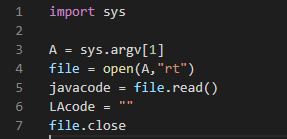
2021 compiler term project 1

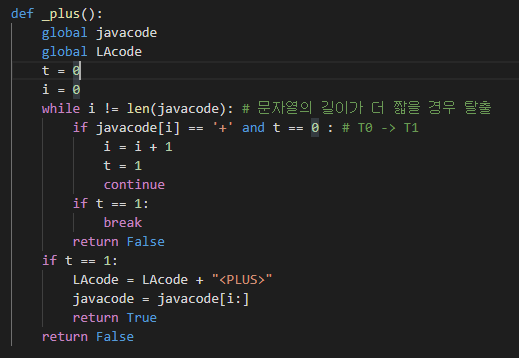
The definition of tokens and their regular expressions, the DFA transition graph or table for recognizing the regular expressions는 documentation.pdf 에 있습니다.

**<lexical analyzer가 작동하는 과정>**



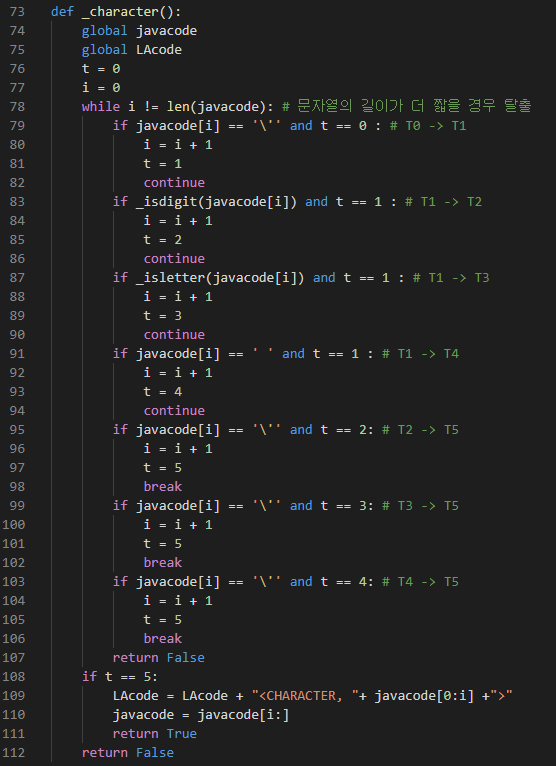
프로그램은 입력 받은 이름의 파일을 읽어서 javacode 라는 string id에 저장하고 토큰을 앞에서부터 하나씩 찾아 나가며, LAcode라는 string id에 토큰을 저장하고 해당하는 원래코드만 javacode에서 지워 나가는 식으로 진행됩니다.

각 함수는 javacode에서 가장 앞의 토큰이 해당하는 토큰인지 확인하고 맞으면 javacode에서 해당 토큰을 지우고 LAcode에 붙여넣습니다. 아니라면 아무것도 하지 않고 지나갑니다

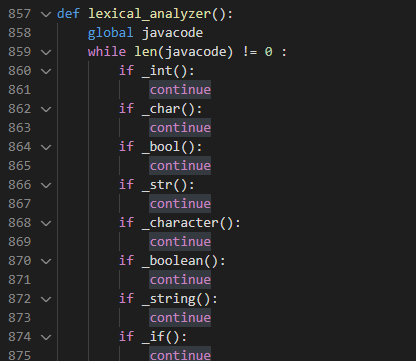
.

위의 코드를 예시로 하면 javacode의 가장 앞에 +기호가 있다면 javacode에서 그 +기호를 지우고 LAcode의 제일 뒤에 <PLUS> 라는 토큰을 추가시킵니다.

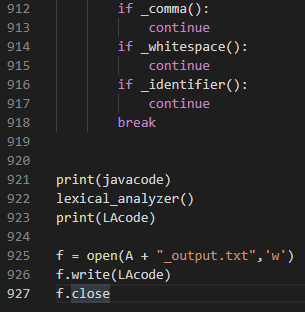
기본적으로 DFA를 바탕으로 코드를 작성하였습니다. 따라서 기본적으로는 같은 틀을 가지고 있습니다.



위 코드에서 t라는 변수는 DFA의 state의 번호를 확인하는 변수입니다. i라는 변수는 해당 토큰의 길이를 체크하는 변수입니다. while문을 계속 돌면서 현재 state에서 어느 방향의 state로 가야 하는지 확인하고 있습니다. 만약 다음 state를 확인하려 하는데 javacode의 길이가 짧아 그 다음 기호를 읽을 수 없다면 index 참조 에러가 발생하기 때문에 확인하고 while문을 탈출할 수 있도록 조건을 만들었습니다. 그리고 토큰이 맞는 것을 확인하고 함수가 끝났으면 True, 토큰이 아닌 것을 확인하고 함수가 끝났으면 False를 출력하도록 했습니다.



~

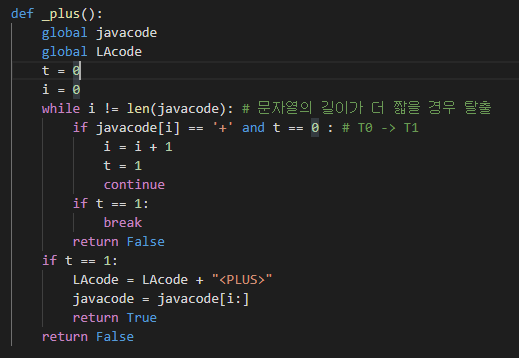


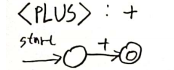
위와 같은 방식으로 만든 모든 토큰확인 함수들을 lexical\_analyzer() 함수로 실행하도록 만들었습니다. 이때, \_identifier() 함수를 가장 밑에 있도록 해서 keyword를 identifier로 오해하는 일이 없도록 설정했습니다. 그리고 \_assignment() 함수보다 \_compare() 함수를 먼저 써서 == 가 나오는 경우 알맞게 나오도록 설정했습니다. \_minus() 함수와 \_integer() 함수의 순서에 관해서는 아래에서 서술하겠습니다.

최종적으로 입력된 코드와 결과를 보여주고 <입력 파일>\_ouput.txt 파일로 결과가 나옵니다.

**<토큰확인 함수들의 구현 방식>**

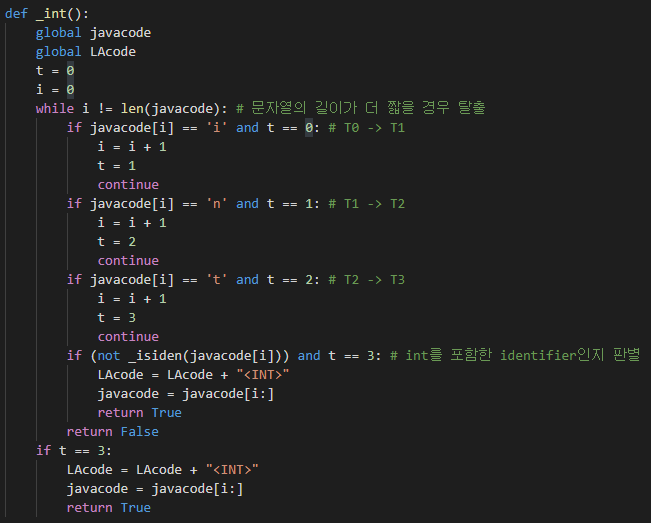
1. +, -, (, ; 와 같은 기호

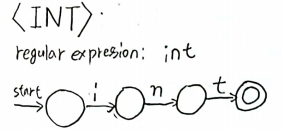




위와 같은 경우는 state가 2가지 밖에 없습니다. 따라서 한번 만 확인하고 i=1임을 확인 한 후에 루프를 나와서 t=1이라면 처음에 +가 있었으므로 LAcode와 javacode를 수정해 주면 됩니다.

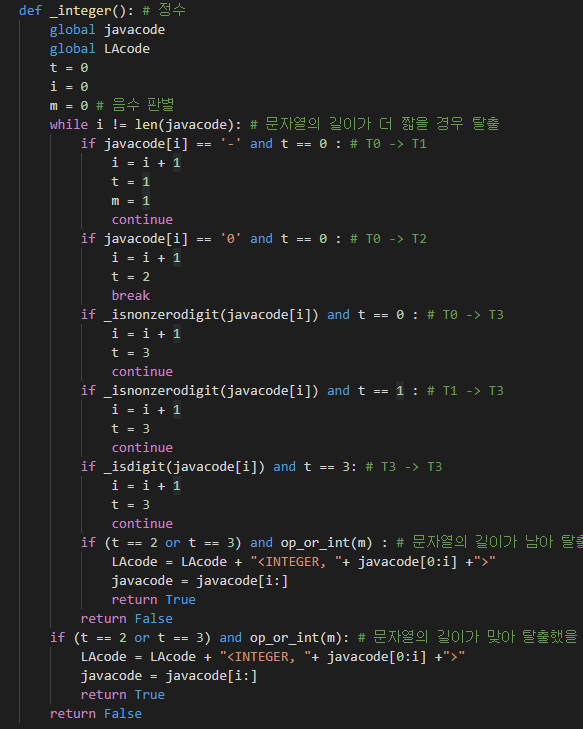
2. int, char, if와 같이 뒤에 다른 문자, 숫자, ‘\_’가 나올 경우 식별자가 되어버리는 경우

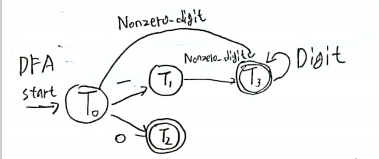




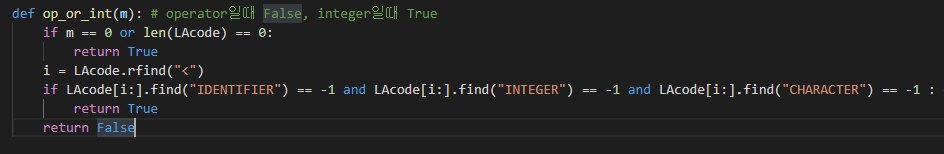
위와 같은 경우는 먼저 DFA에 따라 int가 나오는지 확인합니다. 그 다음에 입력된 코드가 더 이상 없거나 뒤에 알파벳, 숫자, ‘\_‘가 나오지 않는다면 식별자가 될 수 없으므로 int인 것을 확인할 수 있습니다.

3. DFA를 그대로 따라가면 되는 나머지 토큰



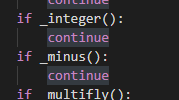


위와 같은 DFA 경우 처음 T0에서 T2로 간다면 바로 종료됨을 알 수 있습니다. 따라서 T0->T2일 때, break을 선언하여 루프를 빠져나오게 했습니다. T3으로 갔을 경우에는 T3에서 다시 T3으로 갈 수 있기에 break 대신에 continue를 선언했고 그 다음에 digit이 나오지 않는다면 -가 연산자로 쓰였는지 아닌지 확인하는 op\_or\_int(m) 를 사용하여 연산자로 사용되지 않았으면 True를 리턴하고 그렇지 않으면 False를 리턴해서 INTEGER가 아님을 알 수 있습니다. 하지만 javacode의 마지막이 숫자로 끝날 경우에 op\_or\_int(m)를 실행하지 않으므로 while문을 탈출했을 때도 다시 확인하여 -가 MINUS인지 INTEGER의 일부분인지 확인할 수 있습니다.



위 3번에서 얘기했던 op\_or\_int(m)의 함수입니다. m이 1일 경우에 제일 앞에 -가 있는 경우입니다.

-가 연산자인지 음수임을 알려주는 기호인지 알기 위해 바로 앞의 토큰이 무엇이었는지 알아볼 필요가 있었습니다. 앞의 토큰이 없을 경우 즉 아무것도 없다면 연산자로 쓰일 수 없습니다. **연산자로 쓰일 때 필요한 것은 앞에 INDENTIFIER, INTEGER 혹은 아스키 코드를 활용한 CHARACTER가 있는 경우일 것**이라고 생각했습니다. 이 세 가지가 모두 없다면 TRUE를 리턴하도록 했고 셋 중 하나라도 있다면 False를 리턴하도록해서 이 수는 정수가 아님을 알 수 있도록 하였습니다.



따라서 lexical\_analyzer 함수에서 MINUS는 INTEGER 뒤에 와서 INTEGER가 아닌 경우에 MINUS를 확인하도록 설정했습니다.