

Compiler Project 1

20172675 이해인

20171666 고재원

To making a Lexical Analyzer, We must have to 1) define Tokens, 2) make Regular Expressions, 3) make an Automata, 4) Also make a program. So, the contents are following :

- Contents -

1.	Making Tokens with given Lexical Specification	
2.	Regular Expression of Tokens	
3.	Making NFA / DFA from Regular Expression	
4.	Making NFA to DFA using Subset Construction	13p
5.	Automata to Program (+ the Priority of '-')	15p
6.	Test Result	21p
7.	Short Review	27p

1. Making Tokens with given Lexical Specification

From Lexical Specification, we can define some Tokens which consist of the given Specification. Here are the Tokens and the contents of the tokens.

* Variable type :

(token name) (token value)
VTYPE : int / char / boolean / string

* Signed Integer :

S_INTEGER : -231232 / 0 / 323 / ...

* Single Character :

SINGLE_CHAR : '3' / 'v' / 'E' / ' ' / ...

* Boolean String :

TRUE : true

FALSE : false

* Literal String :

LIT_STR : "abcdef2" /
"paganini/liszt" / "010-1234-5678" / ...

* Identifiers of variables and functions :

IDENT : _merge_ / _quick_look / main / ...

* Keyword for special statements :

IF : if

ELSE : else

CLASS : class

RETURN : return

* Operator :

OPERATOR : $+ / - \times //$

* Assignment operator :

ASSIGN : $=$

* Comparison :

COMPARISON : $> / < / \geq / \leq / == / !=$

* Terminating symbol :

TERMINATE : $;$

* Pair Symbol :

LSB : $[$

RSB : $]$

LCB : $\{$

RCB : $\}$

LPAREN : $($

RPAREN : $)$

* Separating input arguments in functions :

COMMA : $,$

• A non-empty sequence of ~~wh~~ t , ~~wh~~ n , and blanks :

WHITESPACE : $\backslash t, \backslash n$

2. Regular Expression of Tokens

In this part, We make REs from characteristics of defined Tokens.

! Before we start !

1. digit = $0|1|2|3|4|5|6|7|8|9$
2. single char = $a|b|c|d|..|0|1|2|3|4|5|..|!|@|\#|\$|\%|....$
which defined in ASCII code. (number: 0~126)
3. letter = $a|b|c|...|A|B|C|D|...|Z$
4. epsilon = ϵ

< Regular Expression of Tokens >

* Variable type :

$V_TYPE = int | char | string | boolean$

* Signed Integer :

$S_INTEGER = (-|\epsilonpsilon)(1|2|...|9)(0|1|...|9)^*$

* Single character :

$SINGLE_CHAR = 'single\ char'$

* Boolean string :

TRUE : true

FALSE : false

* Literal String :

LIT-STR = "(single char)⁺"

* Identifiers of variables and functions :

IDENT = "(_ | letter) (letter | digit | _)^{*}"

* Keyword for special statements :

IF : if

ELSE : else

CLASS : class

RETURN : return

* Operator :

OPERATOR = + | - | / | *

* Assignment operator :

ASSIGN = =

* Comparison :

COMPARISON = > | < | >= | <= | == | !=

* Terminating symbol :

TERMINATE = ;

* Pair Symbol:

LSB = [

RSB =]

LCB = {

RCB = }

LPAREN = (

RPAREN =)

* Separating input arguments in functions:

COMMA = ,

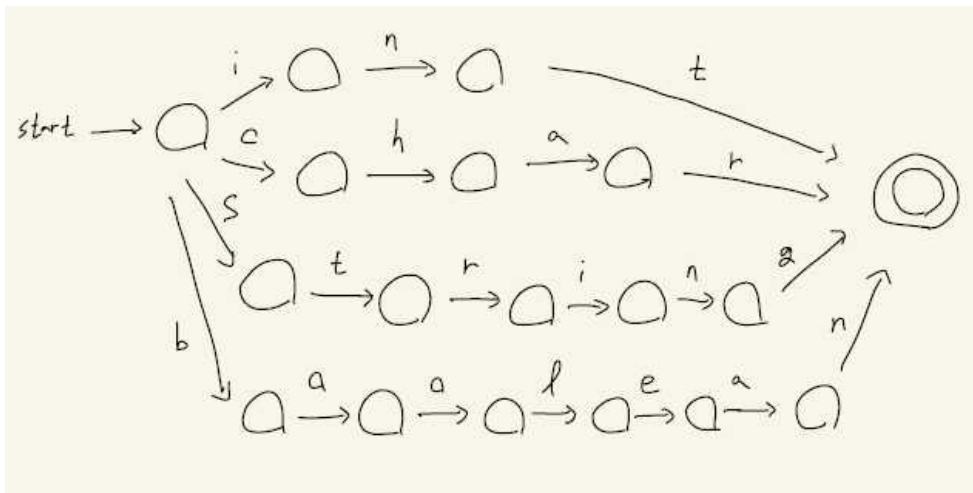
* Blank, \t, \n:

WHITESPACE = blank | \t | \n

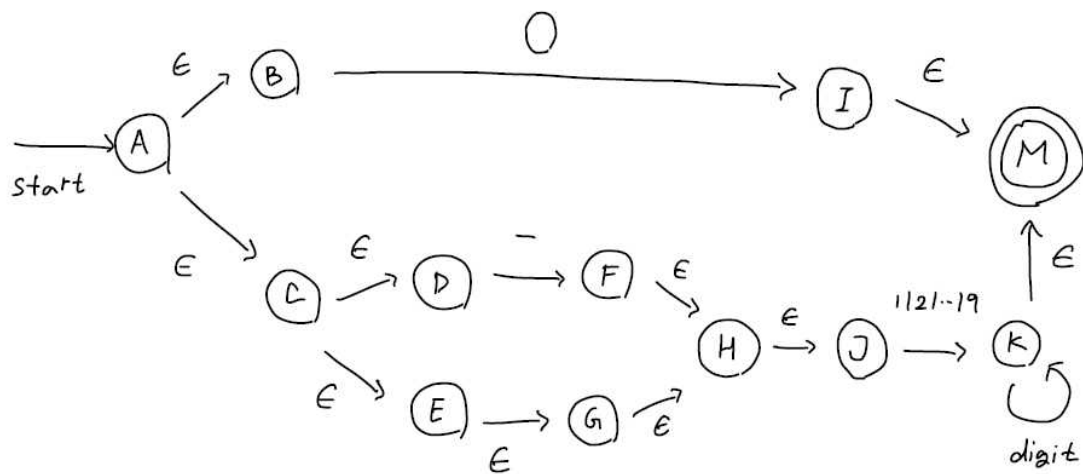
3. Making NFA / DFA from Regular Expression

Here are the Regular Expressions of the Tokens defined in part 2. Some of RE don't have any epsilon or same input for 2 way. So that kind of REs will be made to DFA, not NFA.

1. vTYPE

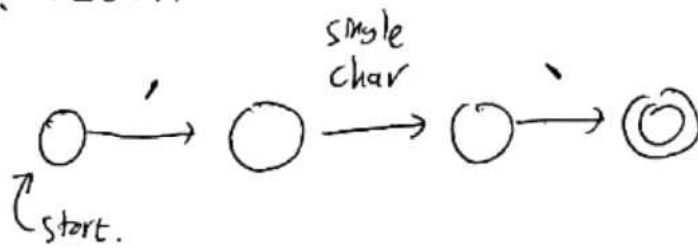


2. S_INTEGER

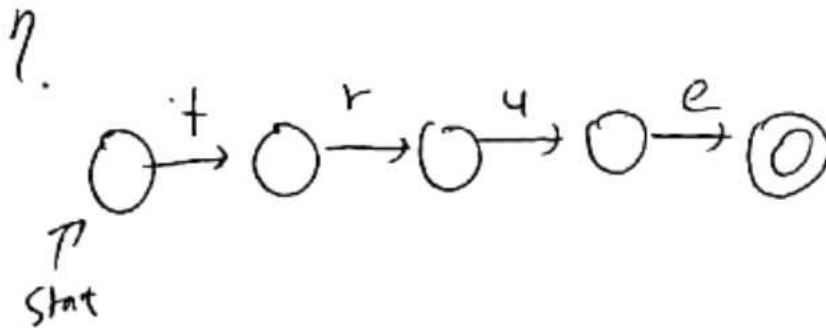


3. SINGLE_CHAR

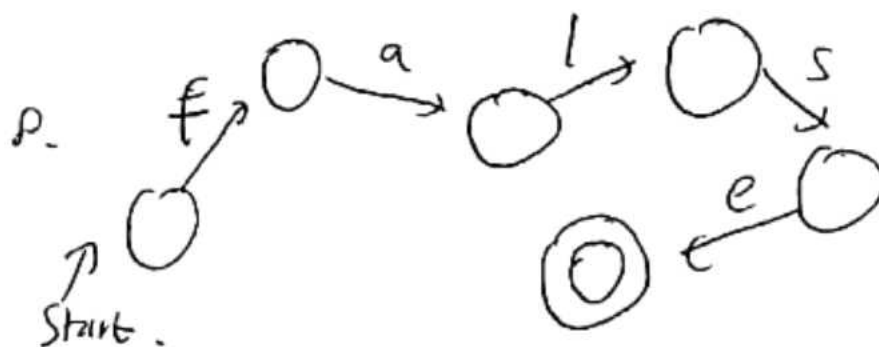
6. S_char.



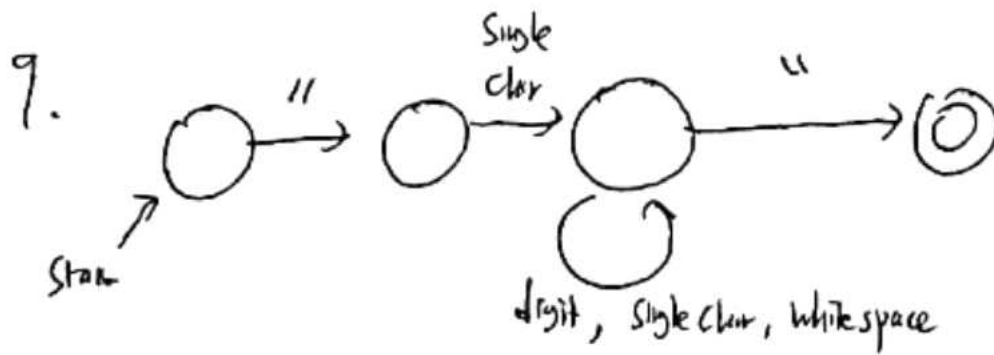
4. TRUE



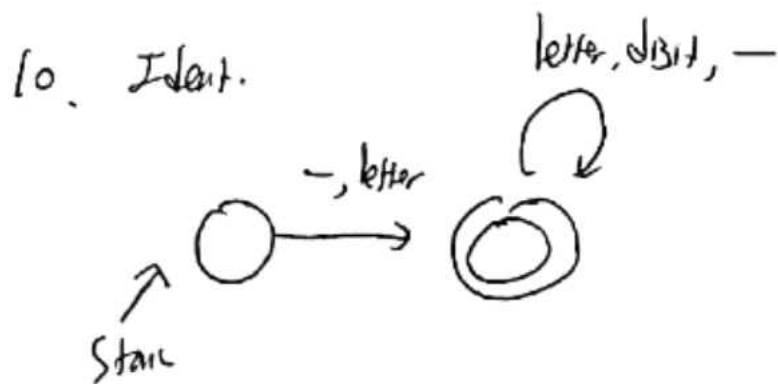
5. FALSE



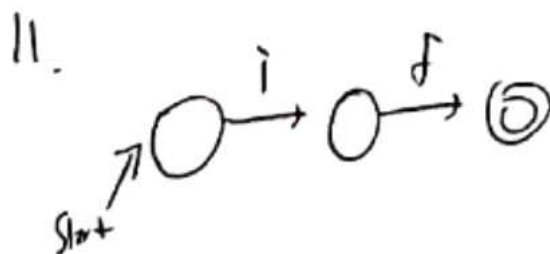
6. LIT_STR



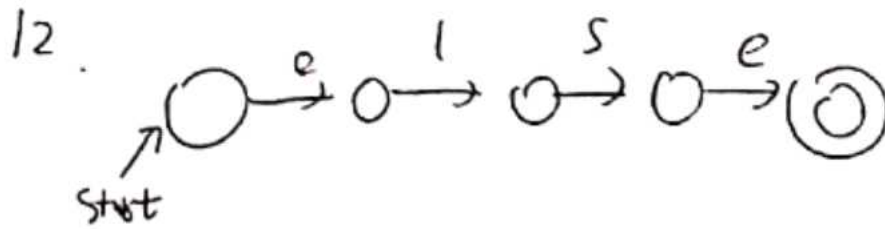
7. IDENT



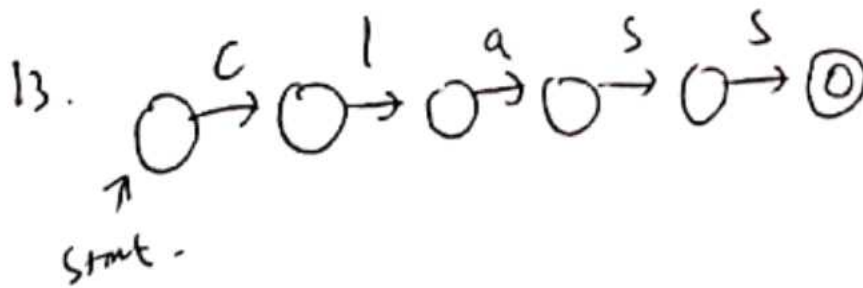
8. IF



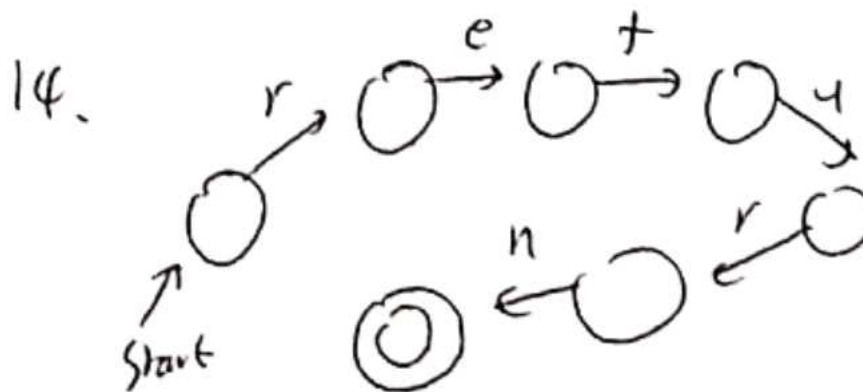
9. ELSE



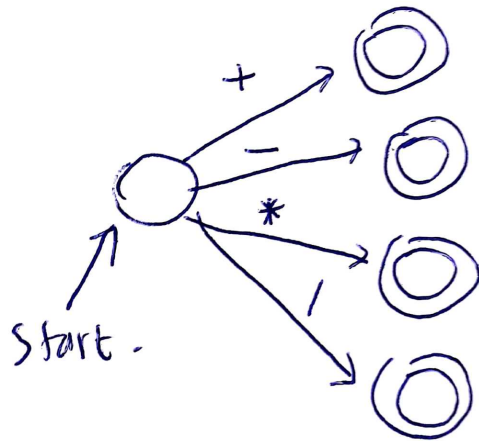
10. CLASS



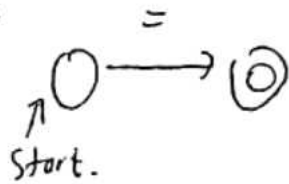
11. RETURN



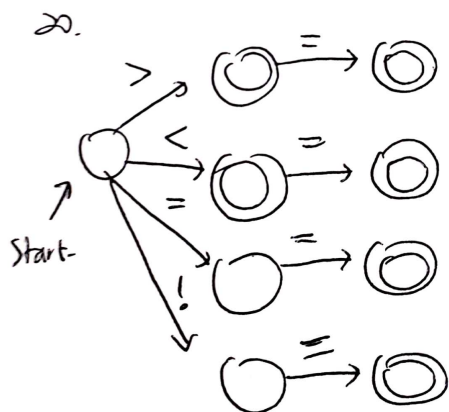
12. OPERATOR



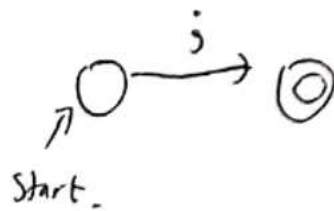
13. ASSIGN



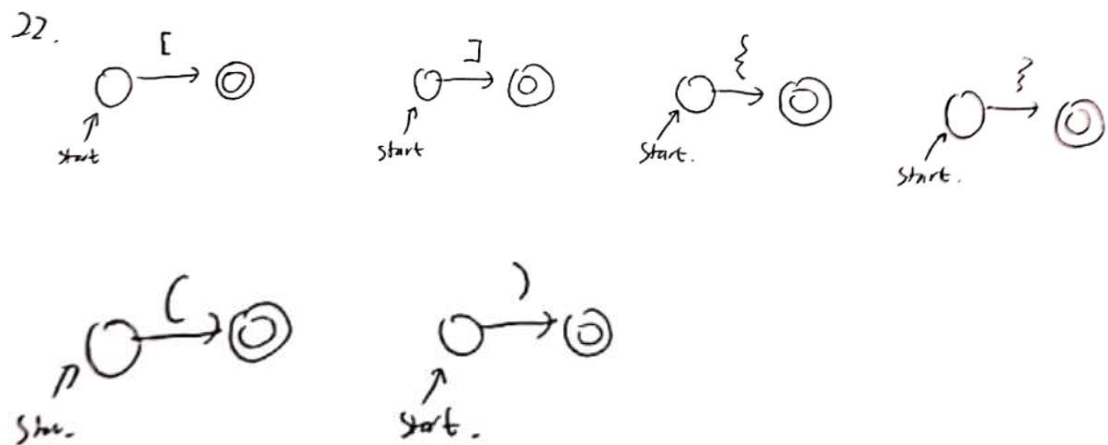
14. COMPARISON



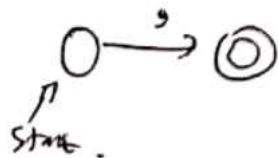
15. TERMINATE



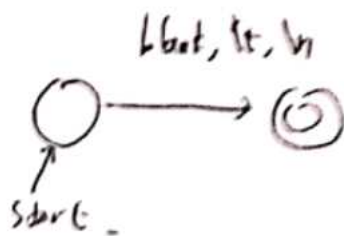
16. LSB, RSB, LCB, RCB, LPAREN, RPAREN



17.COMMA



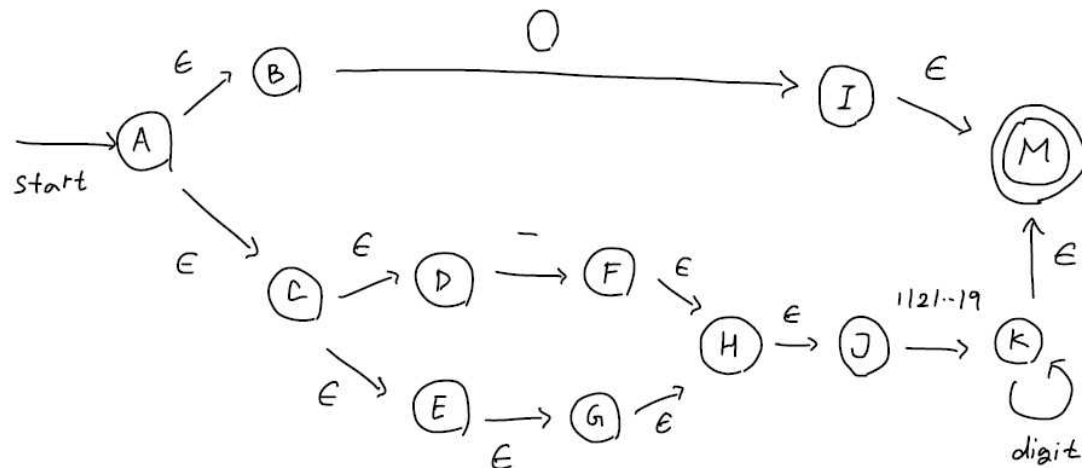
18. WHITESPACE



4. Make NFA to DFA using Subset Construction

By using Subset Construction, We can make NFA to DFA.

S_INTEGER :



1. Closure of start position

$$T_0 = \epsilon\text{-closure}(A) = \{A, B, C, D, E, G, H, J\}$$

$$T_1 = \epsilon\text{-closure}(\delta(T_0, 0)) = \epsilon\text{-closure}(I) = \{I, M\}$$

$$T_2 = \epsilon\text{-closure}(\delta(T_0, -)) = \epsilon\text{-closure}(F) = \{F, H, J\}$$

$$T_3 = \epsilon\text{-closure}(\delta(T_0, 1121..19)) = \epsilon\text{-closure}(K) = \{K, M\}$$

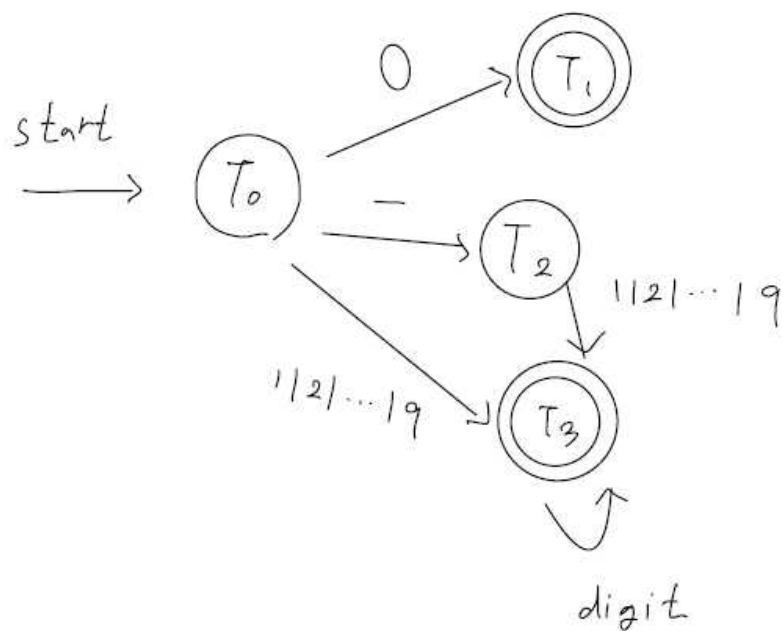
$$\epsilon\text{-closure}(\delta(T_2, 1121..19)) = \epsilon\text{-closure}(K) = \{K, M\}$$

$$\epsilon\text{-closure}(\delta(T_3, \text{digit})) = \epsilon\text{-closure}(K) = \{K, M\}$$

2. Subset Construction

	0	-	1 2 ... 9	digit
T_0	T_1	T_2	T_3	\emptyset
T_1	\emptyset	\emptyset	\emptyset	\emptyset
T_2	\emptyset	\emptyset	T_3	\emptyset
T_3	\emptyset	\emptyset	\emptyset	T_3

3. Result DFA of S_INTEGER



5. Automata to Program

Since we can't use the Library like Regex, We have to think.
What's the way to define the Token of the lexeme?

To solve this problem, we have to think about how are the words made and how can we know the Token of the words.

1. Gathering of characters that have not any means made the word.
2. We can assure the Token of the lexeme because as we see the first letter(character), we can classify the Token.

for example, think about the lexeme: asdf34324.

the first letter is 'a', so it can't be an integer. it might be an identifier or something. So we can guess it.

If the computer can think like that, we can make a computer classify the lexeme by looking at the first letter of the lexeme.

=====

Look at the **first Letter** of the lexeme:

if the Letter is a **Alphabet** :

 this lexeme's Class is LETTER

else if the Letter is a **Number** :

 this lexeme's Class is DIGIT

else :

 this lexeme's Class is UNKNOWN

(Of course, we have to classify it more.)

=====

Then what token can be classified to

DIGIT / LETTER / UNKNOWN?

From the given Specified form, we can classify Tokens into Three Classes.

```
=====
LETTER : if / else / while / return / class/ vtype / identifier /..
DIGIT : signed_integer
UNKNOWN : bracket series / operator / comparison / comma /
          terminator / literal_string / single_char
          /identifier (start with '_' ) / signed_integer (start with '-')
=====
```

So, by classifying the lexeme, we can reduce our choice to define Token. If lexeme is DIGIT, we just check the s_integer DFA to whether this lexeme is qualified to DFA or not. If it is not qualified, then it's an error. If it is qualified, then this lexeme is a signed integer. Then we just put these to symbol table or stream for save.

```
=====
Lexcal analyzer :
```

if Class is LETTER :

```
    check Identifier / vtype / .... DFA :
        if there is a DFA that pass the lexeme:
            lexeme belongs to that Token
        else:
            it's an error
```



```

else if Class is DIGIT :
    check s_integer DFA :
        if there is a DFA that pass the lexeme:
            lexeme belongs to that Token
        else:
            it's an error
else:
    check operator / comparison / .... DFA :
        if there is a DFA that pass the lexeme:
            lexeme is belong to that Token
        else:
            it's an error

```

after checking DFA,

if we can define the Token:

Then save it to the Symbol table.

=====

And think about the DFA. Do we have to check each DFA in Class?
 We can, but it's waste of time. So, We have some ideas.

How about Combine the DFAs that were in the same Class?

For example, If the Class of the lexeme is a LETTER, then we can
 check the combined DFA that can detect

if / else / while / return / class/ vtype / identifier /..
 all of Tokens in Class LETTER.

and Of course, combine the DFA is also not a hard work. In

program, We just use SWITCH to check the whole Tokens in same Class.

=====

SIMPLE COMBINED UNKNOWN DFA:

```
SWITCH ( one letter ):  
    case +: Token is OPERATOR  
    case *: Token is OPERATOR  
    ...  
    case :: Token is TERMINATE  
    case _:  
        Check more detail  
        ...  
        Token is IDENT  
    default:  
        ERROR
```

=====

Sudden Question : How to Define the '-'s Token?

It's a hard question. Because the answer is "It depends". We have think all the cases of '-', **also it makes the program very DIRTY....** but rule is rule so Let's think.

'-' can't define Token itself. it must look at the front lexeme's Token or look at the next letter.

From the announcement and natural rule, we define some rules :

=====

When we see '-' from the input string :

See ahead of '-',

1. if that lexeme's Token is a **signed integer** :
 : '-' is **operator** ex) input : 3544(digit) -

2. if that lexeme's Token is **identifier, if, while, return,,, :**
 : '-' is **operator** ex) input : return(return) - 3
 ex) input : chicken23 - wing

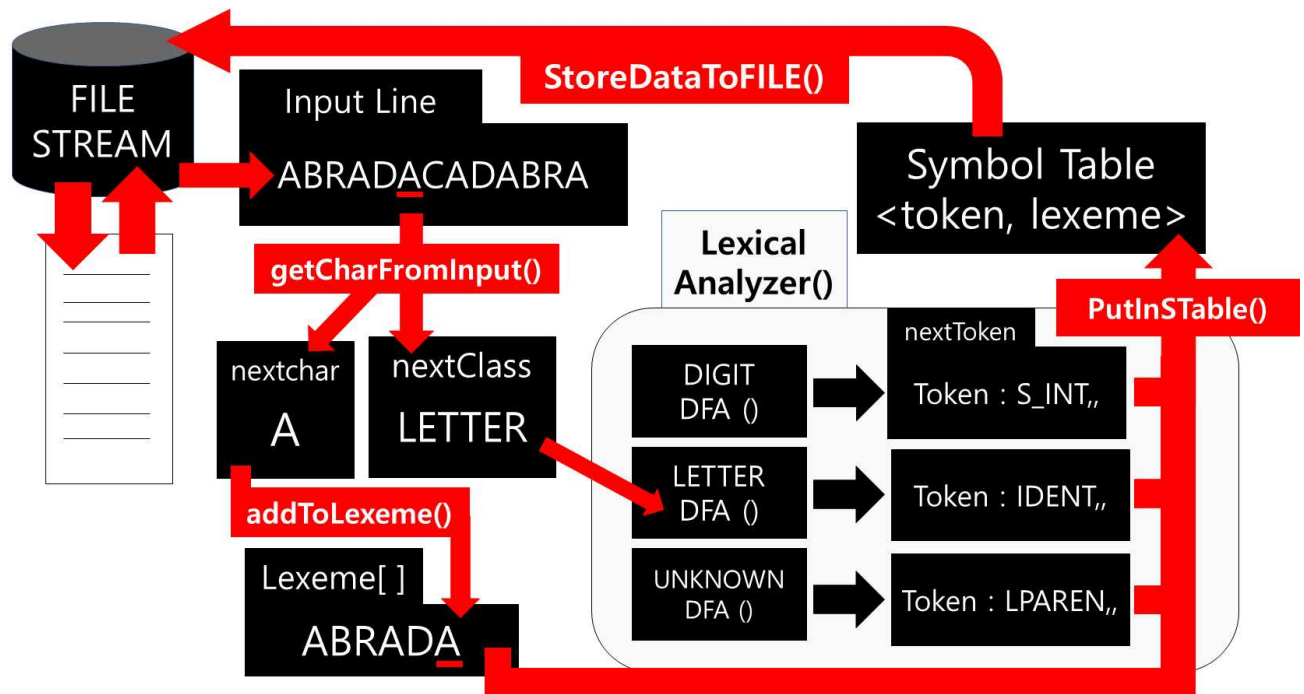
3. if that lexeme's Token is operator :
 : '-' **signed integer** ex) input : jason + -56

4. if that lexeme's **Class is UNKNOWN** and it's Token
 is **not a operator** :
 : '-' is **operator** (as mentioned in announcement)
 ex) input : "NEVERMIND"-543543

=====

Some other problems like classifying the 0011 is solved like these.
So, that's all. From all above, Now we can make some programs.

Program works like this:



I just wrote the important functions.

1. Get the one char from the input line and save it to **nextChar**
2. Classify the lexeme and store the resulting Class to **nextClass**
3. Using nextClass, choose one DFA and check it.
(this part will execute in Lexical analyzer ())
4. Inside of Lexical Analyzer, there are three DFA, and based on charClass, we choose the right DFA and check the lexeme.
5. In DFA, nextChar is stored to Lexeme[] on and on, so we can make the one lexeme.
6. If Token is defined, store the lexeme and Token to the **symbol table**. (we can store these pairs straight to stream.)
7. We do this loop again until reading the whole input.

6. Test Result - without error

before we start, I would like to explain the test cases and background of the program.

Test Cases :

test1.txt - Win

파일(F) 편집(E) ↗

001

0010

0010a0010

0010-10

0010--10

test2.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
int main(){char if123='1';int 0a = a+-1;return -0;}
```

""

""

-0

0abc0

123if0

"smells like t22n.spirit"

-34234324

apple-orange|

test3.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
int main()[
```

```
    char s = 'd';
```

```
    String die = "DIEDIEDIEDIE";
```

```
    if(s <= -10){
```

```
        while( class g != 60 ){
```

```
            alpha - delta;
```

```
        }
```

```
    }
```

```
    return 0;
```

```
]|
```

just like the lines mentioned at the announcement.

Language, Development environment

- wirtten in C, visual studio, window10.

Executed Environment

- Executable binary file is made in Ubuntu (In Window10, using WSL2), compiled in gcc

```
root@DESKTOP-61QKLAH:/mnt/c/download# gcc -o lexical_analyzer lex.c
```

- Executed in Ubuntu (In Window10, using WSL2)

```
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer test.txt  
END Lexical Analyzer. Check the output.txt
```

RESULT :

(Suppose this program is located at C:\download)

PC > 로컬 디스크 (C:) > Download

이름

test3.txt

test2.txt

test1.txt

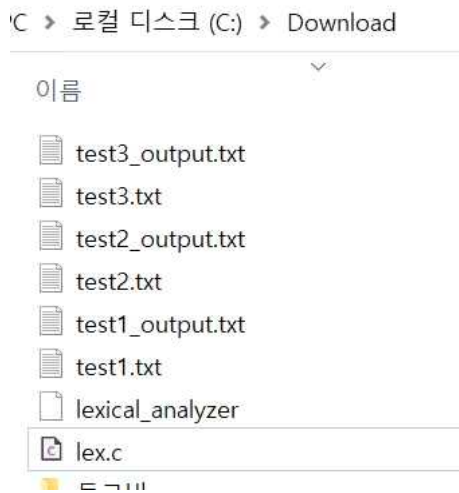
lexical_analyzer

lex.c

<< after the gcc compilation.

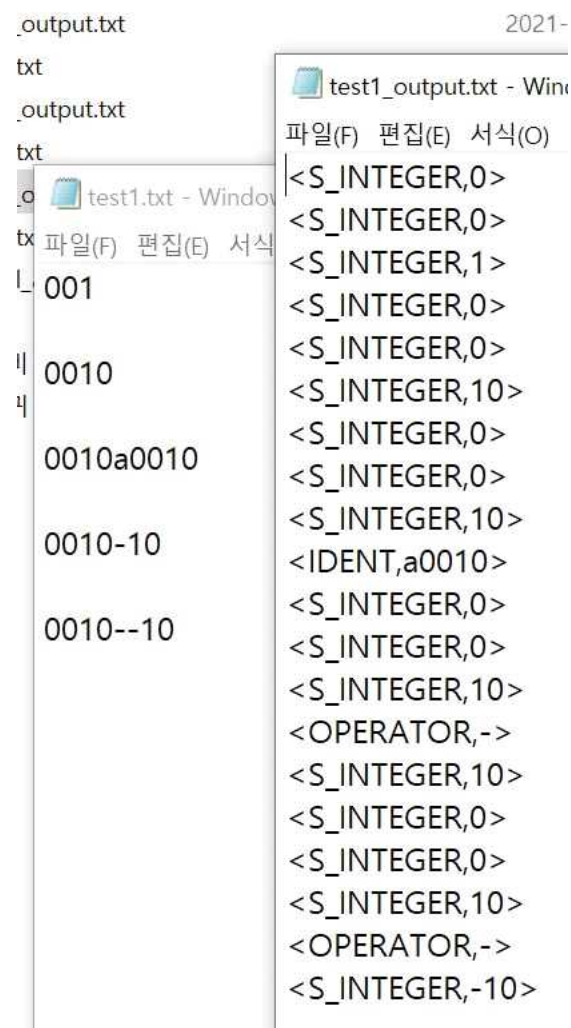
Result_output

```
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer test1.txt  
END Lexical Analyzer. Check the output.txt  
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer test2.txt  
END Lexical Analyzer. Check the output.txt  
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer test3.txt  
END Lexical Analyzer. Check the output.txt
```



We can see that the result is stored in same directory.

Then how about the result txt?



As we can see,

The result is same as the result of the announcement

output.txt	test2_output.txt - W
t	파일(F) 편집(E) 서식(O)
test2.txt - Windows 메모장	<VTYPE,int>
파일(F) 편집(E) 서식(O) 보기(V)	<IDENT,main>
int main(){char if123='1';i	<LAPREN,(>
"	<RPAREN,)>
""	<LCB,{>
-0	<VTYPE,char>
0abc0	<IDENT,if123>
123if0	<ASSIGN,=>
"smells like t22n.spirit"	<S_CHAR,1>
-34234324	<TERMINATE,;>
apple-orange	<VTYPE,int>
	<S_INTEGER,0>
	<IDENT,a>
	<ASSIGN,=>
	<IDENT,a>
	<OPERATOR,+>
	<S_INTEGER,-1>
	<TERMINATE,;>
	<RETURN,return>
	<OPERATOR,->
	<S_INTEGER,0>

Of course,

The result is same as the result of the announcement.

then how about the last one?

test3.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
int main()  
    char s = 'd';  
    String die = "DIEDIEDIEDIE";  
    if(s <= -10){  
        while( class g != 60 ){  
            alpha - delta;  
        }  
    }  
    return 0;  
}
```

test3_output.txt - ...

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
<VTYPE,int>  
<IDENT,main>  
<LAPREN,(>  
<RPAREN,>  
<LSB,[>  
<VTYPE,char>  
<IDENT,s>  
<ASSIGN,=>  
<S_CHAR,d>  
<TERMINATE,;>  
<VTYPE,String>  
<IDENT,die>  
<ASSIGN,=>  
<LIT_STR,DIEDIEDIEDIE>  
<TERMINATE,;>  
<IF,if>  
<LAPREN,(>  
<IDENT,s>  
<COMPARE,<=>  
<S_INTEGER,-10>  
<RPAREN,>
```

As we can see, it shows the result well :)

What if lines has an error?

This program will find the line that occurs error and exit the program. Let's see.

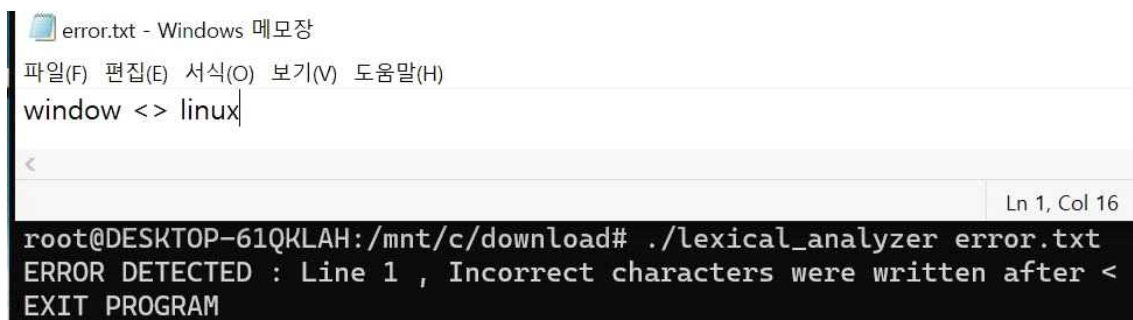
1. When using a wrong Comparison



```
error.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
int britney = spears;
Joker != Batman

Ln 3, Col 1
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer error.txt
ERROR DETECTED : Line 2 , Incorrect characters were written after !
EXIT PROGRAM
root@DESKTOP-61QKLAH:/mnt/c/download#
```

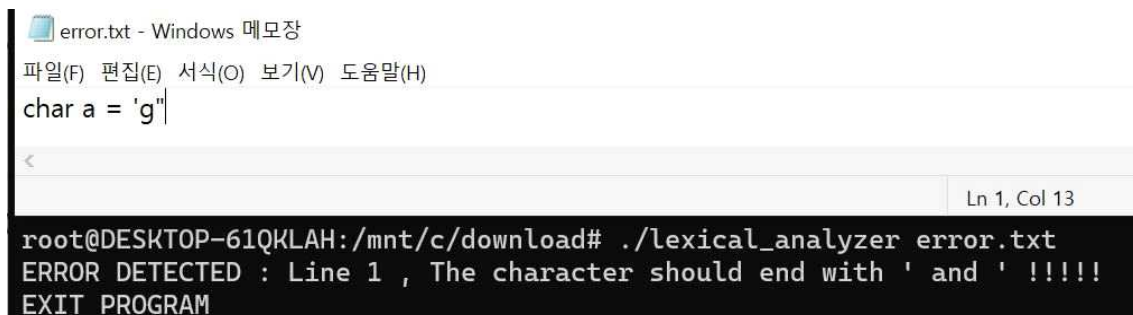
2. When using a wrong Comparison 2



```
error.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
window <> linux

Ln 1, Col 16
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer error.txt
ERROR DETECTED : Line 1 , Incorrect characters were written after <
EXIT PROGRAM
```

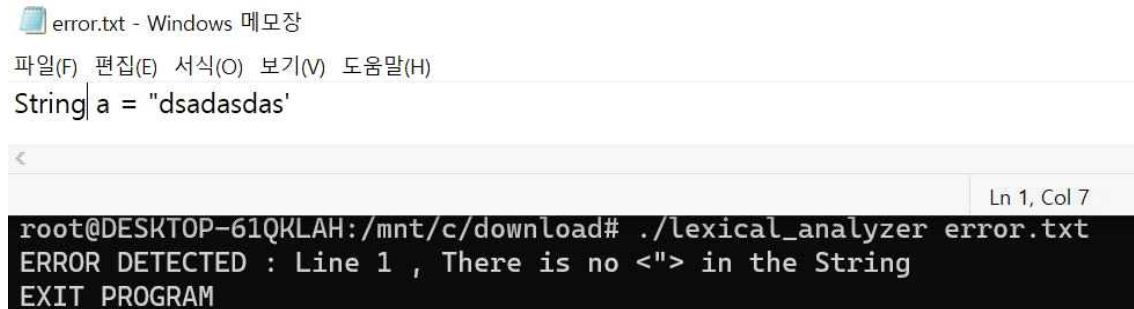
3. When using a wrong quotation mark



```
error.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
char a = 'g"

Ln 1, Col 13
root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer error.txt
ERROR DETECTED : Line 1 , The character should end with ' and ' !!!!!
EXIT PROGRAM
```

4. When using a wrong quotation mark 2



The screenshot shows a Windows Notepad window titled "error.txt - Windows 메모장". The menu bar includes "파일(F)", "편집(E)", "서식(O)", "보기(V)", and "도움말(H)". The text in the window is `String a = "dsadasdas'`. Below this, a terminal window shows the command `root@DESKTOP-61QKLAH:/mnt/c/download# ./lexical_analyzer error.txt` and the output: `ERROR DETECTED : Line 1 , There is no ">" in the String` and `EXIT PROGRAM`. The terminal window also shows the cursor position as "Ln 1, Col 7".

This programs can detect more error, but we think its enough to showing the detecting.

but we left **some** error undetected because that error will be found in **Syntax Analyzer**. So, we leave the error such as `;;;;;` (use terminator more than once),,, etc.

Review

Although it is very hard to design the lexical analyzer. But, from this project, We learned how the lexical analyzer really works, What's the important thing, etc. Also, We learned the difference between Windows and Linux, learned Some tools about Linux. We think this project is worth it. Lastly, We think we have to put more effort to think about how to write a good program...