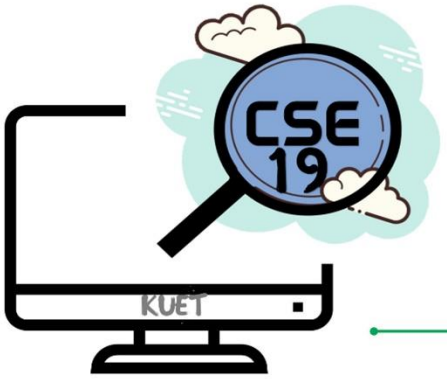
Khulna University of Engineering and Technology

ASSIGNMENT on Flex Tokenization of Self Made Language
Course Name: Compiler Design Laboratory, Course Code: CSE 3212

Submitted From	Submitted to
Name: Naimur Rahman	Nazia Jahan Khan Chowdhury Assistant Professor
Roll: 1907031	Dipannita Biswas Lecturer
Section: A	Dept of CSE, KUET
Dept of CSE, KUET	Date of Submission: 04-10-2023



Objectives:

- ✓ Recognize keywords, identifiers, literals, and operators in the source code
- ✓ Handle whitespace and comments gracefully
- ✓ Generate error tokens/messages for duplicate variables
- ✓ Optimize regular expressions for improved lexer performance
- ✓ Provide clear and informative error messages

Introduction:

Tokenization is a fundamental step in the process of translating human-readable source code into machine-executable instructions. In the realm of programming languages and compilers, tokenization involves breaking down a source code file into its smallest meaningful units, known as tokens. These tokens can represent keywords, identifiers, operators, literals, and other language constructs. Tokenization serves as the initial phase, laying the foundation for subsequent parsing and analysis, making it a crucial step in the compilation or interpretation of programming languages.

ID and Type Design:

Datatypes:

datatype	“vari” similar to int | “varf” similar to float | “varc” similar to char
void	“void”

Type Design:

Int	[+-]?[0-9]+
Float	[+-]?[0-9]+[.][0-9]+([eE][+-]?[0-9]+)?|[+-]?[1-9]+[eE][-+][0-9]+
String	\"[A-Za-z0-9]+\"
ID	[a-zA-Z][a-zA-Z0-9]*

Regular Expression for Types:

Variable Declaration	{datatype}[ ]+{ID}("="({ID}|{int}|{float}|{string}))?([ ]*","[ ]*{ID}("="({ID}|{int}|{float}|{string}))?)*
Value Assignment	{ID}[ ]*"="[ ]*({ID}|{int}|{float}|{string})
Array	"array"[ ]+{datatype}[ ]+{ID}[ ]+"of"[ ]+{int}

Loop and Directives:

Loops and Syntax:

- ✓ “loopw” similar to while loop

  loopw as {variable1} {relational operator} {variable2} {logical operator} . . . . . .

begin

//statements

End

- ✓ “loopf” similar to for loop

loopf with {variable} {start range}...{end range} {increment/decrement}

begin

//statements

End

Loop Regular Expressions for Loop Conditions:

loopw	"loopw"[ ]+"as"[ ]+({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string})([ ]+{Logical_operator}[ ]+({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string}))*[ ]*
loopf	"loopf"[ ]+"with"[ ]+{ID}[ ]+{int}"..."{int}[ ]+[-]?{int}[ ]*

Directives:

Include	"#include<".*">"

Operator Handling:

Arithmatic_operator	"+"|"-"|"*"|"/"|"$"
Logical_operator	"&&"|"||"
Not	"!"
Unary_operator	"++"|"--"
Relational_operator	"ls"|"gr"|"eq"|"ge"|"le"|"ne"
Assignment_operator	"="|"+="|"-="|"/="|"*="

Regular Expressions:

Unary Operator	{ID}{Unary_operator} {Unary_operator}{ID}
Relational Operator	({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string})
Assignment Operator	{ID}[ ]*{Assignment_operator}[ ]*({ID}|{int}|{float}|{string})

Logical Operator	({ID}|{int}|{float}|{string})[ ]*{Logical_operator}[ ]*({ID}|{int}|{float}|{string})
Not	{Not}[ ]*({ID}|{int}|{float}|{string})
Arithmetic Operator	({ID}|{int}|{float}|{string})[ ]*{Arithmatic_operator}[ ]*({ID}|{int}|{float}|{string})

Conditional Operations:

"is" similar to if

is {variable1} {relational operator} {variable2} {logical operator} . . . . . .

{

//statements

}

"oris" similar to else if

oris {variable1} {relational operator} {variable2} {logical operator} . . . . . .

{

//statements

}

"or" similar to else

{

//statements

}

"si" similar to fi

Regular Expressions for Conditions:

"is" "oris"	({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string})([ ]+{Logical_operator}[ ]+({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string}))*[ ]*

Functions:

main	({datatype}|{void})[ ]"main()"[\n\t]*"{"
functions	({datatype}|{void})[ ]+"function"[ ]+{ID}"("({datatype}[ ]+{ID}([ ]*","[ ]*{datatype}[ ]{ID})*)?")"[\n\t]*"{"

Comments:

Single Line Comment \/\/[^\n]*

Multi Line Comment \/*([^*]|*[^\/])**\/

Source File:

```
//Single
/* Multi
line
comment */
#include<stdio.h>
vari main(){
        vari a,b,c,d=a;
        varf c="abc";
        varc c=10;
        return 0;
        d=4;
        a=2;
        b=3;
        c=10;
        c++;
        c--;
        c gr 5;
        5 ne 4;
        array vari h of 10;
        array vari d of 3;
        is c gr d
        {
        d=6;
        }
        oris a ne c
        {
        c=1;
        }
        si
        loopw as a eq b
        begin
        a=5;
        end
        loopf with a 1...3 1
        begin
        b=9;
        end
}

varf function add(vari x, vari y){
}

void function sub(){
}
```

Flex File:

```
%{

// header files
#include<string.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

// keyword and datatype string collection
const char* keywords[] = {"vari","varf","varc","begin","end","return","function","with","is","or","oris","si","loopf","loopw","ls","gr","eq","ge","le","ne","void","as","array","of",NULL};
const char* datatype[] = {"vari","varf","varc",NULL};
const char* relation[] = {"ls","gr","le","ge","eq","ne",NULL};
const char* logic[] = {"&&","||",NULL};

// check if a new variable name is a keyword
int is_keyword(const char* word) {
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(keywords[i], word) == 0) {
            return 1;
        }
    }
    return 0;
}

// check if it is a logical operator
int is_logic(const char* word) {
    for (int i = 0; logic[i] != NULL; i++) {
        if (strcmp(logic[i], word) == 0) {
            return 1;
        }
    }
    return 0;
}

// check if it is a relational operator
int is_relation(const char* word) {
    for (int i = 0; relation[i] != NULL; i++) {
        if (strcmp(relation[i], word) == 0) {
            return 1;
        }
    }
    return 0;
}

// check if a new variable is a datatype
int is_datatype(const char* word) {
    for (int i = 0; datatype[i] != NULL; i++) {
        if (strcmp(datatype[i], word) == 0) {
            return 1;
        }
    }
    return 0;
}

// symbol table linked list that adds new values into head
```

```
struct Symbol_Table {
    char* name;
    char* datatype;
    char* value;
    struct Symbol_Table* next;
};

// head pointer for symbol table linked list
struct Symbol_Table* head = NULL;

// insert a new variable into symbol table
void insert_variable(const char* name, const char* datatype, const char* value) {
    struct Symbol_Table* new_symbol = malloc(sizeof(struct Symbol_Table));
    new_symbol->name = strdup(name);
    new_symbol->datatype = strdup(datatype);
    new_symbol->value = strdup(value);
    new_symbol->next = head;
    head = new_symbol;
}

// check if a variable is already declared
int is_declared(const char* name) {
    struct Symbol_Table* tmp = head;
    while (tmp != NULL) {
        if (strcmp(tmp->name, name) == 0) {
            return 1;
        }
        tmp = tmp->next;
    }
    return 0;
}

// update the value of a variable in the symbol table
int update_value(const char* name, const char* value) {
    struct Symbol_Table* tmp = head;
    while (tmp != NULL) {
        if (strcmp(tmp->name, name) == 0) {
            free(tmp->value); // free previous value
            tmp->value = strdup(value); // update new value
            return 1;
        }
        tmp = tmp->next;
    }
    return 0;
}

// variables needed for operations
int var_count=0, statement_count=0, function_count=0, header_count=0, temp_var_count=0, i=
0;
int Unary_operator_count=0, Relational_operator_count=0, Assignment_operator_count=0,
Logical_operator_count=0, Arithmatic_operator_count=0;
int mcmt_count=0, scmt_count=0, array_count=0;
int is_declared_check=0, conditional_count=0, loop_count=0;
%}

/* labels for future need */
%x WHILE_CONDITION
%x WHILE_BODY
%x FOR_CONDITION
%x FOR_BODY
```

```
%x IS_BODY
%x IS_CONDITION

/* defining regex */
char [a-zA-Z]
digit [0-9]
special [_@]
space " "
newline "\n"
tabline "\t"
datatype "vari"|"varf"|"varc"
void "void"
Arithmatic_operator "+"|"-"|"*"|"/"|"$"
Logical_operator "&&"|"||"
Not "!"
Unary_operator "++"|"--"
Relational_operator "ls"|"gr"|"eq"|"ge"|"le"|"ne"
Assignment_operator "="|"+="|"-="|"/="|"*="
int [+-]?[0-9]+
float [+-]?[0-9]+[.][0-9]+([eE][+-]?[0-9]+)?|[+-]?[1-9]+[eE][-+][0-9]+
string \"[A-Za-z0-9]+\"
ID [a-zA-Z][a-zA-Z0-9]*
scmt \/\/[^\n]*
mcmt \/\*([^*]|\*[^\/])*\*\/

%%

{scmt} {
    //single line comment detection
    scmt_count++;
    printf("Single Line Comment\n");
}

{mcmt} {
    //multi line comment detection
    mcmt_count++;
    printf("Multiple Line Comment\n");
}


{datatype}[ ]+{ID}("="({ID}|{int}|{float}|{string}))?([ ]*","[ ]*{ID}("="({ID}|{int}|{float}|{string}))?)* {
    //variable detection
    char* token = strtok(yytext, " ,\t");
    char* type;
    int invalid = 0;
    temp_var_count=0;

    while (token != NULL) {
        if (!is_datatype(token)) {
            if (is_keyword(token)) {
                printf("ERROR: Reserved keyword can't be used as a variable name: %s\n", token);
                invalid = 1;
                break;
            }
            if (is_declared(token)) {
                printf("ERROR: Variable '%s' is already declared.\n", token);
                invalid = 1;
                break;
```

```
            } else {
                char* init_value = strchr(token, '=');
                if (init_value != NULL) {
                    *init_value = '\0'; // Null-terminate the variable name
                    const char* varName = token;
                    if (is_declared(varName)) {
                        printf("ERROR: Variable '%s' is already declared.\n", varName);
                        invalid = 1;
                        break;
                    }
                    if (is_keyword(varName)) {
                        printf("ERROR: Reserved keyword can't be used as a variable name:
%s\n", varName);
                        invalid = 1;
                        break;
                    }
                    Assignment_operator_count++;
                    printf("Assignment operator found\n");
                    init_value++; // Move to the value part
                    insert_variable(token, type, init_value);
                    printf("Variable name: %s, Datatype: %s, Initialized with: %s\n",
token, type, init_value);
                } else {
                    insert_variable(token, type, "");
                    printf("Variable name: %s, Datatype: %s\n", token, type);
                }
                temp_var_count++;
                var_count++;
            }
        } else {
            type = token;
        }
        token = strtok(NULL, " ,\t");
    }
    if (!invalid) {
        printf("Total variables declared: %d\n", temp_var_count);
    }
}

{ID}[ ]*"="[ ]*({ID}|{int}|{float}|{string}) {
    //value updating of previously declared variables
    char* var = strtok(yytext, " =");
    char* val = strtok(NULL, " =");
    if (!is_declared(var)) {
        printf("ERROR: Variable '%s' not declared before assignment.\n", var);
    }
    else {
        printf("ASSIGNMENT: Variable '%s' assigned value '%s'\n", var, val);
        update_value(var, val); // Update the variable value
    }

}

"array"[ ]+{datatype}[ ]+{ID}[ ]+"of"[ ]+{int} {
    //array declaration
    char* token = strtok(yytext, " \t");
    char* type;
    int invalid = 0;
    while (token != NULL) {
        if(strcmp(token, "array")==0){
```

```
230
231         }
232         else if (!is_datatype(token)) {
233             if (is_keyword(token)) {
234                 printf("ERROR: Reserved keyword can't be used as a variable name: %s\n",
    token);
235                 invalid = 1;
236                 break;
237             }
238             if (is_declared(token)) {
239                 printf("ERROR: Variable '%s' is already declared.\n", token);
240                 invalid = 1;
241                 break;
242             } else {
243                 insert_variable(token, type, "");
244                 printf("Variable name: %s, Datatype: %s\n", token, type);
245             }
246             array_count++;
247             break;
248         }
249         else{
250             type = token;
251         }
252         token = strtok(NULL, " \t");
253     }
254     if (!invalid) {
255         printf("Array declared\n");
256     }
257 }
258
259
260 "loopw"[ ]+"as"[ ]+ {
261     //while loop syntax
262     BEGIN(WHILE_CONDITION);
263 }
264
265 <WHILE_CONDITION>({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|
    {float}|{string})([ ]+{Logical_operator}[ ]+({ID}|{int}|{float}|{string})[ ]+
    {Relational_operator}[ ]+({ID}|{int}|{float}|{string}))*[ ]* {
266     //while loop condition
267     char* token = strtok(yytext, " \t");
268     while (token != NULL) {
269         if(is_relation(token)){
270             Relational_operator_count++;
271             printf("Relational operator found\n");
272         }
273         else if(is_logic(token)){
274             Logical_operator_count++;
275             printf("Logical operator found\n");
276         }
277         token = strtok(NULL, " \t");
278     }
279     printf("WHILE LOOP: Condition\n");
280     loop_count++;
281     BEGIN(WHILE_BODY);
282 }
283
284 <WHILE_BODY>[ \t]*"begin"  { printf("WHILE LOOP: Start\n"); }
285
286 <WHILE_BODY>[ \t]*"end"  {
287     printf("WHILE LOOP: End\n");
```

```
    BEGIN(INITIAL);
}

<WHILE_BODY>.*  { printf("WHILE LOOP Code: %s\n", yytext); }


"loopf"[ ]+"with"[ ]+ {
    //for loop syntax
    BEGIN(FOR_CONDITION);
}

<FOR_CONDITION>{ID}[ ]+{int}"..."{int}[ ]+[-]?{int}[ ]* {
    //for loop condition
    printf("FOR LOOP: Condition\n");
    loop_count++;
    BEGIN(FOR_BODY);
}

<FOR_BODY>[ \t]*"begin"  { printf("FOR LOOP: Start\n"); }

<FOR_BODY>[ \t]*"end"  {
    printf("FOR LOOP: End\n");
    BEGIN(INITIAL);
}

<FOR_BODY>.*  { printf("FOR LOOP Code: %s\n", yytext); }


"is"[ ]+ {
    //if structure
    printf("IS BLOCK\n");
    is_declared_check=1;
    conditional_count++;
    BEGIN(IS_CONDITION);
}

"oris"[ ]+ {
    //else if structure
    if (is_declared_check) {
        printf("ORIS BLOCK\n");
        conditional_count++;
        BEGIN(IS_CONDITION);
    } else {
        printf("ERROR: 'oris' without preceding 'is'\n");
    }
}

"or"[ ]* {
    //else structure
    if (is_declared_check) {
        printf("OR BLOCK\n");
        conditional_count++;
        BEGIN(IS_BODY);
    } else {
        printf("ERROR: 'or' without preceding 'is'\n");
    }
}

<IS_CONDITION>({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|
{float}|{string})([ ]+{Logical_operator}[ ]+({ID}|{int}|{float}|{string})[ ]+
```

```
{Relational_operator}[ ]+({ID}|{int}|{float}|{string}))*[ ]* {
    //if or else if condition check
    char* token = strtok(yytext, " \t");
    while (token != NULL) {
        if(is_relation(token)){
            Relational_operator_count++;
            printf("Relational operator found\n");
        }
        else if(is_logic(token)){
            Logical_operator_count++;
            printf("Logical operator found\n");
        }
        token = strtok(NULL, " \t");
    }
    printf("IS CONDITION\n");
    BEGIN(IS_BODY);
}

<IS_BODY>[ \t]*"{" { printf("START OF CODE BLOCK\n");}

<IS_BODY>[ \t]*"}" {
    printf("END OF CODE BLOCK\n");
    BEGIN(INITIAL);
}

"si" {
    //denotes end of if block and simmilar to fi
    if(is_declared_check){
        printf("END OF IS BLOCK\n");
        is_declared_check=0;
        BEGIN(INITIAL);
    }
    else{
        printf("ERROR: 'si' without preceding 'is'\n");
    }

}

<IS_BODY>.* { printf("CODE: %s\n", yytext); }


; {statement_count++;}


({datatype}|{void})[ ]"main()"[ \n\t]*"{" {
    //check main function
    function_count++;
    printf("Main Function\n");
}

({datatype}|{void})[ ]+"function"[ ]+{ID}"("({datatype}[ ]+{ID}([ ]*","[ ]*{datatype}[ ]
{ID})*)?")"[ \n\t]*"{" {
    //function declaration
    function_count++;
    printf("Function Declaration\n");
}


"#include<".*">" {
    //header file
```

```
    header_count++;
    printf("Header File\n");
}


{ID}{Unary_operator} {
    //operators check
    Unary_operator_count++;
    printf("Unary operator found\n");
}

{Unary_operator}{ID} {
    Unary_operator_count++;
    printf("Unary operator found\n");
}

({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string}) {
    Relational_operator_count++;
    printf("Relational operator found\n");
}

{ID}[ ]*{Assignment_operator}[ ]*({ID}|{int}|{float}|{string}) {
    Assignment_operator_count++;
    printf("Assignment operator found\n");
}

({ID}|{int}|{float}|{string})[ ]*{Logical_operator}[ ]*({ID}|{int}|{float}|{string}) {
    Logical_operator_count++;
    printf("Logical operator found\n");
}

{Not}[ ]*({ID}|{int}|{float}|{string}) {
    Logical_operator_count++;
    printf("Logical operator found\n");
}

({ID}|{int}|{float}|{string})[ ]*{Arithmatic_operator}[ ]*({ID}|{int}|{float}|{string}) {
    Arithmatic_operator_count++;
    printf("Arithmatic operator found\n");
}


. {
    //ignores everything else
}

%%

int yywrap()
{
    return 1;
}

int main()
{
    yyin = fopen( "sample.txt", "r" );
    yylex();
    printf("%d variables declared\n", var_count);
    printf("%d arrays declared\n", array_count);
    printf("%d functions declared\n", function_count);
```

```
    printf("%d headers declared\n", header_count);
    printf("%d Unary operators\n", Unary_operator_count);
    printf("%d Relational operators\n", Relational_operator_count);
    printf("%d Assignment operators\n", Assignment_operator_count);
    printf("%d Logical operators\n", Logical_operator_count);
    printf("%d Arithmatic operators\n", Arithmatic_operator_count);
    printf("%d single line comments\n", scmt_count);
    printf("%d multiple line comments\n", mcmt_count);
    printf("%d conditional statements\n", conditional_count);
    printf("%d loops\n", loop_count);
    printf("%d statements\n", statement_count);
    return 0;
}
```

Output File:

```
C:\Users\never\OneDrive\Documents\Project>app
Single Line Comment
Multiple Line Comment
Header File
Main Function
Variable name: a, Datatype: vari
Variable name: b, Datatype: vari
Variable name: c, Datatype: vari
Assignment operator found
Variable name: d, Datatype: vari, Initialized with: a
Total variables declared: 4
ERROR: Variable 'c' is already declared.
ERROR: Variable 'c' is already declared.
ASSIGNMENT: Variable 'd' assigned value '4'
ASSIGNMENT: Variable 'a' assigned value '2'
ASSIGNMENT: Variable 'b' assigned value '3'
ASSIGNMENT: Variable 'c' assigned value '10'
Unary operator found
Unary operator found
Relational operator found
Relational operator found
Variable name: h, Datatype: vari
Array declared
ERROR: Variable 'd' is already declared.
IS BLOCK
Relational operator found
IS CONDITION
START OF CODE BLOCK
CODE:   d=6;
END OF CODE BLOCK
ORIS BLOCK
Relational operator found
IS CONDITION
START OF CODE BLOCK
CODE:   c=1;
END OF CODE BLOCK
END OF IS BLOCK
Relational operator found
WHILE LOOP: Condition
WHILE LOOP: Start
WHILE LOOP Code:        a=5;
WHILE LOOP: End
FOR LOOP: Condition
FOR LOOP: Start
FOR LOOP Code:  b=9;
FOR LOOP: End
Function Declaration
Function Declaration
4 variables declared
1 arrays declared
3 functions declared
1 headers declared
2 Unary operators
5 Relational operators
1 Assignment operators
0 Logical operators
0 Arithmatic operators
1 single line comments
1 multiple line comments
2 conditional statements
2 loops
14 statements
```

Discussion:

Tokenization is the process of breaking down a given text or source code into smaller, meaningful units known as tokens, with wide-ranging applications in fields like natural language processing, information retrieval, programming languages, data parsing, security, and machine learning. In NLP, it enables language analysis and understanding; in information retrieval, it facilitates efficient text searching; in programming languages and compilers, it transforms source code into language constructs; in data processing, it simplifies structured data handling; in security, it enhances data protection, and in machine learning, it's often a critical preprocessing step. The choice of tokenization strategy depends on the specific context and desired outcomes, making it a foundational step in various data processing and analysis tasks.

Conclusion:

Tokenization plays a vital role in transforming unstructured or structured data into manageable and meaningful units across diverse domains. Whether it's making human language computationally accessible, enabling efficient information retrieval, or serving as a fundamental step in compiling programming languages, tokenization is a foundational process that paves the way for more advanced data analysis and interpretation. Its flexibility and adaptability to specific contexts underscore its significance in modern data-driven applications, contributing to improved language understanding, search functionality, code execution, data security, and machine learning capabilities.

Reference:

- ✓ Lab Lectures
- ✓ CHAT GPT