PL Assignment #6: Cute17 Parser

과제물 부과일 : 2017-04-13(목)
Program Upload 마감일 : 2017-04-19(수) 23:59:59(추가 제출 기한 없음)

문제

Cute17 문법에 따라 작성된 program이 문자열에 저장되어 있다. 이를 input으로 하여, 프로그램의 syntax tree를 구성 하시오. (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, Cute17으로 작성된 program이 아래와 같을 경우

`'( + 2 3 ) 또는 ( quote ( + 2 3 ) )`

이와 같은 프로그램의 출력결과는 모두 다음과 같다.

```
'( [PLUS] [INT: 2] [INT: 3] )
```

Cute17의 문법

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

Cute17의 특징

괄호를 써서 프로그램이 표현되는 Cute17는 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)
+
```

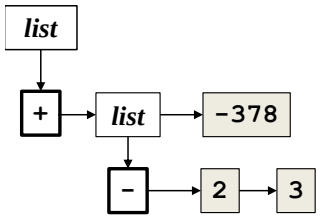
```
a
[ID:a]
```

```
'a
'[ID:a]
```

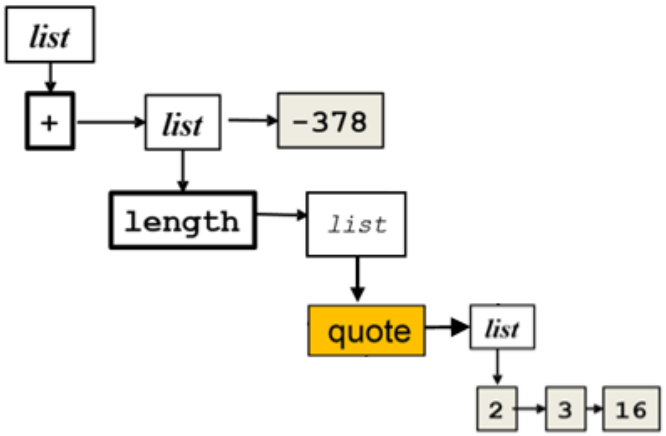
Quote 내부 구조

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

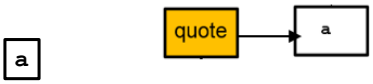
```
(+ (- 2 3) -378)
```




```
(+ (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. id 나 기타 expression에 대한 quote는 quote노드의 next 로 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.)

주의사항

기호 '는 quote함수를 줄여쓰는 것이다. 따라서 겉으로 봤을 때, 구조가 다르게 나와야 할 것 같지만 내부적으로 가지는 구조는 같아야 한다. '(+ 2 3)이 있다고 할 때, 이 노드를 출력하면 ' (+ 2 3)과 같은 식으로 출력이 되지만 내부적인 자료구조는 (quote (+ 2 3))인 구조를 가져야한다.

Programming(지난 주에 이어서 할 것!)

1. **노드의 자료구조**

   노드는 Python 의 특징을 이용한다. Python 은 타입이 실행중에 정의된다. 노드의 value 의 타입은 type 변수가 가지는 값에 따라 타입이 정해진다. 주의할 점은 노드가 List 타입일 경우에는 노드를 value 로 가진다.

```
class Node (object):

    def __init__(self, type, value=None):
        self.next = None
        self.value = value
        self.type = type

    def set_last_next(self, next_node):
        if self.next is not None:
            self.next.set_last_next(next_node)

        else:
            self.next = next_node

    def __str__(self):
        result = ''

        if self.type is TokenType.ID:
            result = '[' + NODETYPE_NAMES[self.type] + ':' + self.value + ']'
        elif self.type is TokenType.INT:
            result = '[' + NODETYPE_NAMES[self.type] + ':' + self.value + ']'
        elif self.type is TokenType.LIST:
            result = '('+str(self.value)+')'
        else:
            result = '['+NODETYPE_NAMES[self.type]+']'

        # fill out
        # next 노드에 대해서도 출력하도록 작성
        # recursion 이용
        # Quote에 대해서도 출력하도록 작성
```

2. 프로그램을 수행하는 프로그램 예시

```
class BasicPaser(object):

    def __init__(self, token_list):
        """
        :type token_list:list
        :param token_list:
        :return:
        """
        self.token_iter = iter(token_list)

    def _get_next_token(self):
        """
        :rtype: Token
        :return:
        """
        next_token = next(self.token_iter, None)
        if next_token is None:
            return None
        return next_token

    def parse_expr(self):
        """
        :rtype : Node
        :return:
        """
        token = self._get_next_token()
        # 하나의 Token을 create_node()를 이용하여 Node로 만들어 반환함
        '""":type :Token"""'
        if token is None:
            return None
        result = self._create_node(token)
        return result

    def _create_node(self, token):
        # 토큰을 Node로 만듬
        if token is None:
            return None
        elif token.type is CuteType.INT:
            return Node(TokenType.INT,  token.lexeme)
        elif token.type is CuteType.ID:
            return Node(TokenType.ID,   token.lexeme)
        elif token.type is CuteType.L_PAREN:
            return Node(TokenType.LIST, self._parse_expr_list())
        elif token.type is CuteType.R_PAREN:
            return None
        elif  # 조건 작성, INT, ID, L_PAREN, R_PAREN을 제외한 나머지 경우:
            # “#T”, “#F”, ‘+’, ‘-‘, ‘*’, ‘/’, ‘<’, ‘>’, ‘=’, “define”, “cond”, “not”, “car”,
            # “cdr”, “cons”, “eq?”, “atom?”, “null?”, 에 대해서 작성
            # 조건에 맞는 노드를 반환하도록 작성
            # 기호 ‘와 quote에 대해서도 작성
            # quote의 노드 구조에 맞게 작성(Quote 내부구조 참조)

        else:
            return None
```

```
        def _parse_expr_list(self):
        # Token이 '('일 경우, list 형태로 만들어서 반환
        head = self.parse_expr()
        '"":type :Node""'
        if head is not None:
            head.next = self._parse_expr_list()
        return head
```

3. 테스트

```
def Test_BasicPaser():
    test_cute = CuteScanner("( + ( - 3 String 2 ) ( ) -378 )")
    test_tokens = test_cute.tokenize()
    print test_tokens
    test_basic_paser = BasicPaser(test_tokens)
    node = test_basic_paser.parse_expr()

    print node


Test_BasicPaser()
```

유의사항

- __str__과 _create_node만을 수정하고, 다른 함수는 수정하지 말 것
- 변수와 메소드를 추가해서는 안됨
- "\'"와 quote로 만들어진 노드는 구조가 같아야 한다.
  - ex) '(+ 2 3) 과 (quote (+ 2 3)) 의 노드 구조는 같아야 한다.
  - 결과는 ' ([PLUS] [INT: 2] [INT: 3]) 와 같이 같은 결과가 나와야 한다.
  - 내부적으로 가지는 구조와 출력해서 보여지는 구조가 다르니 주의해야 한다.

실행결과

1. 입력값이 "(+ (+ 2 3) (quote (+ 4 5)))" 일 때,

```
([PLUS] ([PLUS] [INT:2] [INT:3]) ' ([PLUS] [INT:4] [INT:5]))
```

2. 입력값이 "(quote (+ 3 2))"일 때,

```
' ([PLUS] [INT:3] [INT:2])
```