Lab 9

COMP9021, Session 2, 2015

1 Using linked lists to represent polynomials

Extend the program that implements a class `Polynomial` from the previous lab to implement the functions `__add__()`, `__sub__()`, `__mul__()` and `__truediv__()`.

Next is a possible interaction.

```
$ python
...
>>> from polynomial import *
>>> poly_6 = Polynomial('-2x + 7x^3 +x   - 0  + 2 -x^3 + x^23 - 12x^8 + 45 x ^ 6 -x^47')
>>> print(poly_6)
-x^47 + x^23 - 12x^8 + 45x^6 + 6x^3 - x + 2
>>> poly_7 = Polynomial('2x^5 - 71x^3 + 8x^2 - 93x^4 -6x + 192')
>>> poly_8 = Polynomial('192 -71x^3 + 8x^2 + 2x^5 -6x - 93x^4')
>>> poly_9 = poly_7 + poly_8
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_8)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_9)
4x^5 - 186x^4 - 142x^3 + 16x^2 - 12x + 384
>>> print(poly_7 * poly_7)
4x^10 - 372x^9 + 8365x^8 + 13238x^7 + 3529x^6 + 748x^5 - 34796x^4 - 27360x^3 + 3108x^2 - 2304x + 36864
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_7 - poly_7)
0
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_9 / poly_7)
2
>>> print(poly_9)
4x^5 - 186x^4 - 142x^3 + 16x^2 - 12x + 384
>>> print(poly_7)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> poly_10 = Polynomial('-11x^4 + 3x^2 + 7x + 9')
>>> poly_11 = Polynomial('5x^2 -8x - 6')
>>> poly_12 = poly_10 * poly_11
>>> print(poly_12)
-55x^6 + 88x^5 + 81x^4 + 11x^3 - 29x^2 - 114x - 54
>>> print(poly_12 / poly_10)
5x^2 - 8x - 6
>>> print(poly_12 / poly_11)
-11x^4 + 3x^2 + 7x + 9
>>> poly_13 = poly_6 * poly_7
```

```
>>> print(poly_13 / poly_6)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_13 / poly_7)
-x^47 + x^23 - 12x^8 + 45x^6 + 6x^3 - x + 2
```

2 ☞ Using a stack to evaluate fully parenthesised expressions

Modify the program `postfix.py` from the 9th lecture so that a stack is used to evaluate an arithmetic expression written in infix, fully parenthesised, and built from natural numbers using the binary +, -, * and / operators. *Fully parenthesised* means that all expressions of the form e + e', e - e', e * e' and e / e' are surrounded by a pair of parentheses, brackets or braces. Of course a simple solution would be to replace all brackets and braces by parentheses and call `eval()`, but here we want to use a stack.

Hint: think of popping when and only when a closing parenthesis, bracket or brace is being processed.

Here is a possible interaction:

```
$ python
...
>>> from exercise_2 import *
>>> expression = FullyParenthesisedExpression('2')
>>> expression.evaluate()
2
>>> expression = FullyParenthesisedExpression('(2 + 3)')
>>> expression.evaluate()
5
>>> expression = FullyParenthesisedExpression('[(2 + 3) / 10]')
>>> expression.evaluate()
0.5
>>> expression = FullyParenthesisedExpression('(12 + [{[13 + (4 + 5)] - 10} / (7 * 8)])')
>>> expression.evaluate()
12.214285714285714
```

3 Back to context free grammars

A *context free* grammar is a set of *production rules* of the form

```
symbol_0 --> symbol_1 ...  symbol_n
```

where `symbol_0`, ..., `symbol_n` are either *terminal* or *nonterminal symbols*, with `symbol_0` being necessarily nonterminal. A symbol is a nonterminal symbol iff it is denoted by a word built from underscores or uppercase letters. A special nonterminal symbol is called the *start symbol*. The language *generated* by the grammar is the set of sequences of terminal symbols obtained by replacing

a nonterminal symbol by the sequence on the right hand side of a rule having that nonterminal symbol on the left hand side, starting with the start symbol. For instance, the following, where `EXPRESSION` is the start symbol, is a context free grammar for a set of arithmetic expressions.

```
EXPRESSION --> TERM SUM_OPERATOR EXPRESSION
EXPRESSION --> TERM
TERM --> FACTOR MULT_OPERATOR TERM
TERM --> FACTOR
FACTOR --> NUMBER
FACTOR --> (EXPRESSION)
NUMBER --> DIGIT NUMBER
NUMBER --> DIGIT
DIGIT --> 0
...
DIGIT --> 9
SUM_OPERATOR --> +
SUM_OPERATOR --> -
MULT_OPERATOR --> *
MULT_OPERATOR --> /
```

Moreover, blank characters (spaces or tabs) can be inserted anywhere except inside a number. For instance, `(2 + 3) * (10 - 2) - 12 * (1000 + 15)` is an arithmetic expression generated by the grammar.

Verify that the grammar is *unambiguous*, in the sense that every expression generated by the grammar has a unique evaluation.

Write down a program that prompts for an expression, checks whether it can be generated by the grammar, and in case the answer is yes, evaluates the expression, following this kind of interaction:

```
$ python exercise_3.py
Input expression: 2
The expression evaluates to: 2
$ python exercise_3.py
Input expression: 2 * 2
The expression evaluates to: 4
$ python exercise_3.py
Input expression: (2 + 3) * (10 - 2) - 12 * (1000 + 15)
The expression evaluates to: -12140
$ python exercise_3.py
Input expression: 2 + +3
Incorrect syntax
```