Programming Project #2: 수식 인터프리터 개발

소프트웨어학과 3학년

201520882학번 천윤서

1. 서론

- 과제 소개

Recursive Descent Parsing기법을 이용해서 이항 연산자(+, -, *, /, =)와 단항 연산자(-), 상수와 변수 대입 기능을 가진 계산기를 만드는 과제이다. Flex를 통해서 입력된 문장을 분석하여 토큰 리스트를 만들고, 토큰 리스트들을 이용해서 lookahead를 보고 Recursive Descent Parsing을 함과 동시에 Syntax Tree를 생성하게 된다. 만들어진 Syntax Tree를 잘 처리하여, 최종적인 계산 결과를 얻는 것이 목적이다.

- 구현된 부분과 구현되지 않은 부분.

본 프로그램은 flex를 통해 입력된 문장을 분석하여 토큰 리스트를 만들고, 토큰 리스트를 가리키는 lookahead를 보고 각 Nonterminal을 나타내는 Recursive Descent Parsing 함수를 통해서 문법이 완전한지 검사하면서 Syntax Tree를 생성하고, 생성된 Syntax Tree가 가진 토큰을 이용해서 결과값을 계산한다. 따라서, 과제에서 요구하는 부분은 전부 구현하였다. 다만 선택 사항인 멀티라인은 지원하지 않았다. 또한, 한 문장에서 최대로 분석될 수 있는 토큰의 개수는 Default 1,000개이다.

2. 문제 분석

- grammar rule 분석

A -> id A' | F' T' E'
A' -> = A | T' E'
E -> T E'
E' -> + T E' | - T E' | ε
T -> F T'
T' -> * F T' | / F T' | ε
F -> id | F'
F' -> (A) | inum | fnum | - F

Grammar Rule은 위와 같다. 본 과제에서는 사칙 연산(+, -, *, /)를 전부 정의하기에 +TE' 옆에 | -TE'을, *FT' 옆에 | /FT'을 추가해야 한다. A, A'은 할당 연산자와 관련된 문법으로, id=id=id=....식으로 계속 할당 연산자를 확장 시킬 수 있도록 만든다. E, E'와 T, T'이 분리되어서 Syntax Tree가 생성될 때 *가 +보다 우선

순위를 가지게 하고, 연산자를 계속 확장할 수 있다. F, F' 노드를 통해 leaf 노드에 id/inum/fnum가 오고 단항 연산자와 괄호를 계속 확장할 수 있음을 알 수 있다. 괄호를 확장할 때 Assign도 괄호를 허용하려면 (E) 부분을 (A) 로 바꿔야 한다.

이 문법은 하나의 lookahead를 가지고 다음에 어떤 선택을 할 지 예측할 수 있기에 Backtracking이 필요 없는 LL(1)의 Recursive Descent Parsing 함수로 변환할 수 있다. 각각의 Nonterminal에 대응하는 함수들은 재귀적으로 서로 호출하면서, Syntax tree를 반환하고, 필요하다면 Syntax Tree를 파라메터로 넘겨받을 것이다.

하나의 lookahead를 사용하여 Recursive Descent Parsing을 수행할 때는, 이에 맞춰 사용할 FIRST와 FOLLOW를 사용하게 된다. 각 Nonterminal Symbol들의 FIRST와 FOLLOW는 아래와 같다. 각 식의 Lookahead는 보통 FIRST를 이용하며, epsilon이 포함된 경우는 FOLLOW도 같이 사용한다.

	FIRST	**FOLLOW**
A	id, (, inum, fnum, -	), $(코드 상에서는 NEWLINE과 동일하게 취급)
A'	=, *, /, +, -, ε	), $
E	id, (, inum, fnum, -	
E'	-, +, ε	), $
T	id, (, inum, fnum, -	-, +,), $
T'	*, /, ε	-, +,), $
F	id, (, inum, fnum, -	*, /, -, +,), $
F'	(, inum, fnum, -	*, /, -, +,), $

- recursive-descent parsing을 이용한 수식 계산기의 기본개념 정리

추후 자료구조에서 자세히 언급하겠지만, Syntax Tree의 구조체는 계산 편의를 위해서 계산 값을 저장하는 double 변수와 토큰을 저장하는 변수를 가지고 있다. 자식 노드들의 계산 값들을 이용해서 부모 노드의 계산 값을 채우게 되고, 이렇게 연산을 재귀적으로 루트 노드까지 수행하면, 루트 노드의 계산 값이 최종 결과값이 된다. 세부적으로 아래 4가지 경우가 존재하며, 그 외의 경우는 존재할 수 없다.

1. 부모가 사칙연산(+, -, *, /)이며 자식 노드가 둘 다 존재하는 경우.

이항 사칙 연산자. 양쪽 자식 노드의 값을 부모 노드에 해당하는 연산자로 계산하여 부모 노드로 재귀적으로 올린다.

2. 부모가 -이며, 자식 노드가 왼쪽만 존재하는 경우.

단항 연산자. 자식 노드의 값을 반전시켜서 부모 노드의 값으로 대입한다.

3. 부모가 대입 연산자이며, 자식 노드가 둘 다 존재하는 경우.

대입 연산자. 왼쪽 자식 노드에 해당하는 ID의 심볼테이블에 오른쪽 토큰의 값을 대입하며, 오른쪽 자식의 값을 부모 노드에 올린다.

4. 양쪽 다 자식이 없는 경우.

Leaf 노드임을 의미한다. 받아올 자식 노드의 값이 없기에 노드가 가지고 있는 토큰을 값으로 변환하여 저장한다. Leaf 노드는 INTEGER, DOUBLE, ID일 수 있는데 어떤 Token이냐에 따라서 변환 알고리즘이 달라진다.

연산 케이스	알고리즘
INTEGER, DOUBLE	계산 편의를 위해 전부 double형으로 치환한다. 최종 결과값 또한 double이지만 int 판별 알고리즘을 통해 int로 변환하기도 한다.
ID	해당 ID의 심볼테이블의 값을 보고, 값이 들어있는 경우 해당 값으로 치환하여 연산하고, 없는 경우 isDefineError 값을 1로 설정한다.

3. 설계

- 주요 자료구조

해당 프로그램은 struct.h, source.l, main.c 파일로 나누어져 있다. source.l은 1차 과제에서 본인이 작성한 Lexical Analysis를 조금 수정한 어휘 분석 관련 코드가 담겨있고, main.c에는 그 외의 코드들, struct.h는 main.c와 source.l에서 동시에 사용하는 구조체가 담겨있다.

source.l의 코드 중 main.c에서 사용하지 않는 코드들은 보고서에서는 다루지 않겠다. 1차 과제와 거의 변경사항이 없고, 이번 과제의 핵심이 아니기 때문이다.

1. struct.h

source.l과 main.c 양쪽에서 사용하는 구조체가 정의 되어있다. 기능은 아래와 같다.

이름	변수	기능
token	char name[], char value[]	main.c로 넘겨줄 토큰의 구조이다.
symbolTable	char id[], double value, int hasValue	id를 담고 있으며, value를 저장하고, hasValue가 1이면 value를 저장하고 있음을 나타내고, 0이면 아님을 나타낸다.

2. main.c

syntax tree관련 구조체들이 정의되어 있으며, syntax tree는 파싱 과정에서 토큰 리스트에서 토큰을 매칭시켜서 저장하고, 완성된 syntax tree의 토큰들을 이용해서 재귀적으로 계산하여 값을 받아 올라와서 root 노드의 계산값을 반환한다.

이름	변수	기능
syntaxTreeNode	token tok	+ ,- ,id 등의 토큰을 통째로 저장한다.
	syntaxTreeNode* parent	부모 노드의 주소를 저장한다.
	syntaxTreeNode* leftChild, rightChild	양쪽 자식 노드의 값을 저장한다.
	double value	실제 계산할 때만 사용되며, 실제 계산 값을 저장한다.
syntaxTree	syntaxTreeNode* root	Recursive Descent Parsing 함수들이 파라메터로 주고받는 함수이다.

아래는 주요 전역 변수들이다.

변수	기능
extern isLexError	Lexical Error가 발생하면 1로 설정된다.
extern symbolTable *symbol_table	심볼테이블이다. 동적으로 크기가 증가한다.
token tokenTable[]	토큰 테이블이다. 파싱 할 때 이를 lookahead로 읽어나간다.
int lookahead	lookahead이다. 토큰 테이블의 어떤 위치를 가리킨다.
int isSyntaxError, isDefineError	SyntaxError나 DefineError가 발생하면 1로 설정된다.
char defineErrorTemp[]	DefineError가 발생할 때 어떤, ID가 문제인지 임시로 저장한다.

- 프로그램의 module hierarchy 및 module에 대한 설명

1. 외부 함수(Source.l의 함수들)

extern void initTable()	심볼 테이블을 초기화하는 함수이다.
extern token analysisLex(int val)	토큰을 후처리 하는 함수이다. 올바르지 않은 토큰이 들어오면 빈 토큰(val과 name이 전부 '₩0')을 반환한다.

2. SyntaxTree 관련 함수

해당 함수는 Recursive Descent Parsing에서 SyntaxTree를 만드는데 쓰인다.

syntaxTree uniteTree(token tok, syntaxTree left, syntaxTree right)	파라메터의 tok를 부모로 하고, 파라메터의 두 syntaxTree를 자식으로 하는 syntaxTree를 생성/초기화하여 반환한다.
syntaxTree makeNewNode(token tok)	파라메터의 tok를 가지는 새로운 노드 1개짜리 syntaxTree를 만들어 반환한다.

3. Recursive Descent Parsing 보조 함수.

이 함수들은 Recursive Descent Parsing을 할 때 주로 사용되는 함수이다.

void error()	isSyntaxError를 1로 설정한다. 이 함수 뒤에 파싱 함수들은 NullTree를 반환하며, 파싱이 끝나면 계산을 생략하고 에러를 출력한다.
token match(char *input)	input의 문자열과 토큰 리스트의 lookahead가 가리키는 것의 이름이 일치하면 lookahead를 1 증가시키고, 일치하는 토큰을 반환한다. 일치하지 않으면 error를 발생시킨다.
int compareLookahead(char* input)	input으로 들어온 문자열과 lookahead가 가리키는 토큰의 name을 비교해서 맞으면 1을, 틀리면 0을 반환한다.

4. Recursive Descent Parsing 함수

이 함수들은 직접적으로 Parsing을 수행하며, syntax Tree를 서로 반환한다. 각 함수들은 syntaxTree를 반환형으로 가지며, 필요한 경우에만 파라메터로 syntaxTree를 가진다. 예를 들어 A'은 "syntaxTree restassign(syntaxTree parameterTree)"에 대응된다. 각 함수들의 syntaxTree를 다루는 알고리즘은 아래의 표를 그대로 코드화 시켰다.

단말 노드를 만들어 반환할 때는 makeNewNode함수를 사용하였고, 트리를 합칠 때는 uniteTree 함수를 사용하였다. 예를 들어, A' -> =A를 표현하는 함수 안에서 =을 매치시키는 match(char *input)함수는 매칭된 토큰을 반환하며, 이 반환된 토큰과 파라메터로 들어온 트리와 A를 나타내는 assign()의 결과로 받은 트리를 uniteTree함수의 파라메터로 넘긴다. =를 부모로 가지고 파라메터로 들어온 트리와 assign()에서 받은 트리를 자식으로 가지는 트리가 완성되며, 이를 반환하면 된다.

식	작업
F'->(A)	A가 반환한 tree를 반환한다.
F'->inum	inum 단말 노드를 만들어 반환한다.
F'->fnum	fnum 단말 노드를 만들어 반환한다.
F'->-F	F가 반환한 트리를 단말로 가지고 -를 노드로 한 트리를 반환한다.
F->id	id 단말 노드를 만들어 반환한다.
F->F'	F'이 반환한 트리를 반환한다.
T'->*FT'	파라메터로 받은 트리와 F에서 반환된 트리를 자식으로 가지는 *노드를 만들어 T'에 파라메터로 전달하고, 우측의 T'이 반환한 트리를 반환한다.
T'->ε	파라메터로 받은 트리를 반환한다.
T->FT'	F가 반환한 트리를 T'의 파라메터로 전달하고, 우측의 T'이 반환한 트리를 반환한다.
E'->+TE'	파라메터로 받은 트리와 T에서 반환된 트리를 자식으로 가지는 +노드를 만들어 E'에 파라메터로 전달하고, 우측의 E'이 반환한 트리를 반환한다.
E'->ε	파라메터로 받은 트리를 반환한다.
E->TE'	T가 반환한 트리를 E'의 파라메터로 전달하고, 우측의 E'이 반환한 트리를 반환한다.
A'->=A	파라메터와 A에서 받은 트리를 자식으로 가지는 =노드를 만들어서 반환한다.
A'->T'E'	파라메터로 받은 트리를 T'으로 전달하고, T'이 반환한 트리를 E'에 전달한 뒤, 우측의 E'이 반환한 트리를 반환한다.
A->idA'	id 단말 노드를 만들어서 A'에 파라메터로 전달하고, 우측의 A'이 반환한 트리를 반환한다.
A->F'T'E'	F'이 만든 트리를 T'에 전달, T'이 반환한 트리를 E'에 전달하고 우측의 E'이 반환한 트리를 반환한다.

아래 표는 각 nonterminal에 표현하는 함수들이 lookahead를 보고 어떤 동작을 하는지를 나타낸다. 이와 동시에 syntax tree를 다루기도 하나, 위에서 서술한 만큼, 중복해서 언급하지는 않겠다. 각 lookahead는 if문을 통해 분기되고, 실제 식에 그대로 대응되어 토큰을 매칭 시키고, 재귀적으로 서로를 부른다.

예를 들어, A -> idA'|F'T'E'이라면, assing() 함수에서 lookahead가 ID 토큰일 때 match("ID")로 매칭을 일으키고, A'에 대응되는 restassign(~)을 부른다. lookahead가 LPAREN, INTEGER, DOUBLE, MINUSOP라면, restfactor()-F', restterm(~) – T', restexpr(~) – E'을 차례로 호출하고, 나머지 경우에는 에러를 일으킨다. 이런 방식으로 함수들은 서로를 재귀적으로 호출할 것이고, 문법이 맞다면 error를 일으키지 않고 무사히 호출을 끝마칠 것이다.

그 외에 표에 없는 경우는 Error인 경우이다.

이름	실제동작	
	lookahead	동작(왼쪽부터)
assign	id	id 토큰 매칭, restassign 호출.
	(, inum, fnum, -	restfactor, restterm, restexpr 호출.
restassign	=	= 토큰 매칭, assign 호출.
	*, /, -, +,), ₩n($와 동일한 취급)	restterm, restexpr 호출.
expr	id, (, inum, fnum, -	term, restexpr 호출.
restexpr	+	+ 토큰 매칭, term, restexpr 호출.
	-	- 토큰 매칭, term, restexpr 호출.
	), ₩n	호출이나 토큰 매칭 없음.
term	id, (, inum, fnum, -	factor, restterm 호출.
restterm	*	* 토큰 매칭, factor, restterm 호출
	/	/ 토큰 매칭, factor, restterm 호출
	-, +,), ₩n	호출이나 토큰 매칭 없음.
factor	id	id 매칭.
	id, (, inum, fnum	restfactor 호출.
restfactor	(	(토큰 매칭, assign 호출,) 토큰 매칭
	inum	정수 토큰 매칭
	fnum	더블 토큰 매칭
	-	- 토큰 매칭, factor 호출

5. 연산 관련 함수

아래 함수는 void recursive Calculate(syntaxTreeNode *t)를 보조하는 함수이다.

double getNumberValue(token tok)	입력 받은 INTEGER, DOUBLE 토큰이 나타내는 실제값을 double로 반환해준다.
double getIdValue(token tok)	입력 받은 ID 토큰의 symboltable을 참조하여 값이 있는지 확인하여 double값을 반환해준다. 가진 값이 없다면 isDefineError를 1로 설정한다.

void recursiveCalculate(syntaxTreeNode *t) 함수는 앞에서 다룬 수식 계산기의 기본 정리를 코드화 시킨 함수이다. 가장 처음 들어가는 *t는 root Node이며, 각 자식의 double 값을 각 자식 노드를 recursiveCalculate 함수로 넘겨서 계산하여 구한다. 이렇게 계산된 값을 앞의 수식 계산기의 기본 정리 파트에서 서술한대로 처리하여 부모 노드의 double값을 구한다. (부모가 사칙연산이며 자식 노드가 둘 다 존재하는 이항 연산자의 경우 ,부모가 -이며 자식 노드가 왼쪽만 존재하는 단항 연산자의 경우, 부모가 대입 연산자이며 자식 노드가 둘 다 존재하는 경우, 양쪽 다 자식이 없는 리프 노드의 경우.)

6. main 함수

메인 함수에서는 토큰을 yylex()를 통해 분석 받아서 newline 이 나올 때까지의 토큰 리스트를 채우고 isLexError 가 1 인지 확인한다. 1 이라면 다음 단계는 생략되어 lexical error 를 띄우고, 0 이라면 토큰 리스트와 lookahead 를 가지고 Nonterminal 이자 Start Symbol 인 assign()을 호출하여 syntaxTree 를 받는다. 받은 syntaxTree 의 root 를 recursiveCalculate 함수에 집어넣는데, 이 과정은 isSyntaxError 가 1 이라면 실행되지 않고, SyntaxError 를 띄운다. 매칭이 전부 끝나서 lookahead 가 NEWLINE(=$)을 가리키지 않아도 SyntaxError 를 띄운다. recursiveCalculate 가 끝나고, defineError 가 있는지 점검하고 없다면 tree 의 root 가 가지고 있는 double 값을 결과로서 출력한다.

7. 보조 함수

초기화, 메모리 해제 등의 어휘 분석이나 구문 분석에 속하지 않은 기능들을 모듈화 한 함수들이다. relaseTree 의 경우 이 코드에서는 syntaxTree 가 있는 경우 반드시 들어가지만, 이를 쓰지 않는다고 하여도 프로그램은 정상 작동한다.

이름	기능
void initState()	하나의 입력된 문장이 끝날 때마다 초기화 해야 하는 Error 변수, index 변수, 토큰 테이블 등을 초기화한다.
void printNumber(double d)	입력된 더블 값이 소수점이 없을 경우 integer 로 변환해서 출력한다.
void releaseTree(syntaxTreeNode *root)	root 를 포함한 전체 트리의 메모리를 해제하는 함수이다.

4. 수행 결과

- 정상 실행

```
chun@ubuntu:~/workspace/compiler/homework2$ ./a
>a=b=3
3
>c=2*3
6
>(1+6)*(2+a)/c
5.833333
>d=3.14
3.140000
>((a+b)*d)
18.840000
>3/2
1.500000
>-(-(6+3))
9
>-7+8
1
>
```

```
>val = 10
10
>val
10
>val + 10
20
>i = j =  20
20
>val = val + i
30
>(val + val) * 3
180
>-val - 100
-130
>
```

```
note@DESKTOP-EEMOJ9I:/mnt/c/hw2_201520882$ ./a.out
>-5 + -10
-15
>--(5+10)
15
>-(-(5+10))
15
>a=b=10
10
>a = ( b= 10)
10
>a=(b=(1*3+6))
9
>
```

- 비정상 실행

```
chun@ubuntu:~/workspace/compiler/homework2$ ./a
>abc + 10
abc는 정의되지 않음.
>value + + i
error: syntax error
>val << 10
error: lexical error
>3$ + 6
error: lexical error
>3 = a
error: syntax error
>b + c
c는 정의되지 않음.
>(1+3
error: syntax error
>3+*2
error: syntax error
>5+4-
error: syntax error
>a = * 3
error: syntax error
>
```

```
note@DESKTOP-EEMOJ91:/mnt/c/hw2_201520882$ ./a.out
>(a+b
error: syntax error
>(1+3+4
error: syntax error
>1+
error: syntax error
```