COMPILER PROJECT (lexical analyzer)

20182345 이건호

<tokens , regular expressions>

(DIGIT=0|1|2|3|4|5|6|7|8|9)

(NON_0_DIGIT= 1|2|3|4|5|6|7|8|9)

(LETTER=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)

[VTYPE]=int | INT|char | CHAR

[SIGNED_INTEGER]=0 | (-| ε)(NON_0_DIGIT)(DIGIT)*

[STRING]="(LETTER|DIGIT|)*"

[ID]=LETTER(LETTER|DIGIT)*

[KEYWORD]=if|IF|else|ELSE|while|WHILE|return|RETURN

[ARITHMETIC_OP]=+|-|*|/

[ASSIGNMENT_OP]==

[COMPARISON_OP]=<|>|==|!=|<=|>=

[SEMI]=;

[L_BRACKET]={

[R_BRACKET]=}

[L_PAREN]=(

[R_PAREN]=)

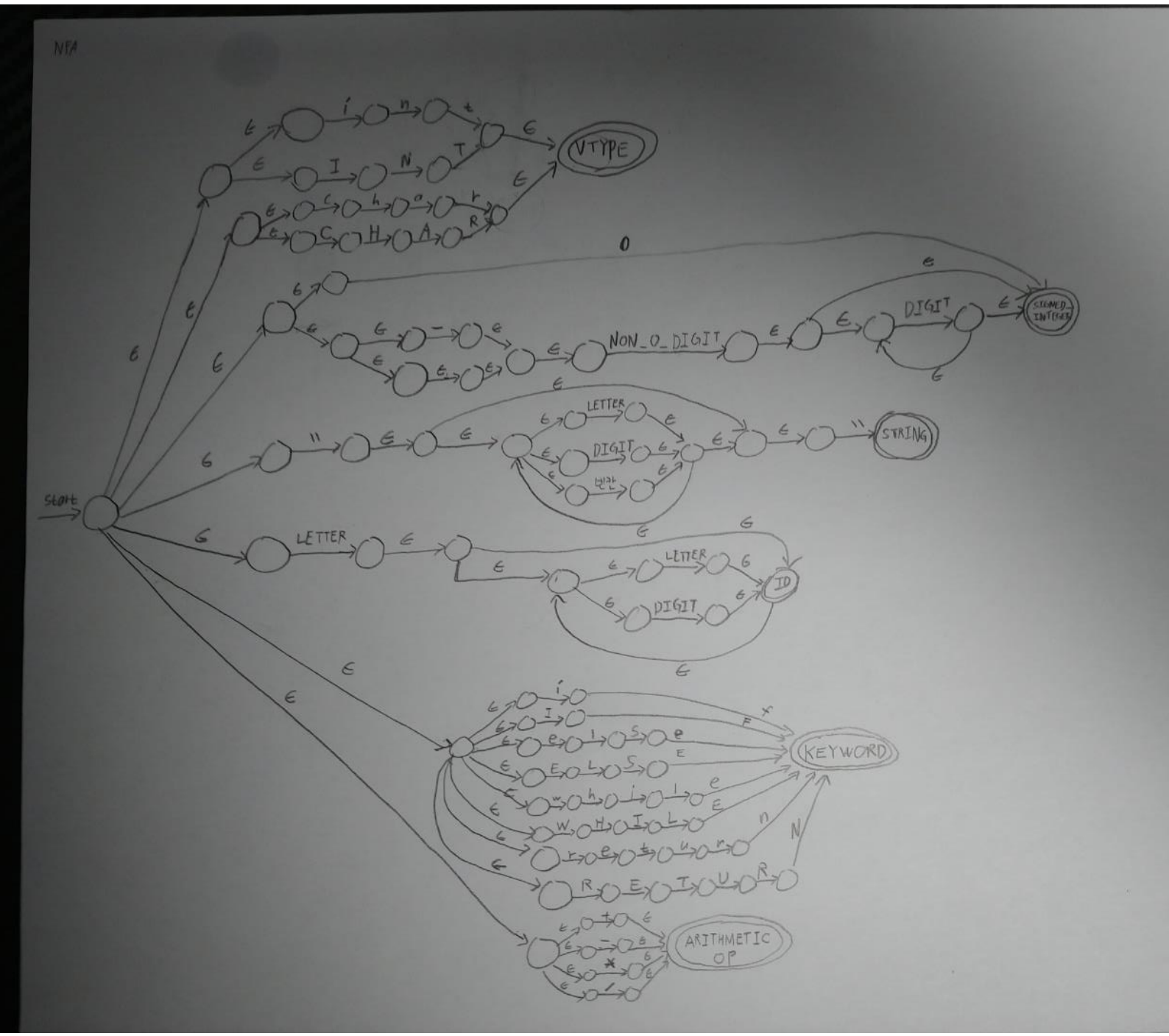
[SEPARATING]=,

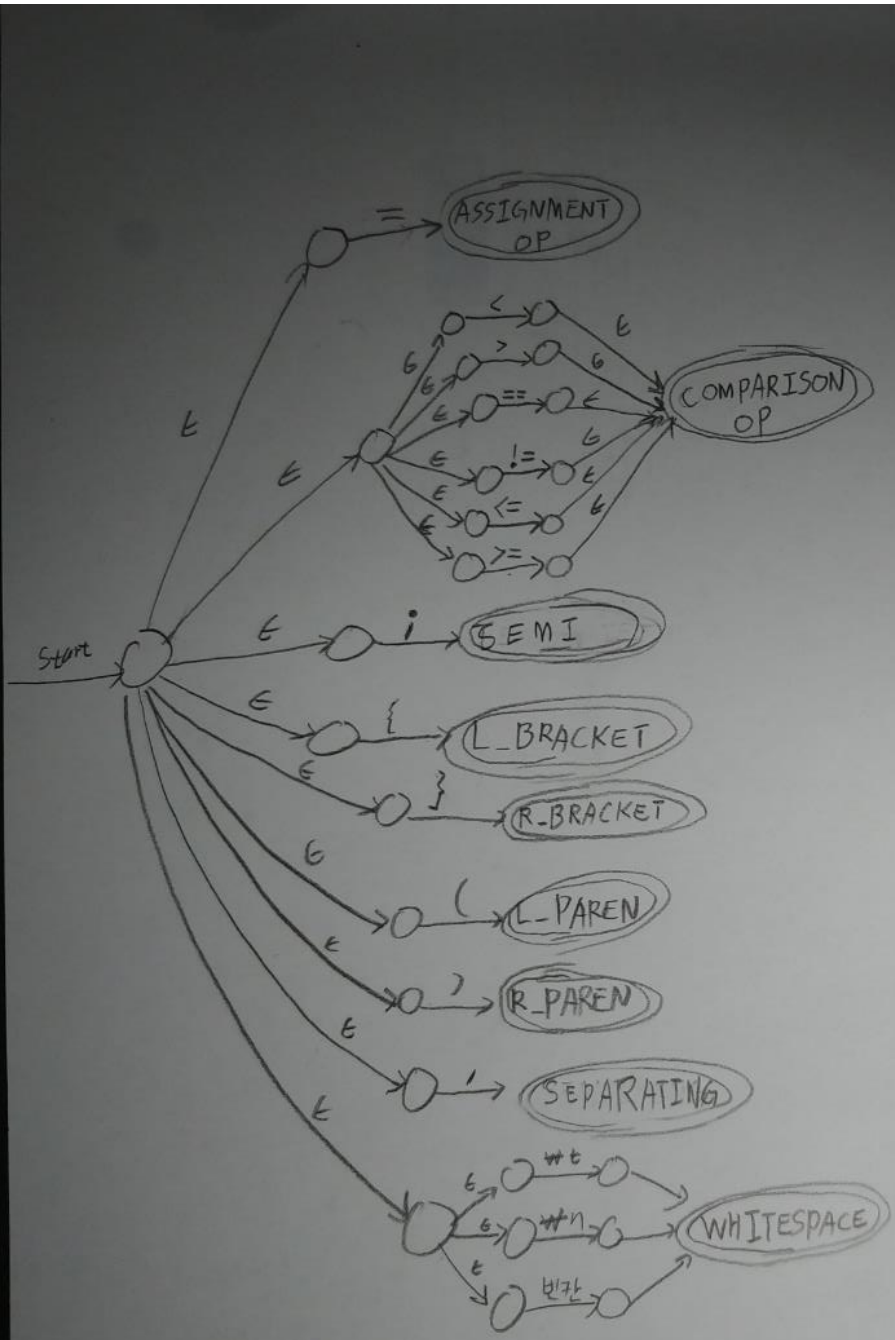
[WHITESPACE]=₩t | ₩n | blank(빈칸) |₩r(리눅스에서 이걸 안 썼더니 윈도우에서 생성한 텍스트파일의 줄바꿈을 인식못해서 추가해줌)

op 의-와 음수의-는 바로앞에 op(arithmetic 이든 assignment 든 comparison 이든)가 있을 경우에만 음수의-이고 그 외에는 op 처리

<NFA transition graph>

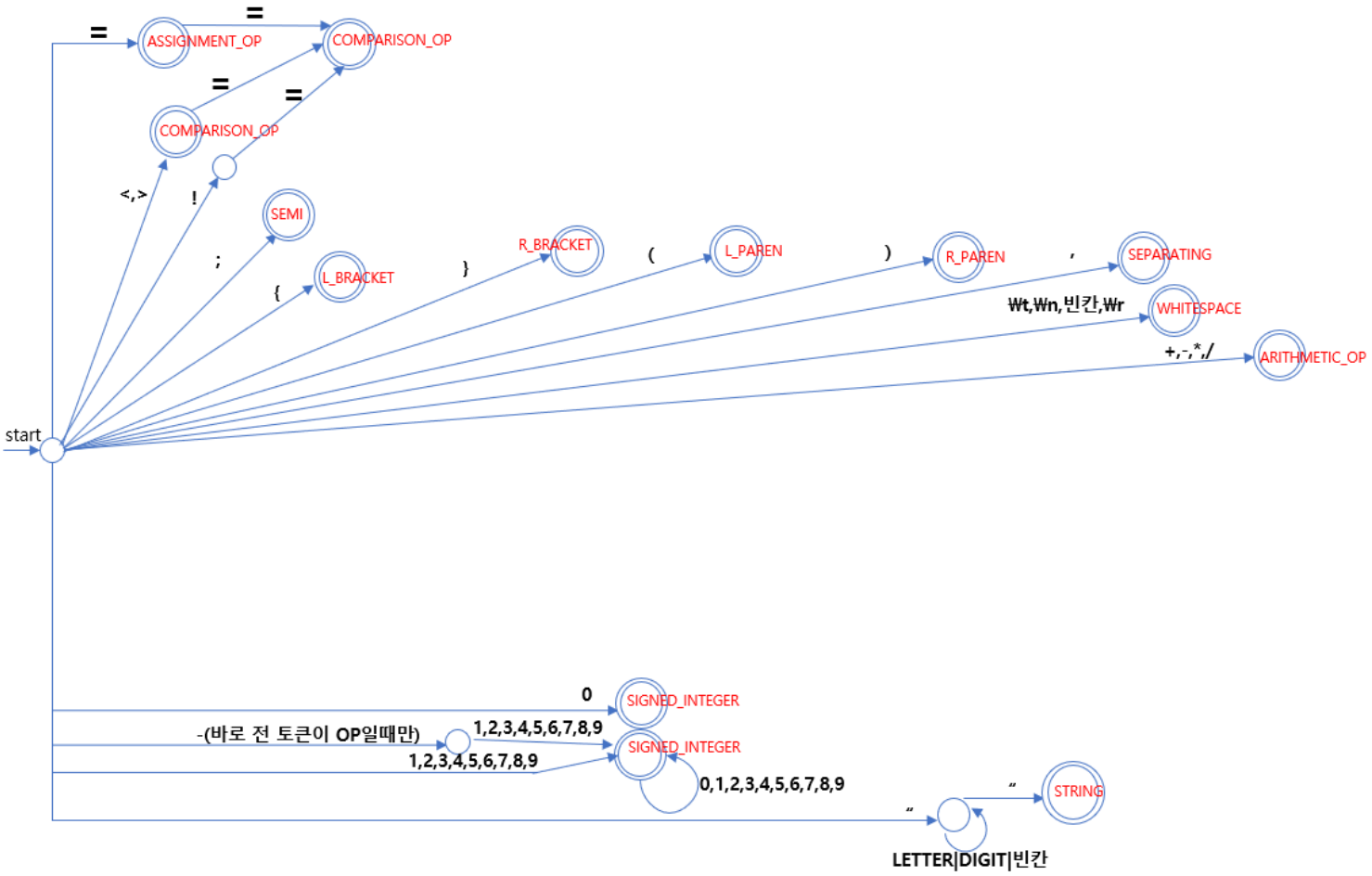
종이 2 장에 나눠서 그렸습니다. (같은 start 지점을 지님)

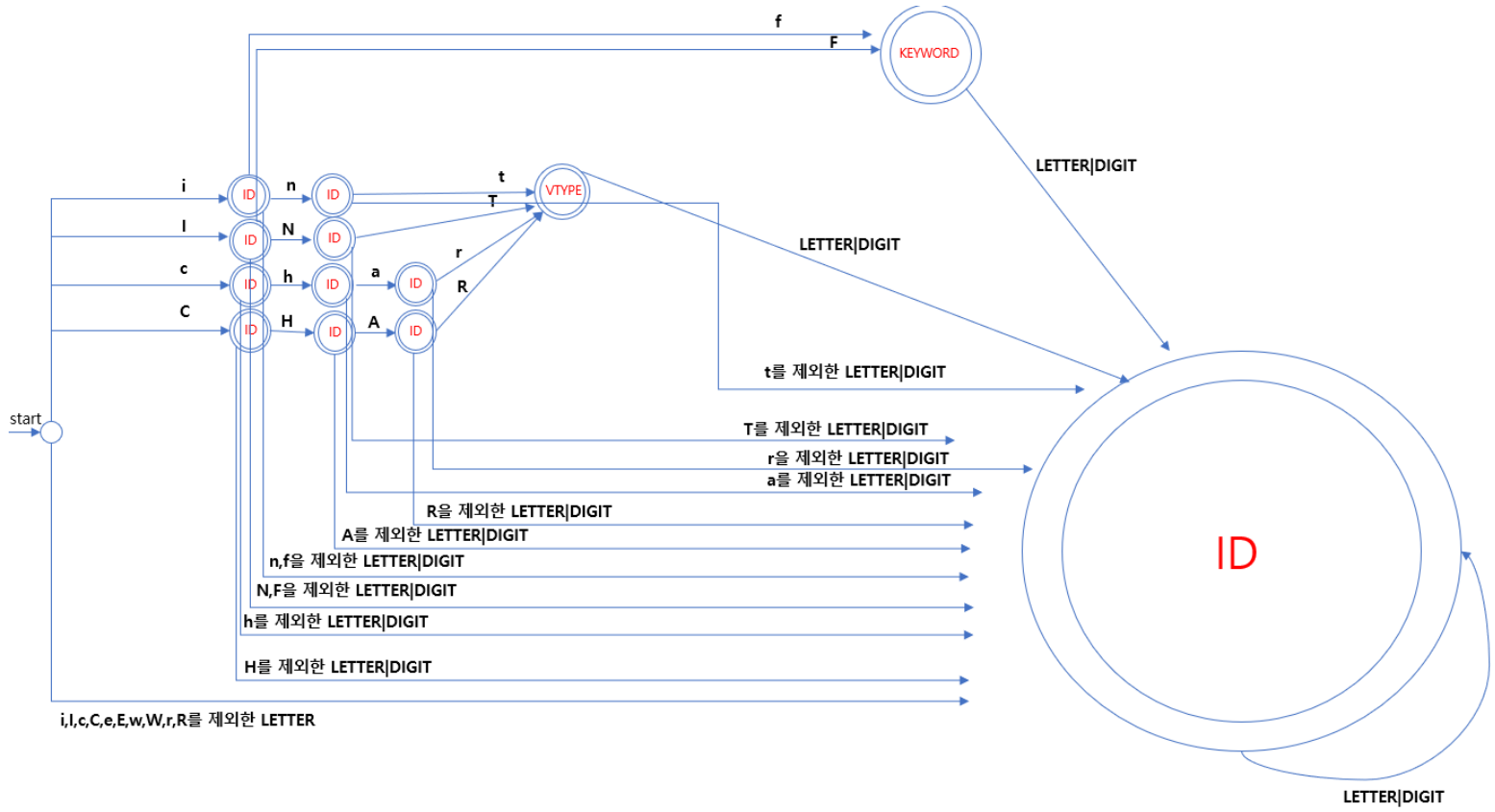


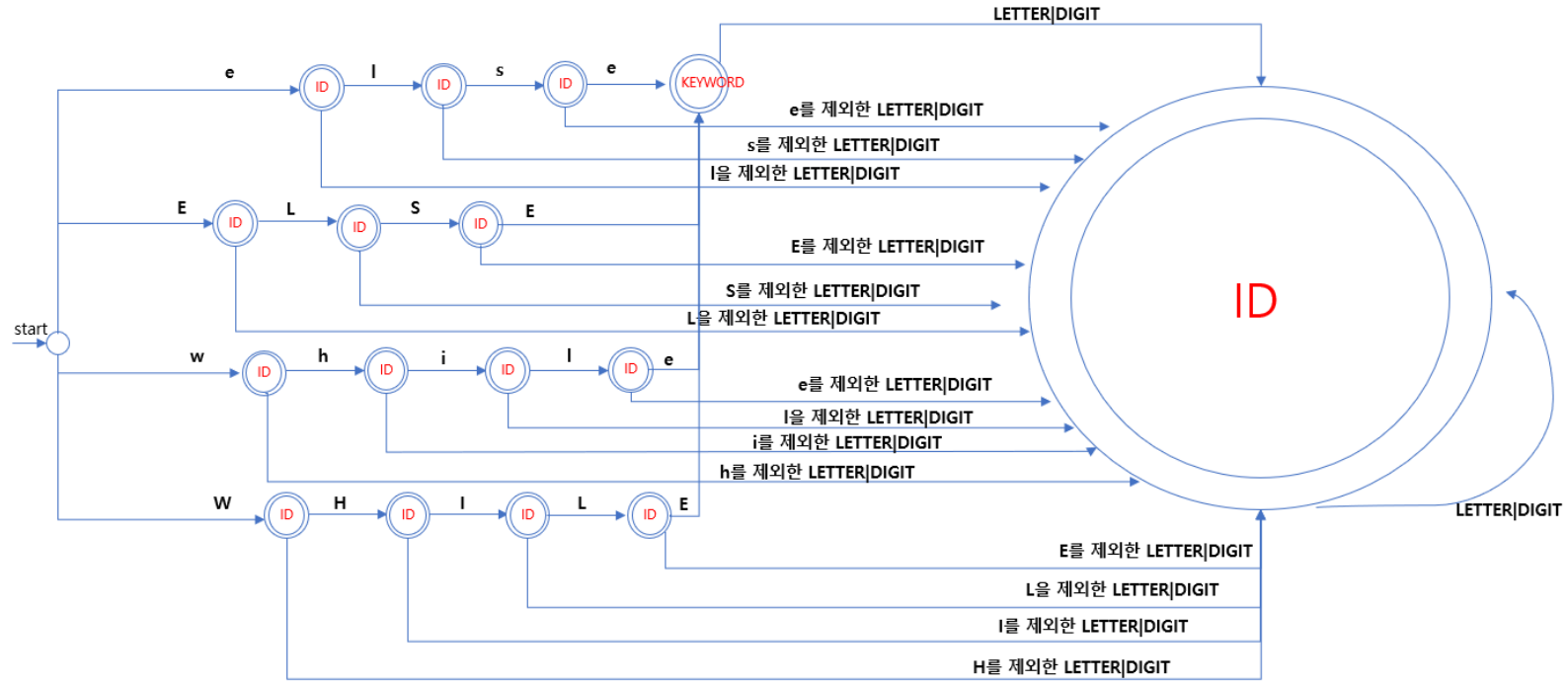

Start
=
ASSIGNMENT OP
<
>
==
!=
<=
>=
COMPARISON OP
;
SEMI
{
L_BRACKET
}
R_BRACKET
(
L_PAREN
)
R_PAREN
/
SEPARATING
\t
\n
빈칸
WHITESPACE

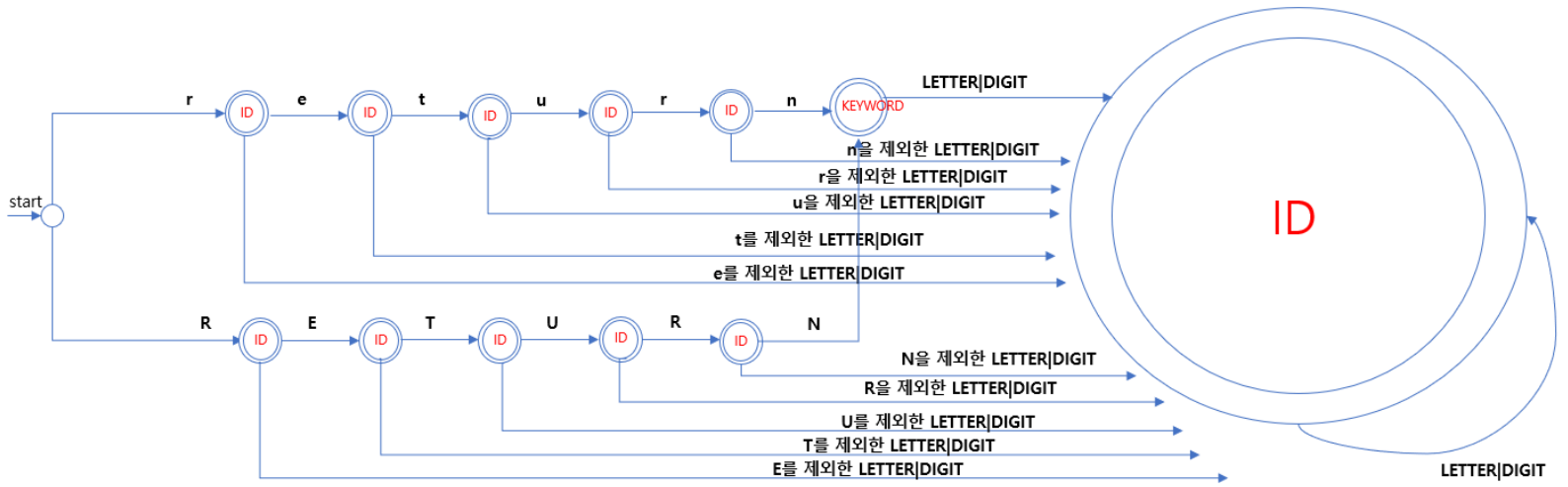
<DFA transition graph>

파워포인트를 이용해서 그렸습니다. 4 장으로 나눠서 그렸으며 모두 같은 start 지점을 지님.






LETTER|DIGIT
e
l
s
e
ID
ID
ID
KEYWORD
e를 제외한 LETTER|DIGIT
s를 제외한 LETTER|DIGIT
l을 제외한 LETTER|DIGIT
E
L
S
E
E를 제외한 LETTER|DIGIT
S를 제외한 LETTER|DIGIT
L을 제외한 LETTER|DIGIT
start
w
h
i
l
e
e를 제외한 LETTER|DIGIT
l을 제외한 LETTER|DIGIT
i를 제외한 LETTER|DIGIT
h를 제외한 LETTER|DIGIT
W
H
I
L
E
ID
LETTER|DIGIT
E를 제외한 LETTER|DIGIT
L을 제외한 LETTER|DIGIT
I를 제외한 LETTER|DIGIT
H를 제외한 LETTER|DIGIT


LETTER|DIGIT
r
e
t
u
r
n
ID
KEYWORD
n을 제외한 LETTER|DIGIT
r을 제외한 LETTER|DIGIT
u을 제외한 LETTER|DIGIT
start
t를 제외한 LETTER|DIGIT
e를 제외한 LETTER|DIGIT
R
E
T
U
R
N
N을 제외한 LETTER|DIGIT
R을 제외한 LETTER|DIGIT
U를 제외한 LETTER|DIGIT
T를 제외한 LETTER|DIGIT
E를 제외한 LETTER|DIGIT
ID
LETTER|DIGIT

<상세 설명>

먼저 main 함수부터 흐름을 설명하겠습니다.

```
int main(int argc,char* argv[])
{
    string input_txt;
    input_txt=argv[1];

    ifstream in(input_txt);
    if (in.is_open())
    {
        std::cout << "유효한 파일 이름입니다. Parser를 실행합니다.\n\n " << endl;
    }
    else
    {
        std::cout << "파일을 찾을 수 없습니다!" << std::endl;
        return 0;
    }


    int line=1;
    while (!in.eof()) //한줄씩 lexical analyser 수행
    {
        string input;
        getline(in, input);
        lexical_analyser lexical_analyser(input,line);
        lexical_analyser.add_token_table(input);
        line++;
    }

    input_txt+=".out";
    ofstream output(input_txt);
    if (output.fail())
    {
        std::cerr << "Error!" << std::endl;
        return -1;
    }
    if(error==false) //에러가 없을 경우에 lexical analyser한 결과 output파일 생성
    {

        for (int i = 0; i < token.size(); i++)
        {
            cout << "<" << token[i].first << "," << token[i].second << ">\n";
            output << "<" << token[i].first << "," << token[i].second << ">\n";
        }
    }
    else //에러가 있을경우 error report를 output파일로 생성
    {
        output<<error_string;
    }
}
```

main 함수의 인자로 lexical analyzer 에 집어넣어줄 파일명을 받아줍니다.

그 후 파일이 정상적으로 열렸을 경우에만 실행을 하게 됩니다.

while 문을 통해서 input 파일의 한줄씩 읽으며 lexical 분석을 실행하게 됩니다. 한줄씩 읽기 때문에 에러를 표시해줄 때 몇 번째 줄, 몇 번째 인덱스에서 발생한 것인지 상세히 표시해줄 수 있습니다.

이때 lexical 분석을 위하여 lexical_analyser 라는 class 를 만들어 줬습니다.

해당 클래스의 구성은 다음과 같습니다.

```
class lexical_analyser
{
    private:
        int line;//현재 input파일의 몇번째 줄을 읽고 있는지
        int pointer=0;//현재 읽은 줄의 어떤 글자를 point하고 있는지(인덱스)
    public:
        lexical_analyser(string input,int lineNum)
        {
            line=lineNum;
        }
        void add_token_table(string input); //lexeme을 나누고 symbol tabl
};
```

먼저 **line** 은 error 를 표시해줄 때 input 파일의 몇 번째 줄에서 문제가 생긴 것인지 표시하기 위해 존재합니다.

pointer 는 input 파일로부터 읽어온 한 줄이 있을 텐데, lexical 분석을 할 때, 그 한 줄의 몇 번째 index 를 가리키고 있는지를 나타내기 위하여 선언해줬습니다.

그리고 **void add_token_table(string input)** 메소드는 본격적으로 lexical 분석을 해주기 위한 장치입니다. input(읽은 한 줄)을 분석해주는 역할을 합니다.

```
vector<pair<string,string>> token;
bool error=false;
string error_string;
```

lexical 분석 과정에서 생성되는 결과물은(<token name, token value>형식**) 전역변수로 선언된 vector<pair<string,string>>token** 에 저장됩니다. pair 와 vector 를 혼합하여서 token name 과 token value 가 짝을 이루면서 순서대로 쉽게 저장될 수 있게 하였습니다. 또한 bool 타입의 error 전역변수는 lexical 분석 과정에서 error 발생여부를 확인하기 위한것이며 string error_string 은 에러내용 기록용입니다.

그럼 이번 프로젝트의 핵심인 **add_token_table(lexical 분석 과정)**이 어떻게 작동하는지에 대하여 설명하겠습니다.

```
bool is_digit(char a)
{
    if(a=='0'||a=='1'||a=='2'||a=='3'||a=='4'||a=='5'||a=='6'||a=='7'||a=='8'||a=='9')  return true;
    else return false;
}
bool is_non_0_digit(char a)
{
    if(a=='1'||a=='2'||a=='3'||a=='4'||a=='5'||a=='6'||a=='7'||a=='8'||a=='9')  return true;
    else return false;
}
bool is_letter(char a)
{
    if(a=='a'||a=='b'||a=='c'||a=='d'||a=='e'||a=='f'||a=='g'||a=='h'||a=='i'||a=='j'||a=='k'||a=='l'
        return true;
    else return false;
}
```

위 세 함수는 lexical 분석 과정에서 자주 쓰일 기능을 구현한 것입니다.

is_digit 은 해당 캐릭터가 0~9 에 속하는지, is_non_0_digit 은 1~9 에 속하는지,

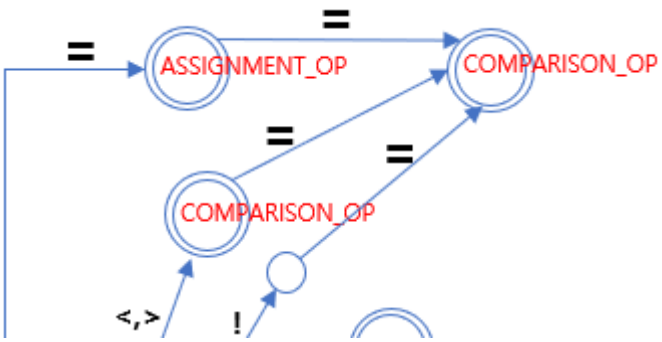
is_letter 는 해당 캐릭터가 알파벳 소,대문자에 속하는지를 판별해주는 기능을 합니다.

```
void write_token_table(string input,int start_index,int end_index,string token_name)
{
    string temp=input.substr(start_index,end_index-start_index+1);
    pair<string,string> p;
    p.first=token_name; p.second=temp;
    token.push_back(p);
}
```

위 함수는 lexical 분석과정에서 적절히 lexeme 을 나눴으면 그 lexeme 에 해당하는 token 의 이름과 lexeme(=token value)를 앞서 설명한 vector<map<string,string>> token 이라는 전역변수에 저장해주는 기능을 합니다. **즉, symbol table(token table)에 추가해 주는 기능입니다.**

그럼 lexical analyzer 가 작동하는 걸 비슷한 방식으로 분석되는 token 카테고리로 나눠서 설명하겠습니다.

먼저 **ASSIGNMENT_OP** 토큰, **COMPARISON_OP** 토큰 입니다.



DFA 에서는 이 부분입니다.

```
if (input[pointer] == '=') // =기호와 ==기호 구분
{
    token_name = "ASSIGNMENT_OP";
    pointer++;
    if (pointer < input.length() && input[pointer] == '=')
    {
        token_name = "COMPARISON_OP";
        pointer++;
    }
    end_index=pointer-1;
    write_token_table(input,start_index,end_index,token_name);
}
```

먼저 인식된 문자가 '='라면 일단은 token 을 ASSIGNMENT_OP 로 설정합니다. 그리고 pointer 를 하나 증가시킨 후, 다음 문자를 검사하였을 때 또 '='가 나왔다면

'=='이기 때문에 token 을 COMPARISON_OP 로 설정해 줍니다. 이렇게 이중 if 문을 사용하여서 =와 ==를 구분하도록 처리하였고 해당되는 token 과 value 를 앞서 말했던 write_token_table 이라는 함수를 이용하여 테이블에 저장합니다.

참고로 lexical 분석 단계에서 pointer 는 항상 인식된 token 다음을 가리키도록 구현했습니다. 그렇기 때문에 매 과정마다 pointer++가 있고 pointer 가 input 의 길이를 넘겼는지 확인하고 있습니다.

```
else if(input[pointer]=='<' || input[pointer]=='>') // <,>,<=,>=기호 구분
{
    token_name="COMPARISON_OP";
    pointer++;
    if(pointer < input.length() && input[pointer] == '=')
    {
        token_name="COMPARISON_OP";
        pointer++;
    }
    end_index=pointer-1;
    write_token_table(input,start_index,end_index,token_name);
}
```

인식된 문자가 =이 아니라 <이거나 >라면 먼저 token 을 COMPARISON_OP 로 설정합니다. 앞에서와 마찬가지로 다음 문자를 검사했을 때 =이라면 <=이거나 >=인 것이기 때문에 TOKEN 은 여전히 COMPARISON_OP 입니다. 그리고 분석된 결과를 테이블에 저장해줍니다.

```
else if(input[pointer]=='!') // !=구분
{
    token_name="error";
    pointer++;
    if(pointer < input.length() && input[pointer] == '=')
    {
        token_name="COMPARISON_OP";
        pointer++;
        end_index = pointer - 1;
        write_token_table(input, start_index, end_index, token_name);
    }
    else // 이어서 =이 오지않으면 에러처리. table에 !는 추가해주지 않음. 오류메시지만 출력
    {
        cout<<"ERROR:"<<line<<"번째줄 "<<start_index+1<<"번 인덱스에 =이 와서 !=을 완성해야하지만 오지 않았습니다.\n";
        error=true;
        error_string+="ERROR:"+to_string(line)+"번째줄 "+to_string(start_index+1)+"번 인덱스에 =이 와서 !=을 완성해야하지만 오지 않았습니다.\n";
    }
}
```

인식된 문자가 !라면 일단은 token 을 error 로 설정합니다. 왜냐하면 이번에 과제로 구현한 컴파일러에서 !단독으로 쓰이는 토큰은 없기 때문입니다. 다음 문자를 검사했을 때 =가 나온다면 !=이기 때문에 token 을 COMPARISON_OP 로 설정하고 분석된 결과를 테이블에 저장해줍니다. 하지만 다음 문자가 =가 아니라면 !혼자 왔기때문에 ERROR 를 표시해줍니다.

이번엔 **SEMI** 토큰, **L_BRACKET** 토큰, **R_BRACKET** 토큰, **L_PAREN** 토큰, **R_PAREN** 토큰, **SEPARATING** 토큰, **WHITESPACE** 토큰 입니다.

```
else if(input[pointer]==';') // ;구분
{
    token_name="SEMI";
    pointer++;
    end_index = pointer - 1;
    write_token_table(input, start_index, end_index, token_name);
}
```

위 토큰들은 단일 문자로 분석된다는 특징이 있습니다. 그렇기 때문에 비교적 간단합니다. SEMI 토큰을 예로 들면 인식된 문자가 ;라면 token 을 SEMI 로 설정하고 테이블에 추가해줍니다. 나머지 토큰들도 동일하기 때문에 생략하겠습니다.

이번엔 **ARITHMETIC_OP** 토큰, **SIGNED_INTEGER** 토큰입니다. 이 두 토큰사이에서는 고민 해줘야 할 사안이 있습니다. 바로 '-'기호를 어떻게 인식해줘야 하는가입니다. 연산기호로써의 '-'인지 아니면 음수를 나타내는 '-'인지가 그 문제입니다.

저는 이 문제를 OP 다음에 오는(ASSIGNMENT,COMPARISON,ARITHMETIC) '-'기호라면 SIGNED_INTEGER 에 속하는 '-'인 것으로 처리하였고 그렇지 않은 경우는 연산자로서의, 즉 ARITHMETIC_OP 의 '-'기호인 것으로 처리하였습니다. 이렇게 한다면 a=-1+2 같은 경우에 -1 이런식으로 묶어지고, a<-3 이 경우도 OP 다음에 온 -이기때문에 -3 으로 묶어지게 됩니다. 즉, 상식적인 프로그래밍 환경에서는 제가 처리한 방식으로 '-'기호를 어느 토큰에 넣어줄지를 구분할 수 있을 것이라 생각합니다.

```
else if(input[pointer]=='-' && (token.back().first=="ARITHMETIC_OP"||token.back().first=="ASSIGNMENT_OP"||token.back().first=="COMPARISON_OP")) // -의 경
{
    token_name="error";
    pointer++;
    if(pointer < input.length() && is_non_0_digit(input[pointer]))
    {
        token_name="SIGNED_INTEGER";
        pointer++;
        while(pointer<input.length() && is_digit(input[pointer]))
        {
            token_name = "SIGNED_INTEGER";
            pointer++;
        }
        end_index = pointer - 1;
        write_token_table(input, start_index, end_index, token_name);
    }
    else // 음수이므로 -뒤에 숫자가 와야하는데 안왔으므로 에러처리
    {
        cout<<"ERROR:"<<line<<"번째줄 "<<start_index+1<<"번 인덱스에 0이 아닌 숫자가 와서 음수를 구성해야하지만 오지 않았습니다.\n";
        error=true;
        error_string+="ERROR:"+to_string(line)+"번째줄 "+to_string(start_index+1)+"번 인덱스에 0이 아닌 숫자가 와서 음수를 구성해야하지만 오지 않았습니다.\n";
    }
}
```

위 코드를 보면 인식된 기호가 -이고 바로 이전에 저장된 token 이 OP 관련 token 이라면 SIGNED_INTEGER 로 TOKEN 을 설정해주는 걸 알 수 있습니다. 이때 EROOR 처리도 해주었는데 음수를 나타내는 -인 것이 확정되었기 때문에 그 뒤 문자를 검사하여서 그 문자가 0 이아닌 숫자가 아니면 error 처리를 해주었습니다.

```
else if(input[pointer]=='+'||input[pointer]=='-'||input[pointer]=='*'||input[pointer]=='/')
{
    token_name="ARITHMETIC_OP";
    pointer++;
    end_index = pointer - 1;
    write_token_table(input, start_index, end_index, token_name);
}
```

ARITHMETIC_OP 는 음수의 -인 경우를 제외하고는 전부 연산자-이기 때문에 +,-,*,/를 한번에 합쳐서 else if 문에 넣어줬습니다.(음수의 -를 구분하는 else if 문 뒤에 위치함) 그렇게 하여 간단히 구분해줄 수 있습니다.

```
else if(input[pointer]=='0') // 단순 숫자 0 구분
{
    token_name="SIGNED_INTEGER";
    pointer++;
    end_index = pointer - 1;
    write_token_table(input, start_index, end_index, token_name);
}
else if(is_non_0_digit(input[pointer])) //양의 정수인지 확인
{
    token_name="SIGNED_INTEGER";
    pointer++;
    while ((pointer < input.length()) && is_digit(input[pointer]))
    {
        token_name = "SIGNED_INTEGER";
        pointer++;
    }
    end_index = pointer - 1;
    write_token_table(input, start_index, end_index, token_name);
}
```

그리고 단순 숫자 0 과 양의정수도 인식하여서 SIGNED_INTEGER 토큰으로 설정해주고 테이블에 저장해줬습니다.

이번엔 **STRING** 토큰입니다.

```
else if(input[pointer]=='"') // "로 시작하고 "로 끝나는 String인지 구분. 저장할때 value는 "는 제외하고 저장함
{
    token_name="error";
    pointer++;
    while((pointer < input.length()) && (is_letter(input[pointer]) || is_digit(input[pointer]) || input[pointer]==' '))
    {
        token_name="error";
        pointer++;
    }
    if((pointer < input.length()) && input[pointer]=='"')
    {
        token_name="STRING";
        pointer++;
        end_index=pointer-2;
        if(start_index==end_index) //빈 string일 경우 빈string 넣어줌.
        {
            string temp = "";
            pair<string, string> p;
            p.first = token_name;
            p.second = temp;
            token.push_back(p);
        }
        else
        {
            start_index+=1;
            write_token_table(input, start_index, end_index, token_name);
        }
    }
    else
    {
        cout<<"ERROR:"<<line<<"번째줄에서 \"기호가 나타나고 닫는 \"기호가 나오지 않았거나, letter,digit,빈칸 이외에 다른 문자가 등장하였습니다.\n";
        error=true;
        error_string+="ERROR:"+to_string(line)+"번째줄에서 \"기호가 나타나고 닫는 \"기호가 나오지 않았거나, letter,digit,빈칸 이외에 다른 문자가 등장하였습니다.\n";
    }
}
```

먼저 "문자가 인식되면 letter 또는 digit 또는 빈칸이 더 이상 안 나올때까지 계속하여 읽습니다. 그러던 와중에 letter,digit,빈칸에 속하지 않는 문자가

인식되거나 끝까지 읽었음에도 "가 나타나지 않는다면 ERROR 를 표시해줍니다. 그렇지 않는다면 테이블에 "~~~" 에서 큰따옴표를 제거한 ~~~부분만 token value 로 저장해주고 token 은 STRING 으로하여서 테이블에 저장해줍니다.

이번엔 **VTYPE** 토큰 중 int,INT , 그리고 **KEYWORD** 토큰중 if,IF 에 관한 처리입니다.

이렇게 묶은 이유는 네 lexeme 이 i 로 시작한다는 공통점이 있기 때문입니다.

소스코드 257 줄~396 줄까지가 해당 내용인데, 너무 길어서 스크린샷은 생략하겠습니다. 어떤 원리로 구현을 했냐를 int 와 if 를 예시로 설명하겠습니다. 먼저 i 문자를 인식합니다. 그러면 이제 선택지는 4 개로 나눠집니다.

1. i 뒤에 나오는 문자가 letter 나 digit 이 아니라서 i 혼자서만 token 으로 저장되는 경우
2. i 뒤에 나오는 문자가 f 라서 token 이 KEYWORD 로 설정되고 그 value 는 if 로 설정된 경우(그러나 f 문자 이후의 문자가 letter 나 digit 이 나올 경우 ID 토큰으로 변경됨. 그 후부터는 ID 를 검사해서 저장해줌)
3. i 뒤에 나오는 문자가 n 이라서 int 가 될 수 있는 경우. 이 분기가 선택된다면 추가로 다음 문자를 검사해서 t 인지아닌지 여부를 판단해야함.(n 일때는 선택지가 3 개로 나뉨. in 자체가 ID 인경우, 다음 문자가 t 여서 int 로 갈 경우,t 가아닌 letter 나 digit 이 와서 ID 검사 단계로 넘어가는 경우)
4. i 뒤에 나오는 문자가 f 나 n 이 아닌 letter 이거나 digit 이어서 if,int 의 가능성이 사라지고 token 이 ID 로 고정되는 상태. 이때는 LETTER 나 DIGIT 이 더 이상 나오지 않을때까지 읽어서 그 값을 ID 토큰의 value 로 설정하고 테이블에 저장되게 됨.

INT,IF 도 똑 같은 선택지를 갖게 됩니다. 이를 조건문과 while 문을 이용하여 구현하였습니다. 제가 그린 DFA 를 그대로 구현한 방식입니다.

VTYPE 토큰의 char,CHAR 도 마찬가지로 비슷한 방식입니다.

```
else if(input[pointer]=='c') //char(VTYPE),ID구분
{
    token_name="ID";
    pointer++;
    if((pointer < input.length()) && input[pointer]=='h')
    {
        token_name="ID";
        pointer++;
        if((pointer < input.length()) && input[pointer]=='a')
        {
            token_name="ID";
            pointer++;
            if((pointer < input.length()) && input[pointer]=='r')
            {
                token_name="VTYPE";
                pointer++;
                while ((pointer < input.length()) && (is_letter(input[pointer]) || is_digit(input[pointer])))
                {
                    token_name = "ID";
                    pointer++;
                }
                end_index = pointer - 1;
                write_token_table(input, start_index, end_index, token_name);
            }
            else branch_to_ID(input,&pointer,'r',start_index,end_index);
        }
        else branch_to_ID(input,&pointer,'a',start_index,end_index);
    }
    else branch_to_ID(input,&pointer,'h',start_index,end_index);
}
```

char 을 예시로 들면 먼저 c 를 인식합니다. 그리고 다음에 오는 문자가 h 인지 검사합니다. 만약 아니라면? else 문을 통하여 branch_to_ID 함수를 사용합니다.

```
void branch_to_ID(string input,int* pointer, char c,int start_index,int end_index) // 특정문자(여기선 인자인 c)를 제외한 LE
{
    string token_name="ID";
    if((*pointer < input.length()) && input[*pointer]!=c && (is_letter(input[*pointer]) || is_digit(input[*pointer])) )
    {
        token_name="ID";
        *pointer=*pointer+1;
        while((*pointer < input.length()) && (is_letter(input[*pointer]) || is_digit(input[*pointer])))
        {
            token_name = "ID";
            *pointer=(*pointer)+1;
        }
        end_index = (*pointer) - 1;
        write_token_table(input, start_index, end_index, token_name);
    }
    else //특정문자까지만 ID 임. 그 후는 ID형식에 맞지않거나 끝까지 읽은 경우
    {
        end_index=*pointer-1;
        write_token_table(input, start_index, end_index, token_name);
    }
}
```

이 함수의 역할이 무엇이냐면 예를 들어 위 상황에서 c 다음 h 가 올 것이라 기대했지만 오지 못한 상황입니다. 그럼 위 함수에 h 를 인자로 넘겨줍니다.

그러면 branch_to_ID 함수 내에서는 c 다음에 오는 문자가 h 가 아닌 LETTER 나 DIGIT 이라면 token 을 ID 로 설정하고 LETTER 나 DIGIT 이 더 이상 안 나올때까지 읽어들여서 해당 lexeme 을 ID 토큰의 value 로 테이블에 저장해줍니다. 하지만 c 다음에 오는 문자가 LETTER 나 DIGIT 이 아니라면 c 혼자서 ID 토큰의 value 인 것이기 때문에 테이블에 <ID,c>형태로 저장해줍니다. 이와 같은 역할을 해 주는 함수입니다.

이 함수를 if 문마다 else 로 짝지어서 반복해서 적용해주면 VTYPE(char,CHAR)인지 ID 인지를 구분해 줄 수 있게됩니다. (char 이 인식되었다면 그다음문자가 LETTER 나 DIGIT 이 아닌경우에만 <VTYPE,char>로 저장이되고 그렇지 않다면 VTYPE 이 아닌 그냥 ID 토큰이기 때문에 ID 토큰 처리를 해주게됩니다.)

그럼 이제 남은 것이 **KEYWORD** 토큰(앞에서 처리한 if,IF 를 제외한 else,ELSE,while,WHILE,return,RETURN)뿐인데 이는 바로 전에 설명한 VTYPE 토큰인 char,CHAR 를 인식하는 방법과 완벽히 동일한 방법으로 구분합니다. 다른점이 있다면 단지 완성된 결과가 VTYPE 토큰이냐 KEYWORD 토큰이냐 뿐입니다.

```
else if(input[pointer]=='e') //else(KEYWORD),ID구분
{
    token_name="ID";
    pointer++;
    if((pointer < input.length()) && input[pointer]=='l')
    {
        token_name="ID";
        pointer++;
        if((pointer < input.length()) && input[pointer]=='s')
        {
            token_name="ID";
            pointer++;
            if((pointer < input.length()) && input[pointer]=='e')
            {
                token_name="KEYWORD";
                pointer++;
                while ((pointer < input.length()) && (is_letter(input[pointer]) || is_digit(input[pointer])))
                {
                    token_name = "ID";
                    pointer++;
                }
                end_index = pointer - 1;
                write_token_table(input, start_index, end_index, token_name);
            }
            else branch_to_ID(input,&pointer,'e',start_index,end_index);
        }
        else branch_to_ID(input,&pointer,'s',start_index,end_index);
    }
    else branch_to_ID(input,&pointer,'l',start_index,end_index);
}
```

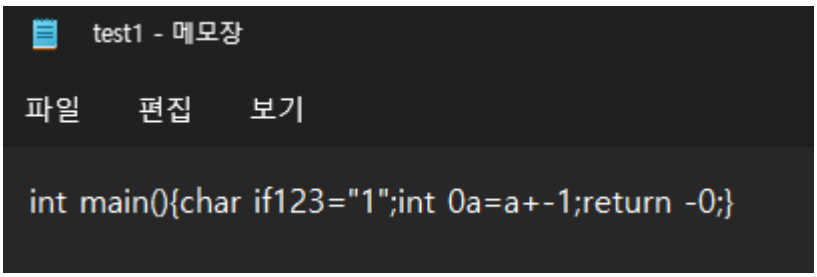
그 예로 else 와 ID 를 구분해주는 부분을 보면 char(VTYPE)과 ID 를 구분해주는 부분과 매우 유사한 방식인 것을 알 수 있습니다. 각 if 문마다 짝지어진 else 문이 있고 그 else 문 마다 branch_to_ID 가 실행되고 있는 것을 볼 수 있습니다. 보이는 차이는 단지 'else'가 완성이 되었을 때 더 이상 LETTER 나 DIGIT 이 인식이 안된다면 token 을 KEYWORD 로 설정하고 테이블에 저장해준다는 차이 뿐입니다.

이와 같은 방식으로 lexical_analyzer 프로그램을 만들었습니다.

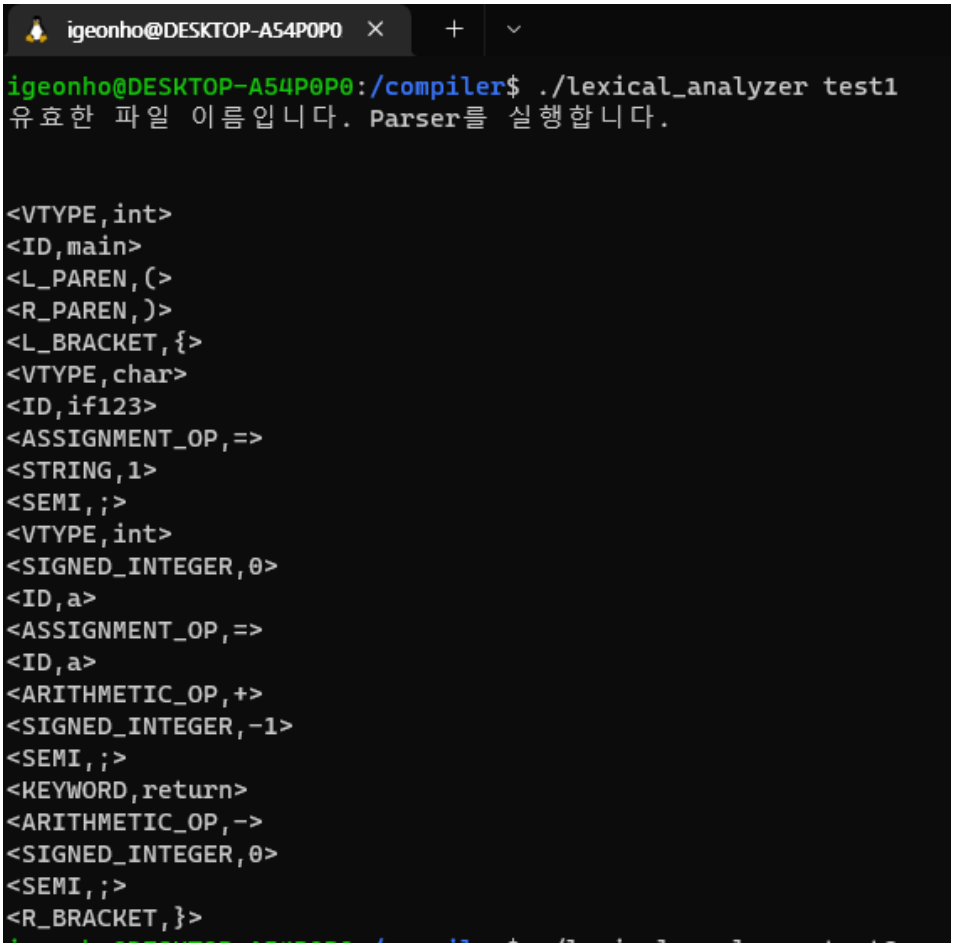
<INPUT,OUPUT example>

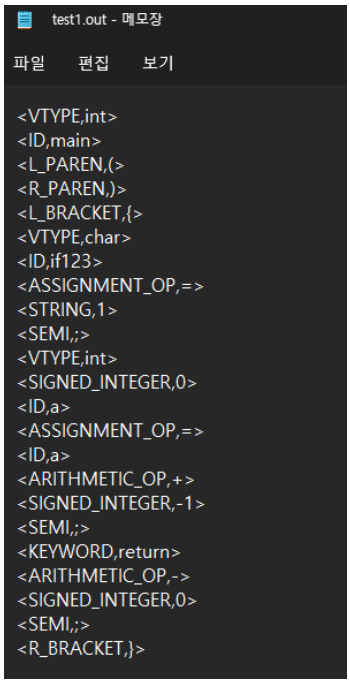
먼저 교수님께서 이클래스에 올려준 예시를 실행시킨 결과를 첨부하겠습니다.

input: test1



output: test1.out

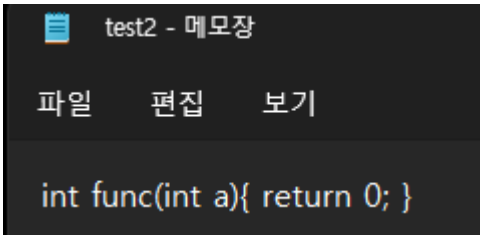




output 파일 생성됩니다

정상적으로 token 이 분류된 것을 볼 수 있습니다.

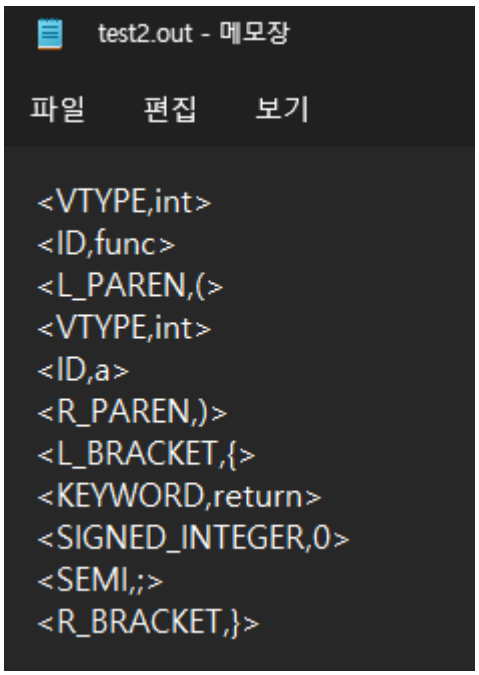
input: test2



output: test2.out

```
igeonho@DESKTOP-A54P0P0:/compiler$ ./lexical_analyzer test2
유효한 파일 이름입니다. Parser를 실행합니다.


<VTYPE,int>
<ID,func>
<L_PAREN,(>
<VTYPE,int>
<ID,a>
<R_PAREN,)>
<L_BRACKET,{>
<KEYWORD,return>
<SIGNED_INTEGER,0>
<SEMI,;>
<R_BRACKET,}>
```

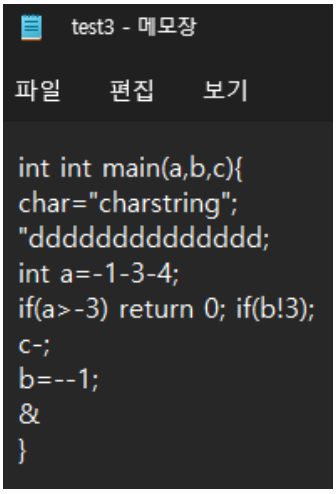


output 파일 생성됩니다

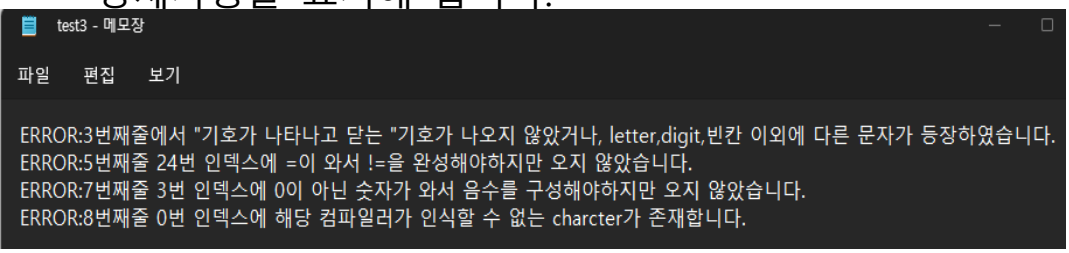
정상적으로 token 이 분류된 것을 볼 수 있습니다.

이번에는 ERROR 를 처리하는 상황을 보여주겠습니다.

input: test3



output: 이때 test3.out 에는 에러내용이 기록됩니다. 또한 콘솔에 에러에 대한 상세사항을 표시해 줍니다.




```
igeonho@DESKTOP-A54P0P0:/compiler$ ./lexical_analyzer test3
유효한 파일 이름입니다. Parser를 실행합니다.


ERROR:3번째줄에서 "기호가 나타나고 닫는 "기호가 나오지 않았거나, letter,digit,빈칸 이외에 다른 문자가 등장하였습니다.
ERROR:5번째줄 24번 인덱스에 =이 와서 !=을 완성해야하지만 오지 않았습니다.
ERROR:7번째줄 3번 인덱스에 0이 아닌 숫자가 와서 음수를 구성해야하지만 오지 않았습니다.
ERROR:8번째줄 0번 인덱스에 해당 컴파일러가 인식할 수 없는 charcter가 존재합니다.
```

에러 메시지를 보면 3 번째줄에서 string 이 구성되어야하는데 "는 안오고 string 을 구성하지 않는 요소인;만 왔기 때문에 error 를 띄웠습니다.

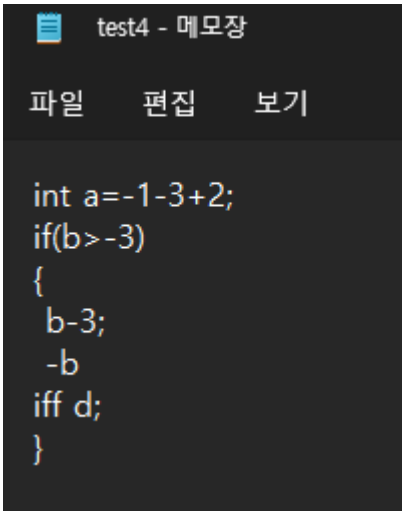
두번째 error 는 5 번째줄에서 b!=3 으로 표시가 되어야하는데 b!3 으로만 되어있기에 error 를 띄웠습니다.

세번째 error 는 7 번째줄에서 =뒤에 오는 -는 음수를 나타내는 -라고 정의했기 때문에 0 이아닌 숫자가 오지 않고 바로 또 -가 왔기 때문에 error 를 띄웠습니다.

네번째 error 는 제가 구현한 lexical analyzer 는 &문자에 대한 처리가 없기 때문에 error 를 띄웠습니다.

마지막으로 추가로 정상적으로 lexical 분석되는 예시입니다.

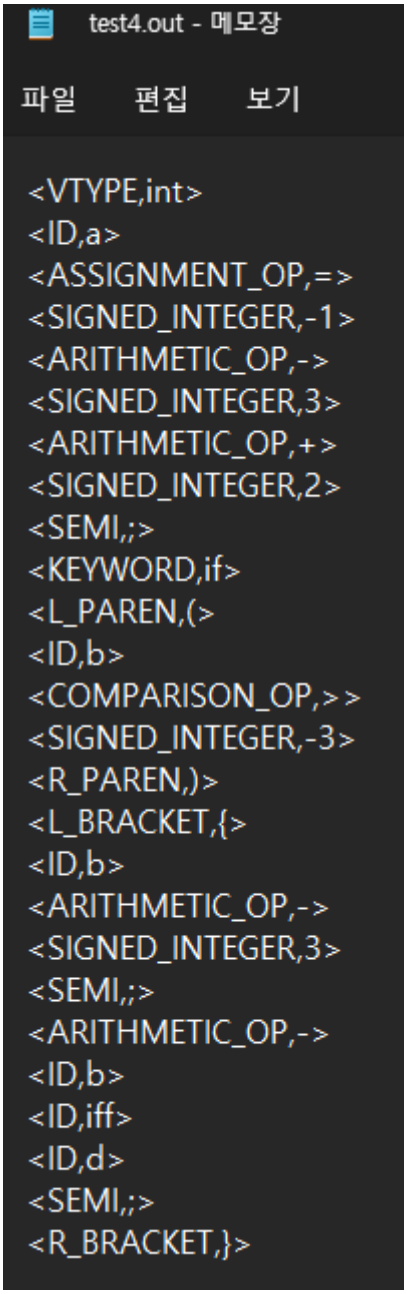
input: test4



output: test4.out

```
igeonho@DESKTOP-A54P0P0:/compiler$ ./lexical_analyzer test4
유효한 파일 이름입니다. Parser를 실행합니다.


<VTYPE,int>
<ID,a>
<ASSIGNMENT_OP,=>
<SIGNED_INTEGER,-1>
<ARITHMETIC_OP,->
<SIGNED_INTEGER,3>
<ARITHMETIC_OP,+>
<SIGNED_INTEGER,2>
<SEMI,;>
<KEYWORD,if>
<L_PAREN,(>
<ID,b>
<COMPARISON_OP,>>
<SIGNED_INTEGER,-3>
<R_PAREN,)>
<L_BRACKET,{>
<ID,b>
<ARITHMETIC_OP,->
<SIGNED_INTEGER,3>
<SEMI,;>
<ARITHMETIC_OP,->
<ID,b>
<ID,iff>
<ID,d>
<SEMI,;>
<R_BRACKET,}>
```



COMPILER PROJECT (syntax analyzer)

20182345 이건호

<CFG>

CFG G: $S' \rightarrow CODE$ (dummy production)

01: CODE → VDECL CODE | FDECL CODE | ϵ

02: VDECL → ~~vtype id semi~~ (VTYPE ID SEMI)

03: FDECL → ~~vtype id lparen~~ ARG ~~rparen lbrace~~ BLOCK RETURN ~~rbrace~~ (VTYPE ID L_PAREN R_PAREN L_BRACKET R_BRACKET)

04: ARG → ~~vtype id~~ MOREARGS | ϵ (VTYPE ID)

05: MOREARGS → ~~comma vtype id~~ MOREARGS | ϵ (SEPARATING VTYPE ID)

06: BLOCK → STMT BLOCK | ϵ

07: STMT → VDECL | ~~id assign~~ RHS ~~semi~~ (ID ASSIGNMENT_OP SEMI)

08: STMT → ~~if lparen~~ COND ~~rparen lbrace~~ BLOCK ~~rbrace~~ else ~~lbrace~~ BLOCK ~~rbrace~~ (KEYWORD L_PAREN R_PAREN L_BRACKET R_BRACKET KEYWORD L_BRACKET R_BRACKET)

09: STMT → ~~while lparen~~ COND ~~rparen lbrace~~ BLOCK ~~rbrace~~ (KEYWORD L_PAREN R_PAREN L_BRACKET R_BRACKET)

10: RHS → EXPR | ~~literal~~ (STRING)

11: EXPR → TERM addsub EXPR | TERM (ARITHMETIC_OP)

12: TERM → FACTOR multdiv TERM | FACTOR (ARITHMETIC_OP)

13: FACTOR → ~~lparen~~ EXPR ~~rparen~~ | ~~id~~ | ~~num~~ (L_PAREN R_PAREN ID SIGNED_INTEGER)

14: COND → FACTOR ~~comp~~ FACTOR (COMPARISON_OP)

15: RETURN → ~~return~~ FACTOR ~~semi~~ (KEYWORD SEMI)

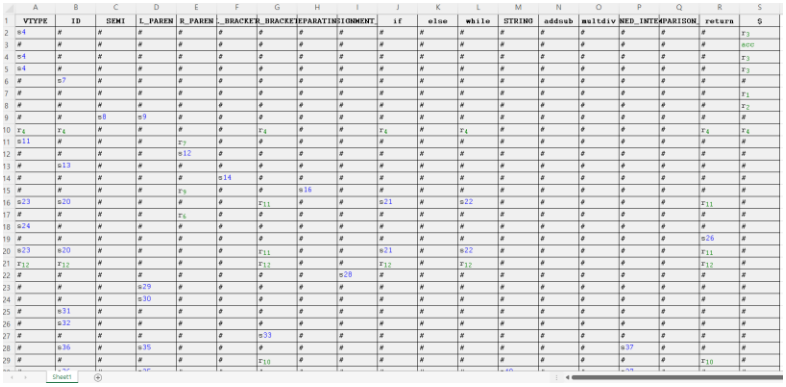
제 lexical analyzer 과정에서 나온 토큰정보에 맞춰서 CFG 를 위와 같이 수정하였습니다. 초록색으로 동그라미 친 것은, 예를들어 if,else,while,return 은 묶어서 KEYWORD 토큰으로 되어있기 때문에 syntax analyzer 과정에서 읽은 토큰이 KEYWORD 토큰일때는 그 토큰이 지닌 value 를 얻어내는 과정을 한번 더 거치게 됩니다.(if 인지 else 인지 while 인지 return 인지...)

참고로 '{'는 L_BRACE 가 맞지만 처음 lexical_analyzer 를 설계할 때 {를 실수로 L_BRACKET 으로 토큰을 지정하였습니다. '}'도 마찬가지입니다.

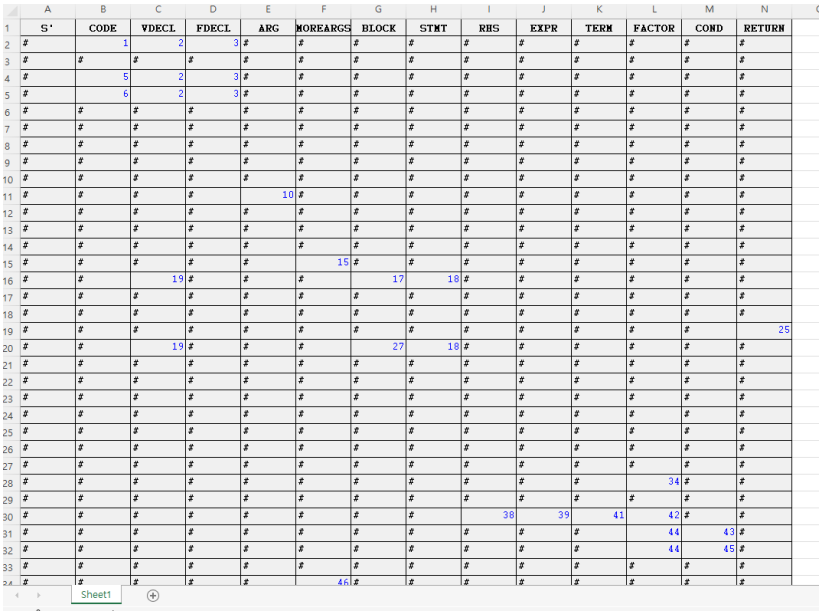
<SLR parsing table >

교수님께서 알려주신 사이트를 기반으로 테이블을 만들었습니다.

1. 먼저 사이트에 CFG 를 넣어주고 SLR-parsing table 을 얻습니다.
2. ACTION 부분과 GOTO 부분을 각각 나눠서 엑셀로 복붙했습니다.



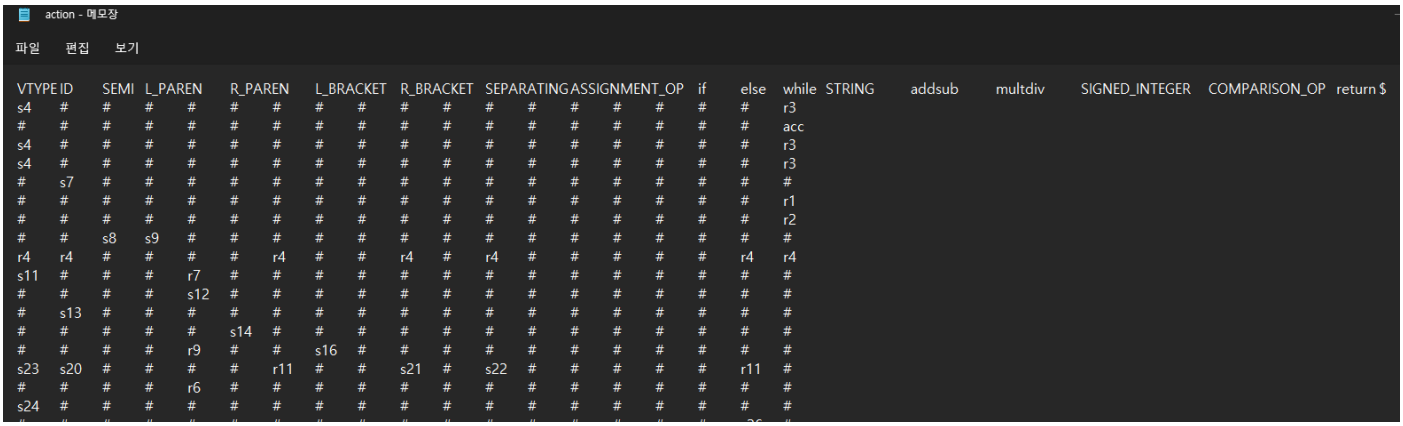
(엑셀로 옮긴 ACTION 테이블)



(엑셀로 옮긴 GOTO 테이블)

이 과정에서 빈 칸에는 '#'기호를 넣어줬습니다. #기호를 통해 빈 칸을 구분하고 에러를 검출할 것이기 때문입니다.

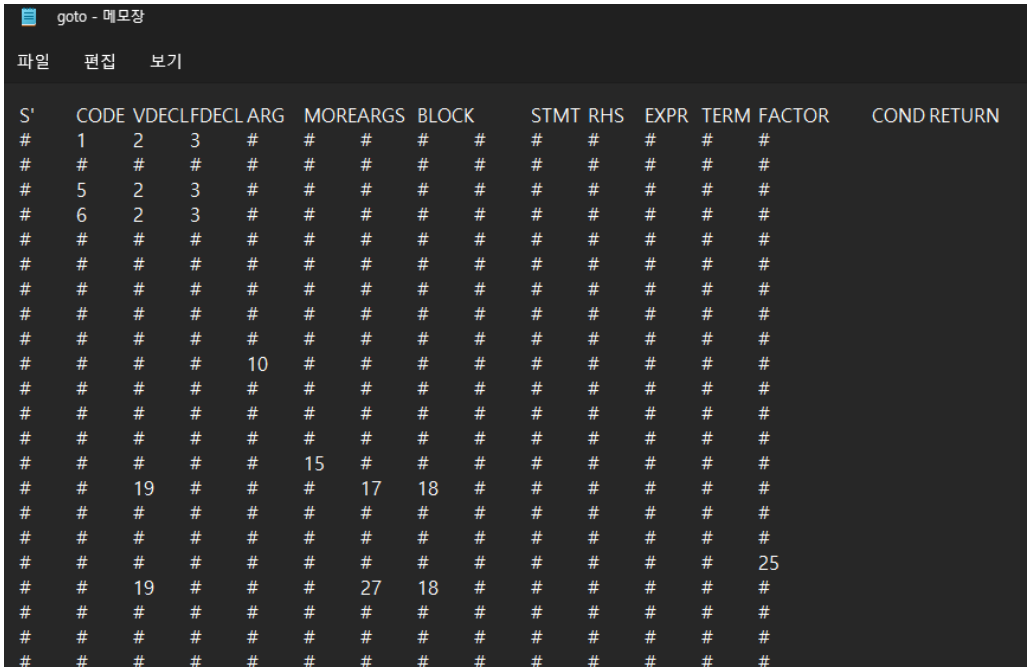
3. 이 엑셀로 수정한 정보를 텍스트파일로 옮겼습니다.(각각 action.txt 와 goto.txt 로 옮김)



action - 메모장

파일 편집 보기

VTYPE	ID	SEMI	L_PAREN	R_PAREN	L_BRACKET	R_BRACKET	SEPARATING	ASSIGNMENT_OP	if	else	while	STRING	addsub	multdiv	SIGNED_INTEGER	COMPARISON_OP	return	$
s4	#	#	#	#	#	#	#	#	#	#	r3							
#	#	#	#	#	#	#	#	#	#	#	acc							
s4	#	#	#	#	#	#	#	#	#	#	r3							
s4	#	#	#	#	#	#	#	#	#	#	r3							
#	s7	#	#	#	#	#	#	#	#	#	#							
#	#	#	#	#	#	#	#	#	#	#	r1							
#	#	#	#	#	#	#	#	#	#	#	r2							
#	#	s8	s9	#	#	#	#	#	#	#	#							
r4	r4	#	#	#	r4	#	#	r4	#	r4	#							
s11	#	#	#	r7	#	#	#	#	#	#	#							
#	#	#	#	s12	#	#	#	#	#	#	#							
#	s13	#	#	#	#	#	#	#	#	#	#							
#	#	#	#	#	s14	#	#	#	#	#	#							
#	#	#	#	r9	#	#	s16	#	#	#	#							
s23	s20	#	#	#	#	r11	#	#	s21	#	s22	#	#	#	#	r11	#	
#	#	#	#	r6	#	#	#	#	#	#	#	#	#	#	#	#	#	
s24	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	



goto - 메모장

파일 편집 보기

S'	CODE	VDECL	FDECL	ARG	MOREARGS	BLOCK		STMT	RHS	EXPR	TERM	FACTOR	COND	RETURN
#	1	2	3	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	5	2	3	#	#	#	#	#	#	#	#	#		
#	6	2	3	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	10	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	15	#	#	#	#	#	#	#		
#	#	19	#	#	#	17	18	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	25		
#	#	19	#	#	#	27	18	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		
#	#	#	#	#	#	#	#	#	#	#	#	#		

이제 이 텍스트파일의 정보를 불러와서 syntax 분석에 사용할 것입니다.

<상세 설명(syntax_analyzer클래스)>

```
class syntax_analyzer
{
    private:
        vector<map<string, string>> action_table; // action테이블 저장. ex)표에서 row(state)=1 이고 column(터미널)="ID"인 값은 action_table[1]["ID"]로 접근함.
        vector<map<string, string>> goto_table;   // goto테이블 저장. ex)표에서 row(state)=1 이고 column(논터미널)="STMT"인 값은 action_table[1]["STMT"]로 접근함.
        vector<string> terminal;                  // action표에서 사용되는 터미널들을 저장함.
        vector<string> nonterminal;               // goto표에서 사용되는 논터미널들을 저장함.
        stack<int> syntax_stack;                  //SLR파싱 과정에서 사용할 스택
        string next_input_symbol;                 //현재 next_input_symbol이 무엇인지 저장
        pair<int,string> reduce_arr[27]; //reduce를 실행할때 스택에서 몇개나 pop해줘야하는지와 A->a에서 A문자열을 저장해줌. ex)R5이면 reduce_arr[5].first의 값만큼 pop을해줌.
    public:
        void set_terminal(string line);              // action테이블의 terminal의 값을 설정해줌.
        void set_action_table(string line, int row); // action테이블에 파일로부터 읽어온 값을 채워 넣는 메소드

        void set_nonterminal(string line);         // goto테이블의 nonterminal의 값을 설정해줌.
        void set_goto_table(string line, int row); // goto테이블에 파일로부터 읽어온 값을 채워 넣는 메소드
        void set_reduce_arr();//reduce_arr 페어에 값을 넣어줌. 이는 SLR파서 테이블을만드는 사이트에서 사용한 정보를 넣어주는 과정임.
        void check_syntax();
};
```

syntax_analyzer 를 위한 클래스입니다.

action_table 은 만들어놓은 action 테이블 정보가 담긴 텍스트파일을 불러온 후 저장해 놓기 위한 변수입니다. vector<map<string,string>>타입을 사용했기 때문에 예를들어 테이블에서 state=3 이고 읽을 다음 토큰(심볼)이 ID 인경우 action_table[3]["ID"]의 방식으로 테이블의 내용에 접근할 수 있습니다.

goto_table 변수는 action_table 과 동일한 방식입니다.

terminal 변수는 action 테이블의 가장 윗줄인 terminal 들을 저장하고 있습니다.

nonterminal 변수는 goto 테이블의 가장 윗줄인 nonterminal 들을 저장하고 있습니다.

syntax_stack 은 syntax 분석에서 사용할 스택입니다. stl 의 stack 을 사용하였습니다.

next_input_symbol 은 교수님 강의자료를 예시로 보면

$T|*id\$$ 이 상황일때 *가 next_input_symbol 에 저장됩니다.

pair<int,string>reduce_arr[27]변수는 action 테이블에서 Reduce 명령을 실행해야할 때 first 부분에 A->a 일 때 |a|의 값을 저장, 즉 pop 해야할 개수를 저장하고, second 부분에 A 문자열을 저장해줍니다. 이 second 부분은 GOTO 테이블을 검색할 때 사용합니다.

public 부분의 메소드들을 보면 check_syntax()를 제외하고 전부 set 이 붙어 있는데 이 메소드들은 set_ 뒤에있는 변수를 초기설정해주는 메소드들입니다.

void check_syntax()메소드는 실제로 syntax 분석을 해주는 실질적으로 메인역할을 해주는 메소드입니다.

<상세 설명(main 함수 부분)>

main 함수의 흐름을 따라가며 설명하겠습니다. lexcial 분석 과정 이후의 설명입니다.

```
if(error==false) //lexical analyser가 정상적으로 실행되었으면 Syntax analyzer 실행
{
    cout<<"Lexical OK.\n";
    syntax_analyzer syntax_analyzer;
    /*****************************************************/
    //action테이블 설정
    in.close();
    in.clear();//action테이블 값이 있는 파일을 읽어오기 위하여 close,clear한뒤 ifstream을(여기선 in) 재사용함.
    in.open("action.txt");
    string temp;
    getline(in,temp);
    syntax_analyzer.set_terminal(temp);
    int row=0;
    while (!in.eof()) //한줄씩 action 테이블에 넣음
    {
        string input;
        getline(in,input);
        syntax_analyzer.set_action_table(input,row);
        row++;
    }
    /*****************************************************/
```

lexical analyzer 가 정상적으로 실행이 되었다면 syntax analyzer 를 위한 과정이 if 문을 통해 시작됩니다.(위와 같이 error==false 일때)

먼저 action 테이블을 텍스트파일로부터 읽어와서 syntax_analyzer 클래스 내에 있는 vector<map<string,string>> action_table 변수에 저장합니다. 이를 위해서 action.txt 파일을 열어서 syntax_analyzer.set_terminal()메소드를 활용해 먼저 terminal 들을 저장해줍니다.

```
void syntax_analyzer::set_terminal(string line)
{
    int index=0;
    int start_index=0;
    int count=0;
    while(index<line.length())
    {
        if(line[index]=='\t' || line[index]=='\n' || line[index]==' ' || line[index]=='\r') //화이트스페이스 무시
        {
            index++;
        }
        else
        {
            start_index=index;
            while(index<line.length() && line[index]!='\t' && line[index]!='\n' && line[index]!=' ' && line[index]!='\r')
            {
                index++;
            }
            if(index==line.length()-1)
            {
                index++;
            }
            count=index-start_index;
            terminal.push_back(line.substr(start_index,count));
        }
    }
}
```

set_terminal 메소드를 보면 읽은 string 에서 화이트 스페이스을 이용해서 terminal 들을 구분하며 vector<string>terminal 변수에 push 하는식으로 저장해줍니다.

그 다음 syntax_analyzer.set_action_table(input,row);를 이용해서 action 테이블의 실 내용을 줄마다,칸마다 구분해서 저장해줍니다.(화이트스페이스 이용)

```
void syntax_analyzer::set_action_table(string line,int row)
{
    int index = 0;
    int start_index = 0;
    int count = 0;
    int terminal_count=0;
    map<string,string> ma;
    action_table.push_back(ma);
    while (index < line.length())
    {
        if (line[index] == '\t' || line[index] == '\n' || line[index] == ' ' || line[index] == '\r') //화이트스페이스 무시
        {
            index++;
        }
        else
        {
            start_index = index;
            while (index < line.length() && line[index] != '\t' && line[index] != '\n' && line[index] != ' ' && line[index] != '\r')
            {
                index++;
            }
            if (index == line.length() - 1)
            {
                index++;
            }
            count = index - start_index;
            action_table[row][terminal[terminal_count]]=line.substr(start_index, count);
            terminal_count++;
        }
    }
}
```

마찬가지로 이번엔 goto 테이블을 syntax_analyzer 클래스 내에 있는 vector<map<string,string>> goto_table 변수에 저장해줘야 하는데 action_table 을 저장하는 과정과 동일하기 때문에 설명은 생략하겠습니다.(이때 goto_table 의 첫줄은 non-terminal 이기 때문에 terminal 대신 nonterminal 변수를 저장하게되는 차이가 있습니다.)

```
/*****************************************************/
in.close();
in.clear();//goto테이블 값이 있는 파일을 읽어오기 위하여 close,clear한뒤 ifstream을(여기선 in) 재사용함.
in.open("goto.txt");
string temp2;
getline(in,temp2);
syntax_analyzer.set_nonterminal(temp2);
int row2=0;
while (!in.eof()) //한줄씩 action 테이블에 넣음
{
    string input;
    getline(in,input);
    syntax_analyzer.set_goto_table(input,row2);
    row2++;
}
/*****************************************************/
```

다시 main 함수를 보면

```
set_tokens_for_syntax();
syntax_analyzer.check_syntax();
```

set_tokens_for_syntax()함수가 실행됩니다.

이것은 어떤 역할을 하냐면 lexical_analyzer 에서 발생하는 output 은 전역변수인 vector<pair<string,string>> token 에 저장됨. 어떤식으로 저장되냐면 예를들어 3 번째 단어가 '('이면 token[2].first="L_PAREN", token[2].second="(" 이런식으로 저장됩니다. 이를 이때 token.first 에 토큰의 이름이 들어있기 때문에 이를 syntax_analyzer 에 그대로 사용하면 좋겠지만 CFG 과정에서 말했듯이 if,else,while,return 은 KEYWORD 라는 토큰으로 묶여있고 +,-,*,/도 2 개씩 addsub,multdiv 가 아닌 묶여서 ARITHMETIC_OP 토큰으로 묶여있기 때문에 이를 위해서 한번 걸러서 저장해주는 작업이 필요합니다.

즉 vector<pair<string,string>> token 전역변수의 정보를 활용해서

vector<string> tokens_for_syntax 전역변수에 가공한 정보를 넣어주는 역할을 해줍니다. 이 tokens_for_syntax 가 실제 syntax_analyze 과정에서 사용됩니다.

```
//syntax analyzer에서 사용하는 CFG에 맞춰서 symbol table의 내용을 적절히 처리해서 tokens_for_syntax전역변수에 넣어줌.
//이 vector타입 변수 tokens_for_syntax는 syntax_analyzer에서 input으로 사용됨
void set_tokens_for_syntax()
{
    for(int i=0; i<token.size(); i++)
    {
        if (token[i].first == "KEYWORD")
        {
            if(token[i].second =="if" || token[i].second =="IF")
                tokens_for_syntax.push_back("if");
            else if(token[i].second =="else" || token[i].second =="ELSE")
                tokens_for_syntax.push_back("else");
            else if(token[i].second =="while" || token[i].second =="WHILE")
                tokens_for_syntax.push_back("while");
            else if(token[i].second =="return" || token[i].second =="RETURN")
                tokens_for_syntax.push_back("return");
        }
        else if(token[i].first == "ARITHMETIC_OP")
        {
            if(token[i].second=="+" || token[i].second=="-")
                tokens_for_syntax.push_back("addsub");
            else if(token[i].second=="*" || token[i].second=="/")
                tokens_for_syntax.push_back("multdiv");
        }
        else
            tokens_for_syntax.push_back(token[i].first);
    }
    tokens_for_syntax.push_back("$");//마지막에 $기호 넣어줘야함.
}
```

위 코드를 보면 KEYWORD 토큰과 ARITHMETIC_OP 토큰에 대해 CFG 에 쓰이는 terminal 과 맞춰주기 위해 정보를 가공해 주는 걸 볼 수 있습니다.

```
set_tokens_for_syntax();
syntax_analyzer.check_syntax();
```

다시 main 으로 돌아가면 이제 실제 syntax 분석 과정인 check_syntax()메소드를 실행합니다.

```
void syntax_analyzer::check_syntax() //CFG에 따른 syntax 적합성 검사
{
    int index=0;
    syntax_stack.push(0);//스택에 start state 넣어줌
    next_input_symbol=tokens_for_syntax[index]; //next input symbol 초기설정. shift할때마다 바뀔예정
    set_reduce_arr();
    while(1)
    {
        string value=action_table[syntax_stack.top()][next_input_symbol];
        if(value[0]=='s') //shift관련 처리. index수정 및 next_input_symbol 수정 필요함.
        {
            syntax_stack.push(get_number(value));
            index++;
            next_input_symbol=tokens_for_syntax[index];
        }
        else if(value[0]=='r')//reduce관련 처리. goto관련 처리도 여기서 함.
        {
            for(int i=0;i<reduce_arr[get_number(value)].first;i++)
            {
                syntax_stack.pop();
            }
            syntax_stack.push(stoi(goto_table[syntax_stack.top()][reduce_arr[get_number(value)].second]));
        }
        else if(value[0]=='a') //acc관련 처리. 마지막인지?
        {
            cout<<"\nSyntax OK. Accept";
            syntax_error_string+="Syntax OK. Accept";
            break;
        }
```

check_syntax()메소드의 상단부분을 보면 먼저 스택에 초기 state 인 0 을 넣어주게 됩니다. index 변수는 next symbol 이 인덱스로는 몇번째인지를 저장하기 위한 변수입니다. next_input_symbol 은 string 타입으로 tokens_for_syntax[index]의 값을 저장하게됩니다. tokens_for_syntax 는 앞에서 말했듯이 input 으로 받은 파일의 일련의 토큰들을 CFG 에서 사용가능하게 순서를 맞춰서 저장하고 있는 vector 타입 변수입니다. 이것에 현재 index 를 사용하여 가리키고있는 다음 symbol 이 무엇인가를 알 수 있습니다.

그래서 next_input_symbol=tokens_for_syntax[index]라는 문장이 나온 이유입니다.

이제 그다음을 보면 set_reduce_arr()이 나옵니다. 이는 syntax_analyzer 인스턴스의 pair<int,string> reduce_arr[27];를 설정해주기 위한 함수인데, 이는 SLR 파싱 테이블에따라 다르게 코드를 적어줘야 하는 부분입니다. 앞에서 말했듯이 reduce_arr[]멤버변수는 SLR 파싱테이블에서 reduce 명령을 수행해야 할 때 사용하는 변수입니다.

```
void syntax_analyzer::set_reduce_arr()
{
    int intarr[27]={1,2,2,0,3,9,3,0,4,0,2,0,1,4,11,7,1,1,3,1,3,1,3,1,1,3,3};
    string stringarr[27]={"S'","CODE","CODE","CODE","VDECL","FDECL","ARG","ARG","MORE
    for(int i=0;i<27;i++)
    {
        reduce_arr[i].first=intarr[i];
        reduce_arr[i].second=stringarr[i];
    }
}
```

reduce_arr[i].first 에는 A->a 일 때 |a|의 값을 저장, 즉 pop 해야할 개수를 저장해야 하기 때문에

```
S' -> CODE
CODE -> VDECL CODE
CODE -> FDECL CODE
CODE -> ''
VDECL -> VTYPE ID SEMI
FDECL -> VTYPE ID L_PAREN ARG R_PAREN L_BRACKET BLOCK RETURN R_BRACKET
ARG -> VTYPE ID MOREARGS
ARG -> ''
MOREARGS -> SEPARATING VTYPE ID MOREARGS
MOREARGS -> ''
BLOCK -> STMT BLOCK
BLOCK -> ''
STMT -> VDECL
STMT -> ID ASSIGNMENT_OP RHS SEMI
STMT -> if L_PAREN COND R_PAREN L_BRACKET BLOCK R_BRACKET else L_BRACKET BLOCK
R_BRACKET
STMT -> while L_PAREN COND R_PAREN L_BRACKET BLOCK R_BRACKET
RHS -> EXPR
RHS -> STRING
EXPR -> TERM addsub EXPR
EXPR -> TERM
TERM -> FACTOR multdiv TERM
TERM -> FACTOR
FACTOR -> L_PAREN EXPR R_PAREN
FACTOR -> ID
FACTOR -> SIGNED_INTEGER
COND -> FACTOR COMPARISON_OP FACTOR
RETURN -> return FACTOR SEMI
```

사용한 CFG 의 ->우측부분에 있는 단어의 개수를 저장해줍니다.

reduce_arr[i].second 에는 A->a 에서 A 부분을 저장해줍니다. 이 second 부분은 GOTO 테이블을 검색할 때 사용합니다.

예를 들어 action 테이블에서 r5 라는 문자열을 수행해야하면 reduce_arr[5].first 의 수만큼 stack 을 pop()해주고 pop 을 다 수행한후 stack 의 가장 top 인 숫자(state)와 reduce_arr[5].second (A->a 에서 A 부분 문자열)가 교차되는 지점을 goto 테이블에서 찾은 뒤, 그 값을 stack 에 push 해줍니다. 이를 위해서 존재하는 변수입니다.

그 뒤 while 문을 돌면서 syntax error 가 발견되거나 정상적으로 accept 되거나 할때까지 syntax 분석을 계속합니다.

```
while(1)
{
    string value=action_table[syntax_stack.top()][next_input_symbol];
    if(value[0]=='s') //shift관련 처리. index수정 및 next_input_symbol 수정 필요함.
    {
        syntax_stack.push(get_number(value));
        index++;
        next_input_symbol=tokens_for_syntax[index];
    }
    else if(value[0]=='r')//reduce관련 처리. goto관련 처리도 여기서 함.
    {
        for(int i=0;i<reduce_arr[get_number(value)].first;i++)
        {
            syntax_stack.pop();
        }
        syntax_stack.push(stoi(goto_table[syntax_stack.top()][reduce_arr[get_number(value)].second]));
    }
    else if(value[0]=='a') //acc관련 처리. 마지막인지?
    {
        cout<<"\nSyntax OK. Accept";
        syntax_error_string+="Syntax OK. Accept";
        break;
    }
```

while 문 절반을 보면 먼저 value 변수에 action 테이블의 현재 state 와 next symbol 이 교차되는 지점의 값을 저장합니다.

이때 value 의 첫 문자가 s 면 shift and goto 연산을 수행해야 합니다. if(value[0]=='s')가 그 부분인데, 이때 스택에 get_number(value)를 통해서 s 뒤에 존재하는 숫자(state)를 얻은 뒤 스택에 push 합니다. 그리고 splitter 를 움직이듯이 index 를 1 증가시켜줍니다. 그리고 난 후 next_input_symbol 을 새로 갱신해줍니다.

value 의 첫 문자가 r 이면 reduce by 연산을 수행해야 합니다. else if(value[i]=='r')이 그 부분입니다. 앞에서 말했듯이 reduce_arr 변수를 활용해서 A->a 에서 |a|의 해당되는 값만큼 stack 에서 pop()을 수행합니다.
그 후, goto 테이블의현재 state 와 reduce 과정에서 사용된 A->a 에서 A 에 해당되는 문자열의 교차 지점의 값(state)를 스택에 push 해줍니다.

value 의 첫 문자가 a 이면 syntax 가 정상적인 것이므로 Accept 를 해줍니다. 그리고 완료되었기에 break 로 while 문을 빠져나갑니다.

참고로 syntax_error_string 전역변수는 syntax analyzer 결과 및 에러를 output 파일로 표시하기위해 문자를 저장하는 용도입니다.

```
else // 오류처리. 표에서 #일때. (즉 빈칸)
{
    cout<<"\nsynatx에 어긋납니다.\n";
    cout<<token_line[index]<<"번째 줄, ";
    cout<<"'"<<token[index].second<<"'가 올 수 없는 자리입니다\n";
    syntax_error_string+="\nsynatx에 어긋납니다.\n"+to_string(token_line[index])+"번째 줄, "+token[index].second+"'가 올 수 없는 자리입니다\n";
    for(int i=0; i<terminal.size(); i++)
    {
        if(action_table[syntax_stack.top()][terminal[i]]!="#")
        {
            string temp_value=action_table[syntax_stack.top()][terminal[i]];
            stack<int> temp_stack(syntax_stack); // 다음에 올수있는 토큰이 과연 유효할지를 검사하기 위한 스택.
            if(temp_value[0]=='r')
            {
                for (int j = 0; j < reduce_arr[get_number(temp_value)].first; j++)
                {
                   temp_stack.pop();
                }
                temp_stack.push(stoi(goto_table[temp_stack.top()][reduce_arr[get_number(temp_value)].second]));
                if(action_table[temp_stack.top()][terminal[i]]!="#")
                {
                    cout<<terminal[i]<<" ";
                    syntax_error_string+=terminal[i]+" ";
                }
            }
            else    {cout<<terminal[i]<<" ";  syntax_error_string+=terminal[i]+" ";}
        }
    }
    cout<<"<== 이 중에서 하나가 등장해야 문법상 맞을 가능성이 있습니다.";
    cout<<"\nsyntax analyzer를 종료합니다.";
    syntax_error_string+="<== 이 중에서 하나가 등장해야 문법상 맞을 가능성이 있습니다.";
    syntax_error_string+="\nsyntax analyzer를 종료합니다.";
    break;
}
```

이제 action 테이블의 값이 s 도아니고 r 도아니고 a 도 아니면 #인경우인데, 즉 빈칸인 경우인데 이는 syntax error 에 해당됩니다. 먼저 error 가 발생한 line 이 몇번째 줄인지 token_line[index]를 활용해서 출력합니다.
(lexical analyzer 를 만들때는 없던 전역변수인데 여기서 활용하기 위해서 lexical analyzer 과정에서 token 을 추가할때마다 token_line 에 해당 토큰이 몇 번째 줄인지를 저장해주는 과정을 추가했습니다.)

그리고 실제 에러가 난 부분이 어느 단어인지를 표시하기위해 token[index].second 를 사용했습니다. tokens_for_syntax[index]를 안 쓴 이유는, 이곳에는 토큰 이름이 들어 있는 것이지 어떤 문자인지가 안들어있기 때문입니다.(예를들어 {가 아닌 L_BRACKET 이 들어있음.}
참고로 '{'는 L_BRACE 가 맞지만 처음 lexical_analyzer 를 설계할 때 {를 실수로 L_BRACKET 으로 토큰을 지정하였습니다. '}'도 마찬가지입니다.

그리고 terminal.size()는 terminal 의 갯수를 의미합니다. 이 에러처리 부분을 보면 이 terminal 의 갯수만큼 for 문을 돌리는데 왜 그런것이냐면, 에러가난 state 에서 어떤 terminal 을 만나야 정상적인 작동이 될 것인가를 확인하기 위해서입니다. 즉, 와야할 terminal 이 이어서 안왔기 때문에 error 가 나온것이기 때문에 그 와야할 terminal 이 어떤것인지를 하나씩 state 와 연결해보면서 확인하는 것입니다.

	ACTION				
	*	(	)	id	$
1		S4		S5	
2					R(1)
3	S6		R(3)		R(3)
4		S4		S5	
5	R(5)	#	R(5)		R(5)
6		S4		S5	
7			S9		
8			R(2)		R(2)
9	R(4)		R(4)		R(4)

예를 들어서 error 가 state 가 5 일 때 next symbol 이 '(' 여서 빈칸(#)이기 때문에 error 상태라고 가정하면, 그 외 나머지 terminal 들인 *) id $를 대입해보면서 과연 어느 terminal 이 와야지 syntax analyzer 를 성공할 가능성이 생기는지를 확인하는 과정입니다.

```
for(int i=0; i<terminal.size(); i++)
{
    if(action_table[syntax_stack.top()][terminal[i]]!="#")
    {
        string temp_value=action_table[syntax_stack.top()][terminal[i]];
        stack<int> temp_stack(syntax_stack); // 다음에 올수있는 토큰이 과연 유효할지를 검사하기 위한 스택.
        if(temp_value[0]=='r')
        {
            for (int j = 0; j < reduce_arr[get_number(temp_value)].first; j++)
            {
               temp_stack.pop();
            }
            temp_stack.push(stoi(goto_table[temp_stack.top()][reduce_arr[get_number(temp_value)].second]));
            if(action_table[temp_stack.top()][terminal[i]]!="#")
            {
                cout<<terminal[i]<<" ";
                syntax_error_string+=terminal[i]+" ";
            }
        }
        else    {cout<<terminal[i]<<" ";  syntax_error_string+=terminal[i]+" ";}
    }
}
```

그래서 보면 현재 state 에 i 번째 terminal 을 대입했을 때 action 테이블에서의 값이 #(빈칸)이 아니고 r 이면 reduce by 명령을 가상으로 1 번만 수행해보고 그 수행한 결과가 아직 syntax analyzer 를 계속 할 가능성이 있는 경우에는 해당 terminal 을 출력해줍니다. (문법적으로 맞을 가능성이 있기때문에).

또는 현재 state 에 i 번째 terminal 을 대입했을 때 action 테이블에서의 값이 #(빈칸)이 아니고 acc 나 s 인 경우에는 가능성이 있다고 생각해서 해당 terminal 을 출력해주었습니다.

모든 terminal 들을 돌면서 가능한 녀석들을 다 출력해 주었으면

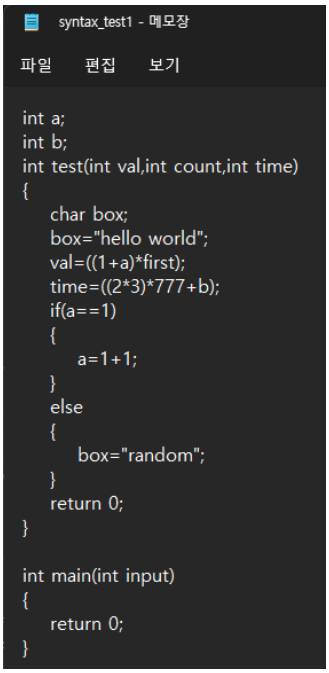
```
cout<<"<== 이 중에서 하나가 등장해야 문법상 맞을 가능성이 있습니다.";
cout<<"\nsyntax analyzer를 종료합니다.";
syntax_error_string+="<== 이 중에서 하나가 등장해야 문법상 맞을 가능성이 있습니다.";
syntax_error_string+="\nsyntax analyzer를 종료합니다.";
break;
```

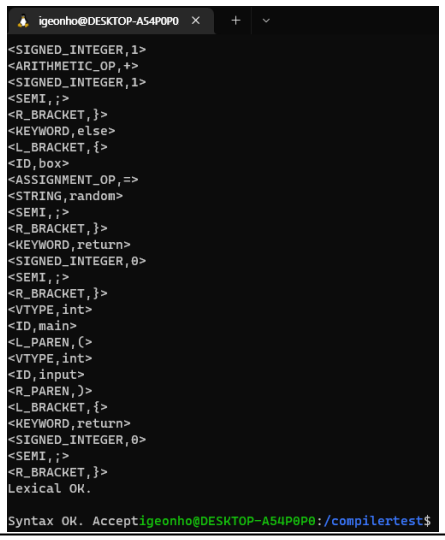
이제 앞에서 출력한 녀석들 중 하나가 등장해야 문법상 맞을 가능성이 있다고 출력해줍니다. 그리고 syntax analyzer 를 종료해줍니다. (break 문 사용해서 while 문 탈출). 그리고 해당 메시지들은 모두 syntax_error_string 에 저장되어서 output 파일로도 생성됩니다.

<INPUT,OUPUT example>

먼저 정상적으로 실행되는 경우를 확인해보겠습니다. (가능한 경우를 최대한 적어봤습니다.)

input: syntax_test1 / 콘솔창결과

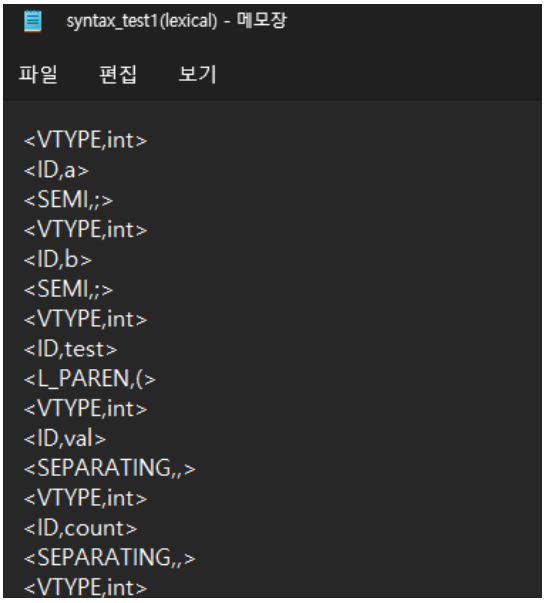


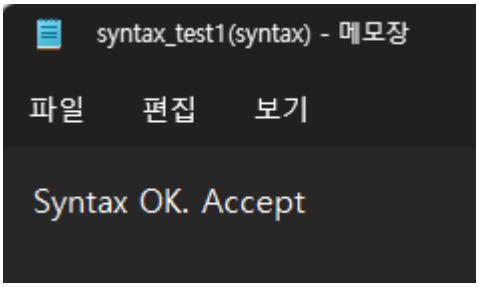


콘솔상에서 lexical analyzer 과정에서 얻은 token 값이 나오고 Lexical OK.가 출력되면서 lexical 분석과정이 성공했다는걸 알려줌.

그 다음 Syntax OK.메시지를 출력하여 Syntax 분석과정에 문제가 없음을 알려줌. 이 두 가지 내용은 각개의 파일(**syntax_test1(lexical).out** 이랑 **syntax_test1(syntax).out**) 로 저장됨

output: syntax_test1(lexical).out 와 syntax_test1(syntax).out



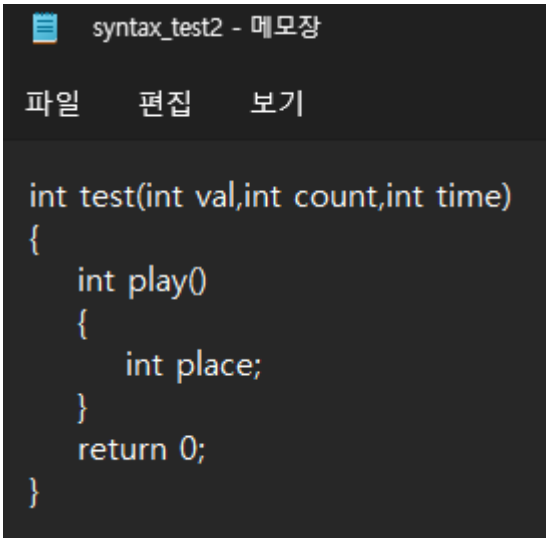


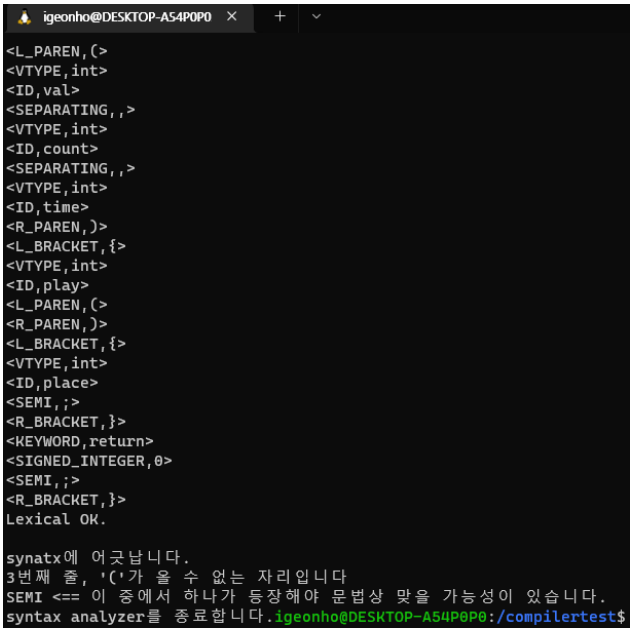
이렇게 lexical 과정의 output 과 syntax 과정의 output 이 각각의 파일로 나오게 프로그램을 짰습니다.

만약 lexcial 과정에서 에러가 있다면 syntax 과정은 실행되지 않고 그렇기에 syntax 과정의 output 파일은 생성되지 않습니다. 이때는 오직 lexical 과정의 error 리포트만 생성됨.

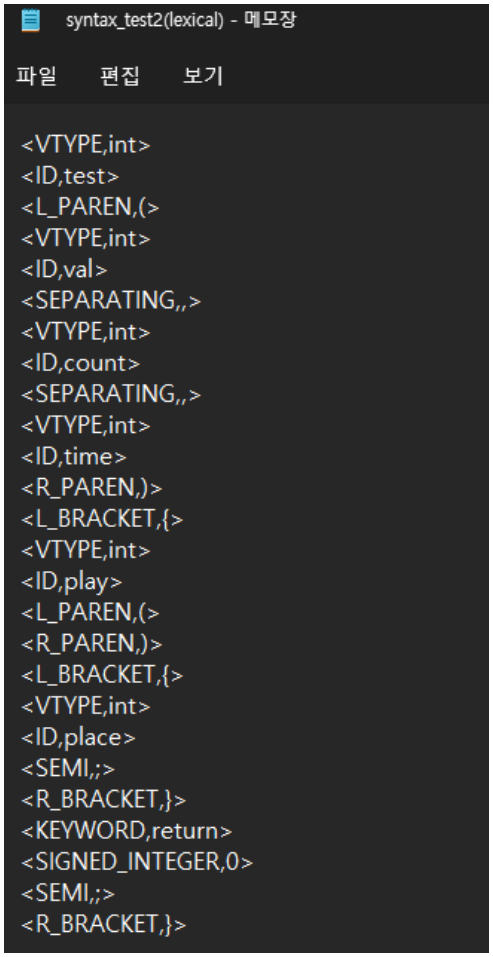
이번에는 CFG 상 함수 속 함수는 불가능하므로 해당 케이스의 syntax error 를 발견하는 예시입니다.

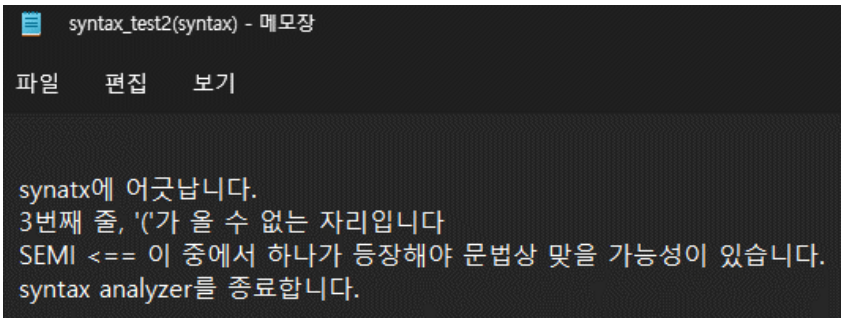
input: syntax_test2 / **콘솔창결과**





output: syntax_test2(lexical).out 와 syntax_test2(syntax).out

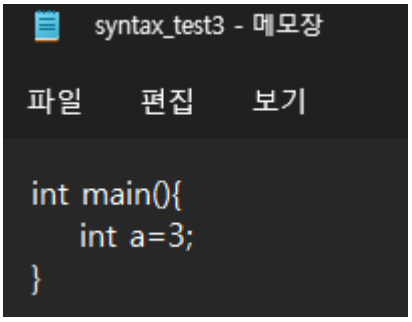


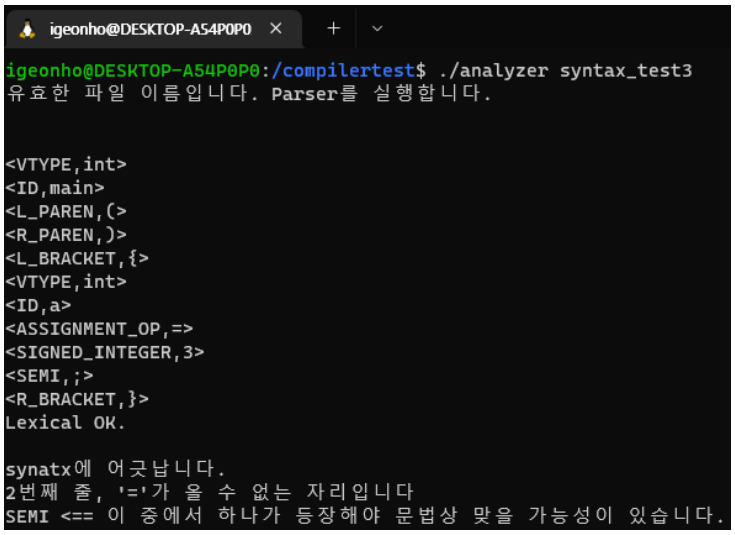


syntax 에러 리포트를 보면 3 번째줄 '('기호에서 문제를 발견한 것을 볼 수 있다. int play(가 되는 순간 이건 함수가 시작된다는 의미이기 때문에 ';'(SEMI)기호가 와야 한다고 알려주고있다. 즉 int play;가 맞는 표현이라고 에러리포트에서 가능성을 제안해주는 것이다.

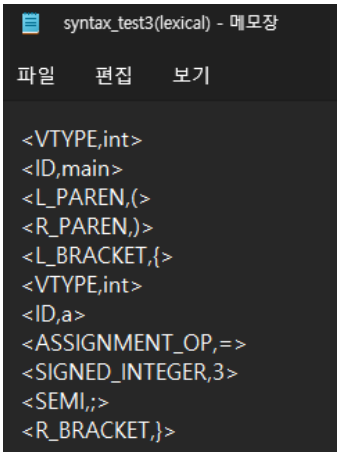
이번에는 CFG 상 선언과 동시에 초기화가 안되기 때문에 해당 케이스의 syntax eror 를 발견하는 예시입니다.

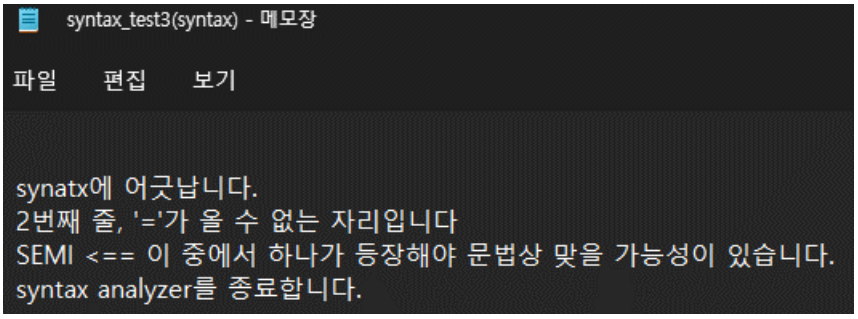
input: syntax_test3 / 콘솔창결과





output: syntax_test3(lexical).out 와 syntax_test3(syntax).out



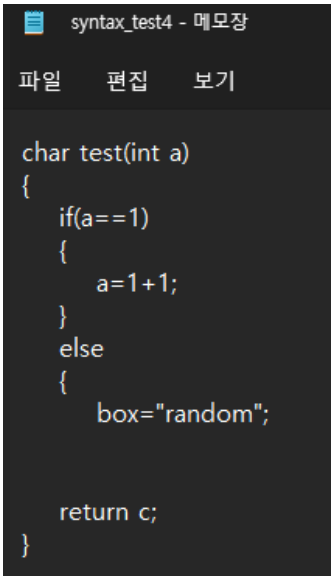


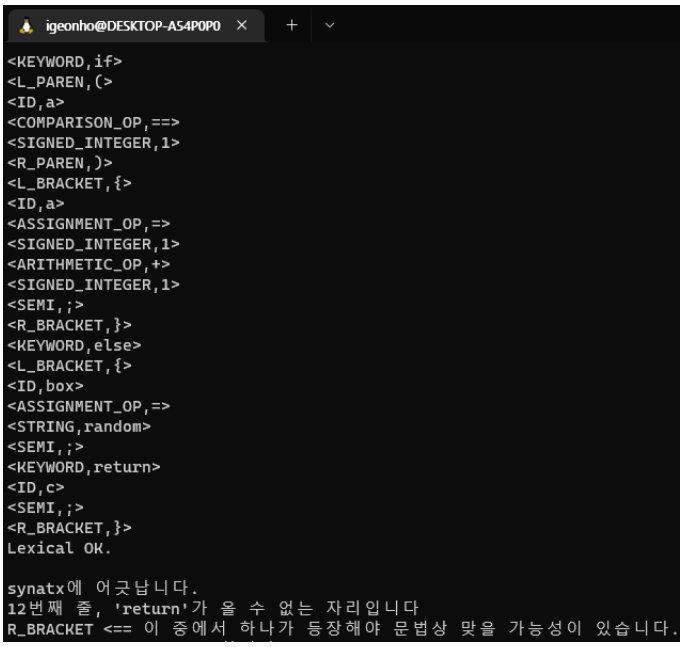
syntax 에러 리포트를 보면 2 번째줄 int a=3;에서 '='기호에서 문제를 발견한 것을 볼 수 있다.

문법상 선언과 동시에 초기화가 안되기 때문에 int a 다음 ';'가 와야 정상적일 것이라고 에러리포트에서 제안하고 있다.

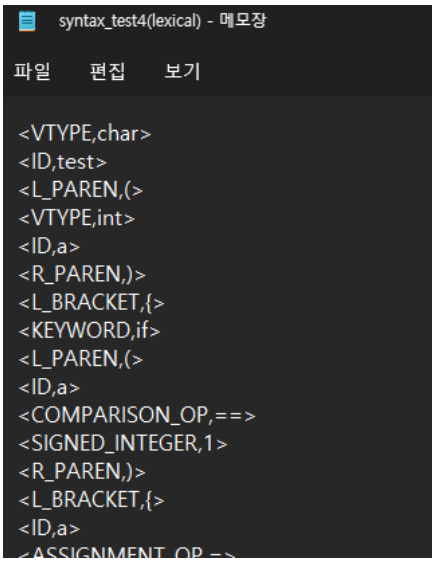
이번에는 {가 나타난 뒤에 }가 오지 않은 경우의 syntax error 를 발견하는 예시입니다.

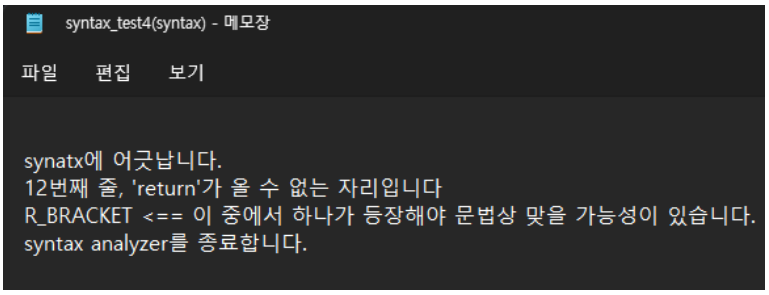
input: syntax_test4 / **콘솔창결과**





output: syntax_test4(lexical).out 와 syntax_test4(syntax).out

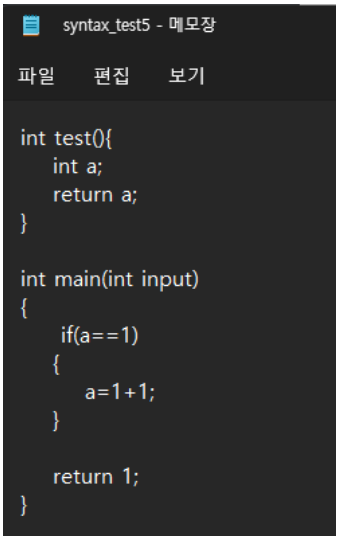


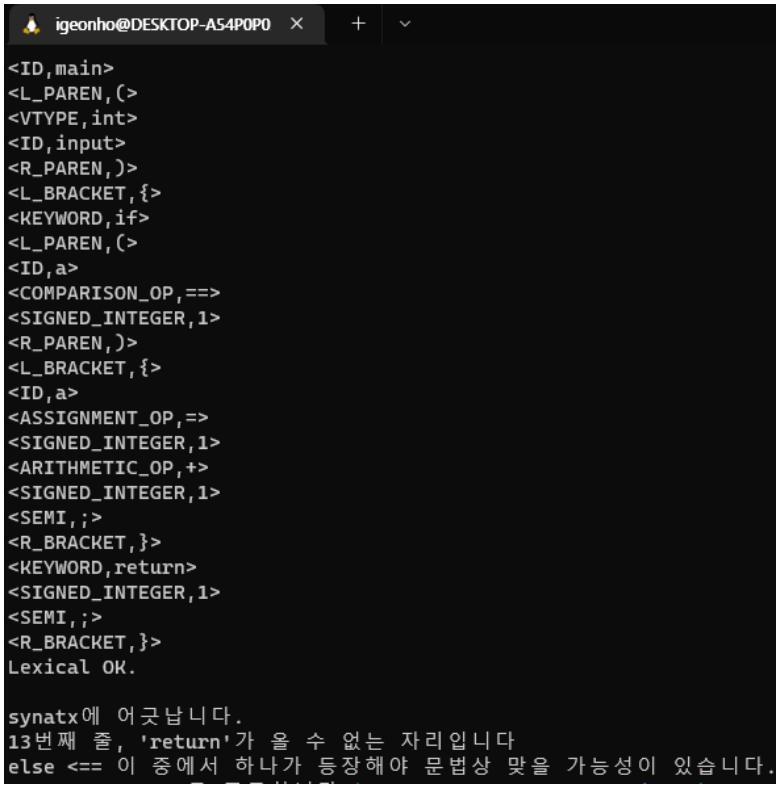


syntax 에러 리포트를 보면 12 번째줄 return 이 올 수 없다고 적혀있다. 이는 else 구문에서 '}'가 나오지 않았는데 return 이 나와버렸기에 뜨는 것이다.
(주어진 CFG 상 if-else 구문속 return 불가능)
그렇기 때문에 '}'이 등장해야 정상적일 것이라고 에러리포트에서 제안하고 있다.

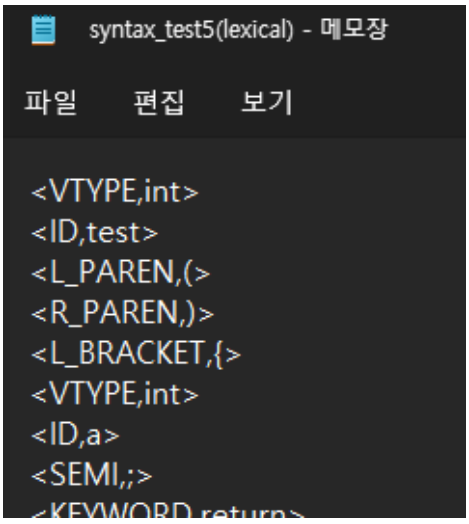
이번에는 if 문만오고 else 문이 안온 경우의 syntax error 를 발견하는 예시입니다. 주어진 CFG 를 따르면 if 만 단독으로 올 수 없습니다.

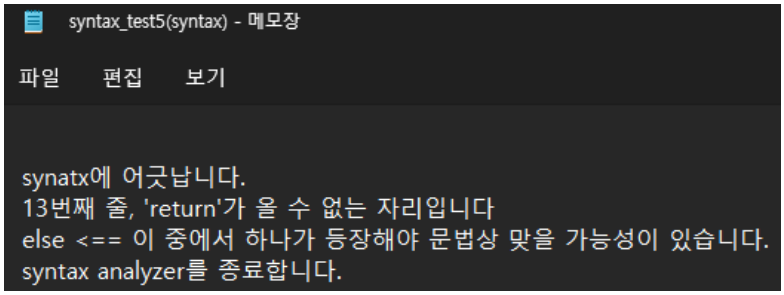
input: syntax_test5 / **콘솔창결과**





output: syntax_test5(lexical).out 와 syntax_test5(syntax).out





syntax 에러 리포트를 보면 13 번째줄 return 이 올 수 없다고 되어있다. 왜냐하면 if 문이 끝났는데 이제 else 문이 와야할 차례인데 return 이 왔기 때문이다.

그렇기 때문에 'else'가 등장해야 문법상 맞을 가능성이 있다고 에러리포트에서 제안하고 있다.

제가 준비한 예시는 여기까지입니다.

<문제점>

문제점이 있었다면 action 테이블과 goto 테이블을 윈도우에서 txt 파일에 저장한 것을 리눅스에서 옮긴 후 그대로 실행하면 정상적으로 실행이 되지 않았습니다.

이는 윈도우와 유닉스의 텍스트를 다루는 방식(개행 문자)때문에 그런 것이었는데 그렇기에 우분투에서 vi 로 action.txt 와 goto.txt 를 열어서 :se ff=unix 명령어를 통해 유닉스텍스트로 변경하여 주었습니다. 이렇게 하니 정상적으로 작동하였습니다.

그리고 '{'이 문자와 '}'이 문자를 L_BRACE 와 R_BRACE 가 아닌 L_BRACKET 과 R_BRACKET 으로 토큰 이름을 잘못 정한 점이 살짝 아쉬운 점이었습니다.

참고로 프로그램을 실행할 때 action.txt 와 goto.txt 는 실행 파일(바이너리)과 같은 디렉토리 내에 위치하여야 합니다. input 파일도 마찬가지입니다.

syntax 에러를 잘 찾고 에러가 발생한 위치(where)와 어떻게 하면 고칠 수 있을까,(why)까지 잘 작동하여서 개인적으로 만족스러운 프로젝트였습니다.