

# *FUNCTION COMPOSITION*

Instructors: Crista Lopes  
Copyright © Instructors.

# Topics

---

- Recursion
- Higher-order functions
- Continuation-Passing Style
- Monads (take 1)
  - ▣ Identity Monad
  - ▣ Maybe Monad



# Recursion

# Prototypical Example



```
fact(n) :  
    if (n <= 1) then 1  
    else n * fact(n-1)
```

# Thinking Recursively

---

- Add numbers in a list
- Print a list of numbers
- Check if a number is in a list

# Tail Recursion

(first-order case)

slide 6

- Function g makes a **tail call** to function f if return value of function f is return value of g

- Example

tail call

not a tail call

$\text{fun } g(x) = \text{if } x > 0 \text{ then } f(x) \text{ else } f(x) * 2$

- Optimization: can pop current activation record on a tail call
  - ▣ Especially useful for recursive tail call because next activation record has exactly same form

# Example of Tail Recursion

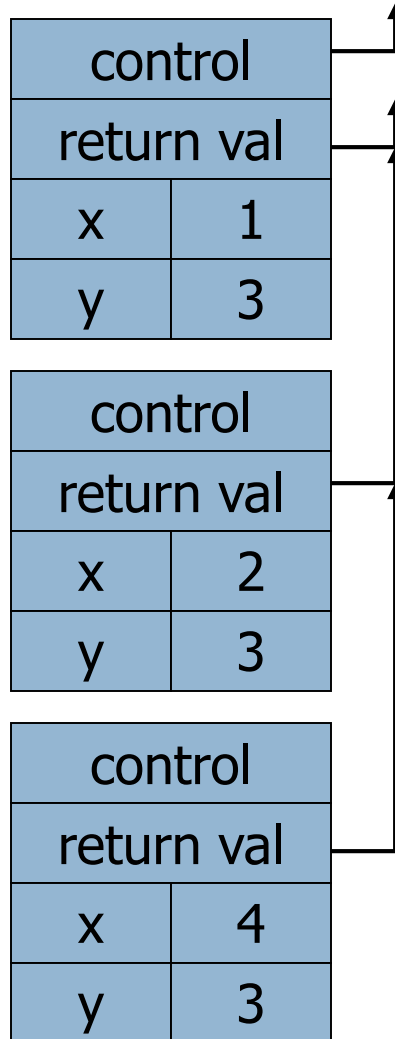
slide 7

Calculate least power of 2 greater than y

$f(1,3)$

control		↑
return val		
x	1	↑
y	3	

```
fun f(x,y) = if x>y  
  then x  
  else f(2*x, y);  
f(1,3) + 7;
```



# Tail Recursion Elimination

slide 8

$f(1,3)$

control		↑
return val		↑
x	1	
y	3	

$f(2,3)$

control		↑
return val		↑
x	2	
y	3	

$f(4,3)$

control		↑
return val		↑
x	4	
y	3	

```
fun f(x,y) = if x>y  
  then x  
  else f(2*x, y);  
f(1,3) + 7;
```

## Optimization

- pop followed by push - reuse activation record in place
- Tail recursive function is equivalent to iterative loop



# Tail Recursion and Iteration

slide 9

$f(1,3)$

control		↑
return val		↑
x	1	
y	3	

$f(2,3)$

control		↑
return val		↑
x	2	
y	3	

$f(4,3)$

control		↑
return val		↑
x	4	
y	3	

```
fun f(x,y) = if x > y  
  then x  
  else f(2*x, y);  
f(1,y);
```

test

loop body

initial value

```
function g(y) {  
  var x = 1;  
  while (!x > y)  
    x = 2*x;  
  return x;  
}
```



# Higher-order functions

# Higher-Order Functions

slide 11

- Function passed as argument
  - ▣ Need pointer to activation record “higher up” in stack
- Function returned as the result of function call
  - ▣ Need to keep activation record of the returning function
- Functions that take function(s) as input and return functions as output are known as functionals

# Return Function as Result

slide 12

- Language feature (e.g., Python, ML, ...)
- Functions that return “new” functions
  - ▣ Example: `fun compose(f,g) = (fn x => g(f x));`
  - ▣ Function is “created” dynamically
    - Expression with free variables; values determined at run-time
  - ▣ Function value is closure =  $\langle \text{env}, \text{code} \rangle$

# Named vs. Anonymous Functions

```
def add(x, y):  
    return x+y
```



This has a name: add

```
lambda (x, y): x+y
```



This does not have a name

The two functions do the same thing

# Named vs. Anonymous Functions

```
def add(x, y):  
    return x+y
```



```
add(3, 4)
```

```
lambda (x, y): x+y
```



?? no name !!



```
(lambda (x, y): x+y) (3, 4)
```

How to call them?

# Lambda: functions with no name

- Python

- ▣ `lambda (x, y) : x + y`

- Java

- ▣ `(int x, int y) -> x + y`

- C#

- ▣ `(int x, int y) => x + y`

- C++

- ▣ `[] (int x, int y) { return x + y; }`

# Assigning names to lambdas

## □ Python

- ▣ `f = lambda (x, y): x + y`

## □ Java

- ▣ `BiFunction<Integer,Integer,Integer> (int x, int y) -> x + y`

## □ C#

- ▣ `Func<int,int> f = (int x,int y) => x+y`

## □ C++

- ▣ `auto f=[] (int x,int y) {return x+y; }`



# Lambda: why?

---

- Sometimes a name is too much overhead!
- In OOP, sometimes we just need a function, and not a method

# Closures

slide 18

- Function value is pair **closure** =  $\langle \text{env}, \text{code} \rangle$ 
  - ▣ Statically scoped function must carry a link to its static environment with it
  - ▣ Only needed if function is defined in a nested block
- When a function represented by a closure is called...
  - ▣ Allocate activation record for call (as always)
  - ▣ Set the access link in the activation record using the environment pointer from the closure

# Closures

- Function with free variables that are bound to values in the enclosing environment

The diagram illustrates the concept of closures. It shows a lambda function and a Python function definition. A box highlights the inner lambda function, and an arrow points to it from the word 'closure'. Another arrow points from the word 'closure' to the inner function definition in the Python code.

```
(lambda (x)
  (lambda (y)
    x+y))
```

closure

```
def makeInc(x):
  def inc(y):
    # x is "closed" in the definition of inc
    return y + x
  return inc
```

# What are closures good for?

- For changing your mind later!
  - ▣ Replaces constants and variables with functions
  - ▣ Replaces conditionals
  - ▣ ...



# Continuations

# Continuations

- Representation of the control state of a program
  - ▣ Data structure available to the programmer instead of hidden
  - ▣ Contains the current stack and point in the computation
- Can be later used to return to that point

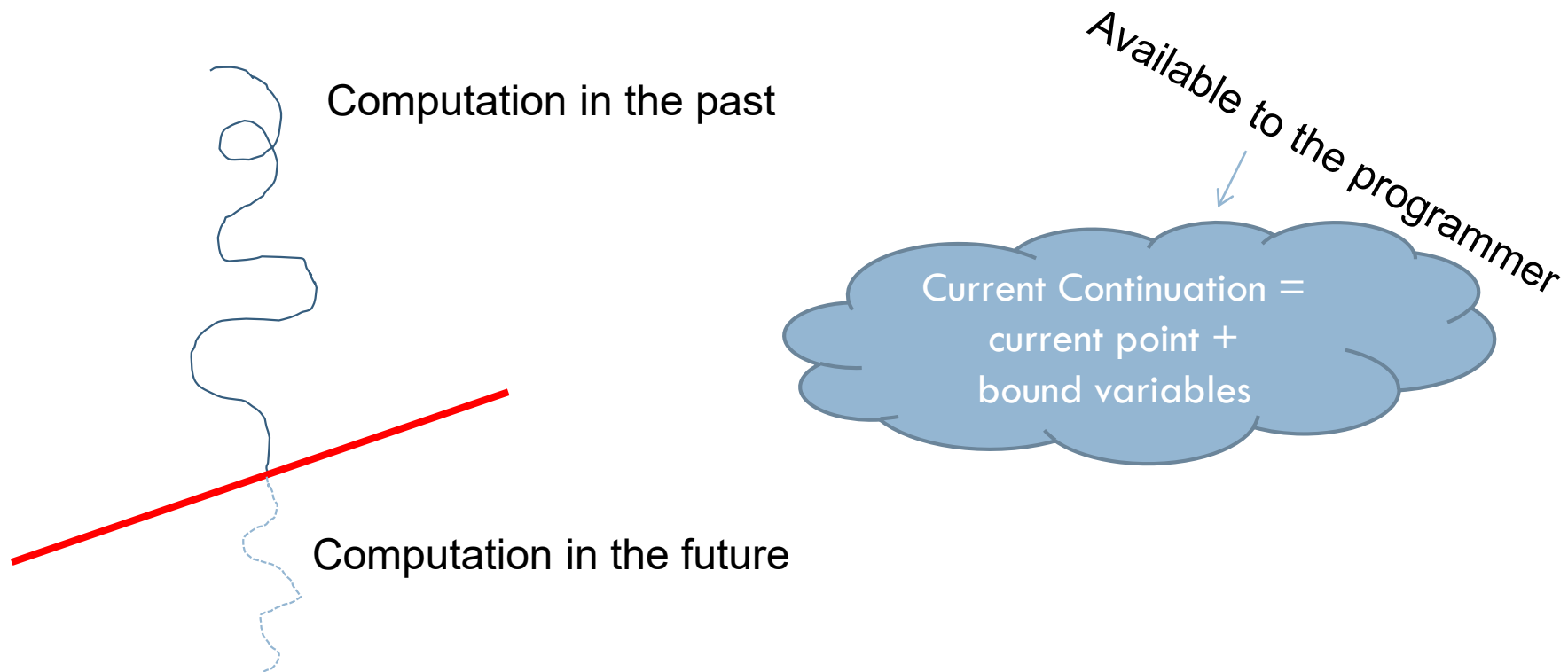
# Remember Goto

```
A: blah  
    blah  
    if something GOTO A else GOTO B  
B: ...
```

Flow control via textual labels mixes computation (beta-reductions) and representation (the text of the program)

# Continuations continued

- Elegant concept for arbitrary flow control



*Note to self: illustrate continuations in Scheme (Wikipedia)*



# What are continuations good for?

- Everything control flow!
  - ▣ Co-routines
  - ▣ Exceptions
  - ▣ Preserving flow  
in non-blocking I/O
    - (rhymes!)

# The continuation nature of exceptions

```
function fact (n) {  
  if (n < 0)  
    throw "n < 0" ;  
  else if (n == 0)  
    return 1 ;  
  else  
    return n * fact(n-1) ;  
}
```

```
function total_fact (n) {  
  try {  
    return fact(n) ;  
  } catch (ex) {  
    return false ;  
  }  
}
```

```
document.write("total_fact(10): " + total_fact(10)) ;  
document.write("total_fact(-1): " + total_fact(-1)) ;
```

# The continuation nature of exceptions – desugaring the previous slide

```
function fact (n, r, t) {  
  if (n < 0)  
    t ("n < 0")  
  else if (n == 0)  
    r(1)  
  else  
    fact(n-1,  
        function (t0) {  
          r (n*t0) ;  
        },  
        t)  
}
```

```
function total_fact (n, ret) {  
  fact (n, ret,  
    function (ex) {  
      ret(false) ;  
    }) ;  
}
```

```
total_fact(10, function (res) {  
  document.write("total_fact(10): " + res)  
}) ;  
  
total_fact(-1, function (res) {  
  document.write("total_fact(-1): " + res)  
}) ;
```

# I/O and continuations

## Blocking (I/O in most systems)

```
contents = fs.ReadFile(path);  
with contents do  
    blah
```

*Blocks here until  
we have the result*

## Non-blocking

```
contents = fs.ReadFileAsync(path);  
with contents do  
    blah
```

*Uh-oh, we still don't  
have it*

How to solve this?

# I/O and continuations

## Non-blocking

```
fs.readFileAsync(path, lambda(contents)
{
    with contents do
        blah
});
```

It's a callback!

**It's the “current continuation” of the blocking form**

JavaScript is FULL of this, so are jquery and node.js



# Monads

# Monads – what is the problem?

- The problem: how to affect the world
- Problem is more prevalent in pure functional programming style
  - ▣ No side-effects
  - ▣ That's right: no side-effects!
- But you've all seen it too!

# No side effects?! Why?

- Easier to test: idempotent functions
- Easier to parallelize
- But the world is ALL about side-effects, right?
  - ▣ Storage, network, UI, ...
  - ▣ Programs affect and control objects and activities in the real world



# Example – a Tracing monad

```
def hypotenuse(x, y):  
    return math.sqrt(math.pow(x, 2) + math.pow(y, 2))
```

Now we want to trace it, or affect the world in it:

```
def hypotenuse(x, y):  
    h = math.sqrt(math.pow(x, 2) + math.pow(y, 2))  
    print "In hypotenuse " + h  
    return h
```



# Example – a Tracing monad

```
def hypotenuse(x, y):  
    h = math.sqrt(math.pow(x, 2) + math.pow(y, 2))  
    return h, "In hypotenuse" + h
```

Signature was                      float, float -> float

Signature now is                  float, float -> float, string

```
> math.pow(hypotenuse(6, 16), 4);
```



# What is a monad?

- It's a container
- An active container... it has behavior to:
  - ▣ Wrap itself around a [typed] value
  - ▣ Bind *functions* together

# What is a monad?

- [A type constructor,  $m$ ]
- A function that builds values of that type  
 $a \rightarrow m\ a$  (what you'd normally call a constructor in OOP)
- A function (bind) that combines values [of that type] with computations that produce values [of that type]  
 $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
- An unwrap function that shows “what's inside”