Day 9

COMP1006/1406

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
  - Assignment 5 and the Project are out!
- a quick look back
  - Bugs
  - Exception handling
- Recursion
- Efficiency

last time...

bugs... exception handling...

Recursion

is the process of repeating something in a `self-similar` way

Recursion

is the process of repeating something in a `self-similar` way

Recursion

recursive definitions

- must have at least one base case
- must have a recursive part (self-similar part)

Recursion

recursive definitions

- must have at least one base case
- must have a recursive part (self-similar part)

recursive methods

- must have at least one base case
- must have a recursive part (self-similar part)

Recursion

recursive definitions

- must have at least one base case
- must have a recursive part (self-similar part)

recursive methods

- must have at least one base case
- must have a recursive part (self-similar part)

recursive data structures

- an implicit base case (typically empty case)
- must have a self-similar part
  (typically an attribute which is itself)

Recursion

Let's look at the factorial function and Fibonacci numbers as a review/introduction to recursion.

- factorial $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

$$fact(n) = n \times fact(n-1)$$

- the n-th Fibonacci number is defined as

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

Recursive Data Types

a **linked list** is an implementation of a list (ADT)

it is the natural implementation of a list using a **recursive** definition:

A **list** is

- empty, or
- a single item (**first**) followed by a list (**rest**)

In languages like scheme or lisp which are based around lists, you use the words **car** (for first) and **cdr** (for the rest of the list).

Recursive Data Types

a basic recursive data type is a **node**

- (usually) contains some data
- contains one or more references to other nodes
  (a link, a pointer, a reference)
  (it may have more than one reference to other nodes)

```
public class Node{
  Object data;
  Node next;
}
```

```
public class Node<T>{
  T data;
  Node next;
}
```

```
public class Node{
  int data;   // or String data;
  Node next;
}
```

Recursive Data Types

a **linked list** in Java

```
public class Node{
  String data;
  Node   next;
}

public class LinkedList{
  Node head;
  Node tail;
  int  size;
}
```

Recursive Data Types

```
public class LinkedList{
  Node head;
  Node tail;
  int size;

  String first(){
    if(head == null){ return head; }
    return head.data;

  LinkedList rest(){
    if(head==null || head.next==null){ return null; }
    LinkedList list = new LinkedList();
    list.head = head.next;
    list.tail = tail;
    list.size = size-1;
    return list;
  }
}
```

Recursive Data Types

a **traversal** of a data structure is a way of visiting each data element in the data structure. For a linked list is just a method of visiting each node in the linked list

traversal of recursive data structures usually require very few lines of code (if we have a nice recursive definition)

but we need to be careful of the details! the base case in particular

Let's try a traversal to do the following:

- print a list
- print a list in reverse order

Recursive Data Types

```
print(list):
  if the list is empty do nothing
  (end function)

  otherwise, the list is not empty

  print the first element of the list

  recursively print the rest of the list
```

```
public void print(LinkedList list){
   if(list.size()==0) return 0;

   System.out.println(list.first());
   print(list.rest());
}
```

Recursive Data Types

How can we print the elements in reverse order?

When using recursion we are implicitly using a **stack** because each functions gets pushed to the function call stack when called.

Recursive Data Types

How can we print the elements in reverse order?

When using recursion we are implicitly using a **stack** because each functions gets pushed to the function call stack when called.

```
public void print(LinkedList list){
   if(list.size()==0) return;

   print(list.rest());
   System.out.println(list.first());
}
```

The code is the same except that two lines are swapped!

Recursive Data Types

What else can we do easily with recursion on a list?

- Add all the numbers in a list
- Find the maximum/minimum in a list
- Create sublist that only contains the even numbers of a given list
- ...

Recursive Data Types

a **binary tree** is a non-linear data structure (think of a family tree)

we can define it as follows

A **binary tree** is

- empty (base case), or
- an item and two binary trees (called left and right)

```
public class Node{
  String data;
  Node   left;
  Node   right;
}

public class BinaryTree{
  Node root;
}
```

Recursive Data Types

What about a traversal for a binary tree? How do we visit each node (data) in a binary tree?

- how did we do this for a linked list?
  - traversal came right from recursive definition
- can we generalize this for binary trees?
  - we have a base case (empty binary tree)
  - we now have two recursive cases instead of one (left and right)

Recursive Data Types

What about a traversal for a binary tree? How do we visit each node (data) in a binary tree?

- how did we do this for a linked list?
  - traversal came right from recursive definition
- can we generalize this for binary trees?
  - we have a base case (empty binary tree)
  - we now have two recursive cases instead of one (left and right)

```
public void traverse(BinaryTree tree){
   if(tree.size()==0) return;

   System.our.println(tree.root.data);
   traverse( tree.root.left );
   traverse( tree.root.right );
}
```

Recursive Data Types

binary tree traversals

- pre-order
  - process current node, visit left subtree, visit right subtree
- in-order
  - visit left subtree, process current node, visit right subtree
- post-order
  - visit left subtree, visit right subtree, then process the current node

It is convention to always visit the left child (subtree) before the right child (subtree).

Recursion

There are different notions/classes of recursion.

- **primitive recursion**
  - a function $f(n)$ is defined over non-negative numbers
  - base case is $n = 0$
  - recursive case is $n > 0$ and calls self with input $n - 1$
- **general recursion**
  - does not need to work over integers (linked lists)
  - can have multiple base cases (Fibonacci numbers)
  - can have multiple recursive cases (binary tree traversal)
- **generative and accumulative recursion**
  - in generative recursion the recursive cases are constructed (generated) from the problem being solved (not based directly on the data's definition)
  - in accumulative recursion, input parameters (accumulators) are added to build up a solution (or to pass extra information to the next function call)

Recursion

A recursive function is often a helper function that is called from another non-recursive function.

The recursive help function will often have more parameters than the main function

```
public int sublist(List list, int start, int end){

    return sublistRecursive(list, start, end, null);

}
```

Examples

Let's look at some more examples

- find the k-th element in a linked list
- add/remove the k-th element in a linked list
- find the maximum element in a binary tree

Tail Recursion

There is overhead involved when using recursion.

- each time a function is called a new activation record is pushed to the stack (this costs time and uses up stack space)

Consider the two recursive functions

```
int sum(LinkedList list){
  if(list.size == 0){ return 0; }
  return list.first() + sum(list.rest());
}
```

```
int sum(LinkedList list, int sum){
  if(list.size==0){ return sum; }
  return sum(list.rest(), sum+list.first()));
}
```

Tail Recursion

The second function is an example of **tail recursion**. In tail recursion, the very last operation of the method (other than the base case) is a recursive call. If the function returns a value then the return value is simply the value returned from the recursive call.

- each time a function is called a new activation record is pushed to the stack (this costs time and uses up stack space)

Some languages (or compilers) can optimize code using tail recursion by only creating a single activation record on the stack and reusing it for each recursive call. This saves time and space. Scheme is a language that guarantees optimized tail recursion. You do not need to worry about running out of stack space when using recursion in Scheme. (why?)