

**EGE UNIVERSITY**

**FACULTY OF ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

***PROGRAMMING LANGUAGES***

***Lexical Analyzer For Star Language***

**2023–2024SPRING SEMESTER**

**PROJECT-1 REPORT**

**DELIVERYDATE**

05/05/2024

**PREPARED BY**

BURAK TAŞKIN 05220000981

İSMAİL ERKAN 05210000278

# CONTENTS

1)Token Cases

2)Other Functions

3)Outputs

# Token Cases

## 1)Identifiers

int classifyToken(char str[], FILE \*outputFile) {

    if (!isalpha(str[0])) {

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

        fclose(outputFile);

        exit(5);

    }

    if (length > MAX\_IDENTIFIER\_SIZE) {

        fprintf(stderr, "Error: Identifier must be shorter than %d chars.\n", MAX\_IDENTIFIER\_SIZE);

        fclose(outputFile);

        exit(5);

    }

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

    return 0;

This part checks whether the given string represents an identifier.

It verifies if the first character of the string is an alphabetic character, as identifiers must start with a letter.

If the string is not an identifier, it prints an error message and exits the program.

If the string is an identifier, it checks if the length of the identifier exceeds the maximum allowed length (MAX\_IDENTIFIER\_SIZE). If it does, it prints an error message and exits the program.

Otherwise, it writes the identifier to the output file.

## 2)Integer Constants

int classifyToken(char str[], FILE \*outputFile) {

    int length = strlen(str);

    bool isInteger = true;

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

        if (!isdigit(str[i])) {

            isInteger = false;

            break;

        }

    }

    if (isInteger) {

        if (length > MAX\_INTEGER\_SIZE) {

            fprintf(stderr, "Error: Integer must be shorter than %d chars.\n", MAX\_INTEGER\_SIZE);

            fclose(outputFile);

            exit(5);

        }

        fprintf(outputFile, "IntConst(%s)\n", str);

        return 0;

    }

This part checks whether the given string represents an integer constant.

It iterates through each character of the string and checks if each character is a digit.

If all characters are digits, it considers the string as an integer constant.

It then checks if the length of the integer constant exceeds the maximum allowed length (MAX\_INTEGER\_SIZE). If it does, it prints an error message and exits the program.

Otherwise, it writes the integer constant to the output file.

## 3) Keywords

char keywords[8][10] = {"int", "text", "is", "loop", "times", "read", "write", "newLine"};

    for (int i = 0; i < 8; ++i) {

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

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

            return 0;

        }

    }

This part checks whether the given string represents one of the predefined keywords.

It compares the given string with a predefined list of keywords (keywords array).

If the string matches any of the keywords, it writes the keyword to the output file.

If the string does not match any keyword, it considers it as a generic identifier and writes it to the output file.

## 4)Operators

int isOperator(char str[], FILE \*outputFile) {

    char operators[5][2] = {"+", "-", "\*", "/"};

    for (int i = 0; i < 4; ++i) {

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

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

            return 0;

        }

    }

    fprintf(stderr, "Error: Invalid Operator(%s)\n", str);

    fclose(outputFile);

    exit(5);

    return 0;

}

This part checks whether the given string represents one of the predefined keywords.

It compares the given string with a predefined list of keywords (keywords array).

If the string matches any of the keywords, it writes the keyword to the output file.

If the string does not match any keyword, it considers it as a generic identifier and writes it to the output file.

## 5)Brackets

int isBracket(char str[], FILE \*outputFile) {

    char brackets[2] = {'{', '}'};

    char bracketNames[2][19] = {"LeftCurlyBracket", "RightCurlyBracket"};

    for (int i = 0; i < strlen(str); i++) {

        for (int j = 0; j < 2; j++) {

            if (brackets[j] == str[i]) {

                fprintf(outputFile, "%s\n", bracketNames[j]);

                break;

            }

        }

    }

    return 0;

}

This function checks whether a character string (or character) represents a bracket ('{' or '}').

When the function receives a character string containing one of the bracket characters, it writes the name of the corresponding bracket to the output file and completes the process

## 6)String Constants

strncat(token, &newChar, 1);

            int tokenLength = strlen(token);

            if (tokenLength > MAX\_STRING\_SIZE) {

                fprintf(stderr, "Error: String must be shorter than %d characters.\n", MAX\_STRING\_SIZE);

                fclose(outputFile);

                exit(5);

            }

            if (tokenLength > 0 && token[tokenLength - 1] == '"') {

                fprintf(outputFile, "String(%s)\n", token);

                activeToken = IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT;

                strcpy(token, "");

                continue;

            }

if (activeToken == STRING) {

        fprintf(stderr, "Lexical Error: String must be closed.\n");

    }

String Length Check:

This part checks whether the length of the string exceeds the maximum allowed length (MAX\_STRING\_SIZE) for a string constant.

If the length exceeds the maximum allowed length, it prints an error message to the standard error stream, closes the output file, and exits the program with an error code.

String Write:

After the length check, it further verifies if the string ends with a double quote ("), indicating the end of the string constant.

If the string ends with a double quote and its length is within the allowed limit, it writes the string constant to the output file with the token type "String".

It then resets the active token to IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT to prepare for the next token processing.

It clears the token buffer by copying an empty string into it, ensuring that it's ready to store the next token.

This code block checks if the program is currently processing a string constant, and if so, it prints a lexical error message indicating that the string must be closed properly.

## 7)Comma

int isComma(char str[], FILE \*outputFile) {

    fprintf(outputFile, "Comma\n");

    return 0;

}

This function checks whether a character string represents a comma (',').

When the function receives a character string containing a comma, it writes the "Comma" token type to the output file and completes the process.

## 

## 8)End Of Line

int isEndOfLine(char str[], FILE \*outputFile) {

    fprintf(outputFile, "EndOfLine\n");

    return 0;

}

This function checks whether a character string represents an end-of-line indicator ('.').

When the function receives a character string containing an end-of-line indicator, it writes the "EndOfLine" token type to the output file and completes the process.

## 9)Comments

strncat(token, &newChar, 1);

            int tokenLength = strlen(token);

            if (tokenLength > 1 && token[tokenLength - 2] == '\*' && token[tokenLength - 1] == '/') {

                activeToken = IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT;

                strcpy(token, "");

                continue;

            }

/////

 if (activeToken == COMMENT) {

        fprintf(stderr, "Lexical Error: Comment block must be closed.\n");

    }

1. **Character Concatenation**:
   * This part appends the new character **newChar** to the **token** buffer using the **strncat** function.
   * It calculates the length of the **token** buffer after the concatenation to determine if the token represents the end of a comment.
2. **Token Check**:
   * After appending the new character, it checks if the last two characters in the **token** buffer represent the end of a comment (**\*/**).
   * If the last two characters are **\*** and **/**, it indicates the end of a comment block.
   * In such case, it sets the active token to **IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT** to prepare for processing the next token.
   * It also clears the **token** buffer by copying an empty string into it, ensuring that it's ready to store the next token.

# Other Functions

## 1)

void changeStateAndToken(char token[], char newChar, enum States newState, int (\*f\_name)(char str[], FILE \*\_outputFile),

                  FILE \*outputFile) {

    if (strcmp(token, "") != 0)

    {

        f\_name(token, outputFile);

    }

    activeToken = newState;

    strcpy(token, "");

    strncat(token, &newChar, 1);

}

This function serves to manage the state and content of a token. Here's a bit more detail:

It takes in a token represented as a character array (token[]) and a new character (newChar). Additionally, it receives the new state of the token (newState), a function pointer to process the token (f\_name), and the output file (outputFile).

Here's how it works:

If the current token (token[]) is not empty, it implies that the token is complete and can be processed. In this case, the function pointer f\_name is called to process the token.

Next, the function updates the active token state (activeToken) with the new state (newState).

It clears the token array (token[]) and then appends the new character (newChar) to it, effectively updating the token content.

# 2)

void processLine(char line[], char token[], FILE \*outputFile) {

    int length = strlen(line);

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

        char newChar = line[i];

        if (activeToken == IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT) {

            // Handling different characters based on the current token

            if (newChar == ' ' || newChar == '\n' || newChar == '\0') {

                changeStateAndToken(token, '\0', IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, classifyToken, outputFile);

                continue;

            } else if (isalnum(newChar) || newChar == '\_') {

                strncat(token, &newChar, 1);

                continue;

            } else if (newChar == '+' || newChar == '-' || newChar == '\*' || newChar == '/' || newChar == ':' ||

                       newChar == '=') {

                changeStateAndToken(token, newChar, OPERATOR, classifyToken, outputFile);

                continue;

            } else if (newChar == '{' || newChar == '}') {

                changeStateAndToken(token, newChar, BRACKETS, classifyToken, outputFile);

                continue;

            } else if (newChar == '"') {

                changeStateAndToken(token, newChar, STRING, classifyToken, outputFile);

                continue;

            } else if (newChar == ',') {

                changeStateAndToken(token, newChar, COMMA, classifyToken, outputFile);

                continue;

            } else if (newChar == '.') {

                changeStateAndToken(token, newChar, ENDOFLINE, classifyToken, outputFile);

                continue;

            } else {

                fprintf(stderr, "Invalid Character(%c)\n", newChar);

                fclose(outputFile);

                exit(5);

                continue;

            }

        }

        else if (activeToken == OPERATOR) {

            // Handling different characters based on the current token

            if (newChar == ' ' || newChar == '\n' || newChar == '\0') {

                changeStateAndToken(token, '\0', IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isOperator, outputFile);

                continue;

            } else if (isalnum(newChar) || newChar == '\_') {

                changeStateAndToken(token, newChar, IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isOperator, outputFile);

                continue;

            } else if (newChar == '+' || newChar == '-' || newChar == '\*' || newChar == '/' || newChar == ':' ||

                       newChar == '=') {

                strncat(token, &newChar, 1);

                int tokenLength = strlen(token);

                if (tokenLength == 2 && strcmp(token, "/\*") == 0) {

                    activeToken = COMMENT;

                    strcpy(token, "");

                } else if (tokenLength > 2 && token[tokenLength - 2] == '/' && token[tokenLength - 1] == '\*') {

                    char subToken[1024] = "";

                    for (int a = 0; a < tokenLength - 2; a++) {

                        strncat(subToken, &token[a], 1);

                    }

                    isOperator(subToken, outputFile);

                    activeToken = COMMENT;

                    strcpy(token, "");

                }

                continue;

            }

            else if (newChar == '{' || newChar == '}') {

                changeStateAndToken(token, newChar, BRACKETS, isOperator, outputFile);

                continue;

            } else if (newChar == '"') {

                changeStateAndToken(token, newChar, STRING, isOperator, outputFile);

                continue;

            } else if (newChar == ',') {

                changeStateAndToken(token, newChar, COMMA, classifyToken, outputFile);

                continue;

            } else if (newChar == '.') {

                changeStateAndToken(token, newChar, ENDOFLINE, isOperator, outputFile);

                continue;

            } else {

                fprintf(stderr, "Invalid Character(%c)\n", newChar);

                fclose(outputFile);

                exit(5);

                continue;

            }

        }

        else if (activeToken == COMMENT) {

            // Handling different characters based on the current token

            strncat(token, &newChar, 1);

            int tokenLength = strlen(token);

            if (tokenLength > 1 && token[tokenLength - 2] == '\*' && token[tokenLength - 1] == '/') {

                activeToken = IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT;

                strcpy(token, "");

                continue;

            }

        }

        else if (activeToken == STRING) {

            // Handling different characters based on the current token

            strncat(token, &newChar, 1);

            int tokenLength = strlen(token);

            if (tokenLength > MAX\_STRING\_SIZE) {

                fprintf(stderr, "Error: String must be shorter than %d characters.\n", MAX\_STRING\_SIZE);

                fclose(outputFile);

                exit(5);

            }

            if (tokenLength > 0 && token[tokenLength - 1] == '"') {

                fprintf(outputFile, "String(%s)\n", token);

                activeToken = IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT;

                strcpy(token, "");

                continue;

            }

        }

        else if (activeToken == BRACKETS) {

            // Handling different characters based on the current token

            if (newChar == ' ' || newChar == '\n' || newChar == '\0') {

                changeStateAndToken(token, '\0', IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isBracket, outputFile);

                continue;

            } else if (isalnum(newChar) || newChar == '\_') {

                changeStateAndToken(token, newChar, IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isBracket, outputFile);

                continue;

            } else if (newChar == '+' || newChar == '-' || newChar == '\*' || newChar == '/' || newChar == ':' ||

                       newChar == '=') {

                changeStateAndToken(token, newChar, OPERATOR, isBracket, outputFile);

                continue;

            } else if (newChar == '{' || newChar == '}') {

                strncat(token, &newChar, 1);

                continue;

            } else if (newChar == '"') {

                changeStateAndToken(token, newChar, STRING, isBracket, outputFile);

                continue;

            } else if (newChar == ',') {

                changeStateAndToken(token, newChar, COMMA, classifyToken, outputFile);

                continue;

            } else if (newChar == '.') {

                changeStateAndToken(token, newChar, ENDOFLINE, isBracket, outputFile);

                continue;

            } else {

                fprintf(stderr, "Invalid Character(%c)\n", newChar);

                fclose(outputFile);

                exit(5);

                continue;

            }

        }

        else if (activeToken == COMMA) {

            // Handling different characters based on the current token

            if (newChar == ' ' || newChar == '\n' || newChar == '\0') {

                changeStateAndToken(token, '\0', IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isComma, outputFile);

                continue;

            } else if (isalnum(newChar) || newChar == '\_') {

                changeStateAndToken(token, newChar, IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isComma, outputFile);

                continue;

            } else if (newChar == '+' || newChar == '-' || newChar == '\*' || newChar == '/' || newChar == ':' ||

                    newChar == '=') {

                changeStateAndToken(token, newChar, OPERATOR, isComma, outputFile);

                continue;

            } else if (newChar == '{' || newChar == '}') {

                changeStateAndToken(token, newChar, BRACKETS, isComma, outputFile);

                continue;

            } else if (newChar == '"') {

                changeStateAndToken(token, newChar, STRING, isComma, outputFile);

                continue;

            } else if (newChar == ',') {

                changeStateAndToken(token, newChar, COMMA, classifyToken, outputFile);

                continue;

            } else if (newChar == '.') {

                changeStateAndToken(token, newChar, ENDOFLINE, isComma, outputFile);

                continue;

            } else {

                fprintf(stderr, "Invalid Character(%c)\n", newChar);

                fclose(outputFile);

                exit(5);

                continue;

            }

        }

        else if (activeToken == ENDOFLINE) {

            // Handling different characters based on the current token

            if (newChar == ' ' || newChar == '\n' || newChar == '\0') {

                changeStateAndToken(token, '\0', IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isEndOfLine, outputFile);

                continue;

            } else if (isalnum(newChar) || newChar == '\_') {

                changeStateAndToken(token, newChar, IDENTIFIERxorKEYWORDSxorINTEGER\_CONSTANT, isEndOfLine, outputFile);

                continue;

            } else if (newChar == '+' || newChar == '-' || newChar == '\*' || newChar == '/' || newChar == ':' ||

                       newChar == '=') {

                changeStateAndToken(token, newChar, OPERATOR, isEndOfLine, outputFile);

                continue;

            } else if (newChar == '{' || newChar == '}') {

                changeStateAndToken(token, newChar, BRACKETS, isEndOfLine, outputFile);

                continue;

            } else if (newChar == '"') {

                changeStateAndToken(token, newChar, STRING, isEndOfLine, outputFile);

                continue;

            } else if (newChar == ',') {

                changeStateAndToken(token, newChar, COMMA, classifyToken, outputFile);

                continue;

            } else if (newChar == '.') {

                changeStateAndToken(token, newChar, ENDOFLINE, isEndOfLine, outputFile);

                continue;

            } else {

                fprintf(stderr, "Invalid Character(%c)\n", newChar);

                fclose(outputFile);

                exit(5);

                continue;

            }

        }

    }

}

This function, **processLine**, is responsible for parsing each character in a line of input. Here's a brief explanation:

1. It iterates over each character in the input line.
2. Depending on the current active token state (**activeToken**), it processes the characters accordingly:
   * For identifiers, keywords, and integer constants, it accumulates alphanumeric characters or underscores into the token until it encounters whitespace or specific operators, brackets, strings, commas, or periods. Then, it calls the **changeStateAndToken** function to handle state transitions and token processing.
   * For operators, brackets, strings, commas, and periods, it changes the token state appropriately and calls **changeStateAndToken** to handle state transitions and token processing.
   * For comments and strings, it accumulates characters until it finds the closing markers ('\*/' for comments and '"' for strings), updating the token state and content accordingly.
   * If it encounters any invalid characters, it prints an error message and exits the program.

In summary, **processLine** manages the parsing of characters within a line, updating the token state and content as needed and handling various types of tokens and their boundaries.

Formun Üstü

# 3) (Main)

int main() {

    // Opening input and output files

    char token[2048] = "";

    FILE \*inputFile;

    char line[2048];

    inputFile = fopen("code.sta", "r");

    if (inputFile == NULL) {

        fprintf(stderr, "Error opening the input file!\n");

        return 1;

    }

    FILE \*outputFile;

    outputFile = fopen("code.lex", "w");

    if (outputFile == NULL) {

        fprintf(stderr, "Error opening the output file!\n");

        return 1;

    }

    // Processing each line of the input file

    while (fgets(line, sizeof(line), inputFile)) {

        processLine(line, token, outputFile);

    }

    // Checking for lexical errors

    if (activeToken == COMMENT) {

        fprintf(stderr, "Lexical Error: Comment block must be closed.\n");

    }

    if (activeToken == STRING) {

        fprintf(stderr, "Lexical Error: String must be closed.\n");

    }

    if (token[0] == '.') {

        fprintf(outputFile, "EndOfLine\n");

    }

    // Closing input and output files

    fclose(inputFile);

    fclose(outputFile);

    return 0;

}

This **main** function serves as the entry point for the lexical analyzer program. Here's a brief explanation:

1. It begins by opening the input file ("code.sta") for reading and the output file ("code.lex") for writing. If any errors occur during file opening, it prints an error message and exits the program.
2. It initializes a token buffer and reads each line from the input file using **fgets**.
3. For each line, it calls the **processLine** function to parse and classify the tokens, writing the results to the output file.
4. After processing all lines, it checks for any remaining unclosed comments or strings. If found, it prints corresponding lexical error messages.
5. If the last character of the input is a period, indicating the end of the line, it writes an "EndOfLine" token to the output file.
6. Finally, it closes both the input and output files before exiting the program with a success status.

In summary, the **main** function orchestrates the overall process of lexical analysis, including file handling, token processing, error checking, and file closing.

Formun Üstü

# Outputs

## İnput1

int a,b,c.

read "First:" a.

newLine.

read "Second:" b.

newLine.

c is a+b.

c is c/2.

write "Result:" c.

## Output1(code.lex)

Keyword(int)

Identifier(a)

Comma

Identifier(b)

Comma

Identifier(c)

EndOfLine

Keyword(read)

String("First:")

Identifier(a)

EndOfLine

Keyword(newLine)

EndOfLine

Keyword(read)

String("Second:")

Identifier(b)

EndOfLine

Keyword(newLine)

EndOfLine

Identifier(c)

Keyword(is)

Identifier(a)

Operator(+)

Identifier(b)

EndOfLine

Identifier(c)

Keyword(is)

Identifier(c)

Operator(/)

IntConst(2)

EndOfLine

Keyword(write)

String("Result:")

Identifier(c)

EndOfLine

## Input2

8kalem.

## Output2(terminal)

Error: Identifier must start with a letter.

## Input3

ankaraaaaaaaaaa.

## Output3(terminal)

Error: Identifier must be shorter than 10 chars.

## Input4

int 3434344839048.

## Output4(terminal)

Error: Integer must be shorter than 8 chars.

## Input5

String "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa".

## Output5(terminal)

Error: String must be shorter than 256 characters.

## Input6

String "ismail.

## Output6(terminal)

Lexical Error: String must be closed.

## Input7

String "ismail". /\*İsmail

## Output7(terminal)

Lexical Error: Comment block must be closed.