RECURSION AND LINKED LISTS 3

COMPUTER SCIENCE 61A

July 1, 2014

1 Termination and Warmup

A function is *recursive* if executing a call to the function requires another call to the same function, called a *recursive call*. A call to a recursive function may *terminate*, or return eventually, if the function only needs to call itself a finite number of times, or may encounter an error if the function needs to call itself an infinite number of times.

1.1 Warmup — What Would Python Do?

1.
```
>>> def blah(n):
...     return blah(n - 1)
>>> blah(2)
```

2.
```
>>> def foo(n):
...     if n == 0:
...         return 2
...     return foo(n%2)
>>> foo(4)
```

   How many times does `foo` call itself? What values of `n` may cause `foo` to run forever?

3.
```
>>> def bar(n, m):
...     if n == 0:
...         return m
...     return bar(n%m, m)
>>> bar(17, 10)
```

   If `bar` returned an answer, what do you know about the values of `n` and `m`?

As you can see, not all recursive functions terminate. These functions all disobey the fundamental principle of solving problems using recursion: the problem you solve in the recursive call must always *reduce towards the base case* — meaning that it must bring the problem closer to a solution.

Base cases are the simplest solution to the problem and serve as stopping points for recursion. When we've reached a base case, *we do not need any further recursive calls to solve the problem*. For example, consider this problem: Alice is in line to buy tickets for a movie, and she wants to know what position she is in line. Alice can't step out of line, or else she'd lose her place. How can she figure out her number in line?

Consider this recursive definition: ask the person in front of her, Bob, for his position in line, and just add one. How does Bob know his position? He can just ask the person in front of him and add one!

This propogates all the way to the front of the line until the second person, Andrew, asks Rohin, the first person in line, for his position. Rohin knows that he's first in line, since there's no one else in front of him, so he tells Andrew that his position in line is 1. Andrew tells the person behind him that he's position 2 in line, and this bubbles back all the way to Alice.

Here Rohin was the base case, since he does not need to ask anyone else to know his position. Asking the person directly in front corresponds to making a recursive call which *reduces towards the base case*. We know that if everyone asks someone closer to the front of the line, we will eventually reach the base case and can start propagating the answer up the line.

2 Linear Recursion

Linear recursion describes functions which include only one recursive call in their body. For example, consider the recursive `fact` function, which computes the factorial of a positive integer `n`:

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)
```

Linear recursion easily ilustrates the *three* common steps in a recursive definition:

1. *Figure out your base case*: Ask yourself, "What is the simplest argument I could possibly get?" The answer should be simple, and is often given by definition. For example, `fact(0)` is 1, by definition.
2. *Make a recursive call that steps towards the base case*: Simplify your problem, and assume that any recursive calls within your function definition will simply work. This is called the "leap of faith" — as you use more recursion, you will get more used to this idea. For `fact`, we make the recursive call `fact(n-1)` — this is the recursive breakdown.
3. *Use your recursive call to solve the full problem*: Remember that we are assuming your recursive call just works. With the result of the recursive call, how can you solve the original problem you were asked? For `fact`, we just multiply $(n-1)!$ by n.

2.1 The Process

Consider the function `remove_alternates`, which takes in a `linked_list` and returns a new `linked_list` which contains every other element of the list.

```
def remove_alternates(lst):
    """
    >>> l = link(1, link(2, link(3, link(4, link(5, empty)))))
    >>> print_linked_list(l)
    <1 2 3 4 5>
    >>> print_linked_list(remove_alternates(l))
    <1 3 5>
    """
```

1. Write an expression which evaluates to the same thing as `remove_alternates` called on `<1 2 3 4 5>` by calling `remove_alternates` on a *slightly smaller* argument. That is, assume that `remove_alternates` already works correctly for smaller arguments - *trust the recursion!*

2. Now generalize this to get a recursive case that works on a general `lst`, assuming `remove_alternates` works perfectly for lists smaller than `lst`. (Remember the constructors and selectors in the `linked_list` data type!) This will be your recursive call.

3. For what sorts of lists would your recursive call not work? Write expressions that work for these base cases.

4. In Questions 1 and 2 you developed the recursive call for `remove_alternates`, and in Question 3 you developed the base case. Put those together into a complete definition of `remove_alternates`.

```
def remove_alternates(lst):
```

2.2 Practice Problems

1. Now define the procedure `every_other`, which returns a list containing every other element starting from the *second*:

```
def every_other(lst):
    """
    >>> l = link(1, link(2, link(3, link(4, link(5, empty)))))
    >>> print_linked_list(l)
    <1 2 3 4 5>
    >>> print_linked_list(every_other(l))
    <2 4>
    """
```

2. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns 9, and `expt(2, -3)` returns 0.125. Assume `power` is an integer. Use recursion.

```
def expt(base, power):
```

3. Define the function `merge`, which takes in two linked lists with elements sorted in ascending order and returns a sorted linked list that contains all the elements of both lists.

```
def merge(lst1, lst2):
    """
    >>> l1 = link(2, link(2, link(5, empty)))
    >>> l2 = link(1, link(5, link(6, empty)))
    >>> lst = merge(l1, l2)
    >>> print_linked_list(lst):
    <1 2 2 5 5 6>
    """
```

4. Write a function `num_to_lst`, which takes in a number and returns a `linked_list` of digits.

```
def num_to_lst(num):
    """
    >>> print_linked_list(num_to_lst(123))
    <1 2 3>
    """
```

5. Now write `lst_to_num`.

```
def lst_to_num(lst):
    """
    >>> l = link(1, link(2, link(3, empty)))
    >>> lst_to_num(l)
    123
    """
```

3 Mutual Recursion

Sometimes a recursive function doesn't directly call itself, but rather calls another function which in turn calls back the original function. These functions are called *mutually recursive*.

As an example, let's try to model a well-known phenomenon of grade school: you forget about half of what you learned over the school year in the summer. Consider this pair of mutually recursive functions:

```
def school_year(grade):
    if grade == 1:
        return 1
    return grade + summer(grade)
def summer(grade):
    return school_year(grade - 1)//2
```

Let's assume that the number of things you learn each school year is equivalent to your grade level, and every summer you forget half of what you learned that school year. School years and summers are numbered as follows: year 1, summer 1, year 2, summer 2, etc. Calling `school_year` on a grade level will tell us how much knowledge you have immediately following that school year, and calling `summer` will tell us how much knowledge you have immediately following that summer.

3.1 Questions

1. Draw an environment diagram for `school_year(3)`.

2. An even number is defined as the successor of an odd number, and an odd number is defined as the successor of an even number. 0 is even. Write two mutually recursive functions `is_even` and `is_odd`, which each take in a positive integer `n` and return a boolean:

```
def is_even(n):



def is_odd(n):
```

3. Two siblings are having an argument, in the classic style of "Is not?"/"Is too!" Each is willing to try `n` times to convince the other, but then gets sick of yelling. Define functions `is_not` and `is_too` which print out their argument. Arguments always start with `is_not()`.

```
def is_not(n):
    """
    >>> is_not(2)
    Is not!
    Is too!
    Is not!
    Is too!
    """



def is_too(n):
```

4 Tree Recursion

Functions that exhibit *tree recursion* contain more than one recursive call in the body. Each call on an argument that isn't one of the base cases results in two or more further calls to the function.

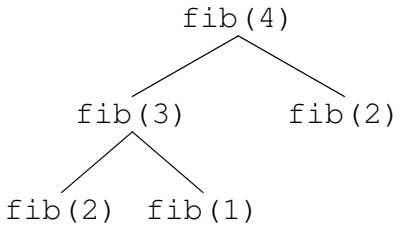
For an analogy, consider this recursive definition of your ancestors: Your ancestors are your mother, your father, your mother's ancestors, and your father's ancestors. In order to find your ancestors, we must find the ancestors of `two` other people.

A simple example of a tree recursive function is `fib`, which computes the n^{th} Fibonacci number:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Tree recursion allows us to explore many different calculations at the same time: in this case, we are exploring two different possibilities (or paths): the $n - 1$ case and the $n - 2$ case.

With the power of recursion, exploring all possibilities like this is very straightforward. You simply try everything using recursive calls for each case, then combine the answers you get back. The different branches of computation form an upside-down tree:



4.1 Questions

1. Draw an environment diagram for `fib(3)`.

2. I want to go up a flight of stairs that has n steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a function `count_stair_ways` that solves this problem for me.

```
def count_stair_ways(n):
```

3. Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynomial $(x + a)^n$. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call its column). Every number in Pascal's triangle is defined as the sum of the item above it and the item that is directly to the upper left of it. If there is a position that does not have an entry, we treat it as if we had a 0 there. Below are the first few rows of the triangle:

```
Item:     0    1    2    3    4    5
Row 0:    1
Row 1:    1    1
Row 2:    1    2    1
Row 3:    1    3    3    1
Row 4:    1    4    6    4    1
Row 5:    1    5    10   10   5    1
...
```

Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle.

```
def pascal(row, column):
```

4. The TAs want to print handouts for their students. However, for some unfathomable reason, both the printers are broken; the first printer only prints multiples of `n1`, and the second printer only prints multiples of `n2`. Help the TAs figure out whether or not it is possible to print an exact number of handouts!

```
def has_sum(sum, n1, n2):
    """
    >>> has_sum(1, 3, 5)
    False
    >>> has_sum(5, 3, 5) # 1(5) + 0(3) = 5
    True
    >>> has_sum(11, 3, 5) # 2(3) + 1(5) = 11
    True
    """
```

5. Write a function `deep_sum_linked_list` which returns the sum of all the numbers in a *recursive linked list*. The elements of a recursive linked list are all either numbers or other recursive linked lists. Assume you have `empty` and `is_list` defined.

```
def deep_sum_linked_list(l):
    """
    >>> lst = link(3, link(link(link(5, empty), link(7, empty)), \
        link(2, empty)))
    >>> print_deep_list(lst)
    <3 <<5> 7> 2>
    >>> deep_sum_linked_list(lst)
    17
```

5 Challenge Problem

When we have an expression in Python, `add(1, mul(2, 3))`, we use parentheses to show which operands are being operated on by which function. However, it turns out that when we only have functions with two arguments (binary operators), then you can remove the parentheses and the notation is still unambiguous.

Imagine Rohin decides to use linked lists in order to write expressions in this notation. Your job is to define a function, `interpret`, which takes a linked list, `expr`, and returns the value that the expression would evaluate to. For instance, `<add mul 2 4 7>` is equivalent to the expression `add(mul(2, 4), 7)`, which evaluates to 15.

(Hint 1: `type` is a built-in function returns the type of an object, so `type(x) == int` will evaluate to `True` if x is an integer.)

(Hint 2: Instead of having a function that just returns a number, create a helper function that retuns both a number and the unevaluated part of the linked list.)

Note: In the doctests below, when printing the linked list, we print the functions as just 'add', 'sub', and 'mul', even though this Python would print something longer.

```
def interpret(expr):
    """Interprets Rohin's linked list expression

    >>> from operator import add, sub, mul
    >>> expr1 = link(add, link(mul, link(2, link(4, link(7, empty)))))
    >>> print_linked_list(expr1)
    <add mul 2 4 7>
    >>> interpret(expr1)
    15
    >>> expr2 = link(1, link(3, link(sub, link(5, link(2, empty)))))
    >>> expr2 = link(add, link(mul, link(2, link(add, expr2))))
    >>> print_linked_list(expr2)
    <add mul 2 add 1 3 sub 5 2>
    >>> interpret(expr2)
    11
    """
```