Module 05: Types of recursion

Topics:

- Review of purely structural recursion
- Accumulative recursion
- Generative recursion

Readings:ThinkP 5.8-5.10, 6.5-6.7

Review: Structural Recursion

- Template for code is based on recursive definition of the data, for example:
  - Basic list template
  - Countdown template for natural numbers
- In our recursive call, our recursive data is one step closer to the base case

Recall how Structural Recursion works

```
def factorial(n):
  if n <= 1:
    return 1
  else:
    return n * factorial(n-1)
```

Trace **factorial**

```
factorial (6)
⇨ 6 * factorial(5)
⇨ 6 * (5 * factorial(4))
⇨ 6 * (5 * (4 * factorial(3)))
⇨ 6 * (5 * (4 * (3 * factorial(2))))
⇨ 6 * (5 * (4 * (3 * (2 * factorial(1)))))
⇨ 6 * (5 * (4 * (3 * (2 * 1))))
⇨ 6 * (5 * (4 * (3 * 2)))
⇨ 6 * (5 * (4 * 6))
⇨ 6 * (5 * 24)
⇨ 6 * 120
⇨ 720
```

Now, for something slightly different

- What if we multiplied the numbers "as we go"
- We'd need an additional parameter to remember this product – we call this new parameter an "accumulator"
- To accommodate the extra parameter, we need a helper function.

Alternate approach to **factorial**

```
def remember_fact(product, n0):
    if n0 <= 1:
        return product
    else:
        return remember_fact(
                product * n0, n0-1)

def factorial2(n):
    return remember_fact(1, n)
```

Trace `factorial2`

```
factorial2 (6)
⇨ remember_fact(1, 6)
⇨ remember_fact(6, 5)
⇨ remember_fact(30, 4)
⇨ remember_fact(120, 3)
⇨ remember_fact(360, 2)
⇨ remember_fact(720, 1)
⇨ 720
```

Differences and similarities in implementations

- **`factorial2`** needs a helper function to keep track of the work done so far
- Both implementations are correct, but
  - **`factorial`** does all calculations after reaching the base case
  - **`factorial2`** does the calculations as we go
    - **`product`** is called the "accumulator"
- Mathematically equivalent, but not computationally equivalent.

Accumulative Approach

- This technique is known as structural recursion with an accumulator, or just accumulative recursion.
- A helper function is required
- It may be a bit harder to trace than pure structural recursion due to the helper
- The main function is a wrapper function that sets the initial value of the accumulator(s).
- The main function may also handle special cases.

More on the helper function

- The recursive helper function requires at least two parameters:
  - One to keep track of what has been done on previous recursive calls (the “accumulator”)
  - One to keep track of what remaining to be processed (used to identify the base or stopping cases)
- Some problems may need more than one accumulator or tracker parameters

Accumulative function pattern

```
def acc_template(acc, remaining, …):
   # if at stopping case of remaining
   #   return (answer using acc)
   # else:
   #   return acc_template(updated-acc,
   #                 updated-remaining. …)

def fn(…):
   # process result of calling
   #   acc_template(initial-acc,
   #                initial-remaining, …)
   # Note: consider special cases, as needed
```

Accumulative recursion ...

- *Might* make better use of space
- *Might* help code run faster *(... or not)*
  - *More on this later*
- May lead to a more natural solution
- But may be harder to trace or test

It provides another option when developing a solution

Testing Accumulative Recursive Code

- Test all statements in main function, including
  - base case of helper
  - recursive case(s) of helper
- Be careful: Failing tests could be due to
  - Errors in the helper base case(s)
  - Errors in the helper recursive case(s)
  - Errors in the initial values in call to helper
  - Errors in other parts of the functions

Another accumulative example: Fibonacci numbers

The nth Fibonacci number is the sum of the two previous Fibonacci numbers:

$$f_n = f_{n-1} + f_{n-2},$$

where $f_0=0, f_1=1$.

These numbers grow very quickly!

$$f_5 = 5, \quad f_{10} = 55, \quad f_{15} = 610,$$

$$f_{20} = 6765, \quad f_{25} = 75{,}025,$$

$$f_{30} = 832{,}040, \quad f_{35} = 9{,}227{,}465$$

First attempt: straight from the definition

```
def fib(n):
  '''returns nth Fibonacci number
     fib: Nat -> Nat
     Example: fib(1) => 1,
        fib(10) => 55
  '''
  if n == 0: return 0
  elif n == 1: return 1
  else:
     return fib(n-1) + fib(n-2)
```

But, this is ***very*** slow. Why?

- Consider **`fib (10)`**:
  - **`fib(9)`** is called 1 times
  - **`fib(8)`** is called 2 times
  - **`fib(7)`** is called 3 times
  - **`fib(6)`** is called ***??*** times
  - …
  - **`fib(1)`** is called ***??*** times
- How many times is **`fib(1)`** called to calculate **`fib(n)`** for any value of **`n`**?

Use Accumulative Recursion

- Remember the Fibonacci numbers by storing them in a list:

  `[0,1,1,2,3,5,8...]`

- But
  - Need fast access to two most recent numbers
  - Equally fast to get them from the beginning or end of list – let's build the list in increasing order (as we would by hand)

Use Accumulative Recursion

- Accumulate previous Fibonacci numbers in a list built in increasing order (calls it **`fibs`**)
- The next number is **`fibs[-1]+fibs[-2]`**
- Append it to the end of **`fibs`**
- Also, use **`n0`** to keep track of which Fibonacci number is at the end of the list
- Stop when **`n0`** equals **`n`**

An improved Fibonnaci

```
def fib_acc(n, n0, fibs):
    if n0>=n: return fibs[-1]
    else:
        fibs.append(fibs[-1]+fibs[-2])
        return fib_acc(n, n0+1, fibs)

def fib2(n):
    if n==0:
        return 0
    else:
        return fib_acc(n, 1, [0, 1])
```

Tracing **`fib2`**

```
fib2(10)
⇨ fib_acc(10,1,[0,1])
⇨ fib_acc(10,2,[0,1,1])
⇨ fib_acc(10,3,[0,1,1,2])
⇨ fib_acc(10,4,[0,1,1,2,3])
⇨ fib_acc(10,5,[0,1,1,2,3,5])
...
⇨ fib_acc(10,10,[0,1,1,2,3,5,8,13,21,34,55])

⇨ 55
```

Why is `fib2` faster?

- Consider **`fib2(10)`:**
  - **`fib_acc(10,1,...)`** is called 1 time
  - **`fib_acc(10,2,...)`** is called 1 time
  - **`fib_acc(10,3,...)`** is called 1 time
  - **`fib_acc(10,4,...)`** is called ***??*** times
  - …
  - **`fib_acc(10,10,...)`** is called ***??*** times
- How many times is **`fib_acc`** called to calculate **`fib2(n)`** for any value of **`n`**?

Improving **`fib2`**

- Anything wrong with **`fib2`**?
  - Remembered all previous numbers
  - Really only needed last two

Another implementation

```
def fib3_acc(n, n0, last, prev):
    if n0 >= n: return last
    else:
        return fib3_acc(n,n0+1,
                        last+prev, last)

def fib3(n):
    if n==0: return 0
    else: return fib3_acc(n,1,1,0)
```

Design choices

Two important features of a computer program are

- how much time it takes *(more about this later)*
- how much memory it uses.

Often these are in opposition.

You can see much more about these topics in Module 07, and course such as CS 234 or 240,

Reversing a List

```
def invert(lst):
  '''returns a list like lst, but in
        reverse order
     invert:(listof Any)->(listof Any)
     Example:
     invert([1,2,3]) => [3,2,1]'''
  if len(lst)<=1: return lst
  else:
    return invert(lst[1:])+[lst[0]]
```

Tracing `invert`

```
invert([1,2,3,4])
⇨ invert([2,3,4]) + [1]
⇨(invert([3,4]) + [2]) + [1]
⇨((invert([4]) + [3]) + [2]) + [1]
⇨(([4] + [3]) + [2]) + [1]
⇨([4,3] + [2]) + [1]
⇨[4,3,2] + [1]
⇨[4,3,2,1]
```

An accumulative approach

```
def build_rev(L,acc):
    if L == []:
        return acc
    else:
        return build_rev(
                    L[1:],[L[0]]+acc)
def rev(L):
    return build_rev(L,[])
```

Tracing **rev**

```
rev([1,2,3,4]
⇨ build_rev([1, 2, 3, 4], [])
⇨ build_rev([2, 3, 4], [1])
⇨ build_rev([3, 4], [2, 1])
⇨ build_rev([4], [3, 2, 1])
⇨ build_rev([], [4, 3, 2, 1])
⇨[4,3,2,1]
```

Sometimes a natural structurally recursive solution may not exist

A palindrome is a string that reads the same forwards and backwards, for example, **`"abcba"`** or **`"12 33 21"`**.

Write a recursive Python function **`is_palindrome`** that consumes a string (**`s`**), and returns **`True`** if **`s`** is a palindrome, and **`False`** if not. Note that any string of length less than 2 is a palindrome.

This is not structural recursion – but it works!

```
def is_palindrome(s):
  if len(s)<2:
    return True
  else:
    return s[0]==s[-1] and \
           is_palindrome(s[1:-1])
```

Generative Recursion

- Consider new ways (other than the definition of the data) to break into subproblems.
- Requires more creativity in solutions.
- We can do more –
  - Different solutions techniques.
  - Even more problems can be solved.
- More variations – no standard template.

Steps for Generative Recursion

1. Break the problem into any subproblem(s) that seem natural for the problem.
2. Determine the base case(s).
3. Solve the subproblems, recursively if necessary.
4. Determine how to combine subproblem solutions to solve the original problem.
5. TEST! TEST! TEST!

Example: **`gcd`**

- The greatest common divisor (*gcd*) of two natural numbers is the largest natural number that divides evenly into both.
  - *gcd (10, 25) = 5*
  - *gcd (20, 22) = 2*
  - *gcd (47, 21) = 1*
- Exercise: Write **`gcd`** function using the standard count down template.

Euclid's Algorithm for **gcd**

- gcd(m,0) = m
- gcd(m,n) = gcd(n, m mod n)

```
def gcd(m,n):
    if m==0:  return n
    elif n==0:  return m
    else:  return gcd(n, m % n)
```

Tracing **gcd**

```
gcd(25, 10)
⇨ gcd(10, 25 % 10)
⇨ gcd(10, 5)
⇨ gcd(5, 10 % 5)
⇨ gcd(5, 0)
⇨ 5
```

Note that second argument is getting smaller on each recursive call

Comments on **`gcd`**

- Not structural (not counting up or down by 1)
- Generative recursion
  – Still has a base case
  – Still has a recursive case – but problem is broken down in a new way

Example: removing duplicates

```
def singles(lst):
  '''returns a list like lst, containing
       only the first occurrences of each
       element in lst
     singles: (listof X) -> (listof X)
     Examples:
     singles([]) =>  []
     singles([1,2,1,3,4,2]) => [1,2,3,4]
  '''
```

```
def singles(lst):
  if lst==[]: return []
  else:
    first = lst[0]
    rest = lst[1:]
    f_rem = list(filter(lambda x:
                          x!=first, rest))
    return [first] + singles(f_rem)
```

Example: reversing a number

Write a function **`backwards`** that consumes a natural number and returns a new number with the digits in reverse order.

For example,

- **`backwards (6)`** ⇨ **`6`**
- **`backwards (89)`** ⇨ **`98`**
- **`backwards (10011)`** ⇨ **`11001`**
- **`backwards (5800)`** ⇨ **`85`**

A Possible Approach

Consider the number n = 5678

- Divide the number into:
  - Last digit: 8
  - Everything else: 567
- Next, reverse 567
  - Take last digit (7) and "add to" 8 ⇨ 87
  - What's left? 56
- Repeat the process until all digits processed.

Coding the Approach

- Use accumulative recursion
- The helper function will keep track of:
  - The digits that have been reversed so far
  - The digits that still need to be reversed
  - The helper will use generative recursion
    - Counting up or down by 1 doesn't help!

```
def bw_acc(so_far, res):
    if res == 0:
        return so_far
    else:
        next_so_far = so_far*10 + res % 10
        next_res = res // 10
        return bw_acc(next_so_far,
                      next_res)

def backwards(n):
  '''returns the value of n with digits
     reversed.
    backwards: Nat -> Nat
    Example: backwards(123) => 321
  '''
  return bw_acc(0, n)
```

Comments on Generative Recursion

- More choices increases chances for errors
- Design recipe:
  - No general template for generative recursion
  - Contract, purpose, examples are still important
  - Testing as important as ever!
- Structural recursion remains best choice for many problems
- An algorithm can use combinations of different types of recursion

Locally defined functions (inner functions) in Python

- The helper functions used in our accumulative recursion solutions are not generally used any where else
- In Racket, we would have defined them using **`local`**
- In Python, we can simply define the helper function inside the primary function
- You *may* define helper functions locally on assignments, but you are not required to

```
def fib4(n):
    '''returns nth Fibonacci
       fib4: Nat -> Nat
       Example: fib4(10) => 55
    '''

    def acc(n0, last, prev):
      '''returns nth Fibonacci number, where last
           is the n0th, and prev is (n0-1)th
         acc: Nat Nat Nat -> Nat
      '''
      if n0 >= n: return last
      else:
          return acc(n0+1, last+prev, last)
    # Body of fib4
    if n==0: return 0
    else: return acc(1,1,0)
```

Warnings about locally defined functions

- The values of local variables (including parameters) for a function can be used inside locally defined functions, but they cannot be changed inside the local function.
- The values of local variables (including parameters) of an inner function cannot be used outside the local function.
- You may use local functions if you wish, but you are **not** required to. *(Suggest you generally don't.)*

Goals of Module 05

- Understand how to write recursive functions which are not purely structurally recursive.
- Understand how to test such functions.