Recursion

CS 240

Alex Vondrak

ajvondrak@csupomona.edu

Winter 2012

Recursion

Definition

Recursion is the process of defining something in terms of itself

- The base case is the simplest instance of the definition, which requires no self-reference
- The recursive case is a more complex instance of the definition, which relies on self-reference to a simpler case (i.e., an instance closer to being the base case)

Example (Recursive Exponentiation)

Suppose we're dealing with natural numbers $(0, 1, 2, \ldots)$.
Exponentiation can be defined recursively upon the operands a and b:

$$a^0 = 1 \qquad \text{(Base Case: } b = 0\text{)}$$
$$a^b = a \times a^{b-1} \qquad \text{(Recursive Case: } b > 0\text{)}$$

Multiple Choice Question

In Java, how can we use recursion in an arbitrary method, `m`?

(A) Use the special **`recursive`** type declaration

(B) Have a method call to `m` within the definition of `m`

(C) We actually can't have a recursive method

(D) Have a method call (within the definition of `m`) to some other method, m_2, that will eventually call `m` again

Multiple Choice Question

In Java, is it possible to have a recursively-defined data type?

(A) Yes

(B) No

(C) I'm not sure

Multiple Choice Question

The `Node<E>` class is recursively defined!
For a linked list with a `head` of type `Node<E>`, we have the following cases:

- ...
- ...

What is the most basic instance of a `Node<E>`, which contains no further `Node<E>` structure?

(A) A `Node<E>` with **`null`** `data`

(B) A `Node<E>` with a **`null`** `link`

(C) A `Node<E>` with **`null`** `data` and a **`null`** link

(D) Just **`null`**

Multiple Choice Question

The `Node<E>` class is recursively defined!
For a linked list with a `head` of type `Node<E>`, we have the following cases:

- `head` is **`null`** (Base Case)
- `head` is a `Node<E>` instance (Recursive Case)

Why is a `Node<E>` instance the recursive case?

(A) Because `head.link` will also be the head of a linked list

(B) Because `head.link` might be **`null`** or might be another `Node<E>` instance

(C) Because we can use `Node<E>` to represent part of the structure of `head`, which itself is of type `Node<E>`

(D) All of the above

Multiple Choice Question

A recursive method that manipulates linked lists can inherit a structure based on the linked list itself.
Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
    ⋮
}
```

How can such a method possibly be recursive?

(A) Make a call to the `size` method of `this.head.link`

(B) It can't; it takes no arguments

(C) We can call a "helper" method that *is* recursive

(D) None of the above

Multiple Choice Question

A recursive method that manipulates linked lists can inherit a structure based on the linked list itself.
Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    ⋮
}
```

Why can these methods be named the same thing?

(A) Because helper methods can be named the same thing as the primary methods
(B) Because one is **`public`**, the other is **`private`**
(C) Because one takes no arguments, the other takes one
(D) None of the above

Multiple Choice Question

A recursive method that manipulates linked lists can inherit a structure based on the linked list itself.
Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    ⋮
}
```

What is the base case for `size(Node<E>)`?

(A) When `list == null`
(B) When `list.link == null`
(C) When `list.isEmpty()`
(D) When `list.link.isEmpty()`

Multiple Choice Question

Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        ...
    }
    ...
}
```

What should we do is `list` matches the base case?

(A) `return list.data;`

(B) `return list.link;`

(C) `return 1;`

(D) `return 0;`

Multiple Choice Question

Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
   return this.size(this.head);
}

private int size(Node<E> list) {
   if (list == null) {
      return 0;
   }
   ...
}
```

What do we know about the recursive case?

(A) `list != null`

(B) `list` is an instance of `Node<E>`

(C) `list.link` won't give an error

(D) All of the above

Multiple Choice Question

Consider writing a recursive `size` method in the `LinkedList<E>` class.

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return ??? this.size(list.link);
}
```

What should we do to the result of `this.size(list.link)` to get the desired result (the size of `list`)?

(A) Add 1

(B) Add the size of the `head`

(C) Nothing; just return `this.size(list.link)`

(D) None of the above

A Recursive size Method

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

Multiple Choice Question

```
private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

In our recursive size method, in what order do we evaluate the parts of the last line?

(A) First **return**, then `1 +`, then `this.size(list.link)`

(B) First `1 +`, then `this.size(list.link)`, then **return**

(C) First `this.size(list.link)`, then `1 +`, then **return**

(D) None of the above

Tail Recursion

Definition (Tail Position)

An arbitrary method call is in the tail position if it's the return value of a method. That is, a call `m(x, y, z, ...)` is in the tail position if it's the last value calculated before a **`return`**.

Definition (Tail Call)

A method call is a tail-call if it is in the tail position.

Definition (Tail Recursion)

A recursive method is tail-recursive if every recursive call is a tail-call.

Multiple Choice Question

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

Is the call to **this**.size(**this**.head) a tail-call?

(A) Yes

(B) No

(C) I don't know

Multiple Choice Question

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

Is the call to `this.size(list.link)` a tail-call?

(A) Yes

(B) No

(C) I don't know

Multiple Choice Question

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

Is the `public int size()` method tail-recursive?

(A) Yes

(B) No

(C) I don't know

Multiple Choice Question

```
public int size() {
    return this.size(this.head);
}

private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

Is the **private int** `size(Node<E> list)` method tail-recursive?

(A) Yes

(B) No

(C) I don't know

Multiple Choice Question

Let's make `size(Node<E>)` tail-recursive.

```
private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return 1 + this.size(list.link);
}
```

How can we make the recursive call a tail call?

(A) Don't add 1 to the result

(B) Remove it from the **return** statement (just put the result in a variable)

(C) Come up with a different recursive case

(D) None of the above

Multiple Choice Question

Let's make `size(Node<E>)` tail-recursive.

```
private int size(Node<E> list) {
    if (list == null) {
        return 0;
    }
    return this.size(list.link);
}
```

Is `this.size(list.link)` a tail-call this way?

(A) Yes

(B) No

(C) Maybe

Multiple Choice Question

Let's make size(Node<E>) tail-recursive.

```
private int size(Node<E> list, ???) {
    if (list == null) {
        return 0;
    }
    return this.size(list.link, ???);
}
```

How can adding a new parameter help?

(A) We can put the recursive call as an argument, then just have a tail-call to an add method

(B) We could pass a boolean for whether to add 1 to the size or not

(C) We could keep a running total of the number of items we've seen

(D) None of the above

Multiple Choice Question

Let's make `size(Node<E>)` tail-recursive.

```
private int size(Node<E> list, int total) {
    if (list == null) {
        return 0;
    }
    return this.size(list.link, ???);
}
```

Is returning `0` in the base case still correct?

(A) Yes

(B) No

(C) Maybe

Multiple Choice Question

Let's make `size(Node<E>)` tail-recursive.

```
private int size(Node<E> list, int total) {
    if (list == null) {
        return total;
    }
    return this.size(list.link, ???);
}
```

What should put in place of the `???`?

(A) `total`

(B) `list.data + total`

(C) `this.size(list) + 1`

(D) None of the above

Multiple Choice Question

```
public int size() {
   return this.size(this.head);
}

private int size(Node<E> list, int total) {
   if (list == null) {
      return total;
   }
   return this.size(list.link, total + 1);
}
```

What should the initial parameters to `size(Node<E>, int)` be?

(A) `this.head` and 1

(B) `this.head` and 0

(C) `this.head.link` and 1

(D) `this.head.link` and 0

A Tail-Recursive size Method

```
public int size() {
   return this.size(this.head, 0);
}

private int size(Node<E> list, int total) {
   if (list == null) {
      return total;
   }
   return this.size(list.link, total + 1);
}
```

Multiple Choice Question

Let's write a recursive `reverse` (helper) method.

```
private Node<E> reverse(Node<E> list) {
   ⋮
}
```

What is the base case?

(A) When `list == null`

(B) When `list.link == null`

(C) Either of the above

(D) None of the above

Multiple Choice Question

Let's write a recursive `reverse` (helper) method.

```
private Node<E> reverse(Node<E> list) {
    if (list == null) {
        ...
    }
    ...
}
```

What is the reverse of an empty list?

(A) Undefined; the base case should be `list.link == null`

(B) An empty list

(C) Depends on if **`null`** is used as an end-of-list marker or represents an actually empty list

(D) None of the above

Multiple Choice Question

Let's write a recursive `reverse` (helper) method.

```
private Node<E> reverse(Node<E> list) {
    if (list == null) {
        return null;
    }
    ...
}
```

What should the recursive action be (i.e., the call that moves `list` closer to the base case)?

(A) `this.reverse(list.link)`

(B) `this.reverse(list)`

(C) Either of the above could work

(D) None of the above

Multiple Choice Question

Let's write a recursive `reverse` (helper) method.

```
private Node<E> reverse(Node<E> list) {
    if (list == null) {
        return null;
    }
    return ??? this.reverse(list.link);
}
```

What task do we need to perform before returning the final result?

(A) Add `list.data` to the front of `this.reverse(list.link)`

(B) Add `list.data` to the back of `this.reverse(list.link)`

(C) We need to replace the `this.reverse(list.link)` call with a more complex expression

(D) We don't need to do anything

Multiple Choice Question

Let's write a recursive `reverse` (helper) method.

```
private Node<E> reverse(Node<E> list) {
    if (list == null) {
        return null;
    }
    return this.addToBack(list.data, this.reverse(list.link));
}
```

What's wrong with this solution?

(A) We don't have such a method (`this.addToBack`)

(B) It's not tail-recursive

(C) It's inefficient

(D) It violates object-orientation principles

Multiple Choice Question

Let's make `reverse` tail-recursive.

```
private Node<E> reverse(Node<E> list, ???) {
    if (list == null) {
        return ???;
    }
    return ???;
}
```

As with many tail-recursive methods, we'll find a need to carry around an extra parameter (so we do computation upon the argument, rather than upon the recursive result).

What should this extra parameter be?

(A) An **`int`** to keep track of how far along we are

(B) An `E` item to hold onto the `list.data` we have to append

(C) A new `LinkedList<E>` we can tack elements onto

(D) A `Node<E>` to hold the head of the reversed result

Multiple Choice Question

Let's make `reverse` tail-recursive.

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
    if (list == null) {
        return ???;
    }
    return ???;
}
```

What should we return in the base case?

(A) **`null`**

(B) `reversed`

(C) `list`

(D) None of the above

Multiple Choice Question

Let's make `reverse` tail-recursive.

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
    if (list == null) {
        return reversed;
    }
    return ???;
}
```

What has to be the return value here for this to be tail-recursive?

(A) A call to **this**.reverse

(B) A constant value that (no recursive calls)

(C) A computation performed upon the result of a **this**.reverse call

(D) A new object created from the result of a **this**.reverse call

Multiple Choice Question

Let's make `reverse` tail-recursive.

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
   if (list == null) {
      return reversed;
   }
   return this.reverse(???, ???);
}
```

What moves `list` closer to the base case?

(A) Doesn't matter; what matters is `reversed`

(B) `list`

(C) **`null`**

(D) `list.link`

Multiple Choice Question

Let's make `reverse` tail-recursive.

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
    if (list == null) {
        return reversed;
    }
    return this.reverse(list.link, ???);
}
```

What do we pass in as the second parameter (the new `reversed`)?

(A) `this.addToBack(list.data, reversed)`

(B) `new Node<E>(list.data, reversed)`

(C) `null`

(D) `reversed.link`

A Tail-Recursive reverse Method

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
    if (list == null) {
        return reversed;
    }
    return this.reverse(list.link,
                        new Node<E>(list.data, reversed));
}
```

A Closer Look At Recursion

Internally, recursive methods are handled by stacks of call frames (or activation records):

- Every time a method is invoked, we allocate space to store
  - The input parameters' values
  - The return address
  - The method's local variables
  - Potentially other addresses
- Upon allocating the frame, we push it to the call stack
- When we finish executing the method we
  - Restore certain portions of memory
  - Store the proper value in a specific spot (the return value)
  - Pop the activation record
  - Jump to the code at the frame's return address

A Closer Look At Tail Recursion

```
private Node<E> reverse(Node<E> list, Node<E> reversed) {
    if (list == null) {                                          0
        return reversed;                                         0
    }                                                            0
    return                                                       2
            this.reverse(list.link,                              1
                         new Node<E>(list.data, reversed));      1
}
```

Tail Recursion vs "Normal" Recursion

```
private int size(Node<E> list) {
    if (list == null) {              ?
        return 0;                    ?
    }                                ?
    return 1 +                       ?
            this.size(list.link);    ?
}
```

What "statement number" should we consider these lines to be?

(A) All 0

(B) All 1

(C) All 2

(D) We have several statements; we'll need several numbers

Tail Recursion vs “Normal” Recursion

```
private int size(Node<E> list) {
    if (list == null) {                                  0
        return 0;                                        0
    }                                                    0
    return 1 +                                           ?
            this.size(list.link);                        ?
}
```

What “statement number” should this line have?

(A) 0

(B) 1

(C) 2

(D) We have several statements; we’ll need several numbers

Tail Recursion vs "Normal" Recursion

```
private int size(Node<E> list) {
    if (list == null) {                         0
        return 0;                               0
    }                                           0
    return 1 +                                  2
           this.size(list.link);                ?
}
```

What "statement number" should this line have?

(A) 0

(B) 1

(C) 2

(D) We have several statements; we'll need several numbers

Tail Recursion vs "Normal" Recursion

```
private int size(Node<E> list) {
    if (list == null) {                        0
        return 0;                              0
    }                                          0
    return 1 +                                 2
            this.size(list.link);              1
}
```

returnValue = [ ]

(Worked out in class)

(A) Push a new call frame

(B) Change `returnValue`

(C) Advance top call frame to the next statement

(D) Pop the call stack

Tail Recursion vs "Normal" Recursion

```
private int size(Node<E> list, int total) {
    if (list == null) {                             0
        return total;                               0
    }                                               0
    return                                          2
            this.size(list.link, total + 1);        1
}
```

returnValue = [ ]

(Worked out in class)

(A) Push a new call frame

(B) Change `returnValue`

(C) Advance top call frame to the next statement

(D) Pop the call stack

Multiple Choice Question

So, what's the big deal about tail-recursion? Let's address this via a series of questions...

```
private int size(Node<E> list, int total) {
    if (list == null) {                          0
        return total;                            0
    }                                            0
    return                                       2
            this.size(list.link, total + 1);     1
}
```

When we returned from the base case in our tail-recursive `size`, how did the `returnValue` change?

(A) `returnValue = 0`

(B) `returnValue = callstack.peek().total`

(C) `returnValue = returnValue`

Multiple Choice Question

So, what's the big deal about tail-recursion? Let's address this via a series of questions. . .

```
private int size(Node<E> list, int total) {
    if (list == null) {                                    0
        return total;                                      0
    }                                                      0
    return                                                 2
            this.size(list.link, total + 1);               1
}
```

When we returned from the recursive case in our tail-recursive `size`, how did the `returnValue` change?

(A) `returnValue = 0`

(B) `returnValue = callstack.peek().total`

(C) `returnValue = returnValue`

Multiple Choice Question

So, what's the big deal about tail-recursion? Let's address this via a series of questions...

```
private int size(Node<E> list, int total) {
    if (list == null) {                            0
        return total;                              0
    }                                              0
    return                                         2
            this.size(list.link, total + 1);       1
}
```

Since we only ever need one "final" `returnValue`, was there a point to pushing all the call frames?

(A) Yes, to keep track of local variables
(B) Yes, to keep track of return values
(C) No, the call frames just take up space (pushed just to get popped)
(D) No, since the return value's not really tied to particular frames here

Tail-Call Optimization

Definition

A compiler optimization that transforms tail-calls into "jumps". Effectively, it can turn a tail-recursive function into a simple loop.

Example

```
private int size(Node<E> list, int total) {
    if (list == null) {
        return total;
    }
    return this.size(list.link, total + 1);
}
```

Tail-Call Optimization

Definition

A compiler optimization that transforms tail-calls into "jumps". Effectively, it can turn a tail-recursive function into a simple loop.

Example

```java
private int size(Node<E> list, int total) {
    while (!(list == null)) {
        list = list.link;
        total = total + 1;
    }
    return total;
}
```

Languages/Compilers That Support TCO

- Scheme (a Lisp dialect)
- Haskell
- Lua
- Standard ML
- OCaml
- Erlang
- Limited support in .NET (C#, F#, etc.)
- GCC with the proper `-O` flags
- ...

Don't Support TCO

- Python
- The Java Virtual Machine!

Multiple Choice Question

Suppose we have the following method:

```
public boolean method1(int n) {
    if (n == 0) {
        return true;
    }
    return method2(n - 1);
}
```

Is `method1` recursive?

(A) Yes

(B) No

(C) It depends

Multiple Choice Question

Suppose we have the following methods:

```
boolean method1(int n) {
    if (n == 0) {
        return true;
    }
    return method2(n - 1);
}
```

```
boolean method2(int n) {
    if (n == 0) {
        return false;
    }
    return method1(n - 1);
}
```

Is `method1` recursive?

(A) Yes

(B) No

(C) It depends

Multiple Choice Question

Suppose we have the following methods:

```
boolean method1(int n) {
   if (n == 0) {
      return true;
   }
   return method2(n - 1);
}
```

```
boolean method2(int n) {
   if (n == 0) {
      return false;
   }
   return method1(n - 1);
}
```

Is `method2` recursive?

(A) Yes

(B) No

(C) It depends

Mutual Recursion

Definition

An indirect form of recursion where, instead of calling themselves, two methods call each other.

So What's The Big Deal About Recursion?

- Some problems are naturally recursive...
  - Fractals
  - Maze generation
  - Towers of Hanoi
  - Binary search
  - Implanting an original idea in a dreamer's subconscious
- Facilitates equational reasoning (to some extent)
  - Just "plug & chug" definitions
  - Typically not much state
  - More amenable to formal proofs

When Is Recursion “Bad”?

Example

```
public int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    return this.fib(n - 1) + this.fib(n - 2);
}
```