컴파일러

보고서

201523483 소프트웨어학과

이병헌

1. **서론**

* **과제 소개 (개요)**

본 과제의 목표는 lexical analysis 단계를 거쳐 토큰화된 토큰들이 저장된 테이블과 직접 정의한 Grammar rule을 이용하여 syntax tree를 구성하고 해당 syntax tree와 각 identifier, integer, float에 해당하는 값들이 저장된 테이블을 이용하여 입력 값의 토큰들을 파싱하고 XML 구조로 결과물을 출력하는 과제이다.

파싱은 반드시 recursive descent parsing 방식을 이용해야 하며, 이에 따라서 lookahead라는 변수를 사용하여 하강하는 방식으로 진행되어야 한다. 과제에서 다루는 expression은 상수와 변수, binary operators, unary operators, assignment operator로 한정되어 있다.

XML 구조로 파싱을 하기 위해서는 XML 구조가 prefix notation의 형태를 띄고 있다는 점에 착안하여 완전히 새로운 Grammar rule을 정의한 뒤에 들어온 수식(infix)을 우선적으로 prefix notation으로 변환하고 이에 따른 syntax tree를 구성하려 했으나, 굳이 이런 방식을 쓰지 않고 recursive descent parse 방식만을 통해서 문제를 해결할 수 있다는 점을 깨닫고, prefix notation으로 변환하지 않고 syntax tree를 순회하면서 expression tree를 구성하는 알고리즘을 구축하였다. Expression tree만 구성되어 있다면, XML 규칙에 맞추어 출력물을 출력하는 것은 어려운 일이 아니기 때문이다.

출력물을 출력하는 규칙은 만들어진 expression tree의 root부터 traversal을 수행하며 출력을 하면된다. 단, Traverse 함수 안에서는 <token>과 관련된 출력 부분은 preorder traversal을, </token>과 관련된 출력 부분은 postorder traversal을 적용하여야 한다. 따라서 아래 슈도 코드와 같은 로직으로 Traverse 함수를 구성하였다.

Traverse() {

Print <token>

Traverse(left)

Traverse(right)

Print </token>

}

* **구현된 부분과 구현되지 않은 부분**
* **구현된 부분**

기본적인 사칙연산 (+, -, \*, /)은 parsing tree에서 expression tree를 구축하는 알고리즘이 정상적으로 돌아가기에 expression tree로부터 식에 알맞은 (연산 우선순위, associativity를 고려한) XML 코드가 출력된다. 또한 괄호도 recursive하게 구현되어 괄호안의 괄호 등도 처리가 가능하다. (아래 출력문에서 확인 가능)

* **구현되지 않은 부분**

괄호와 assign부분은 부분적으로 동작한다. 하지만 아직 몇가지 예외가 존재한다. 가령 c=(10+a) 등의 식은 정상출력되나, c=(10\*a)와 같은 식은 core dumped가 일어난다. 덧붙여 uminus 부분은 아직 구현이 되지 않았다. 이러한 점들은 토큰을 사용하여 보완한 뒤 코드로 제출할 계획이다.

1. **문제 분석**

* **Grammar rule 분석**

Grammar rule은 교재에 나온 기본 수식에 관한 grammar rule과 거의 유사하며 assign, division operator에 관련된 부분만 추가하였다. Grammar rule은 아래와 같다.

E -> TE`

E` -> +TE` | -TE` | =TE` | ε

T -> FT`

T` -> \*FT` | /FT` | ε

F -> (E) | id

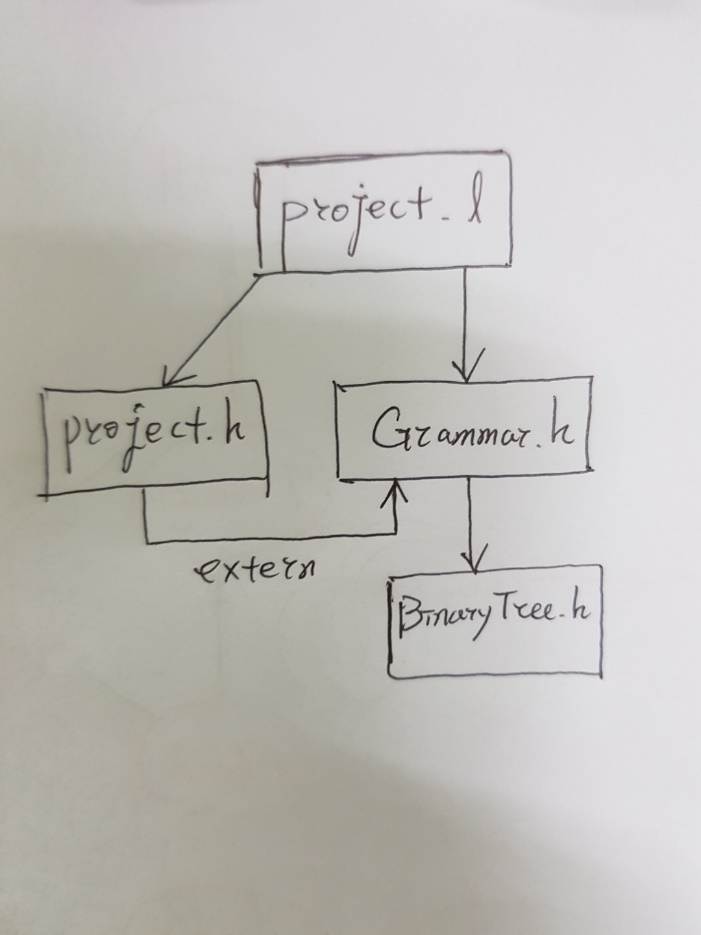
보완된 부분에 관련해서만 설명을 하면, assign operator는 expression E`의 production으로 두었다. 이유는 정의한 알고리즘 상에서 assign의 우선순위와 plus, minus의 우선순위를 같게 두어야 했기 때문이다.

마찬가지로 division이 expression T`의 production으로 들어간 이유도 division의 우선순위를 multiplication의 우선순위와 같게 두었기 때문이다.

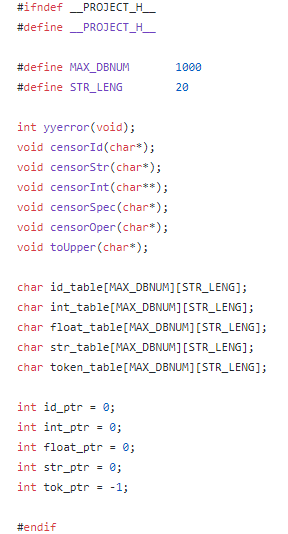
1. **설계**

* **프로그램 module hierarchy 및 module**

이번 과제를 진행함에 있어, Project.h, Grammar.c, Grammar.h, BinaryTree.c, BinaryTree.h를 정의함으로써 프로그램 모듈화를 시도하였다. 또한 Grammar.h에서는 project.h (project lex 파일에 대한 헤더)에 정의된 전역 변수를 extern하여 사용할 수 있게 처리하였다. Module hierarchy는 다음과 같다.

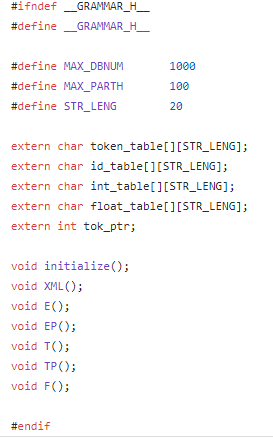


* **주요 자료구조**
* **Project.h**



Project.h은 project.l (lex 파일)에 대한 헤더이며, 이곳에는 identifier, integer, float에 대한 각각의 테이블 들과, token에 대한 테이블 그리고 이에 상응하는 포인터들이 int 타입으로 정의되어 있다.

* **Grammar.h**

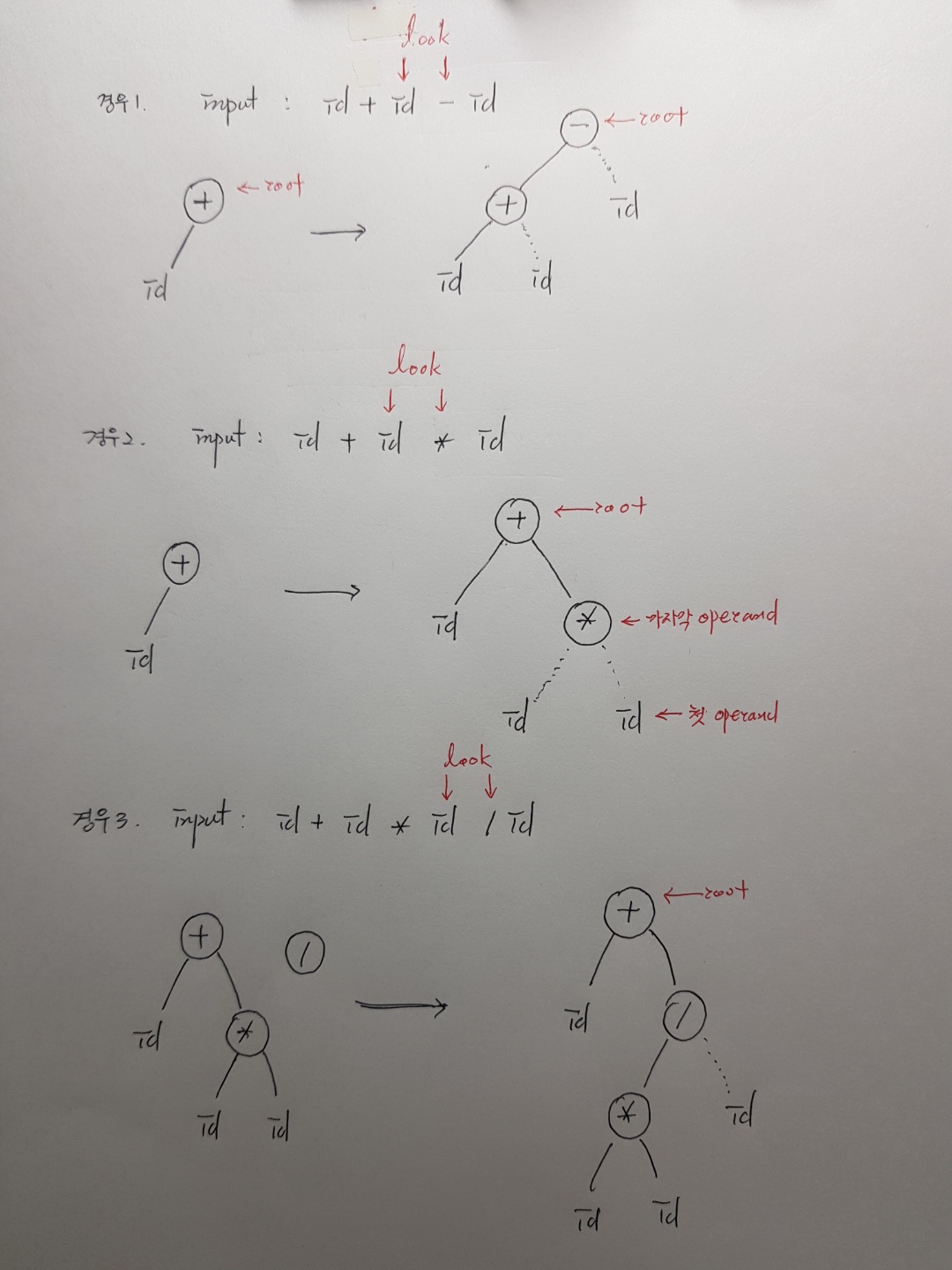


Grammar.h는 Grammar.c에 대한 헤더이며, 이곳에서는 project.h에 정의된 전역 변수들(테이블 및 포인터)을 extern하여 Grammar.c에서 사용할 수 있도록 처리해준다. 또한 Grammar.c에 정의된 함수들 XML(), E(), EP() 등을 선언한다. XML()은 E() (start symbol)에 대한 wrapper함수이며 몇가지 E()와 함께 필요한 몇가지 기능들이 들어가 있고, 프로그램 실행시에 함께 실행되는 주요 함수이다

* **Recursive descent parsing -> expression tree -> XML**
* **Recursive descent parsing -> expression tree**

Recursive descent parsing은 match() 함수와 lookahead라는 변수를 각 EP(),, TP(), F() 함수에서 이용하여 실현한다. 즉, 터미널로만 이루어진 함수 E(), T()에서는 사용하지 않고, 논터미널이 포함된 함수들에서 이용한다. Match() 함수와 lookahead 변수의 이용법은 교재에 나온 내용과 유사하고, 코드에서 확인 가능하므로 생략한다.

Expression tree의 구성 방법은 아래 그림을 참조하여 프로그램의 근간이 되는 expression tree 구성 알고리즘의 일부지만 핵심부분을 설명한다.



(위 그림은에 나오는 tree는 parsing tree가 아닌 expression tree이며, parsing tree로부터 expression tree를 뽑아내는 과정을 설명한다.)

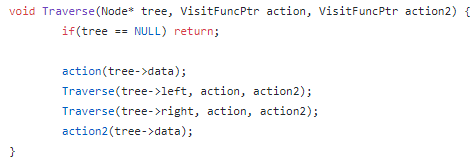
경우1은 input이 id + id – id이고, lookahead가 id에서 -로 넘어가는 경우이다. 이때 expression tree를 구성하는 방식은 우선 F()에서 root노드를 operand로 만들어 놓고 EP()에서 root노드를 정의한다음 operand에 연결하는 방식이다.

경우2는 input이 id + id \* id 이고, lookahead가 id에서 \*로 넘어가는 경우이다. 이때는 경우1과 트리를 구성하는 방식이 달라야한다. 왜냐하면 연산자 우선순위가 다르기 때문이다. 이 경우, lookahead가 id일 때 lookahead\_ptr을 이용하여 그 다음 operator를 검사한다. 검사한 operator (\*)가 -보다 우선순위가 높을 경우에는 현재 lookahead (id)를 operand 포인터에 남겨 놓고 먼저 \* 노드를 root의 right subtree로 달아준 다음 해당 노드의 left subtree로 operand 포인터(id)를 달아준다. 그리고 operand 포인터는 마지막에 \*노드를 가리키도록 설정하며 끝나는데 이유는 두 child가 있는 노드는 피연산자로 봐야하기 때문이다.

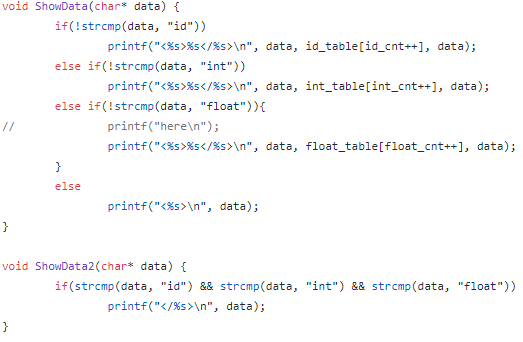
경우3은 경우2에서 / id 가 추가된 경우이다. 사실 경우2와 비슷한(우선순위 비교를 제외하면) 알고리즘으로 동작한다. 경우2와 같이 root의 right subtree로 /를 달아주고 /의 왼편에 operand를 달아주는데, 이 때의 operand는 경우2에서의 마지막 operand (\*)이다.

* **Expression tree -> XML**

Expression tree에서 XML코드로 출력하는 부분은 개요 부분에서 설명한 Traverse()함수 로직을 이용하여 위에서 구성한 tree를 순회하였고, 코드는 아래와 같고, 이는 BinaryTree.c에 정의되어 있다.

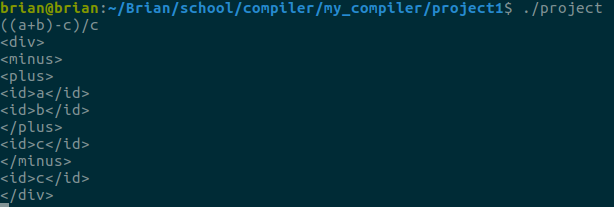
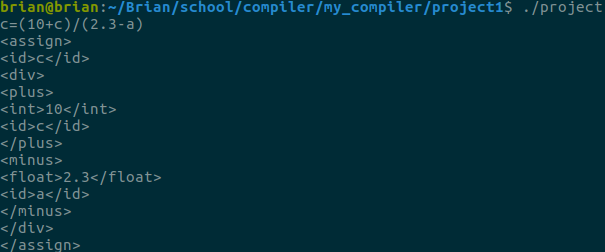
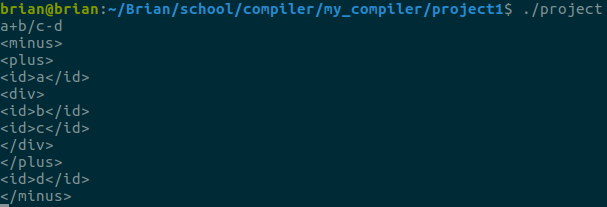
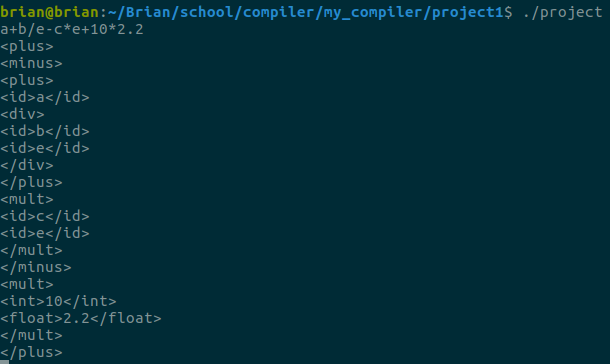
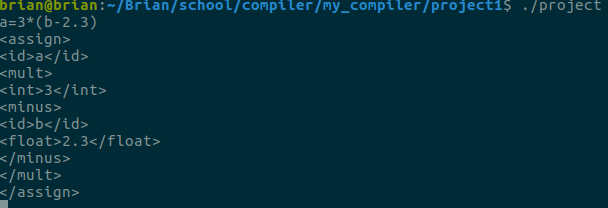


Action과 action2는 함수 포인터로서 해당노드에 도착이 원하는 기능이 수행될 수 있도록 구성하였다. 이에 해당하는 함수는 아래와 같다. 여기서는 action에 해당하는 함수가 ShowData이고, action2에 해당하는 함수가 ShowData2이다.

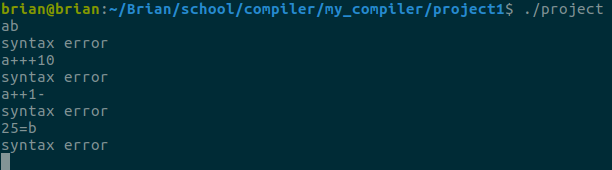


1. **수행결과**

* **정상출력**



* **Syntax error**

****

1. **결론**

이번 과제를 수행하면서 어려웠던 부분은 우선적으로 parsing tree로부터 expression tree를 얻어내는 과정이 어려웠고, 이 점을 해결한 뒤로는 assign, parenthesis, uminus부분의 구현이 어려웠다. 왜냐하면 assign, parenthesis, uminus 부분은 그 자체로 recursion을 포함하고 있었기 때문인데, 이러한 점은 전역변수로 둔 각 flag들을 활용하여 전부는 아니었지만, 해결할 수 있었다.

이번 과제를 하면서 느낀점은 자료구조 과목에서 추상적으로 배운 트리라는 자료구조를 실제 프로그램을 구현할 때 사용할 수 있다는 것에 정말 기뻤고, 트리라는 자료구조가 머릿속에 좀 더 구체화된 느낌을 받았다. 또, 프로그램을 구현하기 위해서는 이전의 설계단계가 정말 중요하다는 점도 다시금 깨달았다. 무작정 코드부터 쓰고 보는 식으로 프로그램을 짜다보니, 프로그램의 규모가 조금만 커지더라도 이후에 여기저기서 문제가 속출하는 사태가 벌어졌다.