each string in the **strIdentifiers** array.
  + If a match is found, return **1** to indicate that the string value represented by **str** is found in the **strIdentifiers** array.
  + If no match is found after iterating through the entire **strIdentifiers** array, return **0** to indicate that the string value represented by **str** is not found in the **strIdentifiers** array.

## toIdent;

void toIdent(char \*str, char \*copied) { // takes the Identifier token from lexical file and takes just identifier name in that token.

    if (strstr(copied, "Identifier")) {

        int i = 11;

        int s = 0;

        while (copied[i] != ')') {

            str[s] = copied[i];

            i++;

            s++;

        }

        str[s] = '\0';

    }

}

**Purpose:** Extracts the identifier name from an "Identifier" token in a lexical file.

**Parameters:**

* **str**: Pointer to a character array where the extracted identifier name will be stored.
* **copied**: Pointer to the "Identifier" token from which the identifier name will be extracted.

**Algorithm:**

1. Search for the substring "Identifier" within the string **copied**.
2. If "Identifier" is found:
   * Initialize index **i** to 11 (position after "Identifier(") and **s** to 0.
   * Iterate through **copied** starting from index **i** until ')' is encountered.
   * Copy each character from **copied** to **str**, incrementing both **i** and **s**.
   * Terminate **str** with a null character once ')' is encountered.

This function effectively isolates the identifier name from the "Identifier" token and stores it in the provided character array **str**.

## endIndex;

int endIndex() { // returns index involving of EndOfLine.

    int temp = currentIndex;

    while (lexLines[temp][0] != '\0') {

        if (strcmp(lexLines[temp], "EndOfLine\n") == 0) { // lexLines[temp] == "EndOfLine"

            break;

        }

        temp++;

    }

    return temp;

}

**Purpose:** Returns the index of the first occurrence of "EndOfLine" within the **lexLines** array, starting from the current index **currentIndex**.

**Algorithm:**

1. Set a temporary variable **temp** to the current index **currentIndex**.
2. While the string at index **temp** in the **lexLines** array is not empty (i.e., not '\0'):
   * Check if the string at index **temp** is equal to "EndOfLine\n" using **strcmp**.
   * If it is, exit the loop.
   * Otherwise, increment **temp** to move to the next index in the array.
3. Return the index **temp**, which represents the index involving the first occurrence of "EndOfLine".

This function efficiently searches for the "EndOfLine" string within the **lexLines** array and returns the corresponding index.

## checkIntIndex;

int checkIntIndex(char \*str) { // controls if given parameter has already exists in int Identifiers.

    if (strstr(str, "Identifier") == 0) {

        return -1;

    }

    char ident[100];

    toIdent(ident, str);

    int i = 0;

    while (i < 10) {

        if (strcmp(intIdentifiers[i], ident) == 0) { // intIdentifiers[i] == str

            return i;

        }

        i++;

    }

    return -1;

}

**Purpose:** Checks if the given parameter representing an integer identifier already exists in the **intIdentifiers** array.

**Parameters:**

* **str**: Pointer to the string containing the identifier to be checked.

**Return Value:**

* Returns the index of the identifier in the **intIdentifiers** array if found; otherwise, returns -1.

**Algorithm:**

1. Check if the string **str** contains the substring "Identifier" using **strstr**.
2. If "Identifier" is not found, return -1 indicating that the given parameter is not an identifier.
3. If "Identifier" is found:
   * Extract the identifier name from the token using the **toIdent** function and store it in the **ident** array.
   * Initialize a counter **i** to 0.
   * Iterate through the **intIdentifiers** array:
     + If the identifier at index **i** in **intIdentifiers** is equal to **ident**, return **i**.
     + Otherwise, increment **i** to check the next element in the array.
4. If the identifier is not found in the **intIdentifiers** array after the loop, return -1.

This function efficiently checks if a given parameter representing an integer identifier already exists in the **intIdentifiers** array and returns its index if found.

## checkStrIndex;

int checkStrIndex(char \*str) { // controls if given parameter has already exists in string Identifiers.

    if (strstr(str, "Identifier") == 0) {

        return -1;

    }

    char ident[100];

    toIdent(ident, str);

    int i = 0;

    while (i < 10) {

        if (strcmp(strIdentifiers[i], ident) == 0) { // strIdentifiers[i] == str

            return i;

        }

        i++;

    }

    return -1;

}

**Purpose:** Checks if the given parameter representing a string identifier already exists in the **strIdentifiers** array.

**Parameters:**

* **str**: Pointer to the string containing the identifier to be checked.

**Return Value:**

* Returns the index of the identifier in the **strIdentifiers** array if found; otherwise, returns -1.

**Algorithm:**

1. Check if the string **str** contains the substring "Identifier" using **strstr**.
2. If "Identifier" is not found, return -1 indicating that the given parameter is not an identifier.
3. If "Identifier" is found:
   * Extract the identifier name from the token using the **toIdent** function and store it in the **ident** array.
   * Initialize a counter **i** to 0.
   * Iterate through the **strIdentifiers** array:
     + If the identifier at index **i** in **strIdentifiers** is equal to **ident**, return **i**.
     + Otherwise, increment **i** to check the next element in the array.
4. If the identifier is not found in the **strIdentifiers** array after the loop, return -1.

This function efficiently checks if a given parameter representing a string identifier already exists in the **strIdentifiers** array and returns its index if found.

## checkLineIndex;

int checkLineIndex(char \*str) { // returns end of the statement index.

    int i = currentIndex;

    while (!strstr(lexLines[i],"EndOfLine") && lexLines[i][0]!='\0') {

        if (strcmp(lexLines[i], str) == 0) { // lexLines[i] == str

            return i;

        }

        i++;

    }

    return -1;

}

**Purpose:** Returns the index of the end of the statement in the **lexLines** array.

**Parameters:**

* **str**: Pointer to the string representing the end of the statement.

**Return Value:**

* Returns the index of the end of the statement in the **lexLines** array if found; otherwise, returns -1.

**Algorithm:**

1. Initialize a counter **i** to the current index **currentIndex**.
2. Iterate through the **lexLines** array starting from index **i**:
   * Check if the string at index **i** does not contain "EndOfLine" and is not an empty string (i.e., not '\0') using **strstr** and **lexLines[i][0]**.
   * If the conditions are met:
     + Check if the string at index **i** is equal to **str** using **strcmp**.
     + If the strings are equal, return **i** as the index of the end of the statement.
   * Increment **i** to move to the next index in the array.
3. If the end of the statement is not found in the **lexLines** array after the loop, return -1.

This function efficiently searches for the end of the statement in the **lexLines** array and returns its index if found.

## toInt;

int toInt(char \*str) { // takes the IntConst token and converts it into int value.

    int start = 0;

    int i = 9;

    char number[100];

    while (str[i] != ')') {

        number[start] = str[i];

        i++;

        start++;

    }

    number[start] = '\0';

    int returnNum = atoi(number);

    return returnNum;

}

**Purpose:** Converts the IntConst token represented as a string into an integer value.

**Parameters:**

* **str**: Pointer to the string containing the IntConst token.

**Return Value:**

* Returns the integer value parsed from the IntConst token.

**Algorithm:**

1. Initialize variables **start** and **i** to 0 and 9 respectively.
2. Declare a character array **number** to store the digits of the integer.
3. Iterate through the characters of the string starting from index 9 until ')' is encountered:
   * Store each digit character in the **number** array.
   * Increment both **i** and **start** to move to the next character and next index in the **number** array.
4. Terminate the **number** array with a null character '\0'.
5. Use the **atoi** function to convert the **number** array to an integer value and store it in **returnNum**.
6. Return the integer value **returnNum**.

This function efficiently parses the integer value from the IntConst token string by extracting the numeric characters and converting them into an integer using the **atoi** function.

## toStr;

void toStr(char \*modify, char \*arranged) { // takes the String token and converts it into string value.

    int start = 0;

    int i = 8;

    char \*string = (char \*)malloc(sizeof(char) \* 260);

    while (arranged[i] != '"') {

        if (start > 256) {

            start++;

            break;

        } else {

            string[start] = arranged[i];

            start++;

            i++;

        }

    }

    string[start] = '\0';

    strcpy(modify, string);

    free(string);

}

**Purpose:** Converts the String token represented as a string into its corresponding string value.

**Parameters:**

* **modify**: Pointer to the string where the converted string value will be stored.
* **arranged**: Pointer to the string containing the String token.

**Algorithm:**

1. Initialize variables **start** and **i** to 0 and 8 respectively.
2. Allocate memory dynamically for a character array **string** to store the characters of the string value.
3. Iterate through the characters of the **arranged** string starting from index 8 until a double quote '"' is encountered:
   * Check if the **start** index is greater than 256, indicating the maximum string length is reached. If so, exit the loop.
   * Otherwise, copy the character at index **i** of **arranged** to the **string** array at index **start**.
   * Increment both **start** and **i** to move to the next character.
4. Terminate the **string** array with a null character '\0'.
5. Copy the contents of the **string** array to the **modify** string using **strcpy**.
6. Free the dynamically allocated memory for the **string** array using **free**.

This function efficiently extracts the string value from the String token string by copying the characters between double quotes '"' into a dynamically allocated character array. It then copies the resulting string into the **modify** string and frees the memory allocated for the temporary array.

## substractTxt;

void substractTxt(char operand1[], char \*subtracted){ // subtracts subtracted from operand1.

    char \*result= strstr(operand1,subtracted);

    int subtractedSize= strlen(subtracted);

    if(result){

        char\* nextSubtracted= result+ subtractedSize;

        int subIndex = nextSubtracted - operand1;

        int length= strlen(operand1)+1 -(subIndex);

        memmove(result,nextSubtracted,length);

    }

}

**Purpose:** Subtracts the **subtracted** string from the **operand1** string.

**Parameters:**

* **operand1[]**: The string from which **subtracted** will be subtracted.
* **subtracted**: Pointer to the string that will be subtracted from **operand1**.

**Algorithm:**

1. Find the first occurrence of the **subtracted** string within the **operand1** string using **strstr** function. Store the result in the pointer variable **result**.
2. Calculate the length of the **subtracted** string using the **strlen** function and store it in the variable **subtractedSize**.
3. If **result** is not **NULL** (indicating **subtracted** is found within **operand1**):
   * Calculate the pointer to the character immediately following the **subtracted** string within **operand1** and store it in the pointer variable **nextSubtracted**.
   * Calculate the index of the character immediately following **subtracted** within **operand1** by subtracting the memory addresses and store it in the variable **subIndex**.
   * Calculate the length of the substring starting from **nextSubtracted** till the end of **operand1** and store it in the variable **length**.
   * Move the characters starting from **nextSubtracted** to the end of **operand1** to the position of **result** using **memmove** function. This effectively removes the **subtracted** string from **operand1**.

This function efficiently removes the first occurrence of the **subtracted** string from the **operand1** string by manipulating pointers and memory addresses, ensuring correct handling even in the presence of overlapping memory regions.

## txtOperator

void txtOperator(char lexLines[][300], int temp) { // operands text values.

    char operand1[270], operand2[270];

     if (strstr(lexLines[temp + 1], "Identif")) {

        int index1 = checkStrIndex(lexLines[temp + 1]);

        strcpy(operand1, strNames[index1]);

    }

    else if (strstr(lexLines[temp + 1], "String")) {

        toStr(operand1, lexLines[temp + 1]);

    }

     if (strstr(lexLines[temp + 3], "Identif")) {

        int index2 = checkStrIndex(lexLines[temp + 3]);

        strcpy(operand2, strNames[index2]);

    }

    else if (strstr(lexLines[temp + 3], "String")) {

        toStr(operand2, lexLines[temp + 3]);

    }

    char operator= lexLines[temp + 2][9];

    int control = operator== '+' || operator== '-';

    switch (operator) {

    case '+':

        strcat(operand1, operand2);

        int i = checkStrIndex(lexLines[temp - 1]);

        strcpy(strNames[i], operand1);

        break;

    case '-':

        substractTxt(operand1, operand2);

        int y = checkStrIndex(lexLines[temp - 1]);

        strcpy(strNames[y], operand1);

        break;

    }

}

**Purpose:** Perform operations on text values based on the given operators in the lexical file.

**Parameters:**

* **lexLines[][300]**: 2D array containing lexical elements.
* **temp**: Index indicating the current position in **lexLines** array.

**Algorithm:**

1. Declare character arrays **operand1** and **operand2** to hold the operands.
2. Determine the type of the first operand (**operand1**) based on the lexical element at **temp + 1** index:
   * If the lexical element contains the string "String", call the **toStr** function to convert it to a string and store it in **operand1**.
   * Otherwise, if it contains "Identif", retrieve the index of the string identifier from **strNames** array using **checkStrIndex** and copy the corresponding string value to **operand1**.
3. Determine the type of the second operand (**operand2**) in a similar manner based on the lexical element at **temp + 3** index.
4. Extract the operator from the lexical element at **temp + 2** index and store it in the variable **operator**.
5. Check if the operator is either '+' or '-'. If it is, set the control variable to true.
6. Using a switch statement, perform the operation based on the operator:
   * If the operator is '+', concatenate **operand2** to **operand1** using **strcat** and update the corresponding string identifier in **strNames** array.
   * If the operator is '-', call the **substractTxt** function to subtract **operand2** from **operand1** and update the corresponding string identifier in **strNames** array.

This function efficiently performs text operations by handling different types of operands and operators, updating the string identifiers accordingly based on the operation performed.

## intOperator;

int intOperator(char lexLines[][300], int temp) { // operands int values.

    int operand1, operand2;

    if (strstr(lexLines[temp + 1], "IntConts")) {

        operand1 = toInt(lexLines[temp + 1]);

    } else if (strstr(lexLines[temp + 1], "Identif")) {

        int index1 = checkIntIndex(lexLines[temp + 1]);

        operand1 = intVar[index1];

    }

    if (strstr(lexLines[temp + 3], "IntConts")) {

        operand2 = toInt(lexLines[temp + 3]);

    } else if (strstr(lexLines[temp + 3], "Identif")) {

        int index2 = checkIntIndex(lexLines[temp + 3]);

        operand2 = intVar[index2];

    }

    char operator= lexLines[temp + 2][9];

    int control = operator== '+' || operator== '-' || operator== '/' || operator== '\*';

    int rtn;

    if (control) {

        switch (operator) {

        case '+':

            rtn = operand1 + operand2;

            break;

        case '-':

            rtn = operand1 - operand2;

            break;

        case '\*':

            rtn = operand1 \* operand2;

            break;

        case '/':

            rtn = operand1 / operand2;

            break;

        }

        if (rtn >= 0 && rtn<=99999999) {

            return rtn;

        }else if(rtn>=99999999){

             printf("Max Integer value is 99999999! ");

             return 0;

        }

         else

            return 0;

    }

    return -1;

}

**Purpose:** Perform operations on integer values based on the given operators in the lexical file.

**Parameters:**

* **lexLines[][300]**: 2D array containing lexical elements.
* **temp**: Index indicating the current position in **lexLines** array.

**Algorithm:**

1. Declare integer variables **operand1** and **operand2** to hold the operands.
2. Determine the type of the first operand (**operand1**) based on the lexical element at index **temp + 1**:
   * If the lexical element contains "IntConst", call the **toInt** function to convert it to an integer and store it in **operand1**.
   * Otherwise, if it contains "Identif", retrieve the index of the integer identifier from **intVar** array using **checkIntIndex** and copy the corresponding integer value to **operand1**.
3. Determine the type of the second operand (**operand2**) in a similar manner based on the lexical element at index **temp + 3**.
4. Extract the operator from the lexical element at index **temp + 2** and store it in the variable **operator**.
5. Check if the operator is one of the valid arithmetic operators (**+**, **-**, **\***, **/**) and set the control variable accordingly.
6. Using a switch statement, perform the operation based on the operator:
   * If the operator is '+', add **operand1** and **operand2**.
   * If the operator is '-', subtract **operand2** from **operand1**.
   * If the operator is '\*', multiply **operand1** by **operand2**.
   * If the operator is '/', divide **operand1** by **operand2**.
7. Check if the result of the operation is within the valid integer range (**0** to **99999999**):
   * If it is, return the result.
   * If it exceeds **99999999**, print a warning message and return **0**.
   * If it is negative or out of range, return **0**.
8. If the control variable is false (meaning an invalid operator), return **-1**.

This function efficiently performs arithmetic operations on integer values, handling different types of operands and operators while ensuring the result is within the specified integer range.

## txtOrInt;

int txtOrInt(char \*lexLine) { // returns 1 for  text, 2 for int about variable type.

    int control = checkStrIndex(lexLine);

    if (control != -1)

        return 1;

    control = checkIntIndex(lexLine);

    if (control != -1)

        return 2;

}

**Purpose:** Determine the type of variable (text or integer) based on the lexical element.

**Parameters:**

* **lexLine**: Pointer to a string representing a lexical element.

**Algorithm:**

1. Call the **checkStrIndex** function to check if the lexical element corresponds to a string identifier:
   * If the return value is not **-1**, indicating that the lexical element is a string identifier, return **1** to signify a text variable.
2. If the lexical element is not a string identifier, call the **checkIntIndex** function to check if it corresponds to an integer identifier:
   * If the return value is not **-1**, indicating that the lexical element is an integer identifier, return **2** to signify an integer variable.
3. If neither condition is met (i.e., the lexical element does not correspond to any known identifier), return **-1** to indicate an unknown variable type.

This function efficiently determines the type of variable (text or integer) based on the provided lexical element, enabling further processing or handling based on the variable type.

## oneRev;

void oneRev() { // processes every statement of star code.

    int temp = currentIndex;

    int endInd = endIndex();

    int control = currentIndex == temp; //if statement is processed, control is false.

    if(control)

    loopControl();

    control = currentIndex == temp;

    if (control)

        strControl();

   control = currentIndex == temp;

    if (control)

        intControl();

    control = currentIndex == temp;

    if (control)

        readControl();

    control = currentIndex == temp;

    if (control)

        writeControl();

    control = currentIndex == temp;

}

**Purpose:** Process each statement of star code.

**Algorithm:**

1. Set the variable **temp** to the current index (**currentIndex**) and determine the index of the end of the line (**endInd**) using the **endIndex** function.
2. Set the variable **control** to **true** if the current index is equal to **temp**. This variable indicates whether a statement has been processed.
3. If **control** is **true**, execute the **loopControl** function to handle loop-related statements.
4. Update the **control** variable to check if a statement has been processed.
5. If **control** is still **true**, execute the **strControl** function to handle string-related statements.
6. Repeat the process for **intControl**, **readControl**, and **writeControl**, updating the **control** variable after each function call.
7. Each function is executed only if the previous statements have not been processed (**control** is **true**), ensuring that each statement is processed sequentially and independently of the others.

This function sequentially processes each statement of the star code, ensuring that each statement is handled appropriately based on its type, and moves on to the next statement if the current statement has been processed.

## numberEnd;

int numberEnd(int endCounterIndex) { // finds statement number in loops having curly brackets.

    int endControl = endCounterIndex;

    int i = 0;

    while (!strstr(lexLines[endControl], "RightCurlyBracket")|| leftBraces[0]!=0) {

        char \*lexLine= lexLines[endControl];

        if(strstr(lexLine,"LeftCurlyBracket")){

            leftBraces[leftIndex++]=1;

        }

        else if(strstr(lexLine,"RightCurlyBracket")){

            leftBraces[--leftIndex]=0;

            i++;

        }

        else if (strstr(lexLine, "EndOfLine")) {

            i++;

        }

        endControl++;

    }

    return i;

}

**Purpose:** Finds the statement number in loops with curly brackets.

**Parameters:**

* **endCounterIndex**: The index indicating the position in the code where the loop ends.

**Return Value:** The number of statements within the loop.

**Algorithm:**

1. Initialize variables **endControl** to **endCounterIndex**, indicating the end of the loop, and **i** to **0** to count the number of statements within the loop.
2. Enter a loop that continues until encountering a **RightCurlyBracket** token and there are no unclosed left curly brackets (**LeftCurlyBracket**).
3. Within the loop:
   * Check if the current line contains a **LeftCurlyBracket** token. If so, increment the count of unclosed left braces (**leftBraces**) and proceed.
   * Check if the current line contains a **RightCurlyBracket** token. If so, decrement the count of unclosed left braces and increment the statement count **i**.
   * Check if the current line contains an **EndOfLine** token. If so, increment the statement count **i**.
   * Move to the next line (**endControl++**).
4. Return the total count of statements **i**, indicating the number of statements within the loop, excluding the curly brackets.

This function iterates through the lines starting from the end of the loop until it finds the corresponding closing curly bracket. Along the way, it keeps track of the number of statements encountered, excluding the curly brackets, by incrementing the counter **i**.

## findRightCurly;

int findRightCurly(int endCounterIndex) { // finds rightcurlybracket index of leftcurlybracket in lexLines.

    while (!strstr(lexLines[endCounterIndex], "RightCurlyBracket")|| rightBraces[0]!=0) {

        char \*lexLine= lexLines[endCounterIndex];

        if(strstr(lexLine,"LeftCurlyBracket")){

            rightBraces[rightIndex++]=1;

        }

        else if(strstr(lexLine,"RightCurlyBracket")){

            rightBraces[--rightIndex]=0;

        }

        endCounterIndex++;

    }

    return endCounterIndex;

}

**Purpose:** Finds the index of the right curly bracket corresponding to a left curly bracket in **lexLines**.

**Parameters:**

* **endCounterIndex**: The index indicating the position in **lexLines** where the left curly bracket is located.

**Return Value:** The index of the right curly bracket corresponding to the left curly bracket.

**Algorithm:**

1. Enter a loop that continues until encountering a **RightCurlyBracket** token and there are no unclosed right curly brackets (**RightCurlyBracket**) or unclosed left curly brackets (**LeftCurlyBracket**).
2. Within the loop:
   * Check if the current line contains a **LeftCurlyBracket** token. If so, increment the count of unclosed right braces (**rightBraces**) and proceed.
   * Check if the current line contains a **RightCurlyBracket** token. If so, decrement the count of unclosed right braces.
   * Move to the next line (**endCounterIndex++**).
3. Return the **endCounterIndex**, which indicates the index of the right curly bracket corresponding to the left curly bracket.

This function iterates through **lexLines** starting from the position of the left curly bracket until it finds the corresponding right curly bracket. Along the way, it keeps track of the count of unclosed right braces to accurately identify the corresponding right curly bracket.

## curlyLoop;

void curlyLoop() { // operands loop with curly brackets.

    int loopValue;

    char \*result1 = strstr(lexLines[currentIndex + 1], "IntConts");

    if (result1) {

        loopValue = toInt(lexLines[currentIndex + 1]);

    }

    char \*result2 = strstr(lexLines[currentIndex + 3], "LeftCurlyBracket");

    int endCounterIndex = currentIndex + 4;

    int rightCurlyBracket = findRightCurly(endCounterIndex);

    int endNumber = numberEnd(endCounterIndex);

    if (result2) {

        currentIndex = endCounterIndex;

        int j = 0;

        while (j < loopValue) {

            int i = 0;

            while (i < endNumber) {

                oneRev();

                i++;

            }

            currentIndex = endCounterIndex;

            j++;

        }

        currentIndex = rightCurlyBracket + 1;

    }

}

**Purpose:** Executes a loop with curly brackets.

**Algorithm:**

1. Search for the integer constant (**IntConts**) operand immediately following the loop keyword.
2. If found:
   * Convert the integer token to its corresponding integer value using **toInt()**.
3. Search for the left curly bracket (**LeftCurlyBracket**) immediately following the loop expression.
4. Determine the index of the right curly bracket corresponding to the left curly bracket using **findRightCurly()**.
5. Determine the number of statements within the loop body using **numberEnd()**.
6. If a left curly bracket is found:
   * Set the current index to the position immediately after the left curly bracket (**endCounterIndex**).
   * Enter a loop that iterates **loopValue** times:
     + Enter another loop that iterates **endNumber** times:
       - Execute one revolution of the star code using **oneRev()**.
     + After completing one iteration of the inner loop, reset the current index to **endCounterIndex**.
   * After completing all iterations of the outer loop, set the current index to the position immediately after the right curly bracket (**rightCurlyBracket + 1**).

This function executes a loop with curly brackets by repeating the statements within the loop body a specified number of times. It iterates through the statements within the loop body, executing each statement in sequence.

## normalLoop;

void normalLoop() { // operands loops without curly brackets.

    int loopValue;

    char \*result1 = strstr(lexLines[currentIndex + 1], "IntConts");

    if (result1) {

        loopValue = toInt(lexLines[currentIndex + 1]);

    }

    char \*result2 = strstr(lexLines[currentIndex + 2], "Keyword(times)");

    int isLoop = result1 && result2;

    if (isLoop) {

        int i = 0;

        currentIndex = currentIndex + 3;

        int temp = currentIndex;

        while (i < loopValue) {

            oneRev();

            currentIndex = temp;

            i++;

        }

        int endInd = endIndex();

        currentIndex = endInd + 1;

    }

}

**Purpose:** Executes loops without curly brackets.

**Algorithm:**

1. Search for the integer constant (**IntConts**) operand immediately following the loop keyword.
2. If found:
   * Convert the integer token to its corresponding integer value using **toInt()**.
3. Search for the keyword "times" (**Keyword(times)**) immediately following the loop expression.
4. Determine if both the integer constant and the "times" keyword are found.
5. If both conditions are met:
   * Set up a loop to iterate **loopValue** times.
   * Increment the current index to the position immediately after the loop expression.
   * Store the current index in a temporary variable **temp**.
   * Enter a loop that iterates **loopValue** times:
     + Execute one revolution of the star code using **oneRev()**.
     + After completing each iteration, reset the current index to **temp**.
   * Determine the index of the end of the line using **endIndex()**.
   * Set the current index to the position immediately after the end of the line.

This function executes loops without curly brackets by repeating the statements within the loop body a specified number of times. It iterates through the statements within the loop body, executing each statement in sequence.

## loopControl;

void loopControl() { // operands loops.

    int endInd = endIndex();

    int temp = currentIndex;

    if (lexLines[temp][0] == '\0') {

        exit(0);

    }

    if (strstr(lexLines[currentIndex], "Keyword(loop)")) {

    char \*result2 = strstr(lexLines[currentIndex + 3], "LeftCurlyBracket");

       if(result2){

        curlyLoop();

       }

       else

        normalLoop();

    }

}

**Purpose:** Controls the execution of loop statements.

**Algorithm:**

1. Determine the index of the end of the current line using **endIndex()** and store it in **endInd**.
2. Store the current index in a temporary variable **temp**.
3. Check if the current line is empty (indicated by the null character **\0** at the beginning). If it is, exit the program.
4. Check if the current line contains the keyword "loop" (**Keyword(loop)**).
5. If the keyword "loop" is found:
   * Search for the left curly bracket immediately following the loop expression (**LeftCurlyBracket**).
   * If a left curly bracket is found:
     + Execute a loop with curly brackets using **curlyLoop()**.
   * If no left curly bracket is found:
     + Execute a normal loop without curly brackets using **normalLoop()**.

This function controls the execution of loop statements, determining whether to execute loops with curly brackets or loops without curly brackets based on the structure of the code. If the code contains a "loop" keyword followed by a left curly bracket, it executes a loop with curly brackets. Otherwise, it executes a normal loop without curly brackets.

## ifNonPrompt

void ifNonPrompt(int temp) { // operand statements with "read" keywords without strings.

                             //read  varName.

    int control = txtOrInt(lexLines[temp + 1]);

    int index;

    int val;

    char strval[270];

    if (control == 1) {

        scanf("%s", &strval);

        index = checkStrIndex(lexLines[temp + 1]);

        strcpy(strNames[index], strval);

    } else if (control == 2) {

        index = checkIntIndex(lexLines[temp + 1]);

        if (scanf("%d", &val)) {

            if (val < 0) {

                val = 0;

            }

            else if(val>99999999){

                printf("Int value cannot be larger than 99999999!");

                val = 0;

            }

            intVar[index] = val;

        } else {

            index = checkIntIndex(lexLines[temp + 1]);

            intVar[index] = 0;

        }

    }

}

**Purpose:** Processes operand statements with "read" keywords that do not involve strings.

**Parameters:**

* **temp**: The index representing the current line in **lexLines**.

**Algorithm:**

1. Determine the type of operand (text or integer) using **txtOrInt()** and store the result in **control**.
2. Declare variables **index** and **val** for storing the index in the respective arrays and the value to be read.
3. If the operand is of type text (control equals 1):
   * Use **scanf()** to read a string from the standard input (**stdin**) and store it in **strval**.
   * Get the index of the operand in the **strIdentifiers** array using **checkStrIndex()** and store it in **index**.
   * Copy the read string **strval** into the **strNames** array at the corresponding index.
4. If the operand is of type integer (control equals 2):
   * Get the index of the operand in the **intIdentifiers** array using **checkIntIndex()** and store it in **index**.
   * Use **scanf()** to read an integer from the standard input (**stdin**) and store it in **val**.
   * If the read integer **val** is less than 0, set it to 0.
   * If the read integer **val** is greater than 99999999, print a message indicating that the integer value cannot exceed this limit, set **val** to 0.
   * Store the integer value **val** in the **intVar** array at the corresponding index.
5. If **scanf()** fails to read an integer, set the value in the **intVar** array at the corresponding index to 0.

This function processes operand statements with "read" keywords that do not involve strings, reading input from the user and storing it in the appropriate array based on the type of operand. If the operand is text, it reads a string and stores it in **strNames**, and if the operand is an integer, it reads an integer and stores it in **intVar**.

## ifString;

void ifString(int temp) { // operand statements with "read" keywords with strings.

                          //read "prompt:", varName.

    if (strstr(lexLines[temp + 2], "Comma")) {

        char prompt[270];

        int isIdent = checkStrIndex(lexLines[temp + 1]);

        if (strstr(lexLines[temp + 1], "String")) {

            toStr(prompt, lexLines[temp + 1]);

        } else if (isIdent != -1) {

            strcpy(prompt, strNames[isIdent]);

        }

        int control = txtOrInt(lexLines[temp + 3]);

        if (control == 2) {

            int val;

            int intIndex = checkIntIndex(lexLines[temp + 3]);

            printf("%s", prompt);

            if (scanf("%d", &val)) {

                if (val < 0) {

                    val = 0;

                }

             else if(val>99999999){

                printf("Int value cannot be larger than 99999999!");

                val = 0;

            }

                intVar[intIndex] = val;

            } else {

                printf("You did not enter a numeric value!");

                intVar[intIndex] = 0;

            }

        }

        if (control == 1) {

            char str[270];

            int txtIndex = checkStrIndex(lexLines[temp + 3]);

            printf("%s", prompt);

            scanf("%s", str);

            strcpy(strNames[txtIndex], str);

        }

    }

}

**Purpose:** Processes operand statements with "read" keywords that involve strings.

**Parameters:**

* **temp**: The index representing the current line in **lexLines**.

**Algorithm:**

1. Check if the operand statement contains a comma (**,**) indicating it involves a string prompt.
2. If a comma is found:
   * Declare a character array **prompt** to store the string prompt.
   * Check if the operand is an identifier using **checkStrIndex()** and store the result in **isIdent**.
   * If the operand is of type string (contains "String"):
     + Extract the string prompt using **toStr()** and store it in **prompt**.
   * If the operand is an identifier:
     + Copy the string stored at the corresponding index in **strNames** into **prompt**.
3. Determine the type of operand after the comma (text or integer) using **txtOrInt()** and store the result in **control**.
4. If the operand after the comma is an integer (control equals 2):
   * Declare an integer variable **val** to store the value read.
   * Get the index of the operand in the **intIdentifiers** array using **checkIntIndex()** and store it in **intIndex**.
   * Print the prompt string using **printf()**.
   * Use **scanf()** to read an integer from the standard input (**stdin**) and store it in **val**.
   * If the read integer **val** is less than 0, set it to 0.
   * If the read integer **val** is greater than 99999999, print a message indicating that the integer value cannot exceed this limit, set **val** to 0.
   * Store the integer value **val** in the **intVar** array at the corresponding index.
   * If **scanf()** fails to read an integer, print a message indicating the failure and set the value in the **intVar** array at the corresponding index to 0.
5. If the operand after the comma is a string (control equals 1):
   * Declare a character array **str** to store the string value.
   * Get the index of the operand in the **strIdentifiers** array using **checkStrIndex()** and store it in **txtIndex**.
   * Print the prompt string using **printf()**.
   * Use **scanf()** to read a string from the standard input (**stdin**) and store it in **str**.
   * Copy the read string **str** into the **strNames** array at the corresponding index.

This function processes operand statements with "read" keywords that involve strings, printing a prompt message if provided and reading input from the user, then storing it in the appropriate array based on the type of operand. If the operand is an integer, it reads an integer and stores it in **intVar**, and if the operand is text, it reads a string and stores it in **strNames**.

## readControl;

void readControl() { // operands read statements.

    int endInd = endIndex();

    int temp = currentIndex;

    if (lexLines[temp][0] == '\0') {

        exit(0);

    }

    if (strstr(lexLines[currentIndex], "Keyword(read)")) {

        if (strstr(lexLines[temp + 2], "EndOfLine")) {

            ifNonPrompt(temp);

        } else if (strstr(lexLines[temp + 4], "EndOfLine")) {

            ifString(temp);

        }

        currentIndex = endInd + 1;

    }

}

**Purpose:** Processes operand statements with "read" keywords.

**Algorithm:**

1. Get the index of the end of the statement using **endIndex()** and store it in **endInd**.
2. Store the current index in **temp**.
3. If the current line in **lexLines** is empty, exit the program.
4. Check if the current line contains the keyword "read" using **strstr()**.
5. If the line contains "read":
   * Check if the operand after "read" is an end of line (**EndOfLine**) using **strstr()**:
     + If true, call **ifNonPrompt(temp)** to process the operand without a string prompt.
   * Otherwise, check if the operand after "read" contains a comma indicating it involves a string prompt:
     + If true, call **ifString(temp)** to process the operand with a string prompt.
6. Move the current index to the end of the processed statement (**endInd + 1**).

This function processes operand statements with "read" keywords. If the operand is followed by an end of line (**EndOfLine**), it processes it without a string prompt using **ifNonPrompt(temp)**. If the operand contains a comma and is followed by a string (**String**), it processes it with a string prompt using **ifString(temp)**. Finally, it moves the current index to the end of the processed statement.

## writeControl;

void writeControl() { // operands write statements.

    int endInd = endIndex();

    int temp = currentIndex;

    if (lexLines[temp][0] == '\0') {

        exit(0);

    }

    if (strstr(lexLines[currentIndex], "Keyword(write)")) {

        while (strcmp(lexLines[temp], "EndOfLine\n")) {

            if (strstr(lexLines[temp], "Identifier")) {

                int checkStr = checkStrIndex(lexLines[temp]);

                char \*writeStr = strNames[checkStr];

                if (checkStr != -1) {

                    printf("%s", writeStr);

                }

                int checkInt = checkIntIndex(lexLines[temp]);

                if (checkInt != -1) {

                    int intVal = intVar[checkInt];

                    printf("%d", intVal);

                }

            }

            else if (strstr(lexLines[temp], "String")) {

                char \*wrtStr = (char \*)malloc(sizeof(char) \* 300);

                toStr(wrtStr, lexLines[temp]);

                printf("%s", wrtStr);

            } else if (strstr(lexLines[temp], "IntConts")) {

                int val = toInt(lexLines[temp]);

                printf("%d", val);

            }

            else if (strcmp(lexLines[temp], "Comma\n") == 0) { // lexLines[temp] == "Comma"

                temp++;

                continue;

            }

            temp++;

        }

        currentIndex = endInd + 1;

    } else if (strstr(lexLines[currentIndex], "newLine")) {

        printf("\n");

        currentIndex = endInd + 1;

    }

}

* **Purpose:** To process and output the content of write statements.
* **Parameters:** None.
* **Algorithm:**
  1. Set endInd to the index of the end of the line.
  2. Set temp to currentIndex to track the current position in lexLines.
  3. Check if the current line is empty. If so, exit the program.
  4. Check if the current line contains the keyword "write".
  5. Iterate through each line until reaching the end of the line:
     + If the line contains an identifier, print the corresponding string or integer value.
     + If the line contains a string, extract and print it.
     + If the line contains an integer constant, convert and print it.
     + If the line is a comma, skip to the next line.
  6. If the current line contains the keyword "newLine", print a new line character.
  7. Update currentIndex to point to the next line after the end of the current line.

## isSpecialLine;

int isSpecialLine(const char \*line) {

    return strcmp(line, "EndOfLine\n") && strcmp(line, "RightCurlyBracket\n");

}

**Purpose:** Checks if a given line is a special line.

**Parameters:**

* **line**: Pointer to a character array representing the line to be checked.

**Return Value:**

* **1** if the line is not "EndOfLine" or "RightCurlyBracket".
* **0** if the line is "EndOfLine" or "RightCurlyBracket".

**Algorithm:**

1. Use **strcmp()** to compare the given line with "EndOfLine\n". If they are not equal, the result is non-zero, indicating that it's not an "EndOfLine".
2. Use **strcmp()** again to compare the given line with "RightCurlyBracket\n". If they are not equal, the result is non-zero, indicating that it's not a "RightCurlyBracket".
3. Return the logical AND of the results from step 1 and step 2.

This function checks if a given line is special, meaning it's not an "EndOfLine" or a "RightCurlyBracket". It returns **1** if the line is special and **0** otherwise.

## typeControl;

int typeControl() { // 1 for intControl, 2 for StringControl.

    int temp = currentIndex;

    int isContinue1=strcmp(lexLines[temp], "EndOfLine\n");

    int isContinue2=strcmp(lexLines[temp], "RightCurlyBracket\n");

    while (isSpecialLine(lexLines[temp])) { // lexLines[temp] != "EndOfLine"

        if (strstr(lexLines[temp], "Keyword(int)")) {

            return 1;

        } else if (strstr(lexLines[temp], "Keyword(text)")) {

            return 2;

        }

        temp++;

    }

}

**Purpose:** Determines the type of control based on the current lexical line.

**Return Value:**

* **1** if the control is for integer.
* **2** if the control is for string.

**Algorithm:**

1. Initialize a variable **temp** to **currentIndex**.
2. Check if the current line is not "EndOfLine" or "RightCurlyBracket" by calling **isSpecialLine(lexLines[temp])**.
3. Inside the loop, check if the current line contains the keyword "int" using **strstr()**. If found, return **1**.
4. Otherwise, check if the current line contains the keyword "text" using **strstr()**. If found, return **2**.
5. If neither keyword is found, move to the next line by incrementing **temp**.
6. Repeat steps 2-5 until a special line is encountered.

This function iterates through the lexical lines starting from the current index until it encounters a special line. During this iteration, it checks for the presence of keywords "int" and "text" to determine the type of control. If "int" is found, it returns **1**; if "text" is found, it returns **2**. If neither keyword is found, it continues to the next line.

## bothInt;

void bothInt(char lexLines[][300], int temp,int i){ //if both are integer identifiers.

       int toCopied = checkIntIndex(lexLines[temp+1]);

       intVar[i]=intVar[toCopied];

}

* **Purpose**: To assign the value of one integer identifier to another when both operands in an expression are integer identifiers.
* **Parameters**:
  + **lexLines**: A 2D array containing lexical lines.
  + **temp**: The current index indicating the position of the operands in **lexLines**.
  + **i**: The index representing the destination integer identifier.
* **Algorithm**:
  + Retrieve the index of the integer identifier to be copied (**toCopied**) from the second operand (at **temp+1** index) using **checkIntIndex**.
  + Assign the value of the integer identifier at **toCopied** index to the integer identifier at index **i** in the **intVar** array.

This function simplifies the assignment operation when both operands are integer identifiers by copying the value from one to the other.

## intControl;

void intControl() { // operands integer statements.

    int endInd = endIndex();

    int temp = currentIndex;

    if (lexLines[temp][0] == '\0') {

        exit(0);

    }

    int isExist = checkIntIndex(lexLines[temp]);

    if (typeControl() == 1 || isExist != -1) {

        while (strcmp(lexLines[temp], "EndOfLine\n")) {

            int k = checkLineIndex("Keyword(int)\n");

            int condition1 = k < temp && k != -1;

            int condition2 = k < temp;

            char str[100];

            strcpy(str, lexLines[temp]);

            char ident[100];

            toIdent(ident, str);

            int isNotSame = !isSameInt(ident) && !isSameTxt(ident);

            if (strstr(str, "Identifier") && condition1 && isNotSame) {

                strcpy(intIdentifiers[lastInt], ident);

                lastInt++;

            } else if (strcmp(str, "Keyword(is)\n") == 0 && condition2) {

                int control = intOperator(lexLines, temp);

                int i = checkIntIndex(lexLines[temp - 1]);

                if(strstr(lexLines[temp+1],"Identifier") && control==-1){

                      bothInt(lexLines,temp,i);

                }

                else if (control == -1) {

                    int value = toInt(lexLines[temp + 1]);

                    if(value>99999999){

                        printf("Max Integer value is 99999999! ");

                        value=0;

                    }

                    intVar[i] = value;

                } else {

                    intVar[i] = control;

                }

            }

            else if (strcmp(str, "Comma\n") == 0) { // lexLines[temp] == "Comma"

                temp++;

                continue;

            }

            temp++;

        }

        currentIndex = endInd + 1;

    }

}

* **Purpose**: This function processes integer statements, handling assignments and declarations involving integer variables.
* **Algorithm**:
  1. **Initialization**: Initialize **endInd** with the index of the end of the current statement, and **temp** with the current index.
  2. **Check for End of File**: If **lexLines[temp][0]** is the null character, indicating the end of the file, exit the program.
  3. **Check for Integer Existence**: Use **checkIntIndex** to check if the current line declares or uses an integer variable. If an integer is found, proceed.
  4. **Processing Loop**: Iterate through the lines until reaching the end of the statement.
     + **Identify Lines**: Extract the current line into the **str** array and the identifier into the **ident** array using **toIdent**.
     + **Check Conditions**:
       - If the line contains an identifier and meets specific conditions (**condition1**, **condition2**, **isNotSame**), add it to the **intIdentifiers** array.
       - If the line is an assignment (**Keyword(is)**), evaluate the expression using **intOperator** and assign the result to the corresponding integer variable. If both operands are integer identifiers, use the **bothInt** function to handle this case.
       - If the line is a comma, continue to the next line.
  5. **Update Current Index**: Set **currentIndex** to **endInd + 1**, indicating the next statement's beginning.

This function handles both integer variable declarations and assignments, ensuring proper storage of integer values and identification of integer identifiers for subsequent use.

## bothStr;

void bothStr(char lexLines[][300], int temp,int i){ //if both are text identifiers.

       int toCopied = checkStrIndex(lexLines[temp+1]);

       strcpy(strNames[i],strNames[toCopied]);

}

* **Purpose**: To copy the value of one text identifier to another.
* **Parameters**:
  + **lexLines**: A 2D array containing lexical lines.
  + **temp**: The current index in **lexLines**.
  + **i**: The index of the text identifier to which the value is copied.
* **Algorithm**:
  + Determine the index of the text identifier to be copied (**toCopied**) using the **checkStrIndex** function.
  + Copy the value of the text identifier at index **toCopied** to the text identifier at index **i** using **strcpy**.

This function simplifies the process of copying the value of one text identifier to another when both identifiers are involved in an assignment statement.

## strControl;

void strControl() { // operands text statements.

    int endInd = endIndex();

    int temp = currentIndex;

    if (lexLines[temp][0] == '\0') {

        exit(0);

    }

    int isExist = checkStrIndex(lexLines[temp]);

    int isNotInt = checkIntIndex(lexLines[temp]);

    if (typeControl() == 2 || isExist != -1) {

        while (strcmp(lexLines[temp], "EndOfLine\n")) {

            int k = checkLineIndex("Keyword(text)\n");

            int condition1 = k < temp && k != -1;

            int condition2 = k < temp;

            char str[280];

            strcpy(str, lexLines[temp]);

            char ident[280];

            toIdent(ident, str);

            int isNotSame = !isSameInt(ident) && !isSameTxt(ident);

            if (strstr(str, "Identifier") && condition1 && isNotSame) {

                strcpy(strIdentifiers[lastStr], ident);

                lastStr++;

            } else if (strcmp(str, "Keyword(is)\n") == 0 && condition2) {

                int i = checkStrIndex(lexLines[temp - 1]);

                char operator= lexLines[temp + 2][9];

                if(operator=='+' || operator=='-'){

                     txtOperator(lexLines,temp);

                }

                else if(strstr(lexLines[temp+1],"Identifier")){

                      bothStr(lexLines,temp,i);

                }

                else{

                toStr(str, lexLines[temp + 1]);

                strcpy(strNames[i], str);

                }

            } else if (strcmp(lexLines[temp], "Comma\n") == 0 && condition2) {

                temp++;

                continue;

            }

            temp++;

        }

        currentIndex = endInd + 1;

    }

}

* **Purpose**: Handle text-related operations like assignments and string concatenation.
* **Parameters**:
  + **lexLines**: 2D array containing lexical lines.
  + **temp**: Current index in **lexLines**.
* **Algorithm**:
  + Check if the current line is empty. If so, exit.
  + Determine if the current line contains a text identifier using **checkStrIndex**.
  + Determine if the current line contains an integer identifier using **checkIntIndex**.
  + Check the type of control using **typeControl**.
  + Iterate through the lexical lines until encountering an end-of-line marker.
  + Check for different conditions such as assignment statements or concatenation operations.
  + For assignment statements, handle cases where the value is another identifier or a literal string.
  + Perform string concatenation if the operator is '+' or '-'.
  + Continue iteration if encountering a comma.
  + Update the current index.

This function facilitates handling text-related operations, including assignments and string concatenation, based on the given lexical lines.

## interpreter;

void interPreter(){ // interpretes every statement.

     int i = 0;

    while (i < 1000) {

       oneRev();

       i++;

   }

}

The **interpreter** function executes each statement in the program. Here's the breakdown:

* **Purpose**: Execute every statement in the program sequentially.
* **Algorithm**:
  1. Initialize a loop counter **i** to 0.
  2. Iterate through the loop until **i** reaches 1000, ensuring the interpreter doesn't run indefinitely.
  3. Inside the loop, call the **oneRev** function to process each statement.
  4. Increment the loop counter **i** after processing each statement.

This function ensures that each statement in the program is executed, limited to a maximum of 1000 iterations to prevent infinite looping.

## main;

int main() {

    fileReading("aa.sta");

    lexLinesCreate("aa.lex");

    interPreter();

    return 0;

}

* **Purpose**: Entry point of the program, reads files, creates lexical lines, and invokes the interpreter.
* **Algorithm**:
  1. Call **fileReading("aa.sta")** to read the contents of the file "aa.sta", which presumably contains the statements of the program.
  2. Call **lexLinesCreate("aa.lex")** to create lexical lines by parsing the file "aa.lex", which likely contains lexical information about the program.
  3. Call **interPreter()** to interpret and execute the program statements.
  4. Return 0 to indicate successful execution.

This **main** function orchestrates the initialization and execution of the program, from reading files to interpreting the program logic.

# *Test Cases*

## Defining Variables and Exceeding Maximum Int Value;

metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturulduOutput;

Here we see that the integer value exceeding maximum limit gave error and became 0.

## Maximum Text Limit;

metin, ekran görüntüsü, yazılım, ekran, görüntüleme içeren bir resim

Açıklama otomatik olarak oluşturuldu

ekran görüntüsü, metin, çizgi, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturulduOutput;

String value truncated back to size 256.

## String Operations and Lines of Code;

metin, ekran görüntüsü, yazılım, ekran, görüntüleme içeren bir resim

Açıklama otomatik olarak oluşturuldu

Output;

metin, yazı tipi, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

## Loops;

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

Output;

metin, yazı tipi, ekran görüntüsü, el yazısı içeren bir resim

Açıklama otomatik olarak oluşturuldu

## metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim Açıklama otomatik olarak oluşturulduRead Operations;

Output;

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

First, varName is “Something…” and sth is “Enter another string: “.

metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturulduAt the line 2, we entered new value for varName and printed it.

At the line 5, we used sth for prompt and we entered “ThisIsAnother?” for varName and printed it. Then line 8 printed the information as you see.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturulduAt the line 10, we entered a value for varName without prompt string and printed it.

At the line 14, we entered nonInteger value for an integer i, the program gave an error and printed a as 0 and program finished.