**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]=(**

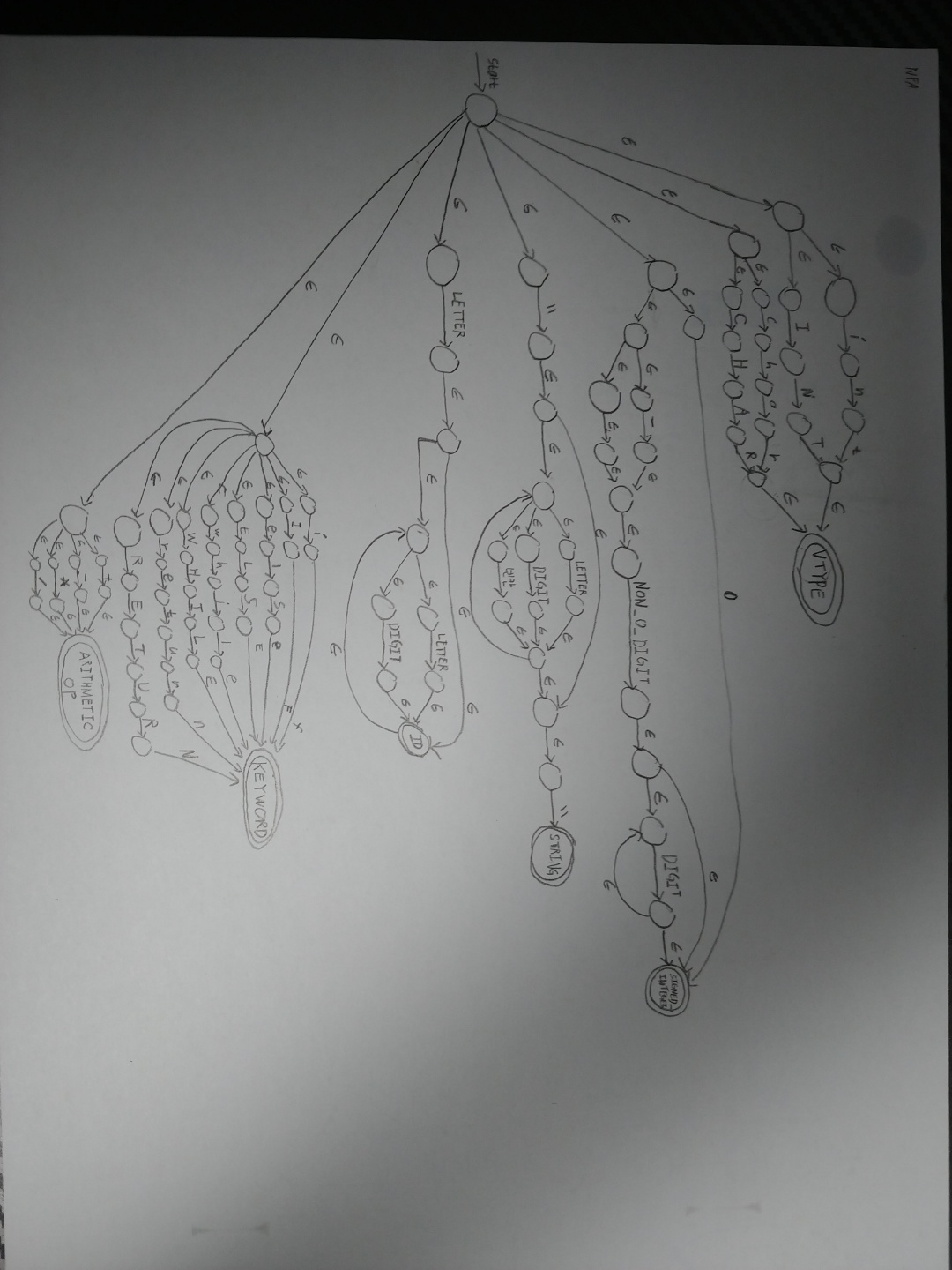
**[R\_PAREN]=)**

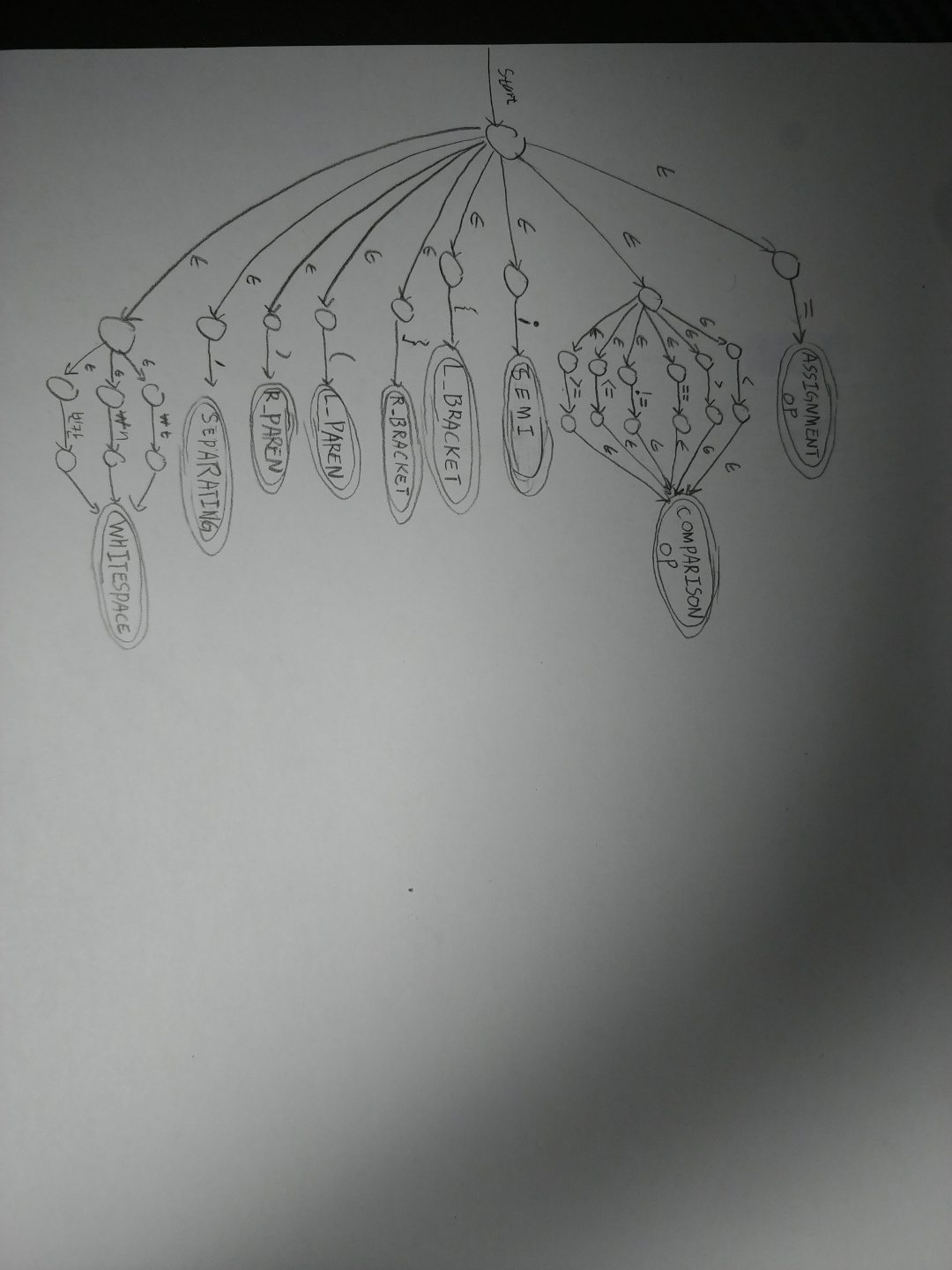
**[SEPARATING]=,**

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

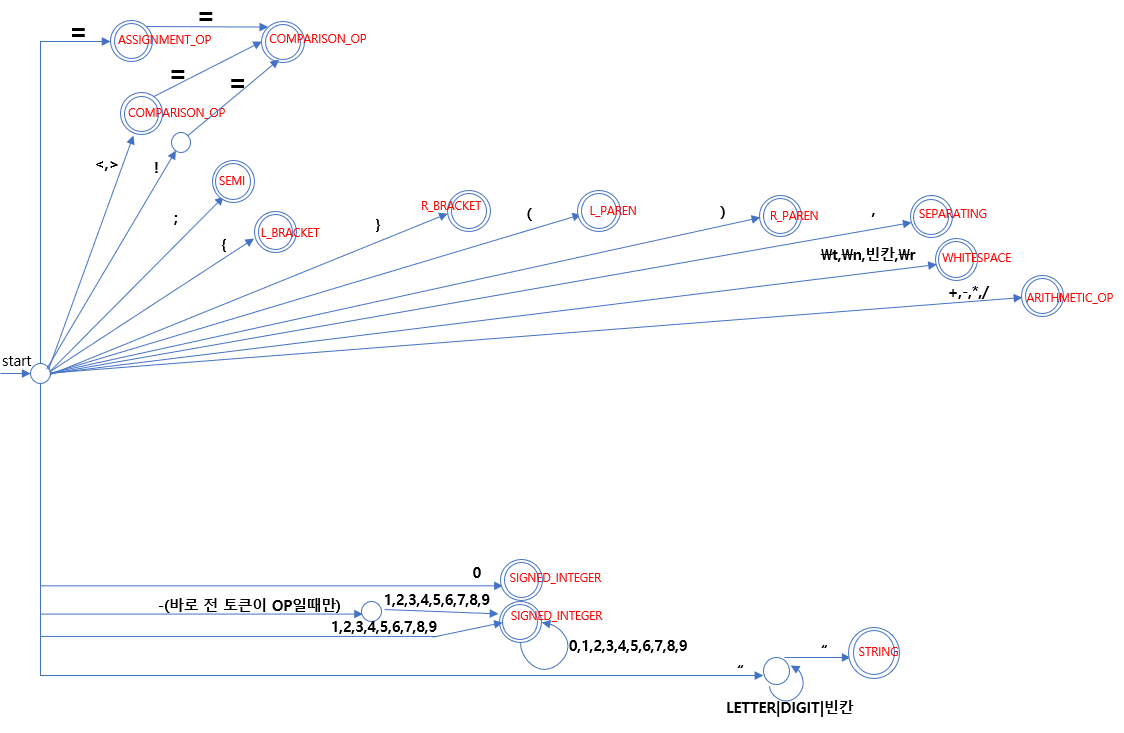
**op의-와 음수의-는 바로앞에 op**(arithmetic이든 assignment든 comparison이든)가 있을 경우에만 음수의-이고 그 외에는 op처리

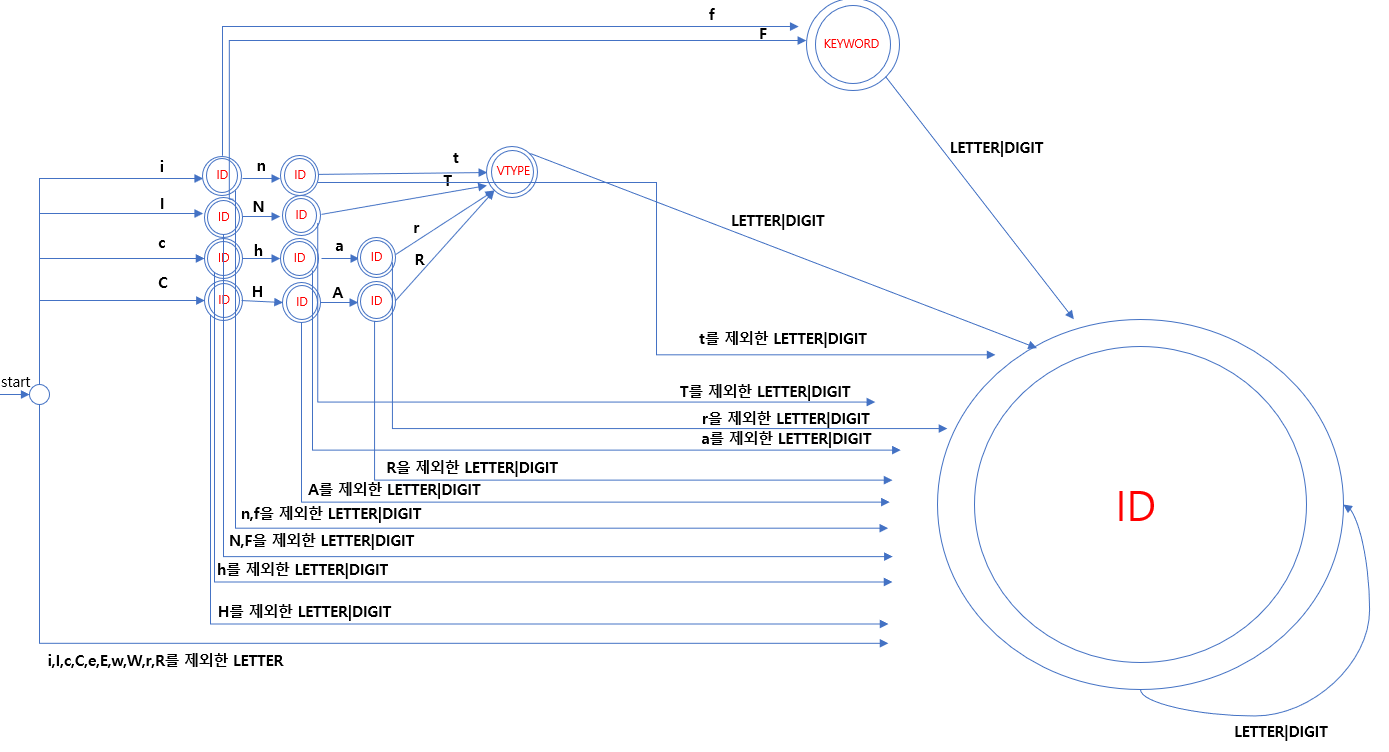
**<NFA transition graph>**

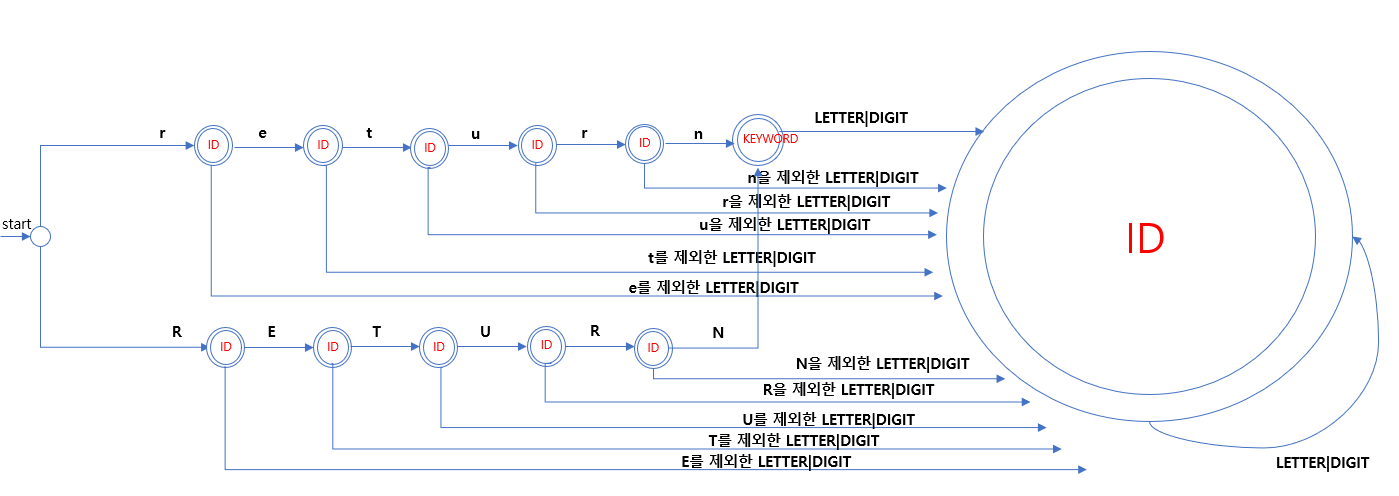
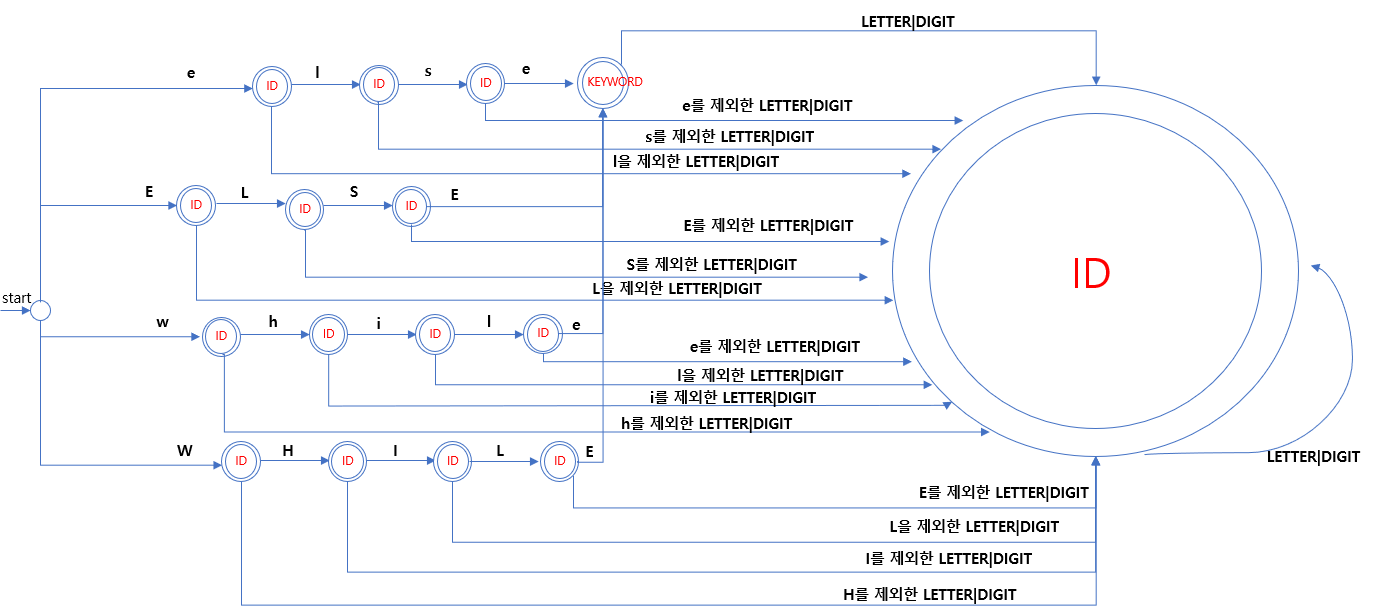
종이 2장에 나눠서 그렸습니다. (같은 start지점을 지님)



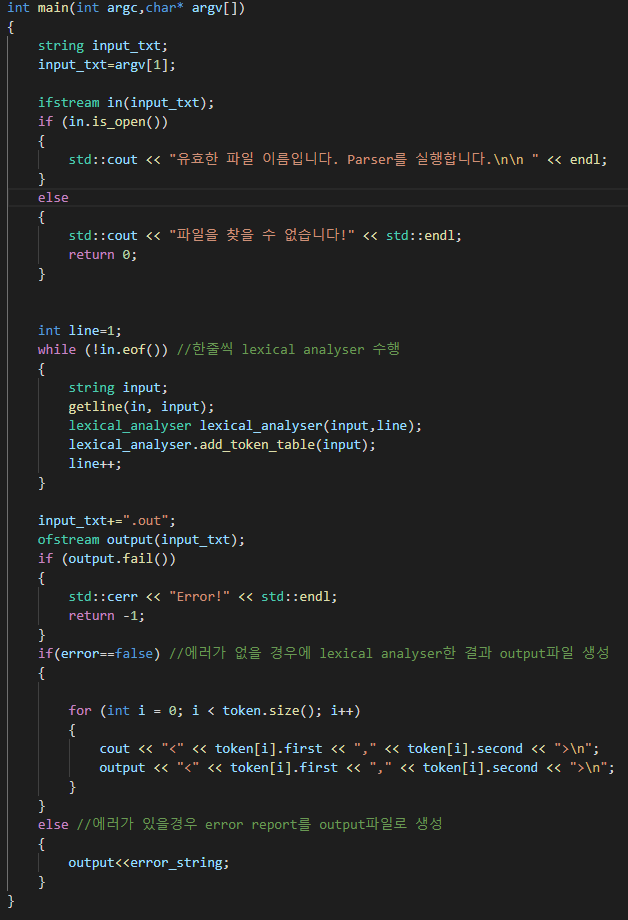
**<DFA transition graph>**

파워포인트를 이용해서 그렸습니다. 4장으로 나눠서 그렸으며 모두 같은 start지점을 지님.



**<상세 설명>**

먼저 main함수부터 흐름을 설명하겠습니다.



main함수의 인자로 lexical analyzer에 집어넣어줄 파일명을 받아줍니다.

그 후 파일이 정상적으로 열렸을 경우에만 실행을 하게 됩니다.

while문을 통해서 input파일의 한줄씩 읽으며 lexical분석을 실행하게 됩니다. 한줄씩 읽기 때문에 에러를 표시해줄 때 몇 번째 줄, 몇 번째 인덱스에서 발생한 것인지 상세히 표시해줄 수 있습니다.

이때 lexical분석을 위하여 lexical\_analyser라는 class를 만들어 줬습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명해당 클래스의 구성은 다음과 같습니다.

먼저 **line**은 error를 표시해줄 때 input파일의 몇 번째 줄에서 문제가 생긴 것인지 표시하기 위해 존재합니다.

**pointer**는 input파일로부터 읽어온 한 줄이 있을 텐데, lexical분석을 할 때, 그 한 줄의 몇 번째 index를 가리키고 있는지를 나타내기 위하여 선언해줬습니다.

그리고 **void add\_token\_table(string input)** 메소드는 본격적으로 lexical분석을 해주기 위한 장치입니다. input(읽은 한 줄)을 분석해주는 역할을 합니다.



lexical분석 과정에서 생성되는 결과물은(<token name, token value>형식**) 전역변수로 선언된 vector<pair<string,string>>token**에 저장됩니다. pair와 vector를 혼합하여서 token name과 token value가 짝을 이루면서 순서대로 쉽게 저장될 수 있게 하였습니다. 또한 bool타입의 error전역변수는 lexical분석 과정에서 error발생여부를 확인하기 위한것이며 string error\_string은 에러내용 기록용입니다.

그럼 이번 프로젝트의 핵심인 **add\_token\_table(lexical분석 과정)**이 어떻게 작동하는지에 대하여 설명하겠습니다.

텍스트, 실내, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

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

is\_digit은 해당 캐릭터가 0~9에 속하는지, is\_non\_0\_digit은 1~9에 속하는지,

is\_letter는 해당 캐릭터가 알파벳 소,대문자에 속하는지를 판별해주는 기능을 합니다.

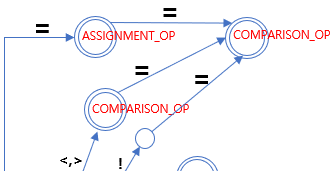
텍스트이(가) 표시된 사진

자동 생성된 설명

위 함수는 lexical분석과정에서 적절히 lexeme을 나눴으면 그 lexeme에 해당하는 token의 이름과 lexeme(=token value)를 앞서 설명한 vector<map<string,string>> token이라는 전역변수에 저장해주는 기능을 합니다. **즉, symbol table(token table)에 추가해 주는 기능입니다.**

**그럼 lexical analyzer가 작동하는 걸 비슷한 방식으로 분석되는 token 카테고리로 나눠서 설명하겠습니다.**

먼저 **ASSIGNMENT\_OP**토큰, **COMPARISON\_OP**토큰 입니다.

DFA에서는 이 부분입니다.

텍스트, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

먼저 인식된 문자가 ‘=’라면 일단은 token을 ASSIGNMENT\_OP로 설정합니다. 그리고 pointer를 하나 증가시킨 후, 다음 문자를 검사하였을 때 또 ‘=’가 나왔다면

‘==’이기 때문에 token을 COMPARISON\_OP로 설정해 줍니다. 이렇게 이중 if문을 사용하여서 =와 ==를 구분하도록 처리하였고 해당되는 token과 value를 앞서 말했던 write\_token\_table이라는 함수를 이용하여 테이블에 저장합니다.

참고로 lexical분석 단계에서 pointer는 항상 인식된 token다음을 가리키도록 구현했습니다. 그렇기 때문에 매 과정마다 pointer++가 있고 pointer가 input의 길이를 넘겼는지 확인하고 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

인식된 문자가 =이 아니라 <이거나 >라면 먼저 token을 COMPARISON\_OP로 설정합니다. 앞에서와 마찬가지로 다음 문자를 검사했을 때 =이라면 <=이거나 >=인 것이기 때문에 TOKEN은 여전히 COMPARISON\_OP입니다. 그리고 분석된 결과를 테이블에 저장해줍니다.



인식된 문자가 !라면 일단은 token을 error로 설정합니다. 왜냐하면 이번에 과제로 구현한 컴파일러에서 !단독으로 쓰이는 토큰은 없기 때문입니다. 다음 문자를 검사했을 때 =가 나온다면 !=이기 때문에 token을 COMPARISON\_OP로 설정하고 분석된 결과를 테이블에 저장해줍니다. 하지만 다음 문자가 =가 아니라면 !혼자 왔기때문에 ERROR를 표시해줍니다.

이번엔 **SEMI**토큰, **L\_BRACKET**토큰, **R\_BRACKET**토큰, **L\_PAREN**토큰, **R\_PAREN**토큰, **SEPARATING**토큰, **WHITESPACE**토큰 입니다.

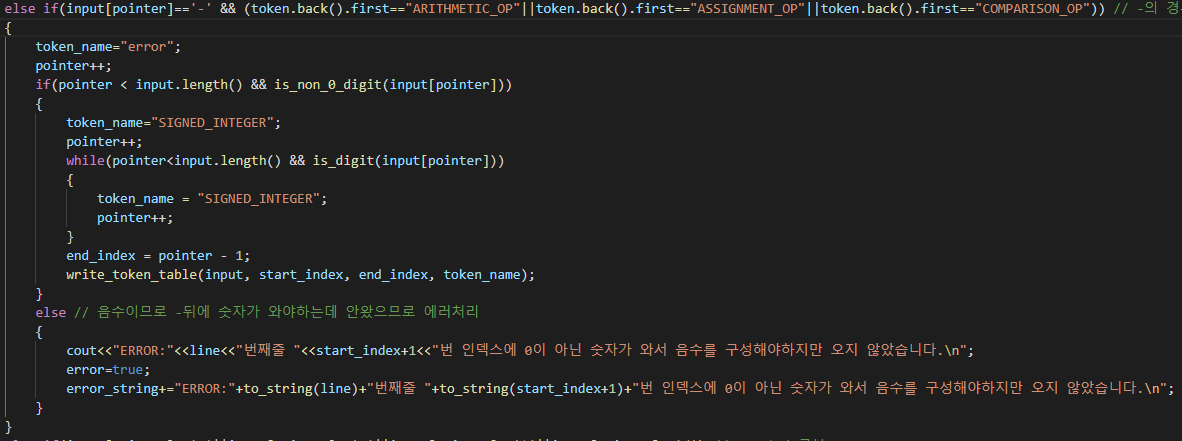
텍스트이(가) 표시된 사진

자동 생성된 설명

위 토큰들은 단일 문자로 분석된다는 특징이 있습니다. 그렇기 때문에 비교적 간단합니다. SEMI토큰을 예로 들면 인식된 문자가 ;라면 token을 SEMI로 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 OP다음에 오는(ASSIGNMENT,COMPARISON,ARITHMETIC) ‘-‘기호라면 SIGNED\_INTEGER에 속하는 ‘-‘인 것으로 처리하였고 그렇지 않은 경우는 연산자로서의, 즉 ARITHMETIC\_OP의 ‘-‘기호인 것으로 처리하였습니다. 이렇게 한다면 a=-1+2같은 경우에 -1이런식으로 묶어지고, a<-3이 경우도 OP다음에 온 -이기때문에 -3으로 묶어지게 됩니다. 즉, 상식적인 프로그래밍 환경에서는 제가 처리한 방식으로 ‘-‘기호를 어느 토큰에 넣어줄지를 구분할 수 있을 것이라 생각합니다.



위 코드를 보면 인식된 기호가 -이고 바로 이전에 저장된 token이 OP관련 token이라면 SIGNED\_INTEGER로 TOKEN을 설정해주는 걸 알 수 있습니다. 이때 EROOR처리도 해주었는데 음수를 나타내는 -인 것이 확정되었기 때문에 그 뒤 문자를 검사하여서 그 문자가 0이아닌 숫자가 아니면 error처리를 해주었습니다.

텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

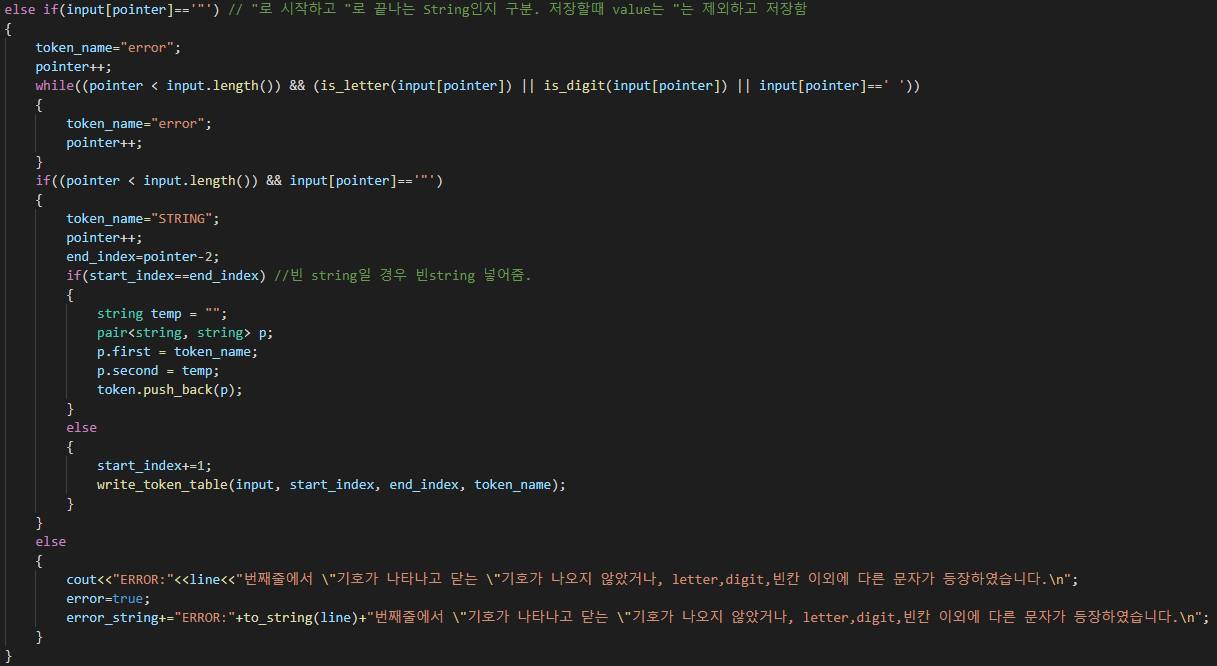
ARITHMETIC\_OP는 음수의 -인 경우를 제외하고는 전부 연산자-이기 때문에 +,-,\*,/를 한번에 합쳐서 else if문에 넣어줬습니다.(음수의 -를 구분하는 else if문 뒤에 위치함) 그렇게 하여 간단히 구분해줄 수 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 단순 숫자0과 양의정수도 인식하여서 SIGNED\_INTEGER토큰으로 설정해주고 테이블에 저장해줬습니다.

이번엔 **STRING**토큰입니다.



먼저 “문자가 인식되면 letter또는 digit또는 빈칸이 더 이상 안 나올때까지 계속하여 읽습니다. 그러던 와중에 letter,digit,빈칸에 속하지 않는 문자가 인식되거나 끝까지 읽었음에도 “가 나타나지 않는다면 ERROR를 표시해줍니다. 그렇지 않는다면 테이블에 “~~~” 에서 큰따옴표를 제거한 ~~~부분만 token value로 저장해주고 token은 STRING으로하여서 테이블에 저장해줍니다.

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

이렇게 묶은 이유는 네 lexeme이 i로 시작한다는 공통점이 있기 때문입니다.

소스코드 257줄~396줄까지가 해당 내용인데, 너무 길어서 스크린샷은 생략하겠습니다. 어떤 원리로 구현을 했냐를 int와 if를 예시로 설명하겠습니다. 먼저 i문자를 인식합니다. 그러면 이제 선택지는 4개로 나눠집니다.

1. i뒤에 나오는 문자가 letter나 digit이 아니라서 i혼자서만 token으로 저장되는 경우
2. i뒤에 나오는 문자가 f라서 token이 KEYWORD로 설정되고 그 value는 if로 설정된 경우(그러나 f문자 이후의 문자가 letter나 digit이 나올 경우 ID토큰으로 변경됨. 그 후부터는 ID를 검사해서 저장해줌)
3. i뒤에 나오는 문자가 n이라서 int가 될 수 있는 경우. 이 분기가 선택된다면 추가로 다음 문자를 검사해서 t인지아닌지 여부를 판단해야함.(n일때는 선택지가 3개로 나뉨. in자체가 ID인경우, 다음 문자가t여서 int로 갈 경우,t가아닌 letter나 digit이 와서 ID검사 단계로 넘어가는 경우)
4. i뒤에 나오는 문자가 f나n이 아닌 letter이거나 digit이어서 if,int의 가능성이 사라지고 token이 ID로 고정되는 상태. 이때는 LETTER나 DIGIT이 더 이상 나오지 않을때까지 읽어서 그 값을 ID토큰의 value로 설정하고 테이블에 저장되게 됨.

INT,IF도 똑 같은 선택지를 갖게 됩니다. 이를 조건문과 while문을 이용하여 구현하였습니다. 제가 그린 DFA를 그대로 구현한 방식입니다.

**VTYPE**토큰의 char,CHAR도 마찬가지로 비슷한 방식입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

char을 예시로 들면 먼저 c를 인식합니다. 그리고 다음에 오는 문자가 h인지 검사합니다. 만약 아니라면? else 문을 통하여 branch\_to\_ID함수를 사용합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 함수의 역할이 무엇이냐면 예를 들어 위 상황에서 c다음 h가 올 것이라 기대했지만 오지 못한 상황입니다. 그럼 위 함수에 h를 인자로 넘겨줍니다.

그러면 branch\_to\_ID함수 내에서는 c다음에 오는 문자가 h가 아닌 LETTER나 DIGIT이라면 token을 ID로 설정하고 LETTER나 DIGIT이 더 이상 안 나올때까지 읽어들여서 해당 lexeme을 ID 토큰의 value로 테이블에 저장해줍니다. 하지만 c다음에 오는 문자가 LETTER나DIGIT이 아니라면 c혼자서 ID토큰의 value인 것이기 때문에 테이블에 <ID,c>형태로 저장해줍니다. 이와 같은 역할을 해 주는 함수입니다.

이 함수를 if문마다 else로 짝지어서 반복해서 적용해주면 VTYPE(char,CHAR)인지 ID인지를 구분해 줄 수 있게됩니다. (char이 인식되었다면 그다음문자가 LETTER나 DIGIT이 아닌경우에만 <VTYPE,char>로 저장이되고 그렇지 않다면 VTYPE이 아닌 그냥 ID토큰이기 때문에 ID토큰 처리를 해주게됩니다.)

그럼 이제 남은 것이 **KEYWORD**토큰(앞에서 처리한 if,IF를 제외한 else,ELSE,while,WHILE,return,RETURN)뿐인데 이는 바로 전에 설명한 VTYPE토큰인 char,CHAR를 인식하는 방법과 완벽히 동일한 방법으로 구분합니다. 다른점이 있다면 단지 완성된 결과가 VTYPE토큰이냐 KEYWORD토큰이냐 뿐입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그 예로 else와 ID를 구분해주는 부분을 보면 char(VTYPE)과 ID를 구분해주는 부분과 매우 유사한 방식인 것을 알 수 있습니다. 각 if문마다 짝지어진 else문이 있고 그 else문 마다 branch\_to\_ID가 실행되고 있는 것을 볼 수 있습니다. 보이는 차이는 단지 ‘else’가 완성이 되었을 때 더 이상 LETTER나 DIGIT이 인식이 안된다면 token을 KEYWORD로 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**

텍스트이(가) 표시된 사진

자동 생성된 설명

**output: test1.out**

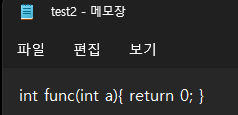
텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명 output파일 생성됩니다

정상적으로 token이 분류된 것을 볼 수 있습니다.

**input: test2**

****

**output: test2.out**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명 output파일 생성됩니다

정상적으로 token이 분류된 것을 볼 수 있습니다.

이번에는 ERROR를 처리하는 상황을 보여주겠습니다.

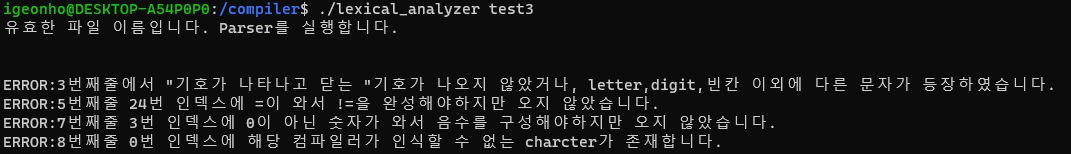
**input: test3**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

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



에러 메시지를 보면 3번째줄에서 string이 구성되어야하는데 “는 안오고 string을 구성하지 않는 요소인;만 왔기 때문에 error를 띄웠습니다.

두번째 error는 5번째줄에서 b!=3으로 표시가 되어야하는데 b!3으로만 되어있기에 error를 띄웠습니다.

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

네번째 error는 제가 구현한 lexical analyzer는 &문자에 대한 처리가 없기 때문에 error를 띄웠습니다.

마지막으로 추가로 정상적으로 lexical분석되는 예시입니다.

**input: test4**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

output: test4.out

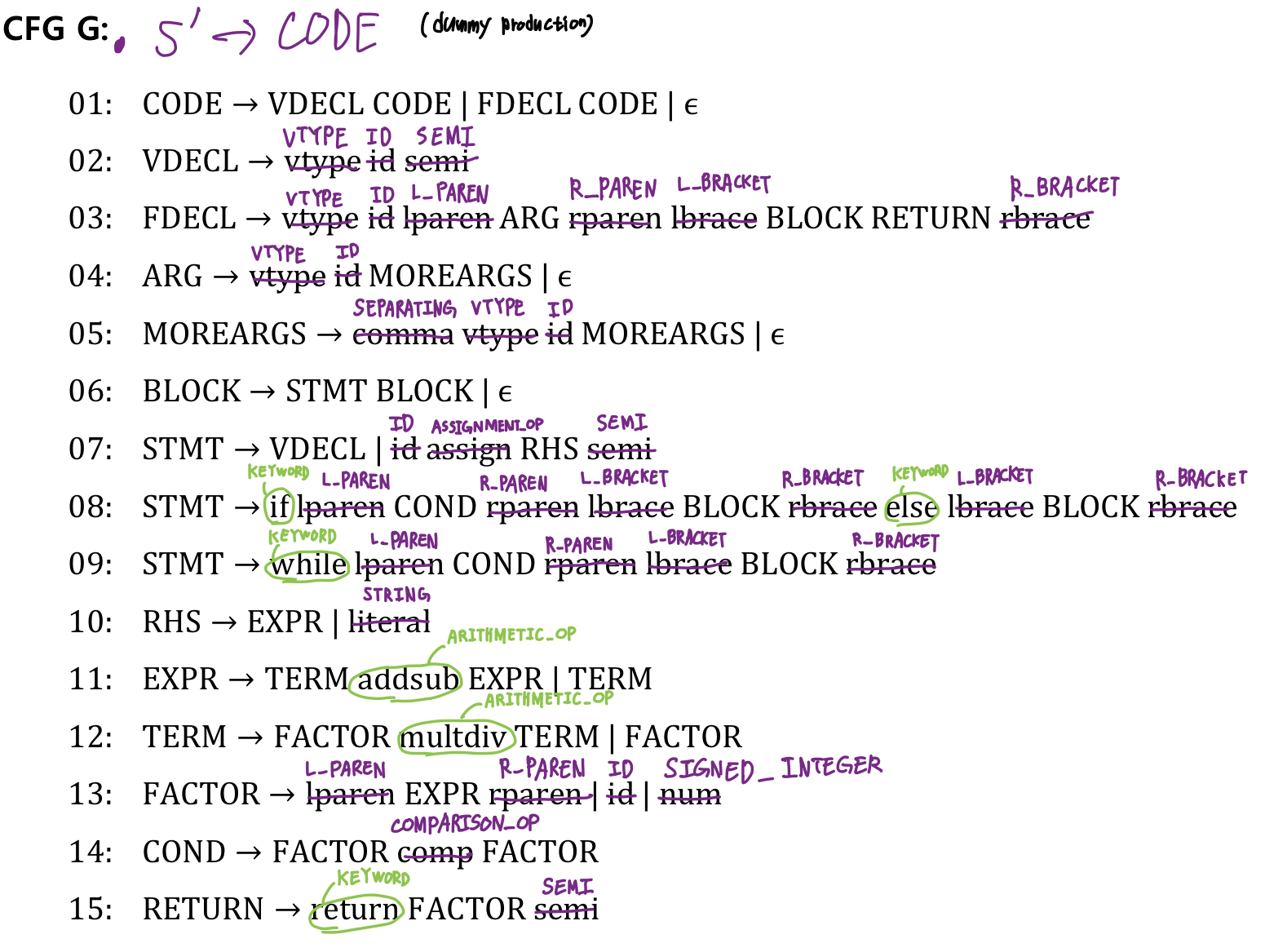
텍스트이(가) 표시된 사진

자동 생성된 설명

**COMPILER PROJECT (syntax analyzer)**

**20182345 이건호**

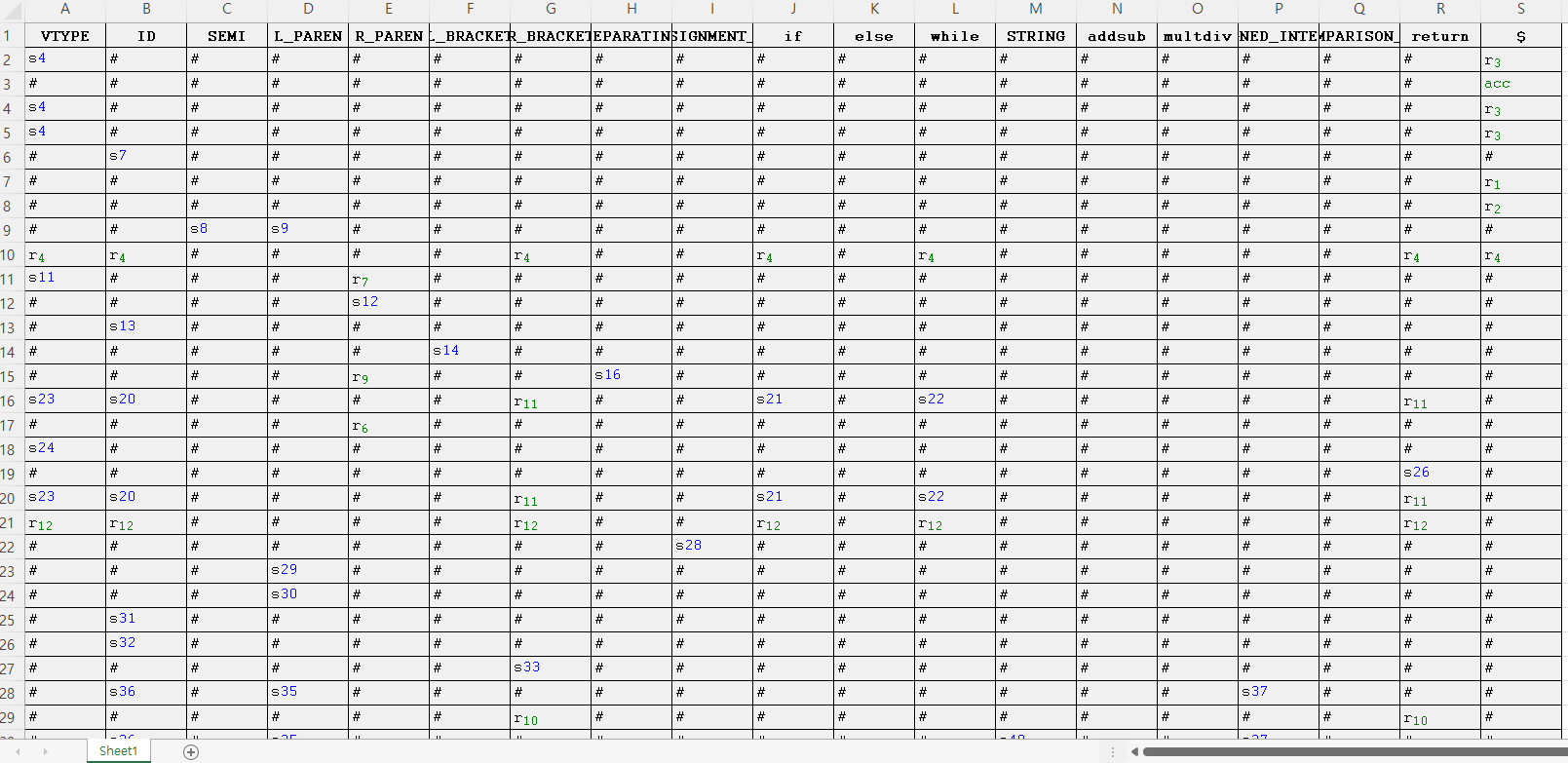
**<CFG>**



제 lexical analyzer과정에서 나온 토큰정보에 맞춰서 CFG를 위와 같이 수정하였습니다. 초록색으로 동그라미 친 것은, 예를들어 if,else,while,return은 묶어서 KEYWORD토큰으로 되어있기 때문에 syntax analyzer과정에서 읽은 토큰이KEYWORD토큰일때는 그 토큰이 지닌 value를 얻어내는 과정을 한번 더 거치게 됩니다.(if인지 else인지 while인지 return인지…)  
  
참고로 ‘{‘는 L\_BRACE가 맞지만 처음 lexical\_analyzer를 설계할 때 {를 실수로   
L\_BRACKET으로 토큰을 지정하였습니다. ‘}’도 마찬가지입니다.

**<SLR parsing table >**

교수님께서 알려주신 사이트를 기반으로 테이블을 만들었습니다.

1. 먼저 사이트에 CFG를 넣어주고 SLR-parsing table을 얻습니다.
2. ACTION부분과 GOTO부분을 각각 나눠서 엑셀로 복붙했습니다.

테이블이(가) 표시된 사진

자동 생성된 설명(엑셀로 옮긴 ACTION테이블)

(엑셀로 옮긴 GOTO테이블)

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

1. 이 엑셀로 수정한 정보를 텍스트파일로 옮겼습니다.(각각 action.txt와 goto.txt로 옮김)

텍스트, 컴퓨터, 스크린샷이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

이제 이 텍스트파일의 정보를 불러와서 syntax 분석에 사용할 것입니다.

**<상세 설명(syntax\_analyzer클래스)>**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

syntax\_analyzer를 위한 클래스입니다.

action\_table은 만들어놓은 action테이블 정보가 담긴 텍스트파일을 불러온 후 저장해 놓기 위한 변수입니다. vector<map<string,string>>타입을 사용했기 때문에 예를들어 테이블에서 state=3이고 읽을 다음 토큰(심볼)이 ID인경우 action\_table[3][“ID”]의 방식으로 테이블의 내용에 접근할 수 있습니다.

goto\_table변수는 action\_table과 동일한 방식입니다.

terminal변수는 action테이블의 가장 윗줄인 terminal들을 저장하고 있습니다.

nonterminal변수는 goto테이블의 가장 윗줄인 nonterminal들을 저장하고 있습니다.

syntax\_stack은 syntax분석에서 사용할 스택입니다. stl의 stack을 사용하였습니다.

next\_input\_symbol은 교수님 강의자료를 예시로 보면

**** 이 상황일떄 \*가 next\_input\_symbol에 저장됩니다.

pair<int,string>reduce\_arr[27]변수는 action테이블에서 Reduce명령을 실행해야할 때 first부분에 A->a일 때 |a|의 값을 저장, 즉 pop해야할 개수를 저장하고, second부분에 A문자열을 저장해줍니다. 이 second부분은 GOTO테이블을 검색할 때 사용합니다.

public부분의 메소드들을 보면 check\_syntax()를 제외하고 전부 set이 붙어 있는데 이 메소드들은 set\_ 뒤에있는 변수를 초기설정해주는 메소드들입니다.

void check\_syntax()메소드는 실제로 syntax분석을 해주는 실질적으로 메인역할을 해주는 메소드입니다.

**<상세 설명(main함수 부분)>**

main함수의 흐름을 따라가며 설명하겠습니다. lexcial분석 과정 이후의 설명입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

lexical analyzer가 정상적으로 실행이 되었다면 syntax analyzer를 위한 과정이 if문을 통해 시작됩니다.(위와 같이 error==false일때)

먼저action테이블을 텍스트파일로부터 읽어와서 syntax\_analyzer클래스 내에 있는 vector<map<string,string>> action\_table 변수에 저장합니다. 이를 위해서 action.txt파일을 열어서 syntax\_analyzer.set\_terminal()메소드를 활용해 먼저 terminal들을 저장해줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

set\_terminal메소드를 보면 읽은 string에서 화이트 스페이스을 이용해서 terminal들을 구분하며 vector<string>terminal변수에 push하는식으로 저장해줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명그 다음 syntax\_analyzer.set\_action\_table(input,row);를 이용해서 action테이블의 실 내용을 줄마다,칸마다 구분해서 저장해줍니다.(화이트스페이스 이용)

마찬가지로 이번엔 goto 테이블을 syntax\_analyzer클래스 내에 있는 vector<map<string,string>> goto\_table변수에 저장해줘야 하는데 action\_table을 저장하는 과정과 동일하기 때문에 설명은 생략하겠습니다.(이때 goto\_table의 첫줄은 non-terminal이기 때문에 terminal대신 nonterminal변수를 저장하게되는 차이가 있습니다.)

텍스트이(가) 표시된 사진

자동 생성된 설명

다시 main함수를 보면

텍스트, 오렌지, 어두운, 설정이(가) 표시된 사진

자동 생성된 설명

set\_tokens\_for\_syntax()함수가 실행됩니다.

이것은 어떤 역할을 하냐면 lexical\_analyzer에서 발생하는 output은 전역변수인 vector<pair<string,string>> token에 저장됨. 어떤식으로 저장되냐면 예를들어 3번째 단어가 ‘(‘이면 token[2].first=”L\_PAREN”, token[2].second=”(“ 이런식으로 저장됩니다. 이를 이때 token.first에 토큰의 이름이 들어있기 때문에 이를 syntax\_analyzer에 그대로 사용하면 좋겠지만 CFG과정에서 말했듯이 if,else,while,return은 KEYWORD라는 토큰으로 묶여있고 +,-,\*,/도 2개씩 addsub,multdiv가 아닌 묶여서 ARITHMETIC\_OP토큰으로 묶여있기 때문에 이를 위해서 한번 걸러서 저장해주는 작업이 필요합니다.

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

vector<string> tokens\_for\_syntax전역변수에 가공한 정보를 넣어주는 역할을 해줍니다. 이 tokens\_for\_syntax가 실제 syntax\_analyze과정에서 사용됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 코드를 보면 KEYWORD 토큰과 ARITHMETIC\_OP토큰에 대해 CFG에 쓰이는 terminal과 맞춰주기 위해 정보를 가공해 주는 걸 볼 수 있습니다.

텍스트, 오렌지, 어두운, 설정이(가) 표시된 사진

자동 생성된 설명

다시 main으로 돌아가면 이제 실제 syntax분석 과정인 check\_syntax()메소드를 실행합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명check\_syntax()메소드의 상단부분을 보면 먼저 스택에 초기state인 0을 넣어주게 됩니다. index변수는 next symbol이 인덱스로는 몇번째인지를 저장하기 위한 변수입니다. next\_input\_symbol은 string타입으로 tokens\_for\_syntax[index]의 값을 저장하게됩니다. tokens\_for\_syntax는 앞에서 말했듯이 input으로 받은 파일의 일련의 토큰들을 CFG에서 사용가능하게 순서를 맞춰서 저장하고 있는 vector타입 변수입니다. 이것에 현재 index를 사용하여 가리키고있는 다음 symbol이 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\_analyzer인스턴스의 pair<int,string> reduce\_arr[27];를 설정해주기 위한 함수인데, 이는 SLR파싱 테이블에따라 다르게 코드를 적어줘야 하는 부분입니다. 앞에서 말했듯이 reduce\_arr[]멤버변수는 SLR파싱테이블에서 reduce명령을 수행해야 할 때   
사용하는 변수입니다.

텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

reduce\_arr[i].first에는 A->a일 때 |a|의 값을 저장, 즉 pop해야할 개수를 저장해야 하기 때문에

텍스트이(가) 표시된 사진

자동 생성된 설명

사용한 CFG의 ->우측부분에 있는 단어의 개수를 저장해줍니다.

reduce\_arr[i].second에는 A->a에서 A부분을 저장해줍니다. 이 second부분은 GOTO테이블을 검색할 때 사용합니다.

예를 들어 action테이블에서 r5라는 문자열을 수행해야하면 reduce\_arr[5].first의 수만큼 stack을 pop()해주고 pop을 다 수행한후 stack의 가장 top인 숫자(state)와 reduce\_arr[5].second (A->a에서 A부분 문자열)가 교차되는 지점을 goto테이블에서 찾은 뒤, 그 값을 stack에 push해줍니다. 이를 위해서 존재하는 변수입니다.

그 뒤 while문을 돌면서 syntax error가 발견되거나 정상적으로 accept되거나 할때까지 syntax분석을 계속합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

while문 절반을 보면 먼저value변수에 action테이블의 현재 state와 next symbol이 교차되는 지점의 값을 저장합니다.

이때 value의 첫 문자가 s면 shift and goto연산을 수행해야 합니다.  
if(value[0]==’s’)가 그 부분인데, 이때 스택에 get\_number(value)를 통해서 s뒤에 존재하는 숫자(state)를 얻은 뒤 스택에 push합니다. 그리고 splitter를 움직이듯이 index를 1증가시켜줍니다. 그리고 난 후 next\_input\_symbol을 새로 갱신해줍니다.

value의 첫 문자가 r이면 reduce by 연산을 수행해야 합니다.  
else if(value[i]==’r’)이 그 부분입니다. 앞에서 말했듯이 reduce\_arr변수를 활용  
해서 A->a에서 |a|의 해당되는 값만큼 stack에서 pop()을 수행합니다.  
그 후, goto테이블의현재 state와 reduce과정에서 사용된 A->a에서 A에 해당되는 문자열의 교차 지점의 값(state)를 스택에 push해줍니다.

value의 첫 문자가 a이면 syntax가 정상적인 것이므로 Accept를 해줍니다.  
그리고 완료되었기에 break로 while문을 빠져나갑니다.

참고로 syntax\_error\_string전역변수는 syntax analyzer결과 및 에러를 output파일로 표시하기위해 문자를 저장하는 용도입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 action테이블의 값이 s도아니고 r도아니고 a도 아니면 #인경우인데, 즉 빈칸인 경우인데 이는 syntax error에 해당됩니다. 먼저 error가 발생한 line이 몇번째 줄인지 token\_line[index]를 활용해서 출력합니다.  
(lexical analyzer를 만들때는 없던 전역변수인데 여기서 활용하기 위해서 lexical analyzer과정에서 token을 추가할때마다 token\_line에 해당 토큰이 몇 번째 줄인지를 저장해주는 과정을 추가했습니다.)

그리고 실제 에러가 난 부분이 어느 단어인지를 표시하기위해   
token[index].second를 사용했습니다. tokens\_for\_syntax[index]를 안 쓴 이유는, 이곳에는 토큰 이름이 들어 있는 것이지 어떤 문자인지가 안들어있기 때문입니다.( 예를들어 {가 아닌 L\_BRACKET이 들어있음.}  
 참고로 ‘{‘는 L\_BRACE가 맞지만 처음 lexical\_analyzer를 설계할 때 {를 실수로   
L\_BRACKET으로 토큰을 지정하였습니다. ‘}’도 마찬가지입니다.

그리고 terminal.size()는 terminal의 갯수를 의미합니다. 이 에러처리 부분을 보면 이 terminal의 갯수만큼 for문을 돌리는데 왜 그런것이냐면, 에러가난 state에서 어떤 terminal을 만나야 정상적인 작동이 될 것인가를 확인하기 위해서입니다. 즉, 와야할 terminal이 이어서 안왔기 때문에 error가 나온것이기 때문에 그 와야할 terminal이 어떤것인지를 하나씩 state와 연결해보면서 확인하는 것입니다.

테이블이(가) 표시된 사진

자동 생성된 설명



예를 들어서 error가 state가 5일 때 next symbol이 ‘(‘여서 빈칸(#)이기 때문에 error상태라고 가정하면, 그 외 나머지 terminal들인 \* ) id $를 대입해보면서 과연 어느 terminal이 와야지 syntax analyzer를 성공할 가능성이 생기는지를 확인하는 과정입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명그래서 보면 현재 state에 i번째 terminal을 대입했을 때 action테이블에서의 값이 #(빈칸)이 아니고 r이면 reduce by명령을 가상으로 1번만 수행해보고 그 수행한 결과가 아직 syntax analyzer를 계속 할 가능성이 있는 경우에는 해당 terminal을 출력해줍니다. (문법적으로 맞을 가능성이 있기떄문에).

또는 현재 state에 i번째 terminal을 대입했을 때 action 테이블에서의 값이 #(빈칸)이 아니고 acc나 s인 경우에는 가능성이 있다고 생각해서 해당 terminal을 출력해주었습니다.

모든 terminal들을 돌면서 가능한 녀석들을 다 출력해 주었으면

텍스트이(가) 표시된 사진

자동 생성된 설명  
이제 앞에서 출력한 녀석들 중 하나가 등장해야 문법상 맞을 가능성이 있다고 출력해줍니다. 그리고 syntax analyzer를 종료해줍니다. (break문 사용해서 while문 탈출). 그리고 해당 메시지들은 모두 syntax\_error\_string에 저장되어서 output 파일로도 생성됩니다.

**<INPUT,OUPUT example>**

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

**input: syntax\_test1 / 콘솔창결과**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

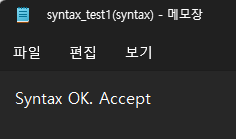
자동 생성된 설명

콘솔상에서 lexical analyzer과정에서 얻은 token값이 나오고 Lexical OK.가 출력되면서 lexical분석과정이 성공했다는걸 알려줌.

그 다음 Syntax OK.메시지를 출력하여 Syntax분석과정에 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**

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 lexical과정의 output과 syntax과정의 output이 각각의 파일로 나오게 프로그램을 짰습니다.

만약 lexcial과정에서 에러가 있다면 syntax과정은 실행되지 않고 그렇기에 syntax과정의 output파일은 생성되지 않습니다. 이때는 오직 lexical과정의 error리포트만 생성됨.

이번에는 CFG상 함수 속 함수는 불가능하므로 해당 케이스의 syntax error를 발견하는 예시입니다.

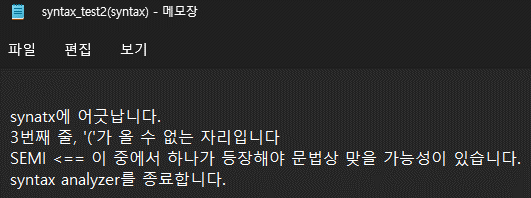
**텍스트이(가) 표시된 사진

자동 생성된 설명input: syntax\_test2 / 콘솔창결과**

텍스트이(가) 표시된 사진

자동 생성된 설명

**output: syntax\_test2(lexical).out 와 syntax\_test2(syntax).out**

텍스트이(가) 표시된 사진

자동 생성된 설명

syntax 에러 리포트를 보면 3번째줄 ‘(‘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 CFG상 선언과 동시에 초기화가 안되기 때문에 해당 케이스의 syntax eror를 발견하는 예시입니다.

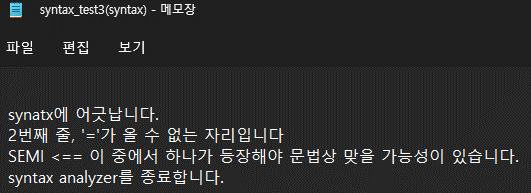
**input: syntax\_test3 / 콘솔창결과**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

**output: syntax\_test3(lexical).out 와 syntax\_test3(syntax).out**

텍스트이(가) 표시된 사진

자동 생성된 설명

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

문법상 선언과 동시에 초기화가 안되기 때문에 int a다음 ‘;’가 와야 정상적일 것이라고 에러리포트에서 제안하고 있다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**input: syntax\_test4 / 콘솔창결과**

**output: syntax\_test4(lexical).out 와 syntax\_test4(syntax).out**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

syntax 에러 리포트를 보면 12번째줄 return이 올 수 없다고 적혀있다. 이는 else구문에서 ‘}’가 나오지 않았는데 return이 나와버렸기에 뜨는 것이다.  
(주어진 CFG상 if-else구문속 return 불가능)   
그렇기 때문에 ‘}’이 등장해야 정상적일 것이라고 에러리포트에서 제안하고 있다.

이번에는 if문만오고 else문이 안온 경우의 syntax error를 발견하는 예시입니다. 주어진 CFG를 따르면 if만 단독으로 올 수 없습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**input: syntax\_test5 / 콘솔창결과**

**output: syntax\_test5(lexical).out 와 syntax\_test5(syntax).out**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

syntax 에러 리포트를 보면 13번째줄 return이 올 수 없다고 되어있다. 왜냐하면 if문이 끝났는데 이제 else문이 와야할 차례인데 return이 왔기 때문이다.

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

제가 준비한 예시는 여기까지입니다.

**<문제점>**

문제점이 있었다면 action테이블과 goto테이블을 윈도우에서 txt파일에 저장한 것을 리눅스에서 옮긴 후 그대로 실행하면 정상적으로 실행이 되지 않았습니다.

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

그리고 ‘{‘이 문자와 ‘}’이 문자를 L\_BRACE와 R\_BRACE가 아닌 L\_BRACKET과 R\_BRACKET으로 토큰 이름을 잘못 정한 점이 살짝 아쉬운 점이었습니다.

참고로 프로그램을 실행할 때 action.txt와 goto.txt는 실행 파일(바이너리)과 같은 디렉토리 내에 위치하여야 합니다. input파일도 마찬가지입니다.

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