PL Assignment #5 : Cute14 Parser2

과제물 부과일 : 2014-04-15(화)
Program Upload 마감일 : 2014-04-21(월) 23:59:59

문제

Cute14 문법에 따라 작성된 program이 as05.txt에 저장되어 있다. 이를 input file로 하여, 프로그램의 syntax tree를 구성하시오.(이 과정을 parsing이라고 한다.) 그리고 syntax tree를 root로부터 pre-order traverse하여 원래 입력된 프로그램과 구조가 동일한 프로그램을 출력해야 한다.(이 과정을 unparsing이라고 한다.)
예를 들어, Cute14으로 작성된 program이 아래와 같을 경우

```
(+ (- 3  2)  -378)
```

이와 같은 프로그램의 출력결과는 다음과 같다.

```
([PLUS] ([MINUS] [INT:3] [INT:2] ) [INT:-378] )
```

Cute14의 문법

```
Program      → Expr | "'" Expr
List         → '(' ItemList ')'
ItemList     → Item ItemList | ε
Item         → Expr | "'" Expr | "quote" Expr
             | '+' | '-' | '+' | '/' | '<' | '>'| '=' | "'"
             | "define" | "cond" | "not" | "lambda"| "car" | "cdr" | "cons"
             | "eq?" | "atom?" | "null?"
Expr         → id    // id에는 define, cond, lambda, ...등의 키워드는 제외됨
             | int    //integer const
             | "#T" | "#F"
             | List
```

추가 설명

- 문법에서 대문자로 시작하는 이름은 non-terminal이고, 소문자로 시작하는 이름인 id와 int는 terminal이다.
- Terminal 중에서 id는 모든 identifier를 총칭하고, int는 모든 정수형 상수를 총칭한다. 따라서 id와 int는 token이라고 볼 수 있으며, token으로 처리하기 위해서 token 이름을 아래와 같이 명명할 수 있다.

```
id          TokenType.ID
int         TokenType.INT
```

```
// id와 int에는 여러 가지가 있을 수 있으므로,
// lexeme으로 구별해 주어야 한다.
```

- 특별한 의미를 가지는 keyword와 해당 token 이름은 다음과 같다.(keyword가 아니면 ID로 간주)

"define"	TokenType.DEFINE
"lambda"	TokenType.LAMBDA
"cond"	TokenType.COND
"quote"	TokenType.QUOTE
"not"	TokenType.NOT
"cdr"	TokenType.CDR
"car"	TokenType.CAR
"cons"	TokenType.CONS
"eq?"	TokenType.EQ_Q
"null?"	TokenType.NULL_Q
"atom?"	TokenType.ATOM_Q

- Boolean 상수는 terminal이며, 해당 token은 다음과 같다.

#T	TokenType.TRUE
#F	TokenType.FALSE

- 특수 문자들은 terminal이며, 다음과 같이 token 이름을 부여할 수 있다.

(	TokenType.L_PAREN
)	TokenType.R_PAREN
+	TokenType.PLUS
-	TokenType.MINUS
*	TokenType.TIMES
/	TokenType.DIV
<	TokenType.LT
=	TokenType.EQ
>	TokenType.GT
'	TokenType.APOSTROPHE

Cute14의 특징

괄호를 써서 프로그램이 표현되는 Cute13은 list가 기본 표현이다. 또한 아래와 같이 각 list의 맨 첫번째 원소를 연산자나 함수 호출로 보고 리스트의 나머지를 피연산자로 간주한 후 evaluate 하게 된다.(다음 예는 > 를 prompt 로 사용하고 있는 인터프리터를 보여준다.)

```
> (+ 2 3)
5
> (* (+ 3 3) 2)
12
```

따라서 만일 상수 list(즉, 프로그램이 아닌 데이터 list) 를 표현하고자 할 때는 특별한 표시를 해야한다. 이 때 사용되는 것이 연산자 "\'" 와 키워드 quote이다.

```
> (+ 1 2)
3
> '(+ 1 2)
(+ 1 2)
> (quote (+ 1 2))
(+ 1 2)
```

연산자 “\’” 와 키워드 quote가 문자나 문자열에 적용되면 문자나 문자열 상수를 의미한다. 그렇지 않은 경우는 변수를 의미한다.

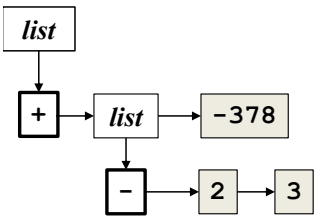
```
> 'a
a
> 'abc
abc
```

만일 quote 후에 여러 item이 나오면 첫 번째 것만 상수로 취하고 나머지는 무시한다. **그러나 편의상, 본 과제에서는 이러한 입력이 없다고 가정한다.**

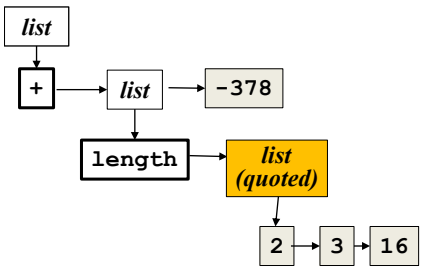
```
> (quote + 1 2)
+
```

다음과 같은 프로그램이 있다고 가정하면, parse tree 는 다음과 같이 된다.

```
(+ (- 2 3) -378)
```




```
(+ (length '(2 3 16)) -378)
```



linked list 의 맨 앞 노드는 연산자나 함수 이름으로 인식한다. 그러나 위 ’(2 3 16)과 같이 list 앞에 ’ 표시가 있거나 (QUOTE (2 3 16)) 과 같이 표현되면 상수 리스트 (데이터)로 인식한다. (그림에서는 색칠된 셀이 가리키는 list)

id 나 기타 expression에 대한 quote 여부도 quoted 태그로 표현된다. 다음은 a 일때(왼쪽)와 ’a 일 때(오른쪽)의 노드 모양이다.

그리고 이들 프로그램의 각 출력결과는 다음과 같다. (키워드 quote를 사용한 경우와 “\’” 를

사용한 경우는 동일하게 출력한다.)

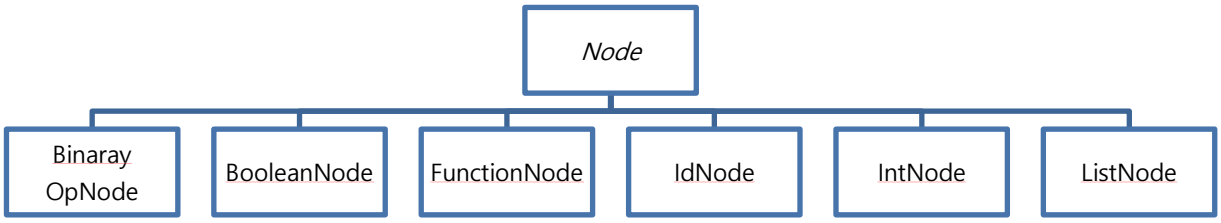
```
 (+ (- 3  2)  -378)
 ([PLUS] ( [MINUS] [INT:3] [INT:2] ) [INT:-378])

(+ (length '(2  3  16))  -378)
([PLUS] ([ID:length] '( [INT:2] [INT:3] [INT:16])) [INT:-378])

a
[ID:a]

'a
'[ID:a]
```

Programming

앞서 과제에서 정의한 getNextToken() 함수를 이용하며, 그 외에도 아래와 같은 자료구조를 사용한다.

1. 노드의 자료구조

Node클래스 수정
- Type필드의 final 키워드 삭제
- getType(), setType()메소드 추가




```
public abstract class Node {
	public enum Type {QUOTED, NOT_QUOTED}
	public Type type;
	Node next;

	public Node(Type type) {
		this.type = type;
		this.next = null;
	}

	public Node getNext(){
		return next;
```

```
	}

	public void setType(Type type){
		this.type = type;
	}


	public void setNext(Node next){
		this.next = next;
	}

	public void setLastNext(Node next){
		if(this.next != null)
			this.next.setLastNext(next);
		else
			this.next = next;
	}

	public Node getNext(){
		return next;
	}
}
…

…
```

2. 프로그램을 parsing하는 클래스 구현 예시

```
public class Parser {
	boolean alreadyParse;
	int pos;
	List<Token> tokenList;

	public Parser(List<Token> list) {
		this.alreadyParse = false;
		this.pos = 0;
		this.tokenList = list;
	}

	private Token getNextToken()
	{
		if(pos < tokenList.size())
			return tokenList.get(pos++);
		else
			return null;
	}

	private void ungetToken()
	{
		if(pos > 0)
			pos--;
	}

	private void errorLog(String err)
	{
		System.out.println(err);
	}
```

```
//호출전에 토큰으로 판별했다면 ungetToken후 호출
//id, int, boolean, list 파싱
private Node parseExpr(){
        Node node = null;
        Token t = getNextToken();


        if(t != null){
                switch(t.type){
                case ID:
                        node = new IdNode(Type.NOT_QUOTED,t.lexme);
                        break;

                …
                …

                default:
                        errorLog("parseExpr switch Error");
                        break;
                }
        }

        return node;
}

//리스트 원소중 키워드, 연산자, id, int, boolean, 중첩리스트 파싱
private Node parseItem()
{
        …
        …
}

//list의 괄호를 뺀 나머지 파싱
private Node parseItemList()
{
        …
        …
}

//호출전에 토큰으로 판별했다면 ungetToken후 호출
//리스트 구조를 파싱
private Node parseList()
{
        …
        …
}

//프로그램을 파싱
public Node parseProgram()
{
        if(alreadyParse){
                errorLog("Already Parse");
                return null;
        }
```

```
		Node list = null;
		Token t = getNextToken();

		if(null != t){
			switch(t.type){
			case APOSTROPHE:
				list = parseExpr(); // 중첩리스트
				list.setType(Type.QUOTED);
				break;

			case ID:
			case INT :
			case TRUE:
			case FALSE:

			case L_PAREN:
				ungetToken();
				list = parseExpr(); // 중첩리스트
				break;

			default:
				ungetToken();
				errorLog("parseSession Error");
				break;
			}
		}

		return list;
	}
```

3. Per-order traverse print class

```
public class Printer {

	PrintStream ps;

	public Printer(PrintStream ps) {
		this.ps = ps;
	}

	//Quoted Node를 '를 이용하여 표현하도록 수정
}
```

4. 테스트

```
public static void main(String[] args){
		read file…

		List<Token> tokens = …
		Parser p = new Parser(tokens);
		Node node = p.parseProgram();

		Printer pt = new Printer(System.out);
		…

}
```