Problem Set 4: Recursion

Handed out: Tuesday, March 12, 2013
Due: Tuesday, March 19, 2013

Introduction

In this problem set, we are going to explore the use of recursion to help solve problems that involve string parsing: 1) testing whether a word contains a set of letters in order, 2) testing whether a word has matching sets of parentheses and 3) evaluating simple mathematical expressions in Polish notation.

These problems should all be solved recursively, not iteratively. You should not use for or while loops in any of these problems.

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Getting Started

Download and unzip ps4.zip. It contains three files: ps4a.py, ps4b.py, and ps4c.py, which correspond to problems 1, 2, and 3, respectively. Running any of these files will run the tester function within that file with some simple test cases we have provided for you. As you fill the methods already defined for you with your own code, use these tests and your own to check your implementation.

Testing your solutions

In all of the files we have provided you, there is a tester function appropriately called `testerFunction`. The functions currently contain a small number of tests. These tests are not exhaustive. You will need to provide additional tests to ensure that your code is working correctly.

We will be grading your tester by running it against functions with subtle bugs and making sure that your tests are able to catch all of them. In particular, you need to make sure your tests cover all the corner cases inherent in the recursive definitions of the functions. **If your tester does not have enough tests to cover the corner cases, you will lose points, even if your implementation of the functions is correct!**

Problem 1: Fredoan

A word is considered *fredoan* if it contains the letters f, r, e, d, and o in it, in that order. For example, we would say that the words "freedom" and "foreshadow" are fredoan because they each contain those five letters in the right order. The word "beforehand" is *not* fredoan because the five letters appear in the wrong order.

In this problem, we want you to write a more generalized function called x_ian(x, word) that returns True if all the letters of x are contained in word in the same order as they appear in x. For example:

```
>>> x_ian('srini', 'histrionic')
True
>>> x_ian('john', 'mahjong')
False
>>> x_ian('dina', 'dinosaur')
True
>>> x_ian('pangus', 'angus')
False
```

You should NOT use built-in functions like "index", "find", or the "in" keyword since they involve implicit iteration, and we're looking for a pure recursive approach. Only use string operations like slicing and indexing (not using the "index" function, but something like string[0]).

```
def x_ian(x, word):
    '''
        Given a string x, returns True if all the letters in x are
        contained in word in the same order as they appear in x.

        x: a string
        word: a string
        returns: True if word is x_ian, False otherwise
    '''
    # your code here
    pass
```

Problem 2: Matching Parentheses

Parentheses operate according to a very simple rule: if you open a parenthesis, you have to close it. For this problem, we are going to write a function that will check if a given expression is properly parenthesized.

The function should return `True` if a string is correctly parenthesized and `False` if it is not. The function should ignore any character that is not an opening or closing round parenthesis.

For example, the following expression is correctly parenthesized:

```
  '( x  (y + z ) w(a b c))(t + x(z))'
```

, but this one is not:

```
  '(x + y (z)'
```

Structuring a recursive solution

Our solution will be in terms of two recursive functions outlined below. These functions can recursively call themselves, or each other.

`checkParenthesis(word):`

The function checks if a string is properly parenthesized. The function returns a boolean `True` or `False` depending on whether the word is properly parenthesized or not. The function is defined recursively as follows:

- Base case 1: Words of length zero are always properly parenthesized.
- Base case 2: If the first letter in a word is a ')', this function is not properly parenthesized (we did not find an opening parenthesis before this closing one).
- Recursive case 1: If the first letter in a word is not a parenthesis or a bracket, the word is properly parenthesized if the rest of the word is properly parenthesized.
- Recursive case 2: If the first letter is a '(', call `completeRound` to find a matching ')' and check parenthesization along the way. If it finds a match, return `False`. Otherwise, check if the rest of the word is correctly parenthesized.

`def completeRound(word):`

The function assumes that word contains the text after an opening round parenthesis has been found (not including the opening parenthesis). It will find a matching closing parenthesis and will check parenthesization between the beginning of word and the matching closing parenthesis. The function returns a pair **`(b, rest)`**, where **`b`** is a boolean describing whether it encountered an parenthesis error, and **`rest`** is the rest of the word after the matching closing round parenthesis.

- Base case 1: If the word is of length zero, it means we can't find a closing round parenthesis. This is an error, so the function should return `(False, something)`. It doesn't matter what `something` is, because it should be ignored.
- Base case 2: If the first character is a closing round parenthesis ')', we have found a match, so we can return `(True, rest)`, where rest is the rest of the word after the closing parenthesis.
- Recursive case 1: If the first letter is not a parenthesis, it can be ignored. The return values should be the same as if the function had been called with word[1:] instead of with word.
- Recursive case 2: Can you figure out the details of what happens when the first character is an opening parenthesis '('? Hint: we need to find a match for this new open parenthesis before we can continue looking for a match for the current open parenthesis in the rest of the word.

Problem 3. Polish Notation Calculator

For this last exercise, you are going to write a simple calculator. The input language for the calculator is called Polish notation. In Polish notation, the mathematical operator comes before its operands (unlike the traditional infix notation we are used to, in which the operator comes in between its operands).

See the Wikipedia article at http://en.wikipedia.org/wiki/Polish_notation#Arithmetic for a more detailed description of Polish Notation. However, the essentials to keep in mind are that in Polish notation, there are two types of expressions:

1. Constants: n

For simplicity, you can assume n is a positive integer.

2. Binary expressions: op e1 e2
   For simplicity, you can assume op can be + or *.
   e1 and e2 are expressions.

For example, the expression '+ 3 5' is a binary expression in which the operator is '+' and e1 and e2 are the constants '3' and '5', respectively. The expression '* + 3 5 4' is a binary expression in which the operator is '*', e1 is the binary expression '+ 3 5', and e2 is the constant '4'. Notice the recursive description of a Polish notation expression.

For implementation purposes, our expressions will be represented as lists of strings, rather than a single string (so the expression '+ 3 5' would be represented as ['+', '3', '5'])

For the following two parts of this problem, we have provided you with a helper function:

```
def isInt(w):
```

which will return True if a given string represents a positive integer, False otherwise. For example, '0' and '456' would return True.

Additionally, for both of the following tasks, you may assume the input expression is a correctly formed Polish Notation expression (i.e. there are no syntax errors).

Converting to Infix Notation

In the first part of this problem, you'll be converting Polish notation expressions into traditional infix notation expressions. Some examples:

['+', '3', '5'] will be converted to (3+5)

['*', '+', '3', '5', '4'] will be converted to ((3+5)*4)

The parentheses are important in infix notation – they convey the order in which the operations should be applied. Note that there are no spaces within the infix expression.

You will do this by filling in the following recursive function:

```
def printExpr(elist):
```

Takes in a list elist representing a Polish Notation expression.

Returns a tuple (infix, rest) where:

- infix is a string with the corresponding infix notation representation.
- rest is the remainder of the list (which has not yet been parsed and converted into infix notation).

When working through a recursive implementation of this function, it will be helpful to consider the following cases:

- Base case 1: the list is empty. In this case we can return the tuple ('', []), because the infix notation representation will be an empty string, and the remainder of the list is also an empty list.
- Base case 2: the first element of the expression is a constant. In this case, the infix representation is simply this constant. The rest of the expression is the expression from the second element onward: expr[1:]
- Recursive case 1: the first element of the expression is the '+' operator. You will then need to recursively call printExpr twice in order to get the infix representations of the following two expressions. This is where it is useful to use the second output of printExpr – the part of the expression that has not been parsed. Once you have the infix notation representation of each of the two expressions, can you figure out how to combine them to form the overall infix expression?
- Recursive case 2: the first element of the expression is the '*' operator. This case should be handled very similarly to Recursive case 1.

Evaluating an Expression in Polish Notation

Your goal is to write a program that, given a list with an expression, returns the value of that expression (simply as a number, not as a string or encapsulated in a list). Some examples:

['+', '3', '5'] should evaluate to 8 (because 3 + 5 = 8)

['*', '+', '3', '5', '4'] should evaluate to 32 (because (3+5)*8 = 32)

To write your calculator, complete the following recursive function:

```
def evalExpr(expr):
```

Evaluates a mathematical expression written in prefix notation.

Returns a tuple (val, rest) where
- val is the value to which an expression evaluates
- rest is the rest of the expression, which has not yet been parsed and evaluated

You will find that you can use the same base and recursive cases you did for the printExpr function; of course, what you should do in each case will be slightly different. In particular, keep in mind that val is a number, rather than a string.

Note that you **may not** use Python's eval function in your code.