

**EGE UNIVERSITY**

**FACULTY OF ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

***PROGRAMMING LANGUAGES***

***Interpreter For Star Language***

**2023–2024 SPRING SEMESTER**

**PROJECT-2 REPORT**

**DELIVERYDATE**

20/05/2024

**PREPARED BY**

İREM NUR ŞENER 05230000198

TUNA BAŞKURT 05230000192

İSMAİL ERKAN 05210000278

BURAK TAŞKIN 05220000981

İçindekiler

[Key Components of Our Code and FlowChart 3](#_Toc166972379)

[FlowChart 3](#_Toc166972380)

[Key Components of the Program: 4](#_Toc166972381)

[Demonstration of Execusion:Explanation of Each Method 5](#_Toc166972382)

[1) raiseError() 5](#_Toc166972383)

[2) skipWhitespace() 5](#_Toc166972384)

[3) skipComment() 5](#_Toc166972385)

[4) isKeyword() 5](#_Toc166972386)

[5) isOperator() 5](#_Toc166972387)

[6) getNextToken() 5](#_Toc166972388)

[7) parseDeclaration() 5](#_Toc166972389)

[8) getVariable() 5](#_Toc166972390)

[9) parseOutput() 6](#_Toc166972391)

[10) parseInput() 6](#_Toc166972392)

[11) parseAssignment() 6](#_Toc166972393)

[12) sizeFunc() 6](#_Toc166972394)

[13) subsFunc() 6](#_Toc166972395)

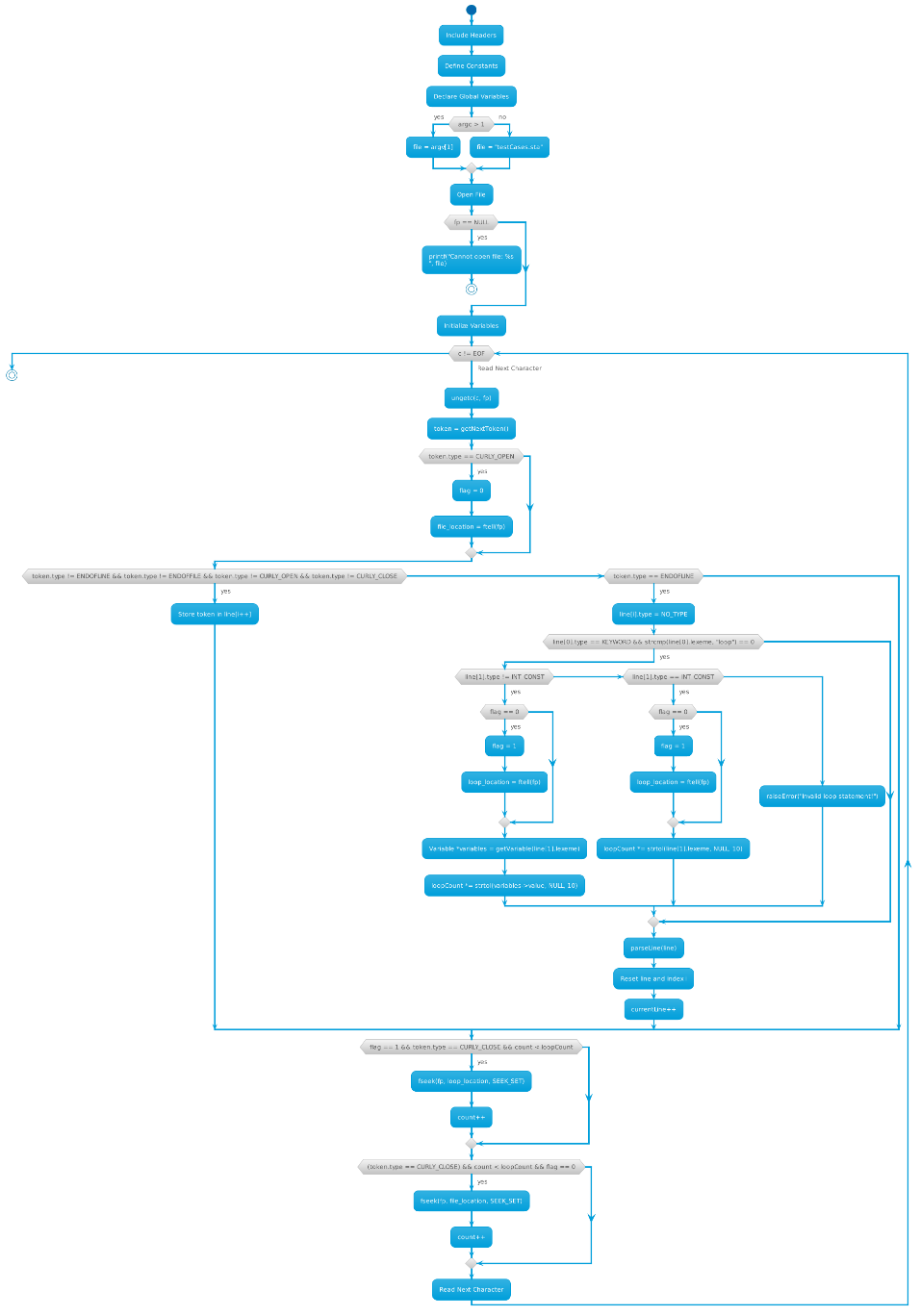
[14) parseArithmeticAssignment() 6](#_Toc166972396)

[15) parseLine() 6](#_Toc166972397)

[16) Main Method() 6](#_Toc166972398)

# Key Components of Our Code and FlowChart

## FlowChart



## Key Components of the Program:

1. **Tokenization**:
   * Our program defines a TokenType enumeration to categorize different types of tokens such as identifiers, constants, operators, keywords, etc.
   * The getNextToken function reads characters from the input file, skips whitespace and comments, and identifies tokens like keywords, operators, identifiers, integer constants, and string constants.
2. **Error Handling**:
   * The raiseError function handles errors by printing an error message and terminating the program. It is used throughout the code to handle invalid input, unrecognized characters, and incorrect syntax.
3. **Variable Management**:
   * Variables are managed using a Variable structure that stores the variable's name, value, and type (integer or text).
   * Functions like parseDeclaration, getVariable, parseAssignment, and parseArithmeticAssignment handle variable declarations, lookups, and assignments.
4. **Command Execution**:
   * Our program supports commands like write and read for outputting and inputting data. The parseOutput and parseInput functions handle these commands.
   * Arithmetic assignments and operations are handled by the parseArithmeticAssignment function, which supports addition and subtraction for both integers and text.
5. **Control Flow**:
   * Our program includes support for loops using the loop keyword. The loop construct allows for repeating a block of code a specified number of times, and the program uses the fseek function to jump back to the start of the loop as needed.
6. **File Handling**:
   * Our program reads the script from a file specified by the user (or a default file). It uses standard file I/O operations (fopen, fgetc, ungetc, ftell, fseek) to read and navigate the file.

# Demonstration of Execusion:Explanation of Each Method

## raiseError()

void raiseError(char\* message)

{

  printf("ERR! Line %d:  %s\n", currentLine, message);;

  exit(1);

}

The **raiseError** function is designed to handle error reporting in the program. When an error occurs, this function is called with a descriptive error message. It prints the error message along with the current line number where the error occurred, making it easier to identify and debug issues in the code. After displaying the error message, the function terminates the program by calling **exit(1)**, indicating that the program has encountered an error and cannot continue. This immediate termination prevents further execution and potential cascading errors.

## skipWhitespace()

char skipWhitespace(char ch) {

  while (isspace(ch)) {

    ch = (char) fgetc(fp);

  }

  return ch;

}

The **skipWhitespace** function is responsible for advancing the file pointer past any whitespace characters in the input stream. It takes a character **ch** as input and repeatedly reads the next character from the file (using **fgetc(fp)**) as long as the current character is a whitespace character (checked using the **isspace** function). Once a non-whitespace character is encountered, the function returns that character. This helps ensure that the parser processes meaningful input by ignoring spaces, tabs, and other whitespace characters.

## skipComment()

char skipComment(char ch) {

  if (ch == '/') {

    ch = (char) fgetc(fp);

    char nextc;

    if (ch == '\*') {

      do {

        if (nextc == EOF) {

          raiseError("Comment cannot be terminated!");

        }

        ch = nextc;

        nextc = (char) fgetc(fp);

      } while (!(ch == '\*' && nextc == '/'));

      return (char) fgetc(fp);

    } else {

      raiseError("Unrecognized character: '/'");

    }

  }

  return ch;

}

The **skipComment** function is designed to handle and skip over block comments in the input file. It takes a character **ch** as input and checks if it is a forward slash (**/**). If so, it reads the next character to determine if it starts a block comment (**/\***). If a block comment is detected, the function enters a loop to skip all characters until it finds the closing comment sequence (**\*/**). During this process, if the end of the file (EOF) is reached before finding the closing sequence, it raises an error indicating an unclosed comment. If the character after **/** is not an asterisk (**\***), it raises an error for an unrecognized character. If no comment is found, it returns the input character. This ensures that comments are ignored during parsing.

## isKeyword()

bool isKeyword (char str[]) {

  const char\* KEYWORDS[] = {"int", "loop", "text", "size", "read", "write"};

  for (int i = 0; i < sizeof(KEYWORDS) / sizeof(KEYWORDS[0]); i++) {

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

      return true;

    }

  }

  return false;

}

The **isKeyword** function checks if a given string **str** is a reserved keyword in the language being parsed. It takes a string **str** as input and compares it against a list of predefined keywords stored in the **KEYWORDS** array. The function iterates through each element of the **KEYWORDS** array, using the **strcmp** function to check if the input string matches any of the keywords. If a match is found, the function returns **true**, indicating that the string is a keyword. If no match is found after checking all keywords, the function returns **false**. This helps in distinguishing keywords from identifiers during the parsing process.

## isOperator()

char isOperator(char ch) {

  if (ch == '+' || ch == '-') {

    return ch;

  }

  if (ch == 'i') {

    long int pos = ftell(fp);

    char nextCh = (char)fgetc(fp);

    if (nextCh == 's') {

      return '=';

    }

    fseek(fp, pos, SEEK\_SET);

  }

  return '\0';

}

The **isOperator** function checks if a given character **ch** is a recognized operator in the language. It initially checks if **ch** is either a plus (**+**) or minus (**-**) operator. If so, it returns the operator character. If **ch** is the letter 'i', the function checks if the subsequent character is 's' (as in "is"). It temporarily stores the current file position using **ftell(fp)** and reads the next character with **fgetc(fp)**. If this character is 's', it indicates the operator "is" (treated as **=**) and returns the **=** character. If the subsequent character is not 's', it restores the file position using **fseek(fp, pos, SEEK\_SET)**. If **ch** is not a recognized operator, the function returns **\0**. This function aids in identifying and handling operators during tokenization.

## getNextToken()

Token getNextToken() {

  Token token;

  token.lexeme = calloc(MAX\_IDENT\_LENGTH, sizeof(char));

  char ch = (char) fgetc(fp);

  //SKIP WHITESPACE and COMMENT

  while(isspace(ch) || ch == '/') {

    ch = skipComment(ch);

    ch = skipWhitespace(ch);

    if (ch == EOF) {

      token.type = ENDOFFILE;

      token.lexeme[0] = '\0';

      return token;

    }

  }

  if (ch == 't') {

      int next\_char = fgetc(fp);

      if (next\_char == 'i' && fgetc(fp) == 'm' && fgetc(fp) == 'e' && fgetc(fp) == 's') {

          token.type = ENDOFLINE;

          strcpy(token.lexeme, "");

          return token;

      }

      else {

          ungetc(next\_char, fp);

      }

  }

  //ENDOFLINE

  if (ch == 'n' && fgetc(fp) == 'e' && fgetc(fp) == 'w' && fgetc(fp) == 'L' &&

      fgetc(fp) == 'i' && fgetc(fp) == 'n' && fgetc(fp) == 'e' && fgetc(fp) == '.') {

      token.type = NEWLINE;

      strcpy(token.lexeme, "");

      return token;

  }

  else

  {

    if (ch == '.')

    {

      token.type = ENDOFLINE;

      strcpy(token.lexeme, "");

      return token;

    }

  }

  //OPERATOR

  char operator = isOperator(ch);

  if (operator != '\0') {

    token.type = OPERATOR;

    token.lexeme[0] = operator;

    token.lexeme[1] = '\0';

    return token;

  }

  //IDENTIFIER

  if (isalpha(ch)) {

    int j = 0;

    while ((isalnum(ch) || ch == '\_' && ch != '\0')) {

      token.lexeme[j++] = ch;

      if(j > MAX\_IDENT\_LENGTH) {

        char errMessage[57];

        sprintf(errMessage, "Identifiers must be smaller or equal than %d characters!", 57);

        raiseError(errMessage);

      }

      ch = (char) fgetc(fp);

    }

    ungetc(ch, fp);

    token.lexeme[j] = '\0';

    token.type = IDENTIFIER;

    if(isKeyword(token.lexeme)){

      token.type = KEYWORD;

    }

    return token;

  }

  //INTEGER

  if (isdigit(ch)) {

    unsigned long long value = 0;

    while (isdigit(ch)) {

      value = value \* 10 + (ch - '0');

      if(value > 4294967295 ) {

        raiseError("Integer value is too big!");

      }

      ch = (char) fgetc(fp);

    }

    if (isalpha(ch) || ch == '\_') {

      raiseError("Invalid identifier, identifiers cannot start with a number!");

    }

    ungetc(ch,fp);

    sprintf(token.lexeme, "%llu", value);

    token.type = INT\_CONST;

    return token;

  }

  // CURLY BRACES

  if (ch == '{')

  {

    token.type = CURLY\_OPEN;

    strcpy(token.lexeme, "{");

    return token;

  }

  if (ch == '}')

  {

    token.type = CURLY\_CLOSE;

    strcpy(token.lexeme, "}");

    return token;

  }

  //PARENTHESIS\_OPEN

  if (ch == '(') {

    token.type = PARENTHESIS\_OPEN;

    strcpy(token.lexeme, "(");

    return token;

  }

  //PARENTHESIS\_CLOSE

  if (ch == ')') {

    token.type = PARENTHESIS\_CLOSE;

    strcpy(token.lexeme, ")");

    return token;

  }

  //COMMA

  if (ch == ',') {

    token.type = COMMA;

    strcpy(token.lexeme, ",");

    return token;

  }

  //STRING CONSTANT

  if (ch == '"' && ch != '\0') {

    int j = 0;

    ch = (char) fgetc(fp);

    while (ch != '"' && ch != '\0') {

      if (ch == EOF) {

        raiseError("String cannot terminated!");

      }

      token.lexeme[j++] = ch;

      ch = (char) fgetc(fp);

    }

    token.lexeme[j] = '\0';

    token.type = STR\_CONST;

    return token;

  }

  char errMessage[50];

  sprintf(errMessage, "Unrecognized character: '%c'!", ch);

  raiseError(errMessage);

}

**getNextToken** function provides a comprehensive approach to tokenize a stream of characters from a file, covering various types of tokens such as whitespace, comments, operators, identifiers, keywords, integers, curly braces, parentheses, commas, and string constants.

### Key Points:

1. **Whitespace and Comments**: The loop checks and skips over any whitespace and comments.
2. **End of File Handling**: If EOF is encountered, the function returns an **ENDOFFILE** token.
3. **End of Line and Dot Handling**: Checks for specific keywords and punctuation indicating the end of a line or other significant token types.
4. **Operator Handling**: Uses **isOperator** to determine if **ch** is an operator and returns the corresponding token.
5. **Identifier and Keyword Handling**: Identifiers are checked for alphanumeric characters or underscores, and their length is validated. Keywords are distinguished using **isKeyword**.
6. **Integer Constants**: The function converts digit sequences into integer constants, ensuring they do not exceed the allowable range.
7. **String Constants**: Handles string literals enclosed in double quotes.
8. **Curly Braces, Parentheses, and Commas**: These are directly checked and returned as corresponding tokens.
9. **Error Handling**: Any unrecognized character leads to an error with a specific message.

## parseDeclaration()

void parseDeclaration(Token \*line, int i)

{

  if (line[0].type != KEYWORD || line[1].type != IDENTIFIER) {

    raiseError("Invalid variable initialization");

  }

  Variable variable;

  strcpy(variable.name, line[i].lexeme);

  variable.value = calloc(1, sizeof(char));

  variable.value = "";

  if (strcmp(line[0].lexeme, "int") == 0) {

    variable.type = INT;

  } else if (strcmp(line[0].lexeme, "text") == 0) {

    variable.type = TEXT;

  } else {

    char errMessage[100];

    sprintf(errMessage, "Unrecognized type: %s!", line[i].lexeme);

    raiseError(errMessage);

  }

  variables[variablesSize++] = variable;

}

1. **Function Definition and Parameters**:
   * **void parseDeclaration(Token \*line, int i)**: The function takes an array of tokens (**line**) and an index (**i**).
2. **Type and Identifier Check**:
   * It checks if the first token is a keyword and the second token is an identifier. If not, it raises an error with **raiseError("Invalid variable initialization")**.
3. **Variable Initialization**:
   * A new **Variable** structure is created.
   * The identifier name from the tokens is copied to the **name** field of the variable.
   * Memory is allocated for the **value** field (though the allocation is redundant due to the immediate reassignment to an empty string).
4. **Type Assignment**:
   * It checks the keyword to determine the variable type (**int** or **text**). If the type is unrecognized, it raises an error with an appropriate message.
5. **Add Variable to List**:
   * The new variable is added to the **variables** array, and the **variablesSize** counter is incremented.

### Summary:

The function validates the declaration syntax, initializes a variable with its name and type, and adds it to a list of variables. If an invalid type or structure is found, it raises an error and stops the program.

Formun Üstü

## getVariable()

Variable\* getVariable(char \*name) {

  for (int i = 0; i < variablesSize; i++)

  {

    if (strcmp(variables[i].name, name) == 0)

    {

      return &variables[i];

**}**

**}**

**char errMessage[50];**

**return &variables[5];**

**}**

The **getVariable** function retrieves a variable by name from the list of variables. It iterates through the variables array, comparing each variable's name with the provided name parameter using **strcmp**. If a match is found, it returns a pointer to that variable. However, if no match is found, it currently returns a pointer to the 6th element of the variables array, which might not be the desired behavior for error handling.

## parseOutput()

**void parseOutput(Token \*line) {**

**Variable variable = \*getVariable(line[1].lexeme);**

**FILE \*fp = fopen("output.txt", "a");**

**for (int i = 0; line[i].type != NO\_TYPE; i++)**

**{**

**Variable variable = \*getVariable(line[i].lexeme);**

**if (variable.type != NO\_TYPE && strcmp(line[i].lexeme, variable.name) == 0 && variable.value != NULL)**

**{**

**printf("%s\n", variable.value);**

**fprintf(fp, "%s\n", variable.value);**

**}**

**if (line[i].type == STR\_CONST && line[i].type != NO\_TYPE && line[i].lexeme != NULL)**

**{**

**printf("%s\n", line[i].lexeme);**

**fprintf(fp, "%s\n", line[i].lexeme);**

**}**

**}**

**}**

This function begins by initializing a variable called variable of type Variable using the lexeme value of the second token in the input array line. It then opens a file named "output.txt" in append mode to prepare for writing output.

Next, the function enters a loop that iterates through each token in the line array until it encounters a token of type NO\_TYPE.

Within the loop, it retrieves variable information based on the lexeme of the current token in the array and stores it in the variable variable. If the obtained variable has a type other than NO\_TYPE, its name matches the lexeme of the current token, and its value is not NULL, the function prints the value of the variable to the console using printf and writes it to the output file using fprintf.

Similarly, if the type of the current token is STR\_CONST and its lexeme is not NULL, the function prints the lexeme of the token to the console and writes it to the output file.

## parseInput()

**void parseInput(Token \*line) {**

**if (line[1].type != STR\_CONST|| line[2].type != COMMA || line[4].type != NO\_TYPE) {**

**raiseError("Invalid input!");**

**}**

**if (line[2].type != COMMA && strcmp(line[1].lexeme, "prompt") != 0) {**

**raiseError("Invalid input!");**

**}**

**Variable prompt = \*getVariable(line[3].lexeme);**

**printf("%s %s", line[1].lexeme, prompt.value);**

**Variable\* variable = getVariable(line[1].lexeme);**

**char buffer[100];**

**fgets(buffer, 100, stdin);**

**buffer[strcspn(buffer, "\n")] = 0;**

**variable->value = calloc(strlen(buffer) + 1, sizeof(char));**

**strcpy(variable->value, buffer);**

**printf("%s\n", buffer);**

**}**

First, the code checks the validity of the input. If the second token is not a string constant, the third token is not a comma, or the fifth token's type is not NO\_TYPE, it raises an error with the message "Invalid input!". This ensures the accuracy of the input.

Next, if the third token is not a comma and the lexeme value of the second token is not "prompt", it again raises an error with the message "Invalid input!". This ensures the appropriateness of the input.

Then, based on the lexeme value of the fourth token, the code initializes a variable named prompt. This provides a hint for the user to enter a value.

The function uses the standard input stream (stdin) to get input from the user and stores this input in a character array named buffer.

Afterwards, by removing the newline character (EOL) from the buffer array, it cleans the user's input.

Then, the function allocates memory to hold the user's input. It allocates memory equal to the length of the input, providing enough space to store it.

After cleaning the input and allocating memory, the function copies the content of the buffer array to the value property of the corresponding variable.

## parseAssignment()

void parseAssignment(Token \*line, int i) {

  if (line[0].type != IDENTIFIER) {

    raiseError("Invalid assignment!");

  }

  Variable \*variable = getVariable(line[0].lexeme);

  if (line[2].type == INT\_CONST){

    if (variable->type != INT) {

      raiseError("Invalid assignment!");

    }

    variable->value = calloc(strlen(line[2].lexeme) + 1, sizeof(char));

    strcpy(variable->value, line[2].lexeme);

  }

  else if (line[2].type == STR\_CONST)

  {

    if (variable->type != TEXT) {

      raiseError("Invalid assignment!");

    }

    variable->value = calloc(strlen(line[2].lexeme) + 1, sizeof(char));

    strcpy(variable->value, line[2].lexeme);

  }

  else if (line[2].type == IDENTIFIER)

  {

    Variable \*variable2 = getVariable(line[2].lexeme);

    if (variable->type != variable2->type) {

      raiseError("Invalid assignment!");

    }

    variable->value = calloc(strlen(variable2->value) + 1, sizeof(char));

    strcpy(variable->value, variable2->value);

  }

  else

  {

    raiseError("Invalid assignment!");

  }

}

The parseAssignment function processes an assignment statement represented by an array of Token objects. It first checks if the first token is an identifier (IDENTIFIER); if not, it raises an error indicating an invalid assignment.

The function retrieves a pointer to the Variable object corresponding to the first token's lexeme. It then checks the type of the third token (line[2]):

If the third token is an integer constant (INT\_CONST), it verifies that the variable is of type INT. If so, it allocates memory for the variable's value and copies the integer constant into it.

If the third token is a string constant (STR\_CONST), it checks that the variable is of type TEXT. It then allocates memory for the variable's value and copies the string constant into it.

If the third token is another identifier (IDENTIFIER), it retrieves the Variable object for this identifier and ensures both variables have the same type. It allocates memory for the first variable's value and copies the second variable's value into it.

If the third token does not match any of these types, or if any type check fails, the function raises an error indicating an invalid assignment. This ensures that assignments are type-safe and correctly handled for constants and variable-to-variable assignments.

## sizeFunc()

int i = 0;

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

i++;

}

return i;

}

This function is designed to find the length of a string. It takes a pointer to a character array, represented as const char \*string, which points to the address of the string.

The function starts with an integer i initialized to zero. Then, it enters a while loop that continues until it reaches the end of the string, indicated by the null character '\0'.

With each iteration of the loop, i is incremented to move to the next character in the string. This process continues until the loop encounters the null character, indicating the end of the string.

When the loop exits upon encountering the null character, the value of i is returned. This value represents the length of the character array, effectively determining the length of the string.

## subsFunc()

char\* subsFunc(const char \*string, int start, int end) {

char \*substring = calloc(end - start + 1, sizeof(char));

int j = 0;

for (int i = start; i < end; i++) {

substring[j++] = string[i];

}

return substring;

}

The first line of the function allocates a memory block named substring using the calloc function. This block allocates space for the substring to be held. Space is allocated for end - start + 1 characters because the null terminator must also be included, in addition to the characters up to the end index.

In the next line, an integer named j is defined and initialized to zero. This is used as an index to add characters of the substring to the substring array.

A loop begins where the variable i starts from the start index and iterates up to the end index. This loop copies the characters from the specified range into the substring array.

During each iteration of the loop, the value of string[i] is assigned to substring[j], and the index j is incremented.

Once the loop completes execution, the substring array contains the characters from the specified range of the string character array. This substring is then returned to the calling location of the function.

## parseArithmeticAssignment()

void parseArithmeticAssignment(Token \*line , int i) {

if (line[0].type != IDENTIFIER || line[3].type != OPERATOR || line[5].type != NO\_TYPE) {

raiseError("Invalid arithmetic assignment!");

}

Variable \*variable1 = getVariable(line[0].lexeme);

if(variable1->type == INT) {

int value1;

int value2;

if (line[2].type != INT\_CONST && line[2].type != IDENTIFIER) {

raiseError("Invalid arithmetic assignment!");

}

if (line[4].type != INT\_CONST && line[4].type != IDENTIFIER) {

raiseError("Invalid arithmetic assignment!");

}

if (line[2].type == INT\_CONST) {

value1 = strtol(line[2].lexeme, NULL, 10);

} else {

Variable \*variable2 = getVariable(line[2].lexeme);

if (variable2->type != INT) {

raiseError("Invalid arithmetic assignment!");

}

value1 = strtol(variable2->value, NULL, 10);

}

if (line[4].type == INT\_CONST) {

value2 = strtol(line[4].lexeme, NULL, 10);

} else {

Variable \*variable2 = getVariable(line[4].lexeme);

if (variable2->type != INT) {

raiseError("Invalid arithmetic assignment!");

}

value2 = strtol(variable2->value, NULL, 10);

}

if (strcmp(line[3].lexeme, "+") == 0) {

variable1->value = calloc(10, sizeof(char));

sprintf(variable1->value, "%d", value1 + value2);

} else if (strcmp(line[3].lexeme, "-") == 0) {

variable1->value = calloc(10, sizeof(char));

sprintf(variable1->value, "%d", value1 - value2);

if (value1 - value2 < 0) {

raiseError("The answer cannot be negative!");

}

} else {

raiseError("Invalid arithmetic assignment!");

}

}

if(variable1->type == TEXT) {

char \*value1;

char \*value2;

if (line[2].type != STR\_CONST && line[2].type != IDENTIFIER) {

raiseError("Invalid arithmetic assignment!");

}

if (line[4].type != STR\_CONST && line[4].type != IDENTIFIER) {

raiseError("Invalid arithmetic assignment!");

}

if (line[2].type == STR\_CONST) {

value1 = line[2].lexeme;

} else {

Variable \*variable2 = getVariable(line[2].lexeme);

if (variable2->type != TEXT) {

raiseError("Invalid arithmetic assignment!");

}

value1 = variable2->value;

}

if (line[4].type == STR\_CONST) {

value2 = line[4].lexeme;

} else {

Variable \*variable2 = getVariable(line[4].lexeme);

if (variable2->type != TEXT) {

raiseError("Invalid arithmetic assignment!");

}

value2 = variable2->value;

}

if (strcmp(line[3].lexeme, "+") == 0) {

variable1->value = calloc(strlen(value1) + strlen(value2) + 1, sizeof(char));

strcpy(variable1->value, value1);

strcat(variable1->value, value2);

} else if (strcmp(line[3].lexeme, "-") == 0) {

if (strlen(value1) < strlen(value2)) {

raiseError("The subtrahend cannot be longer than the minuend!");

}

size\_t resultLength = strlen(value1) - strlen(value2) + 1;

variable1->value = calloc(resultLength, sizeof(char));

char\* found = strstr(value1, value2);

if (found != NULL) {

strncpy(variable1->value, value1, found - value1);

strcat(variable1->value, found + strlen(value2));

} else {

strcpy(variable1->value, value1);

}

} else {

raiseError("Invalid arithmetic assignment!");

}

}

}

This function is designed to perform arithmetic assignment operations. First, the validity of the incoming expression is checked. The expression must contain an identifier variable, followed by an operator, and then optionally followed by any additional expression. If this structure is not met, it raises an error message saying "Invalid arithmetic assignment!". Next, the type of the identifier variable is checked. If the variable type is INT, a numerical operation is performed. The relevant numerical values are obtained, and the operation is performed based on the operator. The result is assigned as the value of the respective variable. If the variable type is TEXT, a textual operation is performed. The relevant textual values are obtained, and the operation is performed accordingly. For example, operations like concatenating strings or subtracting one string from another are carried out. At each step, errors that may occur during the operation are checked, and appropriate error messages are raised if necessary. Once the operation is completed, the value of the respective variable is updated, and this new value is transferred to the code outside the function. This function is crucial for ensuring the correct and consistent operation of the program, and it ensures that arithmetic assignment operations are carried out properly.

## parseLine()

void parseLine(Token \*line)

{

// NEWLINE

if (line[0].type == NEWLINE)

{

printf("\n");

return ;

}

//DECLARATION

if (line[0].type == KEYWORD && strcmp(line[0].lexeme, "loop") == 0)

return ;

if (line[0].type == KEYWORD && strcmp(line[0].lexeme, "int") == 0 || strcmp(line[0].lexeme, "text") == 0) {

for (int j = 0; line[j].type != NO\_TYPE; j++)

{

if (line[j].type == COMMA)

{

parseDeclaration(line, j + 1);

}

}

parseDeclaration(line, 1);

return ;

}

//COMMAND WRITE

if (line[0].type == KEYWORD && strcmp(line[0].lexeme, "write") == 0) {

return parseOutput(line);

}

//COMMAND READ

if (line[0].type == KEYWORD && strcmp(line[0].lexeme, "read") == 0) {

return parseInput(line);

}

//ASSIGNMENT

if (line[1].type == OPERATOR && strcmp(line[1].lexeme, "=") == 0)

{

if(line[3].type == NO\_TYPE)

{

return parseAssignment(line, 1);

}

else if(line[3].type == OPERATOR && line[5].type == NO\_TYPE)

{

return parseArithmeticAssignment(line, 1);

}

else

{

raiseError("Invalid assignment!");

}

}

raiseError("Parsing error!");

}

This function is designed to parse a program line and perform appropriate operations. It takes an array of tokens, line, as its parameter.

First, the function checks if the incoming expression is a NEWLINE. If the expression is a NEWLINE, it prints a blank line to the screen and terminates the function.

Next, the function checks if the expression is a LOOP declaration. If the expression starts with the keyword "loop," the function terminates.

Then, the function checks if the expression is a DECLARATION. If the expression starts with the keywords "int" or "text," one or more variable declaration operations are performed. This operation is carried out through the parseDeclaration function.

The function checks if the expression is a WRITE command. If the expression starts with the keyword "write," the parseOutput function is called.

It checks if the expression is a READ command. If the expression starts with the keyword " read," the parseInput function is called.

Then, the function checks if the expression is an ASSIGNMENT operation. If the expression is an assignment operation, the parseAssignment or parseArithmeticAssignment functions are called, depending on the type of assignment.

Finally, if the expression does not match any of the above conditions, it indicates a parsing error.

At the end of the function, if no operation is performed or if there is an error condition, an error message is raised.

## Main Method()

int main(int argc, char \*argv[]) {

  variables = calloc(10, sizeof(Variable));

  char\* file = "myprog.star";

  long int file\_location;

  long int loop\_location;

  if(argc > 1) {

      file = argv[1];

  }

  fp = fopen(file, "r");

  if(fp == NULL) {

    printf("Cannot open file: %s\n", file);

    return 1;

  }

  Token token;

  char c = (char) fgetc(fp);

  Token\* line = calloc(100, sizeof(Token));

  int i = 0;

  int loopCount = 1;

  int count = 0;

  int flag = 0;

  while (c != EOF)

  {

    ungetc(c, fp);

    token = getNextToken();

    if (token.type == CURLY\_OPEN)

    {

      flag = 0;

      file\_location = ftell(fp);

    }

    if (token.type != ENDOFLINE && token.type != ENDOFFILE && token.type != CURLY\_OPEN && token.type != CURLY\_CLOSE)

    {

      line[i++] = token;

    }

    else if (token.type == ENDOFLINE)

    {

      line[i].type = NO\_TYPE;

      if (line[0].type == KEYWORD && strcmp(line[0].lexeme, "loop") == 0)

      {

        if (line[1].type != INT\_CONST)

        {

          if (flag == 0)

          {

            flag = 1;

            loop\_location = ftell(fp);

          }

          Variable \*variables = getVariable(line[1].lexeme);

          loopCount \*= strtol(variables->value, NULL, 10);

        }

        else if (line[1].type == INT\_CONST)

        {

          if (flag == 0)

          {

            flag = 1;

            loop\_location = ftell(fp);

          }

          loopCount \*= strtol(line[1].lexeme, NULL, 10);

        }

        else

        {

          raiseError("Invalid loop statement!");

        }

      }

      parseLine(line);

      line = calloc(10, sizeof(Token));

      i = 0;

      currentLine++;

    }

    if (flag == 1 && token.type == CURLY\_CLOSE && count < loopCount)

    {

      fseek(fp, loop\_location, SEEK\_SET);

      count++;

    }

    if ((token.type == CURLY\_CLOSE) && count < loopCount && flag == 0)

    {

      fseek(fp, file\_location, SEEK\_SET);

      count++;

    }

    c = (char) fgetc(fp);

  }

}

Initialization:

It starts by allocating memory for an array of Variable structures named variables, presumably to hold variable data.

It sets the default file name as "myprog.star".

It declares variables file\_location and loop\_location to keep track of file positions.

It checks if any command-line argument is provided. If so, it updates the file variable with the provided file name.

File Handling:

It attempts to open the file specified by file in read mode.

If the file opening fails (fp == NULL), it prints an error message and exits the program.

Tokenization and Parsing:

It initializes variables for tokenization and parsing, such as Token token to store tokens, line to store tokens of the current line, and various counters and flags.

It reads characters from the file one by one until the end of the file (EOF) is reached.

For each character read, it retrieves the next token using getNextToken function.

It processes tokens into lines until an end-of-line token is encountered.

If a line starts with the keyword "loop", it checks the loop count and multiplies it with the loop counter.

It then calls the parseLine function to process the current line.

After parsing a line, it resets the line array and other variables to process the next line.

Loop Handling:

If a loop is encountered, it seeks back to the loop's starting position in the file and continues parsing until the loop counter reaches the desired count.

Once the loop completes, it continues parsing from where it left off in the file.

# Test Cases/Screenshots

## 1)VARIABLES and Declarations

**Basit Değişken Deklarasyonu:**

int myInt.

text myString.

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**Birden Fazla Değişkenin Tek Satırda Tanımlanması:** Birden fazla değişkenin tek bir satırda tanımlanması.

int a, b, c, d.

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

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**İlk Değerle Birlikte Deklarasyon:** Değişkenin ilk değeri ile birlikte tanımlanması.

int count is 5.

write count.

**Maksimum String Boyutunun Kontrol Edilmesi:** String değişkenin maksimum boyutunun kontrol edilmesi

text longString is "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam eget dolor nec elit tincidunt suscipit. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Donec ac risus eget lacus dictum viverra. Integer vehicula turpis sit amet fermentum condimentum. Sed in felis quis felis iaculis finibus. Nullam vulputate, lorem quis cursus dignissim, nisl elit varius sem, in pharetra ipsum mi nec est. Phasellus vestibulum dolor nec odio fermentum pharetra. Morbi egestas placerat eros, in aliquam tortor vehicula a. Donec euismod ante in tincidunt congue. Sed feugiat eleifend nibh sit amet ullamcorper.".

write longString.

yazılım, multimedya yazılımı, grafik yazılımı, metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**Maksimum Integer Değeri:** Maksimum integer değerinin kontrol edilmesi.

int bigInteger is 99999999.

Write bigInteger.

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Negatif Değerlerin Kontrolü:** Negatif değer atanmasının kontrol edilmesi.

int negativeInteger is -100.

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

2) ASSIGNMENTS

**Basit Atama İşlemi:** Değişkene basit bir değer atama işlemi.

myString is "Hello world".

write myString.

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

Açıklama otomatik olarak oluşturuldu

ekran görüntüsü, metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Aritmetik İfade ile Atama:** Aritmetik ifade kullanarak bir değişkene değer atama işlemi.

int myInt .

myInt is 0.

myInt is myInt + 1.

write myInt.metin, multimedya yazılımı, yazılım, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**Negatif Değer Atama İşlemi:** Negatif değer atama işlemi. Bu durumda, değer sıfıra eşitlenmelidir.

int myInt.

myInt is 3-1.

write myInt.

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

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Maksimum Integer Değerini Aşan Atama:** Maksimum integer değerini aşan bir değer atama işlemi.

myInt is 1000000000.

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

3)COMMENTS and LINES of CODES

**Yorum İçeren Satır:** Bir kod satırında yorum bulunması durumu.

text myString /\* This line can be divided into two\*/ is "Hello".

write myString.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**Çok Satırlı Yorum:** Çok satırlı bir yorumun kullanımı.

text myString /\* This line can be divided into two

like this. And comments may be within

As you might have noticed, this is a comment. \*/ is "Hello".

write myString.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Kod Satırı Sonlandırma:** Kod satırlarının nokta ile sonlandırılması durumu.

//yukarıdaki örnekler

4) Input / Output

**Okuma (read) Komutu:** Okuma komutunun doğru şekilde çalışması.

read "Enter a number:", varName.

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

Açıklama otomatik olarak oluşturuldu

**Yazma (write) Komutu:** Yazma komutunun doğru şekilde çalışması ve değerlerin boşluklarla ayrılarak yazdırılması.

write "The values are", a, " ", b, " ", c.

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

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Yeni Satır (newLine) Komutu:** Yeni satır komutunun doğru şekilde çalışması.

newLine.

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

Açıklama otomatik olarak oluşturuldu



5)LOOPS

**Temel for Döngüsü:** Temel for döngüsünün doğru şekilde çalışması.

loop 5 times write "\*".

**Kod Bloğu İçeren for Döngüsü:** Kod bloğu içeren for döngüsünün doğru şekilde çalışması.

int i.

i is 1.

loop 10 times

{

write i.

i is i + 1.

}

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**İç İçe Geçmiş Döngüler:** İç içe geçmiş döngülerin doğru şekilde çalışması.

int i.

i is 1.

loop 10 times

{

loop i times write "\*".

newLine.

i is i + 1.

}

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

// 5+6=11 kez yazdırmalı

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

Açıklama otomatik olarak oluşturuldu

6)OPERATIONS

**Metin Birleştirme (Concatenation):** Metin birleştirme işleminin doğru şekilde çalışması.

text Str1 is "like home.", Str2.

Str2 is "There is no place" + Str1.

Write Str1.

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

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu

**Metin Çıkarma (Subtraction):** Metin çıkarma işleminin doğru şekilde çalışması.

text s is "Ohh ice ice baby!".

text s2.

s2 is s - "ice".

write s2. metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

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

Açıklama otomatik olarak oluşturuldu