
# VIT®

## Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

Name: Aditya .

Reg No: 23BCE1344 .

Subject: Compiler Design .

Lab Task: ASSESSMENT-2 .

~ Dr. Sureshkumar WI

# Aim:

***i) To construct a lexical analyzer to identify tokens from:***

***a) A simple statement stored in a linear array.***

***b) A small program (not exceeding 5 lines) stored in a text file.***

***c) A small program (not exceeding 5 lines) input by the user and stored in a text file.***

***ii) To construct a lexical analyzer using the LEX tool to identify tokens from a given input.***

## i) To construct a lexical analyzer

## Algorithm:

1. **Initialize:** Define token types (keywords, identifiers, operators, literals, etc.).
2. **Read Input:**
   - For a simple statement: Read from a linear array.
   - For a small program: Read from a text file.
3. **Tokenize:**
   - Traverse each character in the input.
   - Identify and classify tokens based on predefined rules.
   - Store identified tokens in a list.
4. **Output Tokens:** Print or store the list of tokens.

## Source Code:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
bool isDelimiter(char ch) {
    return (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}');
}
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=');
}
bool validIdentifier(char* str) {
    return !(str[0] >= '0' && str[0] <= '9') && !isDelimiter(str[0]);
}
bool isKeyword(char* str) {
    return (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
```

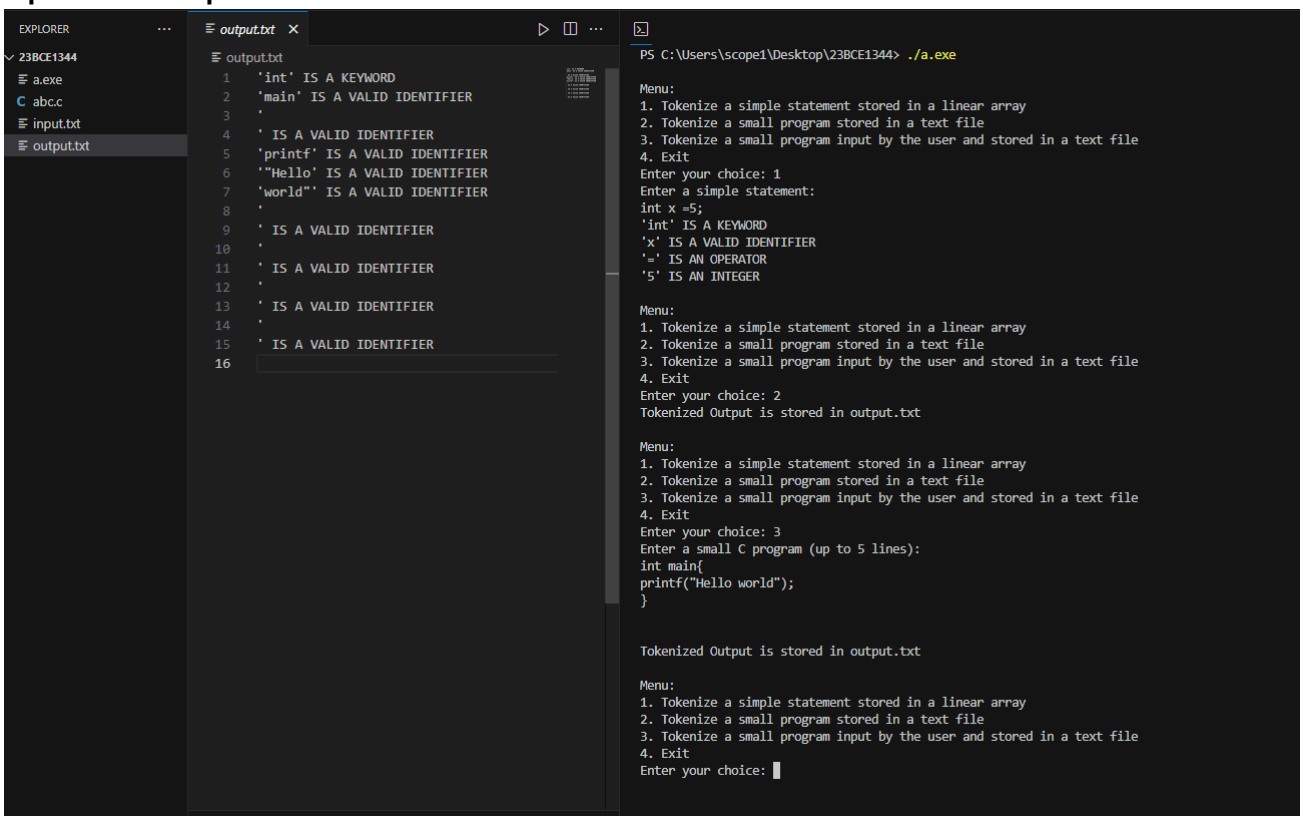
```
                !strcmp(str, "break") || !strcmp(str, "continue") ||
                !strcmp(str, "int") || !strcmp(str, "double") ||
                !strcmp(str, "float") || !strcmp(str, "return") ||
                !strcmp(str, "char") || !strcmp(str, "case") ||
                !strcmp(str, "sizeof") || !strcmp(str, "long") ||
                !strcmp(str, "short") || !strcmp(str, "typedef") ||
                !strcmp(str, "switch") || !strcmp(str, "unsigned") ||
                !strcmp(str, "void") || !strcmp(str, "static") ||
                !strcmp(str, "struct") || !strcmp(str, "goto"));
}
bool isInteger(char* str) {
        int i, len = strlen(str);
        if (len == 0) return false;
        for (i = 0; i < len; i++) {
                if (!isdigit(str[i]) || (str[i] == '-' && i > 0))
                        return false;
        }
        return true;
}
bool isRealNumber(char* str) {
        int i, len = strlen(str);
        bool hasDecimal = false;
        if (len == 0) return false;
        for (i = 0; i < len; i++) {
                if (!isdigit(str[i]) && str[i] != '.' ||
                    (str[i] == '-' && i > 0))
                        return false;
                if (str[i] == '.')
                        hasDecimal = true;
        }
        return hasDecimal;
}
char* subString(char* str, int left, int right) {
        int i;
        char* subStr = (char*)malloc(sizeof(char) * (right - left + 2));
        for (i = left; i <= right; i++)
                subStr[i - left] = str[i];
        subStr[right - left + 1] = '\0';
        return subStr;
}
void parse(char* str, FILE* outputFile) {
        int left = 0, right = 0;
        int len = strlen(str);
        bool expectingIdentifier = false;
        while (right <= len && left <= right) {
                if (!isDelimiter(str[right]))
                        right++;
                if (isDelimiter(str[right]) && left == right) {
                        if (isOperator(str[right]))
                                fprintf(outputFile, "'%c' IS AN OPERATOR\n", str[right]);
                        right++;
                        left = right;
```

```
		} else if (isDelimiter(str[right]) && left != right || (right == len && left != right)) {
			char* subStr = subString(str, left, right - 1);
			if (isKeyword(subStr)) {
				fprintf(outputFile, "'%s' IS A KEYWORD\n", subStr);
				expectingIdentifier = true;
			} else if (expectingIdentifier) {
				if (validIdentifier(subStr)) {
					fprintf(outputFile, "'%s' IS A VALID IDENTIFIER\n", subStr);
				} else {
					fprintf(outputFile, "'%s' IS NOT A VALID IDENTIFIER\n",
subStr);
				}
				expectingIdentifier = false;
			} else if (isInteger(subStr)) {
				fprintf(outputFile, "'%s' IS AN INTEGER\n", subStr);
			} else if (isRealNumber(subStr)) {
				fprintf(outputFile, "'%s' IS A REAL NUMBER\n", subStr);
			} else if (validIdentifier(subStr) && !isDelimiter(str[right - 1])) {
				fprintf(outputFile, "'%s' IS A VALID IDENTIFIER\n", subStr);
			} else if (!validIdentifier(subStr) && !isDelimiter(str[right - 1])) {
				fprintf(outputFile, "'%s' IS NOT A VALID IDENTIFIER\n", subStr);
			}
			free(subStr);
			left = right;
		}
	}
	return;
}
void parseAndPrint(char* str) {
	int left = 0, right = 0;
	int len = strlen(str);
	bool expectingIdentifier = false;
	while (right <= len && left <= right) {
		if (!isDelimiter(str[right]))
			right++;
		if (isDelimiter(str[right]) && left == right) {
			if (isOperator(str[right]))
				printf("'%c' IS AN OPERATOR\n", str[right]);
			right++;
			left = right;
		} else if (isDelimiter(str[right]) && left != right || (right == len && left != right)) {
			char* subStr = subString(str, left, right - 1);
			if (isKeyword(subStr)) {
				printf("'%s' IS A KEYWORD\n", subStr);
				expectingIdentifier = true;
			} else if (expectingIdentifier) {
				if (validIdentifier(subStr)) {
					printf("'%s' IS A VALID IDENTIFIER\n", subStr);
				} else {
					printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
				}
				expectingIdentifier = false;
```

```
                        } else if (isInteger(subStr)) {
                                printf("'%s' IS AN INTEGER\n", subStr);
                        } else if (isRealNumber(subStr)) {
                                printf("'%s' IS A REAL NUMBER\n", subStr);
                        } else if (validIdentifier(subStr) && !isDelimiter(str[right - 1])) {
                                printf("'%s' IS A VALID IDENTIFIER\n", subStr);
                        } else if (!validIdentifier(subStr) && !isDelimiter(str[right - 1])) {
                                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
                        }
                        free(subStr);
                        left = right;
                }
        }
        return;
}
void tokenizeFromArray(char* statement) {
        parseAndPrint(statement);
}
void tokenizeFromFile(char* filename) {
        FILE *file, *outputFile;
        char line[256];
        file = fopen(filename, "r");
        outputFile = fopen("output.txt", "w");
        if (file == NULL || outputFile == NULL) {
                printf("Could not open file.\n");
                return;
        }
        while (fgets(line, sizeof(line), file)) {
                parse(line, outputFile);
        }
        fclose(file);
        fclose(outputFile);
        printf("Tokenized Output is stored in output.txt\n");
}
void tokenizeFromUserInput() {
        FILE *file;
        char line[256];
        int lineCount = 0;
        file = fopen("input.txt", "w");
        if (file == NULL) {
                printf("Could not open file.\n");
                return;
        }
        printf("Enter a small C program (up to 5 lines):\n");
        while (lineCount < 5 && fgets(line, sizeof(line), stdin)) {
                fprintf(file, "%s", line);
                lineCount++;
        }
        fclose(file);
        tokenizeFromFile("input.txt");
}
int main() {
```

```
	int choice;
	char statement[256];
	while (1) {
		printf("\nMenu:\n");
		printf("1. Tokenize a simple statement stored in a linear array\n");
		printf("2. Tokenize a small program stored in a text file\n");
		printf("3. Tokenize a small program input by the user and stored in a text file\n");
		printf("4. Exit\n");
		printf("Enter your choice: ");
		scanf("%d", &choice);
		getchar();
		switch (choice) {
		case 1:
			printf("Enter a simple statement:\n");
			fgets(statement, sizeof(statement), stdin);
			statement[strcspn(statement, "\n")] = '\0';
			tokenizeFromArray(statement);
			break;
		case 2:
			tokenizeFromFile("input.txt");
			break;
		case 3:
			tokenizeFromUserInput();
			break;
		case 4:
			exit(0);
		default:
			printf("Invalid choice. Please try again.\n");
		}
	}
	return 0;
}
```

## Input and Output:



## ii)

## Algorithm :

1. Define Tokens: Specify regular expressions for different token types (keywords, identifiers, operators, literals, etc.) in the LEX file.
2. Write LEX Rules:
    - Use LEX syntax to associate regular expressions with corresponding actions.
    - Actions typically involve printing or storing the identified tokens.
3. Generate Scanner: Run the LEX tool to generate the scanner (lex.yy.c).
4. Compile and Run: Compile the generated C file with a C compiler and run the executable with input data.

## SourceCode:

```
%{
   int COMMENT = 0;
%}

identifier     [a-zA-Z][a-zA-Z0-9]*

%%

#.*                    { printf("\n%s is a Preprocessor Directive", yytext); }

int       |
float     |
main      |
if        |
else      |
printf    |
scanf     |
for       |
char      |
getch     |
while                  { printf("\n%s is a Keyword", yytext); }

"/*"                   { COMMENT = 1; }
"*/"                   { COMMENT = 0; }

{identifier}\(           { if (!COMMENT) printf("\nFunction:\t%s", yytext); }

\{                     { if (!COMMENT) printf("\nBlock Begins"); }
\}                     { if (!COMMENT) printf("\nBlock Ends"); }

{identifier}(\[[0-9]*\])?     { if (!COMMENT) printf("\n%s is an Identifier", yytext); }

\".*\"                 { if (!COMMENT) printf("\n%s is a String", yytext); }

[0-9]+                   { if (!COMMENT) printf("\n%s is a Number", yytext); }

\)(\;)?                { if (!COMMENT) { printf("\t"); ECHO; printf("\n"); } }

\(                     ECHO;
```

```
=                       { if (!COMMENT) printf("\n%s is an Assmt oprtr", yytext); }

\<=       |
\>=       |
\<        |
==                       { if (!COMMENT) printf("\n%s is a Rel. Operator", yytext); }

.|\n                     ;

%%

int main(int argc, char **argv)
{
   if (argc > 1)
   {
      FILE *file;
      file = fopen(argv[1], "r");

      if (!file)
      {
         printf("\nCould not open the file: %s", argv[1]);
         exit(0);
      }

      yyin = file;
   }

   yylex();
   printf("\n\n");

   return 0;
}

int yywrap()
{
   return 0;
}
```

## Input/Output:

```
oslab@oslab-VirtualBox:~/cd$ gedit abc.l
oslab@oslab-VirtualBox:~/cd$ gedit test.c
oslab@oslab-VirtualBox:~/cd$ lex abc.l
oslab@oslab-VirtualBox:~/cd$ ls
abc.l  lex.yy.c  test.c
oslab@oslab-VirtualBox:~/cd$ gcc lex.yy.c
oslab@oslab-VirtualBox:~/cd$ ls
abc.l  a.out  lex.yy.c  test.c
oslab@oslab-VirtualBox:~/cd$ ./a.out
test.c

test is an Identifier
c is an Identifier^C
oslab@oslab-VirtualBox:~/cd$ ./a.out test.c

#include <stdio.h> is a Preprocessor Directive
int is a Keyword
Function:       main(   )

Block Begins
int is a Keyword
a is an Identifier
= is an Assmt oprtr
5 is a Number
char is a Keyword
c is an Identifier
= is an Assmt oprtr
x is an Identifier
Function:       printf(
"Hello world %d %c" is a String
a is an Identifier
c is an Identifier      );

return is an Identifier
0 is a Number
```

## Conclusion:

- Successfully implemented a lexical analyzer that can identify and classify tokens from a simple statement, a small program stored in a file, and a user-input small program stored in a file.
- Successfully constructed a lexical analyzer using the LEX tool that accurately identifies and classifies tokens from the input data based on predefined regular expressions.