Test 2

Hyunki Lee

Panther# 002-34-4677

1.

Share link

First link is Copy repl link.

Second lint is inviting link just in case.

https://repl.it/@todok4636/PLCQ1HyunkiLee

https://repl.it/join/xioryiai-todok4636

Table for tokens

Token Symbol	Token name	Unique Number
$	DOLLAR	01
@	AT	02
%	PERCENT	03
\	DOUBLE_QUOTE	04
‘	SINGLE_QUOTE	05
u	UNSIGNEDL	06
U	UNSIGNEDC	07
l	LONGL	40
L	LONGC	41
ll	LONG_LONGL	10
LL	LONG_LONGC	11
.	DOT	42
f	FLOATING_SUFFL	12
F	FLOATING_SUFFC	13
l	FLOATING_SUFLL	14
L	FLOATING_SUFLC	15
+	PLUS	16
=	EQUAL	17

-	MINUS	18
/	DIVISION	19
*	MULTI	20
%	MODULO	21
!	NOT	22
:	OPEN_CODE	23
;	CLOSE_CODE	24
(	OPEN_FUNC	25
)	CLOSE_FUNC	26
++	INCRE	27
--	DECRE	28
&&	AND	29
||	OR	30

Source Code

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

//define each tokens as unique number

#define DOLLAR 01
#define AT 02
#define PERCENT 03

#define DOUBLE_QUOTE 04

#define SINGLE_QUOTE 05
```

```
#define UNSIGNEDL 06
#define UNSIGNEDC 07
#define LONGL 40
#define LONGC 41
#define LONG_LONGL 10
#define LONG_LONGC 11

#define DOT 42
#define FLOATING_SUFFL 12
#define FLOATING_SUFFC 13
#define FLOATING_SUFLL 14
#define FLOATING_SUFLC 15


#define PLUS 16
#define EQUAL 17
#define MINUS 18
#define DIVISION 19
#define MULTI 20
#define MODULO 21
#define NOT 22
#define OPEN_CODE 23
#define CLOSE_CODE 24
#define OPEN_FUNC 25
#define CLOSE_FUNC 26
#define INCRE 27
#define DECRE 28
```

```
#define AND 29
#define OR 30
#define SPACE 31

#define IDENTIFIER 32
#define DIGIT 33
#define FLOATING 34

#define TOTAL 35

int numOfToken = 0;

//Create struct frame
struct list{
	char pick[35];
	int token;
};
//Type struct elements
struct list reference[TOTAL] = {
	{"$", DOLLAR},
	{"@", AT},
	{"%", PERCENT},

	{"\"", DOUBLE_QUOTE},

	{"'", SINGLE_QUOTE},

	{"u", UNSIGNEDL},
{"U", UNSIGNEDC},
{"l", LONGL},
{"L", LONGC},
{"ll", LONG_LONGL},
{"LL", LONG_LONGC},

{".", DOT},
{"f", FLOATING_SUFFL},
{"F", FLOATING_SUFFC},
{"l", FLOATING_SUFLL},
{"L", FLOATING_SUFLC},

{"+", PLUS},
{"=", EQUAL},
{"-", MINUS},
{"/", DIVISION},
{"*", MULTI},
{"%", MODULO},
{"!", NOT},
{":", OPEN_CODE},
{";", CLOSE_CODE},
{"(", OPEN_FUNC},
{")", CLOSE_FUNC},
{"++", INCRE},
{"--", DECRE},
{"&&", AND},
{"||", OR},
};
// tokens category that where the tokens belong to (This is related to reference
struct)
char tokenCategory[TOTAL+1][50]={
" ",
"Perl Style ID",
"Perl Style ID",
"Perl Style ID",

"Java Style String Literals",

"C Style Character Literals",

"C Style Integer Literals",
"C Style Integer Literals",
"C Style Integer Literals",
"C Style Integer Literals",
"C Style Integer Literals",
"C Style Integer Literals",

"C Style Floating point Literals",
"C Style Floating point Literals",
"C Style Floating point Literals",
"C Style Floating point Literals",
"C Style Floating point Literals",

"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",
"Non-Alphanumeric Special Symbols",

"Perl Style ID",
"C Style Integer Literals",
"C Style Floating point Literals",

};

//Finding tokens
int parsing(char lex[]){
	int i;
	for (i = 0; i < TOTAL; i++){
		if(strcmp(lex,reference[i].pick) == 0){
			return reference[i].token;
		}
	}
```

```
        return IDENTIFIER;

}



//function for printing result
void printResult(int num, char temp[]){
        printf("\n");
        printf("\t\t\t\t%d\t\t\t%s is %s\n",num,temp,tokenCategory[num]);
}

//function for lexical analyzer
void lexi(char temp[], int tempLength){
        int i,j,k;
        int line = 2;
        char c,next;
        char lex[30];
        char z[300];
        //Tokenizing.
        for(i=0; i < tempLength;){
                c = temp[i];

                for(j=0; j<10; j++){
                        z[j]='\0';
                }

                j=0;
                z[j++]=temp[i];
```

```
z[j]='\0';

//Using switch-case to distinguish each token
switch(c){
        case' ':
                i++;
                printf("\n");
                break;

        case'\t':
                i++;
                printf("\n");
                break;

        case '$':
                i++;
                printResult(DOLLAR,z);
                break;

        case '@':
                i++;
                printResult(AT,z);
                break;

        case '%':
                i++;
                printResult(PERCENT,z);
                break;
```

```
case '\"':
        i++;
        printResult(DOUBLE_QUOTE,z);
        break;

case '.':
        i++;
        printResult(DOT,z);
        break;

case '-':
        next = temp[++i];
        if(next=='-'){
                i++;
                printResult(DECRE,z);
                break;
        }else{
                i++;
                printResult(MINUS,z);
                break;
        }

case '+':
        next = temp[++i];
        if(next=='+'){
                i++;
                printResult(INCRE,z);
                break;
            }else{
                i++;
                printResult(PLUS,z);
                break;
            }


        case '=':
            i++;
            printResult(EQUAL,z);
            break;

        case '/':
            i++;
            printResult(DIVISION,z);
            break;

        case '*':
            i++;
            printResult(MULTI,z);
            break;

        case '!':
            i++;
            printResult(NOT,z);
            break;
```

```
case ':':
        i++;
        printResult(OPEN_CODE,z);
        break;

case ';':
        i++;
        printResult(CLOSE_CODE,z);
        break;

case '(':
        i++;
        printResult( OPEN_FUNC,z);
        break;

case ')':
        i++;
        printResult(CLOSE_FUNC,z);
        break;

case '&':
        i++;
        printResult(AND,z);
        break;

case '|':
        i++;
        printResult(OR,z);
```

```
                    break;

                //Set the default tokens. The two categories are alphabet and
digits
                default:
                    if(isalpha(temp[i])){
                        k =0;
                        while(isalpha(temp[i])){
                            lex[k++] = temp[i++];
                        }
                        lex[k]='\0';
                        printResult(parsing(lex),lex);
                        break;

                    //if the digit comes first. print an error.
                    } if(isdigit(temp[0])){
                        printf("\n");
                       printf("Did not match with given Condition");

                    }
                        //if the digit comes and alphabet comes next,
then print an error
                        if(isdigit(temp[i])){
                            if(isalpha(temp[i+1])){
                                printf("\n");
                                printf("Did   not   match   with
given Condition");

                            }
```

```
k = 0;

while(isdigit(temp[i])){
        lex[k++] = temp[i++];
}
if((temp[tempLength]=='u')              ||
(temp[tempLength]=='U')              ||          (temp[tempLength]=='l')              ||
(temp[tempLength]=='L')){

        lex[k++] = temp[i++];
        break;
}
if(temp[i] !='.'){
        lex[k] = '\0';
        printResult(DIGIT,lex);
        break;
}
//Floating number conditions.
else            if(temp[i]=='.'            &&
isdigit(temp[i+1])){

        int check=0;
        lex[k++]='.';
        i++;
        while(isdigit(temp[i])){
                lex[k++] = temp[i++];
        }
        if((temp[i] == 'e') || (temp[i] ==
'E')){

                lex[k++] = temp[i++];
        }
```

```
                                        if((temp[tempLength-1]=='f')
|| (temp[tempLength-1]=='F') || (temp[tempLength-1]=='l') || (temp[tempLength-
1]=='L')){
                                            lex[k++] = temp[i++];
                                        }

                                        while(isdigit(temp[i])){
                                            if(check==0)
                                            lex[k++] = temp[i];
                                            i++;
                                        }
                                        if(check==1){
                                            break;
                                        }
                                        lex[k] = '\0';
                                        printResult(FLOATING,lex);
                                        break;
                                    }
                                }

                        else if(temp[i]=='\n'){
                            i++;
                            if(temp[i+1] != '\n'){
                                printf("\n\nLine No.=%d\n",line++);
                                printf("\n");
                            }
                        }
                        else if(temp[i]=='\t' || temp[i]==' '){
```

```
                    i++;
                }

                else{
                    i++;
                }

            }

        }
        for(i=0; i<10;i++){
            z[i]='\0';
        }
}

int main(){
        //ready to read file, create file pointer.
        FILE *fp;

        int i=0;
        int c;
        char temp[300];
        int tempLength;
        char f[30];

        printf("Test2 Question1\n");
        printf("\n");
        // open file
```

```
	fp = fopen("input.txt","r");
	// print an error when file does not exit
	if(fp == NULL){
		printf("Need a text file, the name should be 'input.txt'");
		printf("\n");
	}
	// check each character until end of file
	while((c = getc(fp)) != EOF){
		temp[i++] = c;
	}
	tempLength = i;
	//close file
	fclose(fp);
	printf("\nLine No.\t\t\tToken ID\t\tExplain\n");

	printf("Line No.=1\n");



	lexi(temp, tempLength);
	return 0;
}
```

2.

I reduce the size of array in replit because replit does not handle the size.

Share link

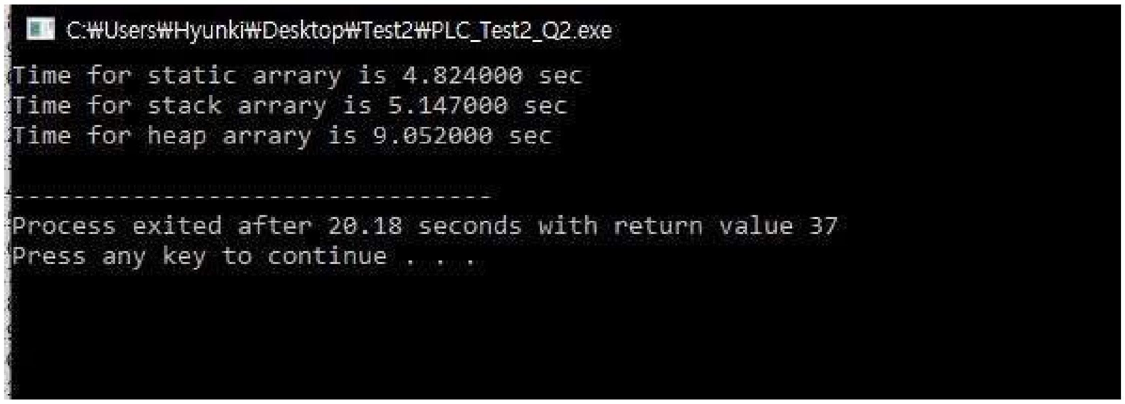
First link is Copy repl link.

Second lint is inviting link just in case

https://repl.it/@todok4636/PLCQ2HyunkiLee

https://repl.it/join/xvblgpgp-todok4636

Output



```
C:₩Users₩Hyunki₩Desktop₩Test2₩PLC_Test2_Q2.exe
Time for static arrary is 4.824000 sec
Time for stack arrary is 5.147000 sec
Time for heap arrary is 9.052000 sec

--------------------------------
Process exited after 20.18 seconds with return value 37
Press any key to continue . . .
```

Explain the result

The static array is the fastest array, and the heap array is the slowest array.

Static variables are addressed directly. It means that static variables have run-time efficiency. In the stack variables case, the variables' declarations are elaborated however, their types are statically bound. Thus, the runtime is not faster than static array. In the heap array case, we need to access them through pointers, and the collection in the heap are disorganized. Therefore, not only cost of references to the variables but cost of runtime to execute a program.

Source Code

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>
```

```
#define ARR_SIZE 30000



//static array
int staticArray(){
	int i;
	static int stArr[ARR_SIZE];

	for(i=0; i<ARR_SIZE; i++){
		stArr[i] = 3;
	}
}



//stack array
int stackArray(){
	int i;
	int stacArr[ARR_SIZE];

	for(i=0; i<ARR_SIZE; i++){
		stacArr[i] = 3;
	}
}

//heap array
int heapArray(){
	int i;
	int *hepArray = (int *) malloc(sizeof(int) * ARR_SIZE);

	for(i=0; i<ARR_SIZE; i++){
		hepArray[i] = 3;
	}
}


//main; count time;
int main(){
	clock_t time;
	int i;
	time = clock();
	//count time for static array
	for(i=0; i<100000;i++){
		staticArray();
	}
	time = clock() - time;
	printf("Time           for           static           arrary           is           %f
sec\n",(float)time/CLOCKS_PER_SEC);


	time = clock();
	//count time for stack array
	for(i=0; i<100000;i++){
		stackArray();
	}
	time = clock() - time;
	printf("Time for stack arrary is %f sec\n",(float)time/CLOCKS_PER_SEC);



	time = clock();
	//count time for heap array
	for(i=0; i<100000;i++){
		heapArray();
	}
	time = clock() - time;
	printf("Time for heap arrary is %f sec\n",(float)time/CLOCKS_PER_SEC);

}
```

3.

EBNF

<prefix>→

<operator> {<operator>} {operand} [<operator>](<operand> <operand>)

<prefix>→

<operator> {<operator>} {({operand} [<operator>])} (<operand> <operand>)

<operator> → + | - | * | / | %

Top-down parsers – The syntax is related with a leftmost derivation, and this build visits each node first (preorder).

Recursive-descent parser – The syntax can be represented by EBNF/BNF.

LL algorithms – The input is read from left to right, and this parser is a leftmost derivation.

4.

Static Scoping – In static scoping case, Lexical analysis part is related. The static scoping is always related to top level environment which means that the static scope of variable is determined before execution. Thus, the static scoping of variable is related with lexical analysis to determine the reference environment.

Dynamic Scoping – In dynamic scoping case, symbol table is related. The dynamic scoping is based on the step that subprograms are called. It means that dynamic scoping is updated when subprograms are executed (keep recent information). When a variable is referencing each time, the symbol table is working to check correction. Thus, the dynamic scoping of variable is related with symbol table to determine the reference environment.

5.

In C programming, we can use ‘define’ and ‘struct’ functions to reserve words. First, using ‘define’ to decide unique numbers for tokens. Second, using ‘struct’ function to reserve words. The skeleton of ‘struct’ is that

```
struct list{
char pick[35];
int token;
};
struct list reference[TOTAL] = {
{"$", DOLLAR},
};
```

Thus, we add some words in ‘struct’ to reserve the words. Using the reserved words is that we need to add ‘case’ for the words in ‘switch’. The example of the ‘switch – case’ is that

```
case '$':
i++;
printResult(DOLLAR,z);
break;
```

6.

Share link

First link is Copy repl link.

Second lint is inviting link just in case.

https://repl.it/@todok4636/PLCQ6HyunkiLee

https://repl.it/join/enfphvyj-todok4636

a. Java if statement

<ifstmt>→ if (<boolexpr>) <statement> [else <statement>]

```
void ifstmt() {
  if (nextToken != IF_CODE)
          error();
  else {
          lex();
          if (nextToken != LEFT_PAREN)
                  error();
          else {
                  lex();
                  boolexpr();
                  if (nextToken != RIGHT_PAREN)
                          error();
                  else {
                          lex();
```

```
                        statement();
                        if (nextToken == ELSE_CODE) {
                                lex();
                                statement();
                        }
                }
        }
  }
}
```

b. Java while statement

<while_stmt> → while "("<boolexpr>")" <statement>

```
                Void whilestmt(){
                        if (nextToken != WHILE_CODE)
                                error();
                        else {
                                lex();
                                if (nextToken != LEFT_PAREN)
                                        error();
                                else {
                                        boolexpr();
                                if (nextToken != RIGHT_PAREN)
                                        error();
                                        else {
                                                statement();
                                        }
```

```
                    }
                }
```

c. logical/mathematical expression

<lexical>→<perlid> {<perlid>} {<literals>} <non-alphanumeric> <non-alphanumeric> {<non-alphanumeric>}

<perlid >→ @ | $ | % | a | b | c | d | e | f | g | h | i | j | k | l | n | m | o | p | q | r | s | t | u | v | w | x | y | z

<literals>→ <java-string> | <c-integer> | <c-character> | <c-floating>

<java-string>→ ”| “

<c-integer>→ (<decimal-constant>[<suffix>]) | (<octal-constant>[<suffix>]) | (<hexadecimal-constant>[<suffix>])

<decimal-constant>→ nonzero-digit | decimal-constant-digit

<octal-constant>→ 0 | octal-constant octal-digit

<hexadecimal-constant>→ (hexadecimal-prefix hexadecimal-digit) | (hexadecimal-constant hexadecimal-digit)

<suffix>→ <unsigned-suffix>[<long-suffix>]

<suffix>→ <unsigned-suffix>[<long-long-suffix>]

<suffix>→ <long-suffix>[<unsigned-suffix>]

<suffix>→ <long-long-suffix>[<unsigned-suffix>]

<unsigned-suffix>→ u | U

<long-suffix>→ l | L

<long-long-suffix>→ ll | LL

<c-character>→ ‘|’

<c-floating>→ <fractional-constant>[<exponent-part>][<floating-suffix>]

<c-floating>→ <digit-sequence><exponent-part>[<floating-suffix>]

<fractional-constant>→ [<digit-sequence>] . <digit-sequence>

<fractional-constant>→ . <digit-sequence>

<exponent-part>→ e [<sign>] <digit-sequence>

<exponent-part>→ E [<sign>] <digit-sequence>

<sign>→ + | -

<digit-sequence>→ <digit>

<digit-sequence>→<digit-sequence> <digit>

<floating-suffix>→ f | l | F | L

<non-alphanumeric>→ + | = | - | / | * | % | ! | (|) | ; | : | ++ | -- | && | '||'

```
void lexical(){
        perlid();
                if(nextToken = perlid().token){
                        perlid();
                }
                lex();
                if(nextToken = literals().token){
                        literals();
                }
                lex();
                nonAlphanumeric();
                nonAlphanumeric();
                lex();

                if(nextToken = nonAlphanumeric().token){
                        nonAlphanumeric();
        }
    }
```

d. mathematical assignment statement

<math>→ (<c-integer> | <c-floating>) <operator> (<c-integer> | <c-floating>) "=" (<c-integer> | <c-floating>)

<math>→ (<c-integer> | <c-floating>) <operator> (<c-integer> | <c-floating>) [(<operator> (<c-integer> | <c-floating>))] "=" (<c-integer> | <c-floating>)

<operator>→ * | - | / | % | +

<c-integer>→ (<decimal-constant>[<suffix>]) | (<octal-constant>[<suffix>]) | (<hexadecimal-constant>[<suffix>])

<decimal-constant>→ nonzero-digit | decimal-constant-digit

<octal-constant>→ 0 | octal-constant octal-digit

<hexadecimal-constant>→ (hexadecimal-prefix hexadecimal-digit) | (hexadecimal-constant hexadecimal-digit)

<suffix>→ <unsigned-suffix>[<long-suffix>]

<suffix>→ <unsigned-suffix>[<long-long-suffix>]

<suffix>→ <long-suffix>[<unsigned-suffix>]

<suffix>→ <long-long-suffix>[<unsigned-suffix>]

<unsigned-suffix>→ u | U

<long-suffix>→ l | L

<long-long-suffix>→ ll | LL

<c-character>→ '|'

<c-floating>→ <fractional-constant>[<exponent-part>][<floating-suffix>]

<c-floating>→ <digit-sequence><exponent-part>[<floating-suffix>]

<fractional-constant>→ [<digit-sequence>] . <digit-sequence>

<fractional-constant>→ . <digit-sequence>

<exponent-part>→ e [<sign>] <digit-sequence>

<exponent-part>→ E [<sign>] <digit-sequence>

<sign>→ + | -

<digit-sequence>→ <digit>

<digit-sequence>→<digit-sequence> <digit>

<floating-suffix>→ f | l | F | L

```
void math(){
        if(nextToken = cInteger().token){
                cInteger();
        }else{
                cFloating();
        }
        lex();
        operator();
        if(nextToken = cInterger().token){
                cInteger();
        }else if(nextToken = cFloating().token){
                cFloating();
        }else{
                lex();
        }
        if(nextToken != '='){
                error();
        }else if(nextToken = cInteger()){
                cInteger();
        }else{
                cFloating();
        }
}
```

7.

RDA usually creates a top-down order parse tree, and EBNF is appropriated for RDA. The general limitation of RDA is that RDA cannot handle left recursive grammar. It means that the RDA with left recursive grammar will not escape the infinity loop. (For instance, <rule1> → <rule1> <statements>)

Thus, we must avoid the left recursive case to create RDA style statements. The RDA process would be like that

1.checking token. If the token does not encounter a condition, then an error will be occurred.

2. If the token meets a condition, calling lex to get to a next token.

3. Repeat first and second steps to reach the goal.

This process is top-down method, and it is not left recursive way. Therefore, the process satisfies RDA rules to create statements.

8.

Using 'our' commend to set global variables. In static scoping case, using 'my' commend to set private variable, hence the declaration of variable with 'my' does not affect to global variable. It means that the static variable is stick with only the function that static variable is in. Otherwise, in dynamic scoping case, using 'local' commend to set temporary variable. The variable affects the value of the global variable. Thus, the most recent variable which is defined with 'local' variable replace the global variable which is defined at the first time.

Share link for perl code:

http://tpcg.io/Vqc0D5lP

Source code:

```
our $staticVal = 0; #set global variable

sub callA{
	return $staticVal;
}

sub staticCall{
	my $staticVal = 3; #'my' is private variable
	return callA();
}
print "Static scoping result is ";
print staticCall();



our $dynamicVal = 0; #set global value

sub callB{
	return $dynamicVal;
}

sub dynamicCall{
	local $dynamicVal = 3; #'local' is temporary variable to global variable
	return callB();
}
print "\n";
print "Dynamic scoping result is ";
print dynamicCall();
```