**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]=(**

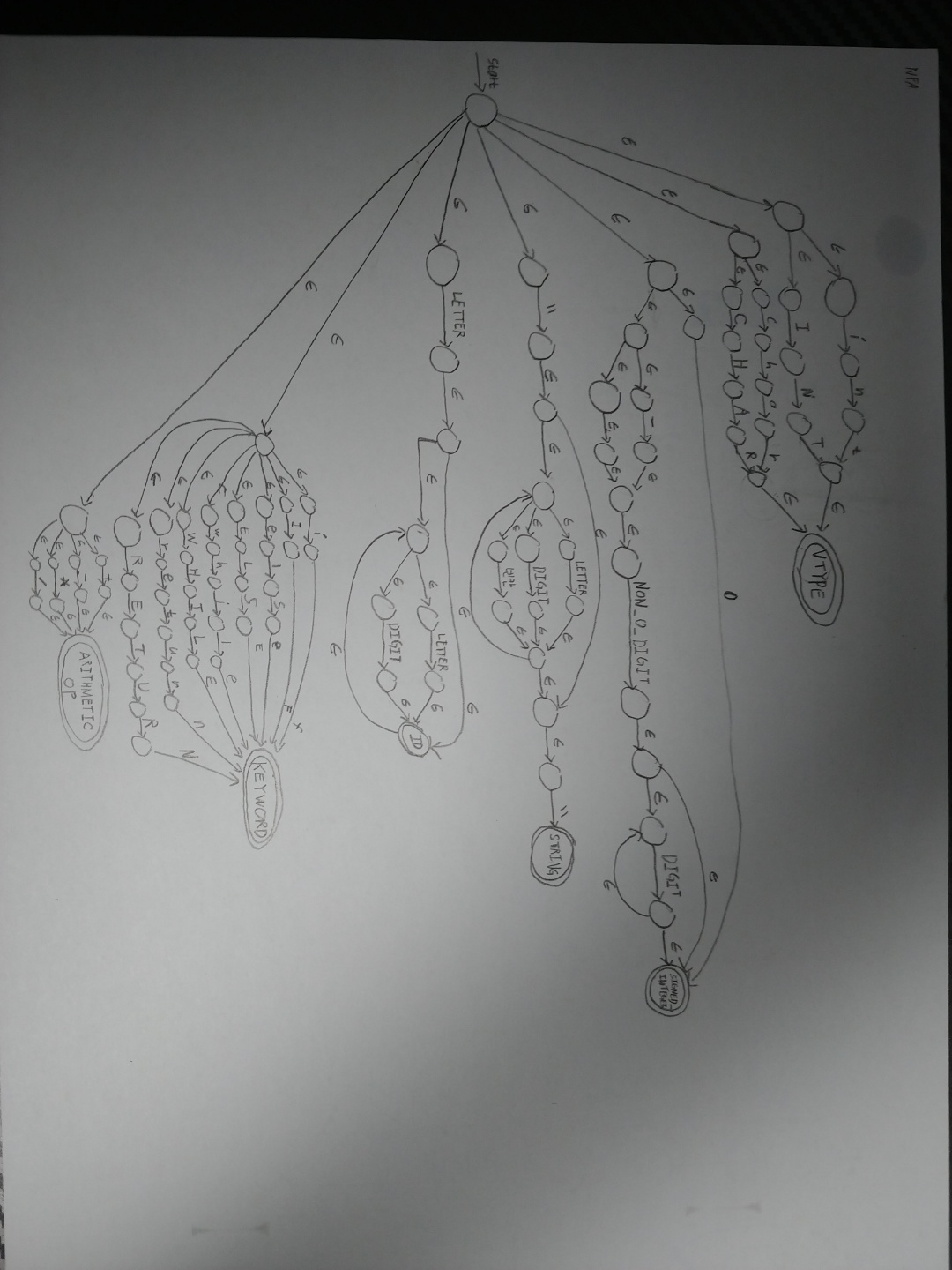
**[R\_PAREN]=)**

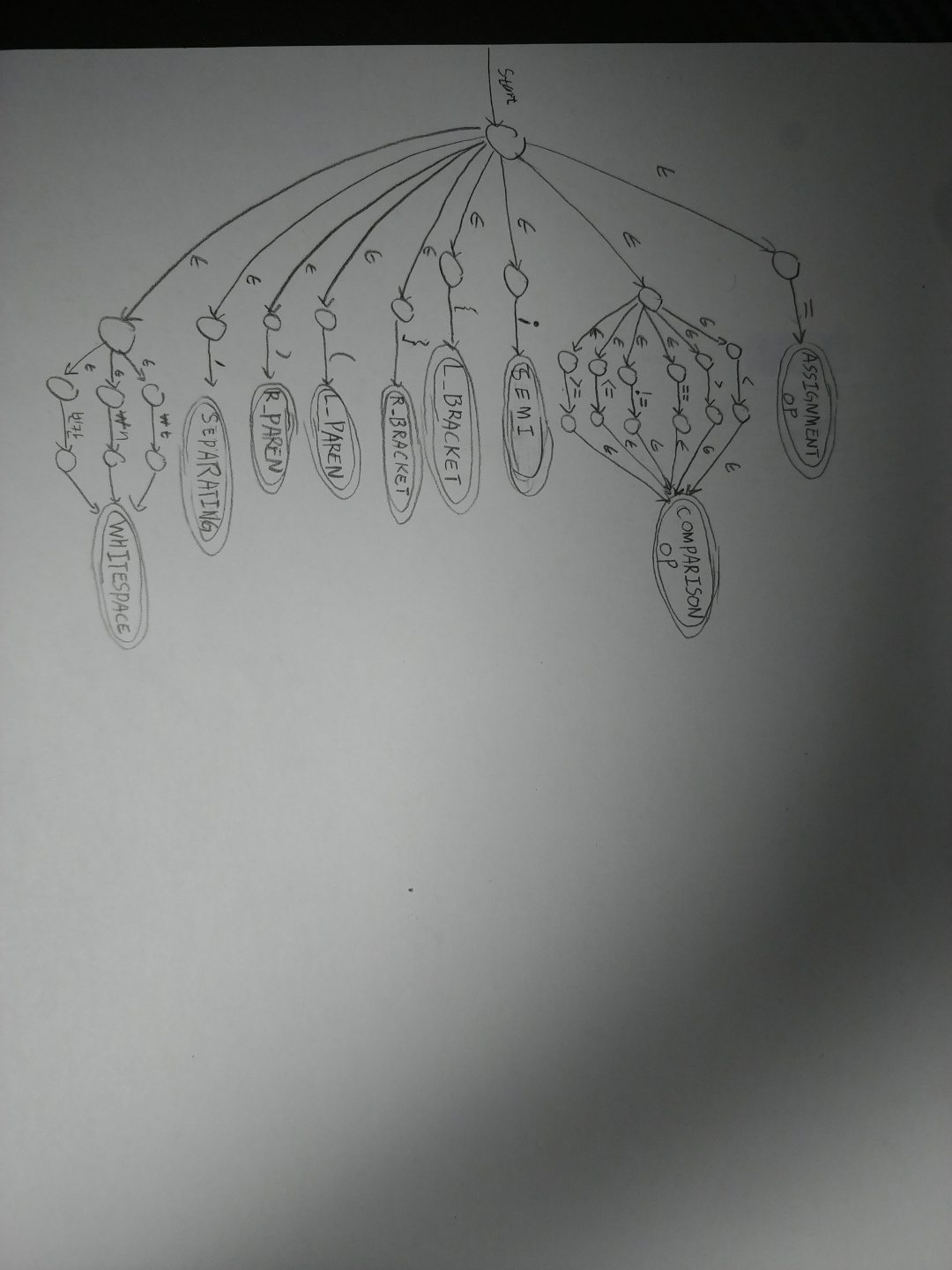
**[SEPARATING]=,**

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

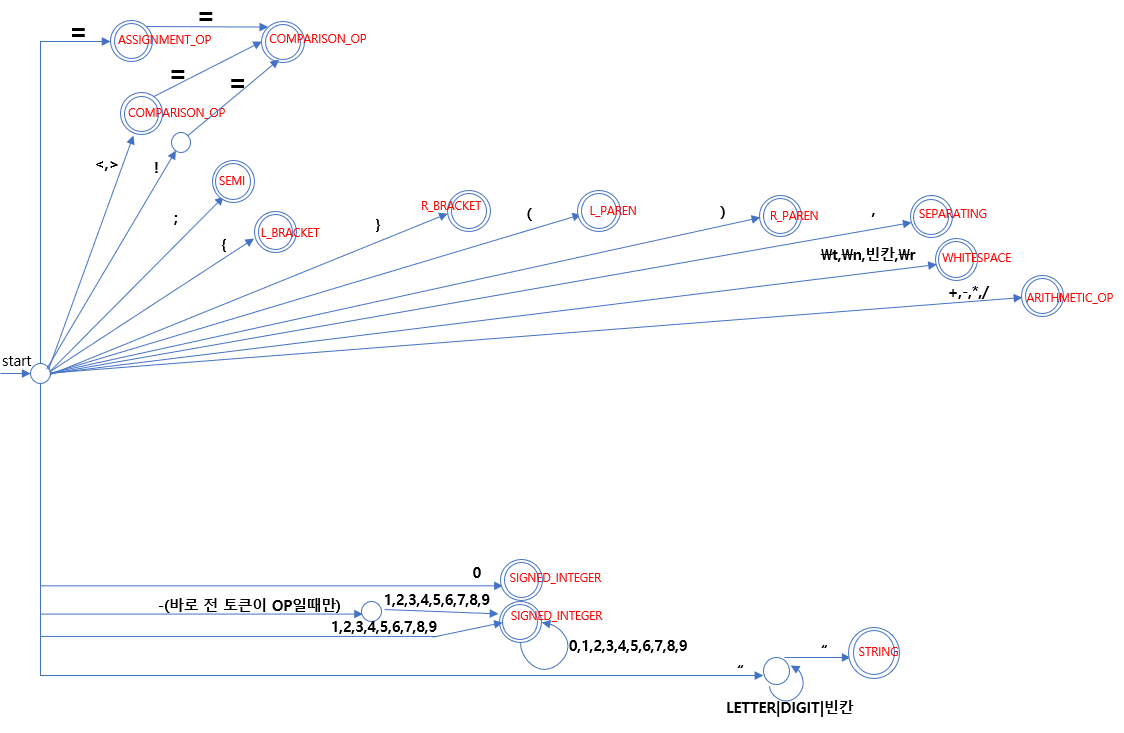
**op의-와 음수의-는 바로앞에 op**(arithmetic이든 assignment든 comparison이든)가 있을 경우에만 음수의-이고 그 외에는 op처리

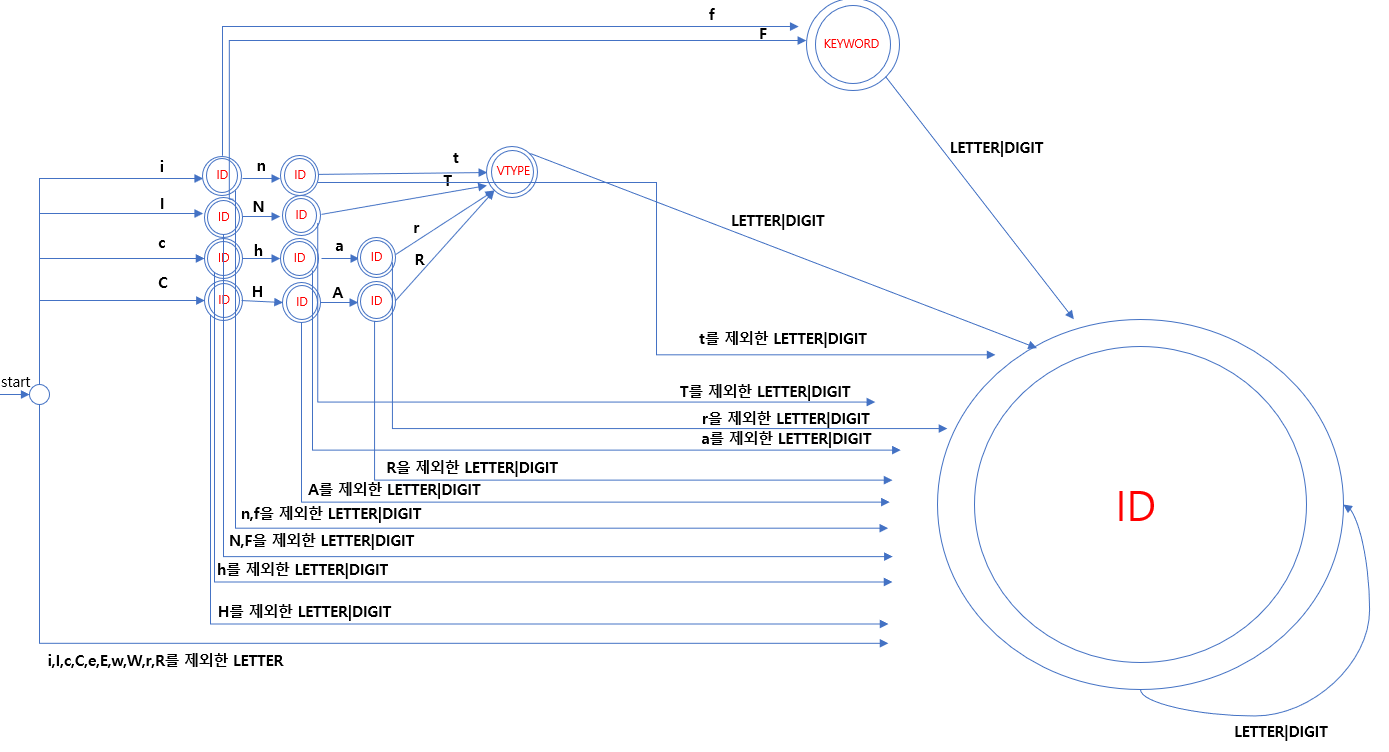
**<NFA transition graph>**

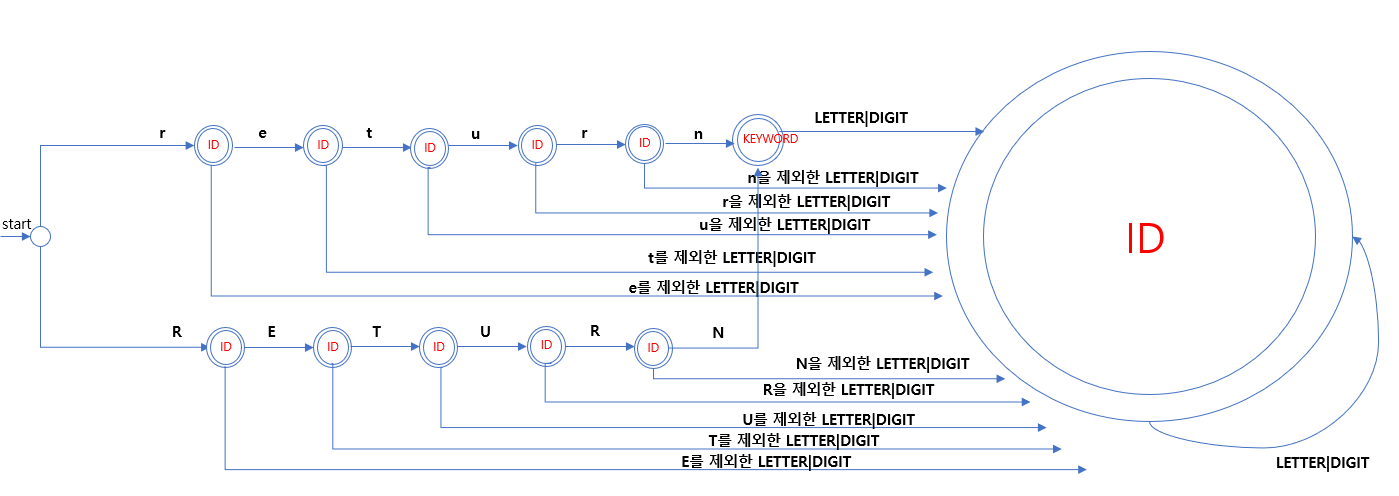
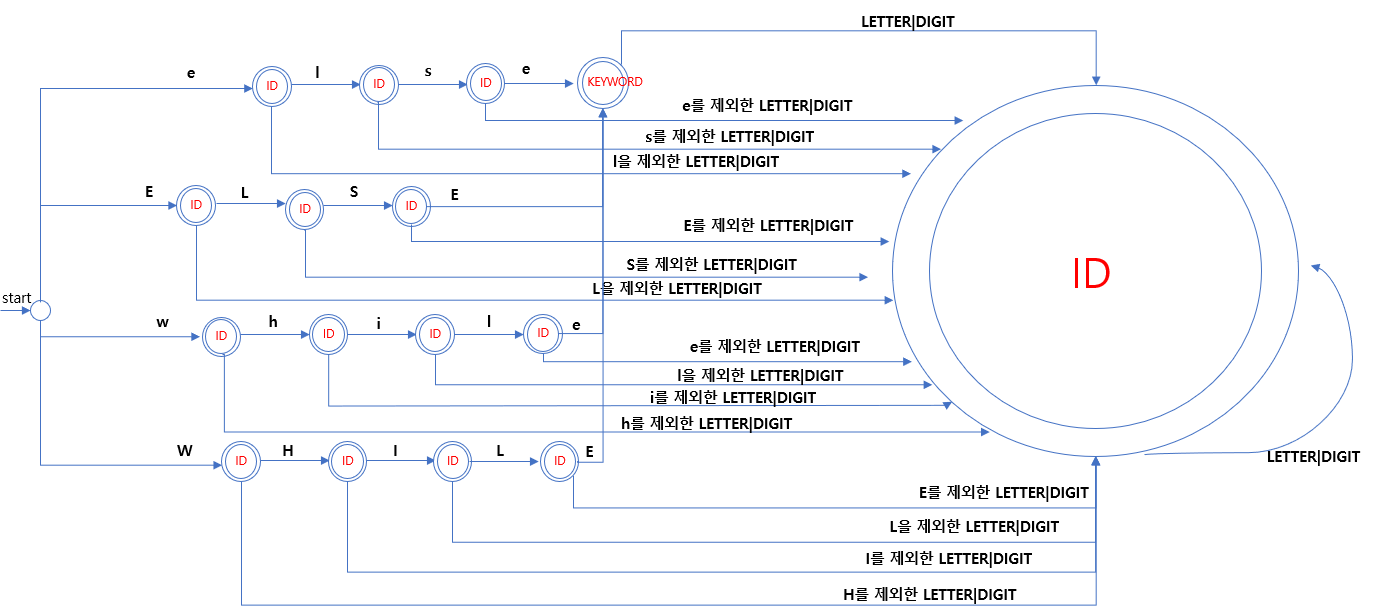
종이 2장에 나눠서 그렸습니다. (같은 start지점을 지님)



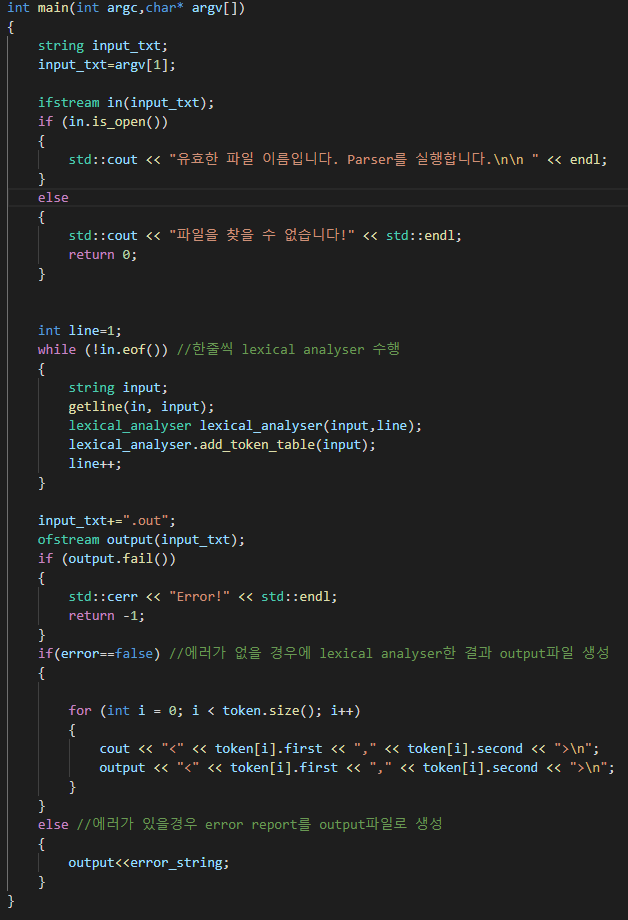
**<DFA transition graph>**

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.

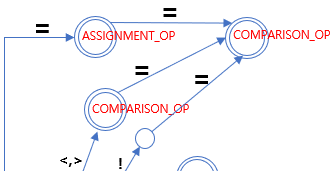
텍스트이(가) 표시된 사진

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.

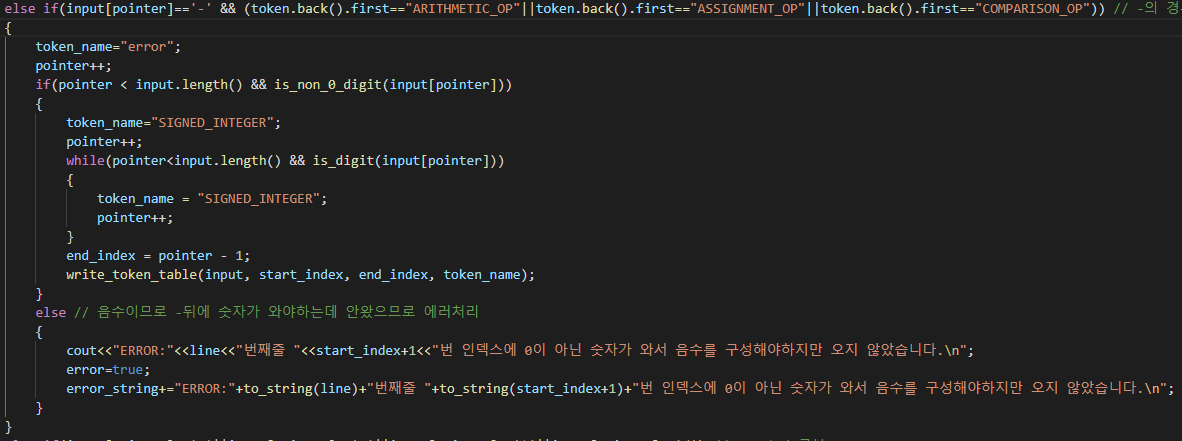
텍스트이(가) 표시된 사진

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

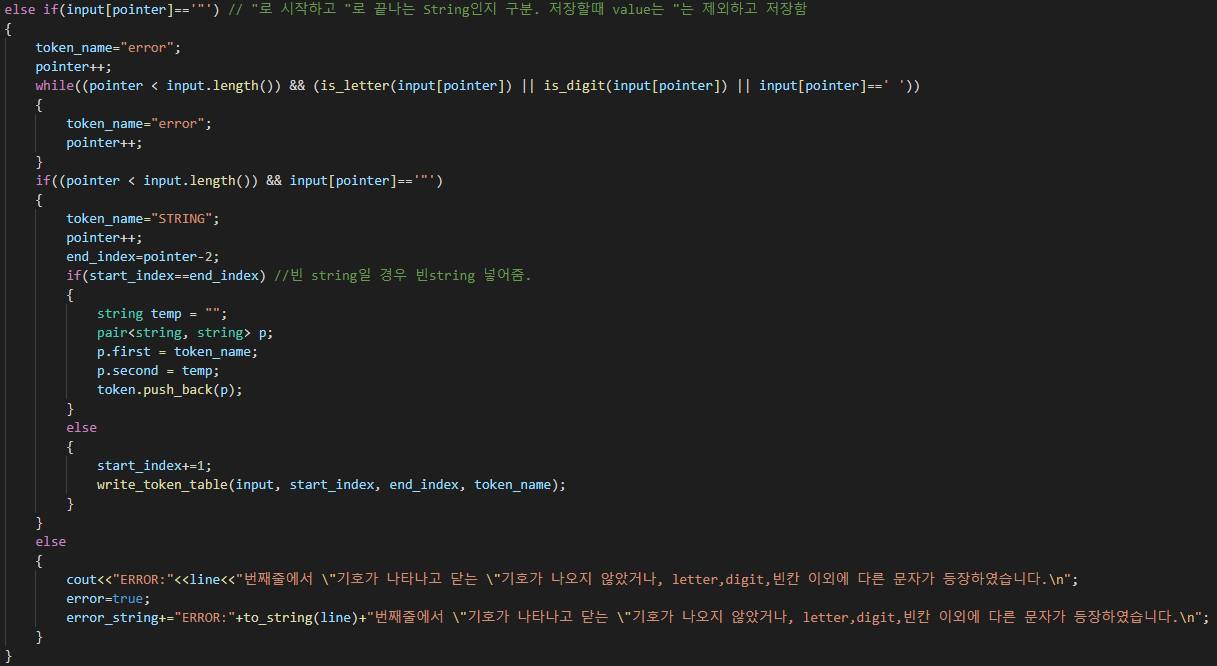
ARITHMETIC\_OP는 음수의 -인 경우를 제외하고는 전부 연산자-이기 때문에 +,-,\*,/를 한번에 합쳐서 else if문에 넣어줬습니다.(음수의 -를 구분하는 else if문 뒤에 위치함) 그렇게 하여 간단히 구분해줄 수 있습니다.

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**

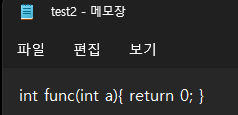
텍스트이(가) 표시된 사진

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.

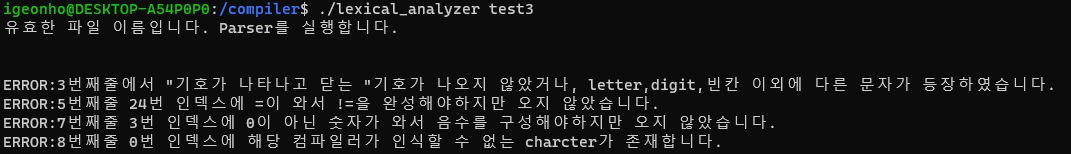
**input: test3**

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