



Khulna University of Engineering and Technology

REPORT on Lexical Analysis of My Language

Course Name: Compiler Design Laboratory,

Course Code: CSE 3212

Submitted From

Name: Naimur Rahman

Roll: 1907031

Section: A

Dept of CSE, KUET

Submitted to

Nazia Jahan Khan Chowdhury
Assistant Professor

Dipannita Biswas
Lecturer

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:

Arithmetic_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}) [] * {Arithmetic_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

`\\/*(*|*[\\V])**`

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:

```
1  %{
2
3  // header files
4  #include<string.h>
5  #include<stdio.h>
6  #include<math.h>
7  #include<stdlib.h>
8
9  // keyword and datatype string collection
10 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};
11 const char* datatype[] = {"vari","varf","varc",NULL};
12 const char* relation[] = {"ls","gr","le","ge","eq","ne",NULL};
13 const char* logic[] = {"&&","||",NULL};
14
15 // check if a new variable name is a keyword
16 int is_keyword(const char* word) {
17     for (int i = 0; keywords[i] != NULL; i++) {
18         if (strcmp(keywords[i], word) == 0) {
19             return 1;
20         }
21     }
22     return 0;
23 }
24
25 // check if it is a logical operator
26 int is_logic(const char* word) {
27     for (int i = 0; logic[i] != NULL; i++) {
28         if (strcmp(logic[i], word) == 0) {
29             return 1;
30         }
31     }
32     return 0;
33 }
34
35 // check if it is a relational operator
36 int is_relation(const char* word) {
37     for (int i = 0; relation[i] != NULL; i++) {
38         if (strcmp(relation[i], word) == 0) {
39             return 1;
40         }
41     }
42     return 0;
43 }
44
45 // check if a new variable is a datatype
46 int is_datatype(const char* word) {
47     for (int i = 0; datatype[i] != NULL; i++) {
48         if (strcmp(datatype[i], word) == 0) {
49             return 1;
50         }
51     }
52     return 0;
53 }
54
55 // symbol table linked list that adds new values into head
```

```

56 struct Symbol_Table {
57     char* name;
58     char* datatype;
59     char* value;
60     struct Symbol_Table* next;
61 };
62
63 // head pointer for symbol table linked list
64 struct Symbol_Table* head = NULL;
65
66 // insert a new variable into symbol table
67 void insert_variable(const char* name, const char* datatype, const char* value) {
68     struct Symbol_Table* new_symbol = malloc(sizeof(struct Symbol_Table));
69     new_symbol->name = strdup(name);
70     new_symbol->datatype = strdup(datatype);
71     new_symbol->value = strdup(value);
72     new_symbol->next = head;
73     head = new_symbol;
74 }
75
76 // check if a variable is already declared
77 int is_declared(const char* name) {
78     struct Symbol_Table* tmp = head;
79     while (tmp != NULL) {
80         if (strcmp(tmp->name, name) == 0) {
81             return 1;
82         }
83         tmp = tmp->next;
84     }
85     return 0;
86 }
87
88 // update the value of a variable in the symbol table
89 int update_value(const char* name, const char* value) {
90     struct Symbol_Table* tmp = head;
91     while (tmp != NULL) {
92         if (strcmp(tmp->name, name) == 0) {
93             free(tmp->value); // free previous value
94             tmp->value = strdup(value); // update new value
95             return 1;
96         }
97         tmp = tmp->next;
98     }
99     return 0;
100 }
101
102 // variables needed for operations
103 int var_count=0, statement_count=0, function_count=0, header_count=0, temp_var_count=0, i=0;
104 int Unary_operator_count=0, Relational_operator_count=0, Assignment_operator_count=0, Logical_operator_count=0, Arithmetic_operator_count=0;
105 int mcmt_count=0, scmt_count=0, array_count=0;
106 int is_declared_check=0, conditional_count=0, loop_count=0;
107 %}
108
109 /* labels for future need */
110 %x WHILE_CONDITION
111 %x WHILE_BODY
112 %x FOR_CONDITION
113 %x FOR_BODY

```

```

114 %x IS_BODY
115 %x IS_CONDITION
116
117 /* defining regex */
118 char [a-zA-Z]
119 digit [0-9]
120 special [_@]
121 space " "
122 newline "\n"
123 tabline "\t"
124 datatype "vari"|"varf"|"varc"
125 void "void"
126 Arithmetic_operator "+"|"-"|"*"|" "/"|" $"
127 Logical_operator "&&"|"||"
128 Not "!"
129 Unary_operator "++"|"--"
130 Relational_operator "ls"|"gr"|"eq"|"ge"|"le"|"ne"
131 Assignment_operator "="|"+="|" -="|" /="|" *="
132 int [+ -]?[0-9]+
133 float [+ -]?[0-9]+[.][0-9]+([eE][+ -]?[0-9]+)?|[+ -]?[1-9]+[eE][+ -]?[0-9]+
134 string \"[A-Za-z0-9]+\"
135 ID [a-zA-Z][a-zA-Z0-9]*
136 scmt \\[/^\\n]*
137 mcmt \\[/^\\n]*\\[/^\\n]*
138
139 %%
140
141 {scmt} {
142     //single line comment detection
143     scmt_count++;
144     printf("Single Line Comment\n");
145 }
146
147 {mcmt} {
148     //multi line comment detection
149     mcmt_count++;
150     printf("Multiple Line Comment\n");
151 }
152
153
154 {datatype}[+ -]{ID}("=({ID}|{int}|{float}|{string}))?([ ]*, "[ ]*{ID}("=({ID}|{int}|
{float}|{string}))?)* {
155     //variable detection
156     char* token = strtok(yytext, " ,\t");
157     char* type;
158     int invalid = 0;
159     temp_var_count=0;
160
161     while (token != NULL) {
162         if (!is_datatype(token)) {
163             if (is_keyword(token)) {
164                 printf("ERROR: Reserved keyword can't be used as a variable name: %s\n",
token);
165                 invalid = 1;
166                 break;
167             }
168             if (is_declared(token)) {
169                 printf("ERROR: Variable '%s' is already declared.\n", token);
170                 invalid = 1;
171                 break;

```



```

172         } else {
173             char* init_value = strchr(token, '=');
174             if (init_value != NULL) {
175                 *init_value = '\0'; // Null-terminate the variable name
176                 const char* varName = token;
177                 if (is_declared(varName)) {
178                     printf("ERROR: Variable '%s' is already declared.\n", varName);
179                     invalid = 1;
180                     break;
181                 }
182                 if (is_keyword(varName)) {
183                     printf("ERROR: Reserved keyword can't be used as a variable name:
184 %s\n", varName);
185                     invalid = 1;
186                     break;
187                 }
188                 Assignment_operator_count++;
189                 printf("Assignment operator found\n");
190                 init_value++; // Move to the value part
191                 insert_variable(token, type, init_value);
192                 printf("Variable name: %s, Datatype: %s, Initialized with: %s\n",
193 token, type, init_value);
194             } else {
195                 insert_variable(token, type, "");
196                 printf("Variable name: %s, Datatype: %s\n", token, type);
197             }
198             temp_var_count++;
199             var_count++;
200         }
201     } else {
202         type = token;
203     }
204     token = strtok(NULL, " ,\t");
205 }
206 if (!invalid) {
207     printf("Total variables declared: %d\n", temp_var_count);
208 }
209 }
210 {ID}[ ]*"="[ ]*({ID}|{int}|{float}|{string}) {
211     //value updating of previously declared variables
212     char* var = strtok(yytext, "=");
213     char* val = strtok(NULL, "=");
214     if (!is_declared(var)) {
215         printf("ERROR: Variable '%s' not declared before assignment.\n", var);
216     }
217     else {
218         printf("ASSIGNMENT: Variable '%s' assigned value '%s'\n", var, val);
219         update_value(var, val); // Update the variable value
220     }
221 }
222 }
223 "array"[ ]+{datatype}[ ]+{ID}[ ]+"of"[ ]+{int} {
224     //array declaration
225     char* token = strtok(yytext, " \t");
226     char* type;
227     int invalid = 0;
228     while (token != NULL) {
229         if(strcmp(token, "array")==0){

```

```

230     }
231     else if (!is_datatype(token)) {
232         if (is_keyword(token)) {
233             printf("ERROR: Reserved keyword can't be used as a variable name: %s\n",
234 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");

```

```

288     BEGIN(INITIAL);
289 }
290
291 <WHILE_BODY>.* { printf("WHILE LOOP Code: %s\n", yytext); }
292
293
294 "loopf"[ ]+"with"[ ]+ {
295     //for loop syntax
296     BEGIN(FOR_CONDITION);
297 }
298
299 <FOR_CONDITION>{ID}[ ]+{int}"..."{int}[ ]+[-]?{int}[ ]* {
300     //for loop condition
301     printf("FOR LOOP: Condition\n");
302     loop_count++;
303     BEGIN(FOR_BODY);
304 }
305
306 <FOR_BODY>[ \t]*"begin" { printf("FOR LOOP: Start\n"); }
307
308 <FOR_BODY>[ \t]*"end" {
309     printf("FOR LOOP: End\n");
310     BEGIN(INITIAL);
311 }
312
313 <FOR_BODY>.* { printf("FOR LOOP Code: %s\n", yytext); }
314
315
316 "is"[ ]+ {
317     //if structure
318     printf("IS BLOCK\n");
319     is_declared_check=1;
320     conditional_count++;
321     BEGIN(IS_CONDITION);
322 }
323
324 "oris"[ ]+ {
325     //else if structure
326     if (is_declared_check) {
327         printf("ORIS BLOCK\n");
328         conditional_count++;
329         BEGIN(IS_CONDITION);
330     } else {
331         printf("ERROR: 'oris' without preceding 'is'\n");
332     }
333 }
334
335 "or"[ ]* {
336     //else structure
337     if (is_declared_check) {
338         printf("OR BLOCK\n");
339         conditional_count++;
340         BEGIN(IS_BODY);
341     } else {
342         printf("ERROR: 'or' without preceding 'is'\n");
343     }
344 }
345
346 <IS_CONDITION>({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|
{float}|{string})([ ]+{Logical_operator}[ ]+({ID}|{int}|{float}|{string})[ ]+

```

```

347 {Relational_operator}[ ]+({ID}|{int}|{float}|{string}))*[ ]* {
348     //if or else if condition check
349     char* token = strtok(yytext, " \t");
350     while (token != NULL) {
351         if(is_relation(token)){
352             Relational_operator_count++;
353             printf("Relational operator found\n");
354         }
355         else if(is_logic(token)){
356             Logical_operator_count++;
357             printf("Logical operator found\n");
358         }
359         token = strtok(NULL, " \t");
360     }
361     printf("IS CONDITION\n");
362     BEGIN(IS_BODY);
363 }
364 <IS_BODY>[ \t]*{" { printf("START OF CODE BLOCK\n");}
365
366 <IS_BODY>[ \t]*}" {
367     printf("END OF CODE BLOCK\n");
368     BEGIN(INITIAL);
369 }
370
371 "si" {
372     //denotes end of if block and similar to fi
373     if(is_declared_check){
374         printf("END OF IS BLOCK\n");
375         is_declared_check=0;
376         BEGIN(INITIAL);
377     }
378     else{
379         printf("ERROR: 'si' without preceding 'is'\n");
380     }
381 }
382 }
383
384 <IS_BODY>.* { printf("CODE: %s\n", yytext); }
385
386
387 ; {statement_count++;}
388
389
390 ({datatype}|{void})[ ]"main()"[ \n\t]*{" {
391     //check main function
392     function_count++;
393     printf("Main Function\n");
394 }
395
396 ({datatype}|{void})[ ]+"function"[ ]+{ID}"("({datatype}[ ]+{ID}([ ]*", "[ ]*{datatype}[ ]
397 {ID}))*?"")"[ \n\t]*{" {
398     //function declaration
399     function_count++;
400     printf("Function Declaration\n");
401 }
402
403 "#include<".*">" {
404     //header file

```

```

405     header_count++;
406     printf("Header File\n");
407 }
408
409
410 {ID}{Unary_operator} {
411     //operators check
412     Unary_operator_count++;
413     printf("Unary operator found\n");
414 }
415
416 {Unary_operator}{ID} {
417     Unary_operator_count++;
418     printf("Unary operator found\n");
419 }
420
421 ({ID}|{int}|{float}|{string})[ ]+{Relational_operator}[ ]+({ID}|{int}|{float}|{string}) {
422     Relational_operator_count++;
423     printf("Relational operator found\n");
424 }
425
426 {ID}[ ]*{Assignment_operator}[ ]*({ID}|{int}|{float}|{string}) {
427     Assignment_operator_count++;
428     printf("Assignment operator found\n");
429 }
430
431 ({ID}|{int}|{float}|{string})[ ]*{Logical_operator}[ ]*({ID}|{int}|{float}|{string}) {
432     Logical_operator_count++;
433     printf("Logical operator found\n");
434 }
435
436 {Not}[ ]*({ID}|{int}|{float}|{string}) {
437     Logical_operator_count++;
438     printf("Logical operator found\n");
439 }
440
441 ({ID}|{int}|{float}|{string})[ ]*{Arithmetic_operator}[ ]*({ID}|{int}|{float}|{string}) {
442     Arithmetic_operator_count++;
443     printf("Arithmetic operator found\n");
444 }
445
446
447 . {
448     //ignores everything else
449 }
450
451 %%
452
453 int yywrap()
454 {
455     return 1;
456 }
457
458 int main()
459 {
460     yyin = fopen( "sample.txt", "r" );
461     yylex();
462     printf("%d variables declared\n", var_count);
463     printf("%d arrays declared\n", array_count);
464     printf("%d functions declared\n", function_count);

```



```
465     printf("%d headers declared\n", header_count);
466     printf("%d Unary operators\n", Unary_operator_count);
467     printf("%d Relational operators\n", Relational_operator_count);
468     printf("%d Assignment operators\n", Assignment_operator_count);
469     printf("%d Logical operators\n", Logical_operator_count);
470     printf("%d Arithmetic operators\n", Arithmetic_operator_count);
471     printf("%d single line comments\n", scmt_count);
472     printf("%d multiple line comments\n", mcmt_count);
473     printf("%d conditional statements\n", conditional_count);
474     printf("%d loops\n", loop_count);
475     printf("%d statements\n", statement_count);
476     return 0;
477 }
```

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 Arithmetic 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