INF 102
CONCEPTS OF PROG. LANGS
FUNCTIONAL COMPOSITION

Instructors: James Jones

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)

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

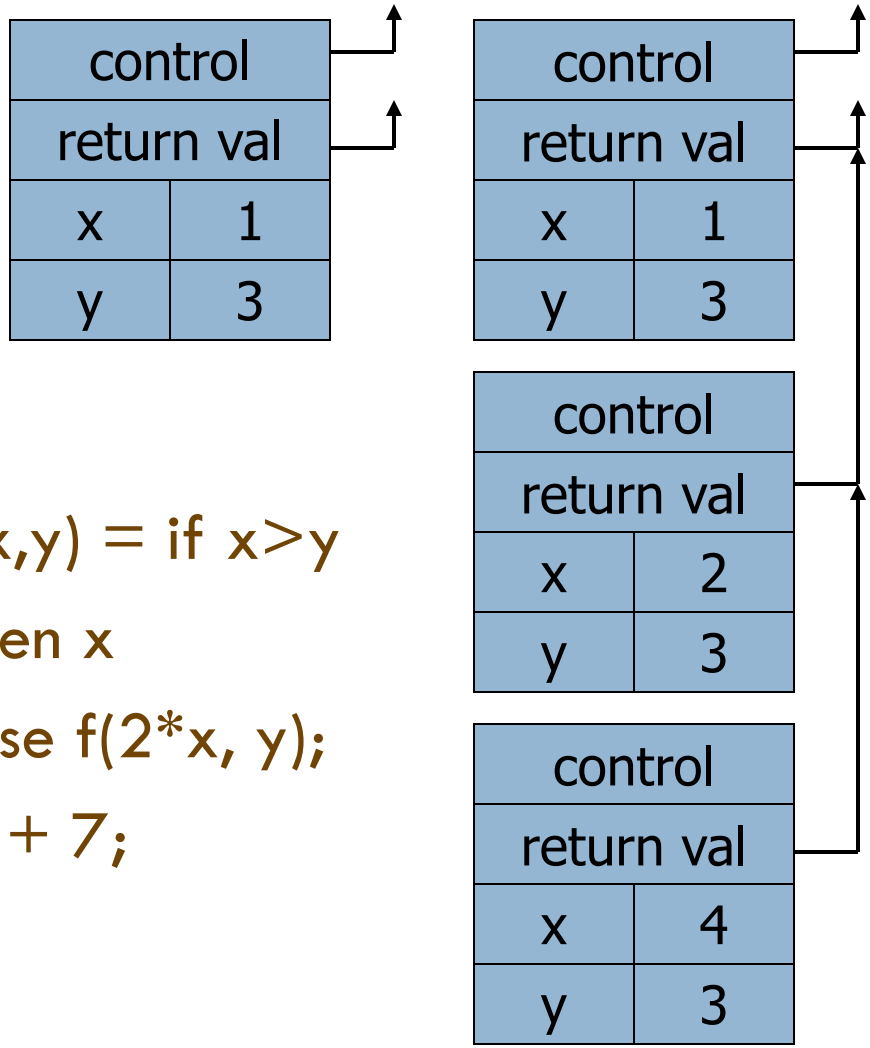

```
fun g(x) = if x>0 then f(x) else f(x)*2
```

- 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

Calculate least power of 2 greater than y

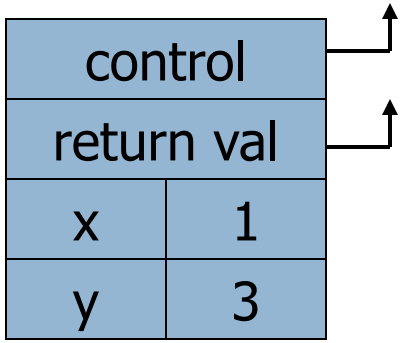
f(1,3)



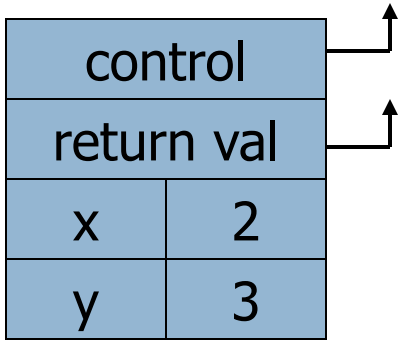

```
fun f(x,y) = if x>y
      then x
      else f(2*x, y);
f(1,3) + 7;
```

Tail Recursion Elimination

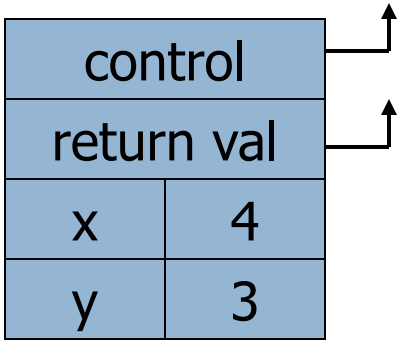
f(1,3)



f(2,3)



f(4,3)




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

Optimization

- Tail recursive function is equivalent to iterative loop

Tail Recursion and Iteration

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);
```

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

test

loop body

initial value

Higher-order functions

Higher-Order Functions

- Function passed as argument
- Function returned as the result of function call
- Functions that take function(s) as input and return functions as output: these are known as functionals

Return Function as Result

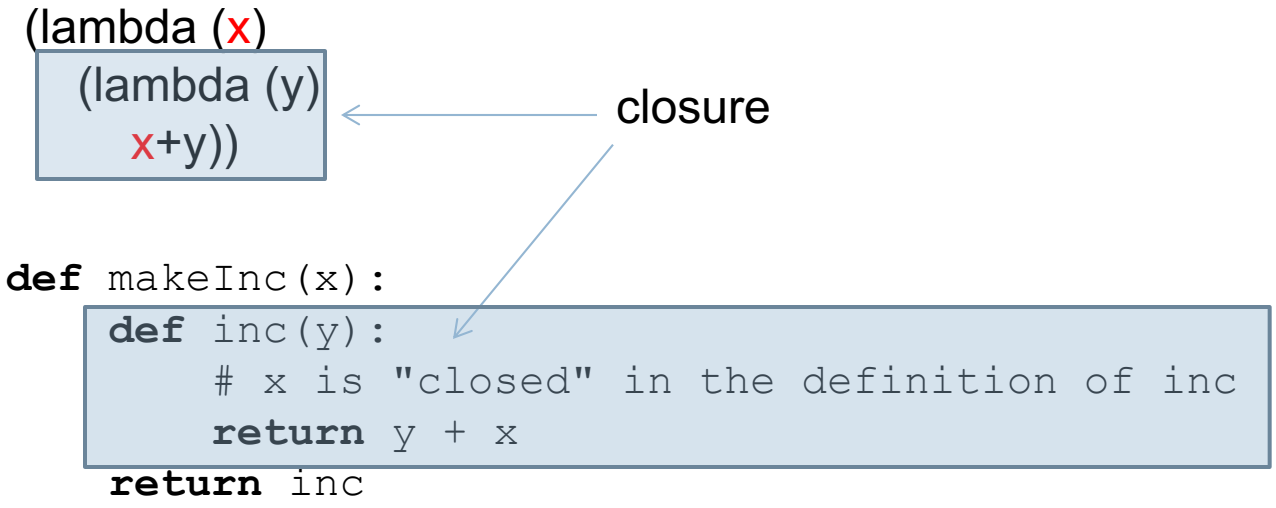
- Language feature (e.g., Python, ML, ...)
- Functions that return "new" functions
  - Example: `fun compose(f,g) = (fn x => g(f x));`
  - Function is "created" dynamically
    - Expression with free variables; values determined at run-time
  - Function value is closure = ⟨env, code⟩
  - Need to maintain environment of the creating function

Closures

- Function value is pair closure = ⟨env, code ⟩
  - 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



What are closures good for?

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

Implementing Closures

- Closures as used to maintain static environment of functions as they are passed around
- May need to keep activation records after function returns
- Possible "stack" implementation:
  - Put activation records on heap
  - Instead of explicit deallocation, invoke garbage collector as needed

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: ...
```

What are continuations good for?

- Co-routines
- Exceptions
- Preserving flow
  in non-blocking I/O

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)) ;
```

Acts as a continuation

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!

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 -> 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 -> (a -> m b) -> m b
- An unwrap function that shows "what's inside"