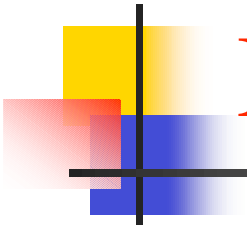
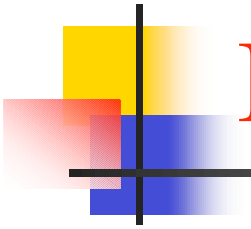
Lambda the Ultimate

Corky Cartwright
Department of Computer Science
Rice University



Motivation for λ-notation

- Often, functions are used only once
- Examples: arguments to functions like
  - `map`,
  - `filter`,
  - `fold`, and many more "higher-order" functions
- Sometimes we want to build new functions in the middle of a computation.
- Local suffices but it is notationally clumsy for this purpose.
- λ provides simpler, more concise notation



Basic Idea

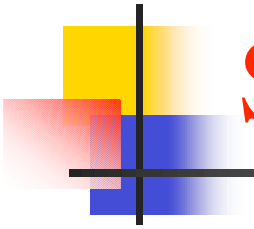
- λ-notation was invented by mathematicians. For example, given
  f (x) = $x^2 + 1$
  what is f? f is the function that maps x to $x^2 + 1$ which we might write as
  $x \rightarrow x^2 + 1$
  The latter avoids naming the function. The notation
  λ x . $x^2 + 1$ evolved instead of $x \rightarrow x^2 + 1$
- In Scheme, we write `(lambda (x) (+ (* x x) 1))` instead of λ x . $x^2 + 1$.
- `(define (f x) (+ (* x x) 1))` abbreviates
  `(define f (lambda (x) (+ (* x x) 1)))`

Why λ?

- The name was used by its inventor
  - Alonzo Church, logician, 1903-1995.
  - Princeton, NJ
  - Introduced lambda in 1930's to formalize math

Church is my academic great-grandfather

Alonzo Church -> Hartley Rogers ->
David Luckham -> Corky Cartwright



Scope for a Lambda Abstraction

- Argument scope:
  `(lambda (`x_1 `...` x_n`)` *body*`)` introduces the variables x_1 ... x_n which have *body* as their scope (except for holes)
- Example:

```
(lambda (x) (+ (* x x) 1)))
```

- Scope for variable introduced by `define`. At the top-level,

```
(define f rhs)
```

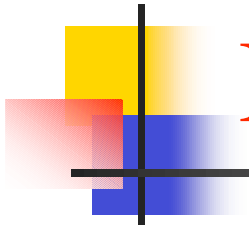
  introduces the variable `f` which is visible everywhere (except inside holes introduced by local definitions of `f`). Inside

```
(local [(define f1 rhs1) ... (define fn rhsn)) body)
```

  the variables f_1 ... f_n have *body* as their scope.
- Recursion comes from `define` not `lambda`!

Many PL researchers are crazy about λ!

Prof.
Phil Wadler
at
CWI,
Amsterdam,
Holland



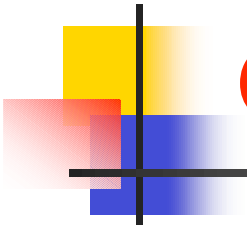
Example

Now we can write the following program concisely

```
(define l '(1 2 3 4 5))
(define a
  (local ((define (square x)
            (* x x)))
    (map square l)))
```

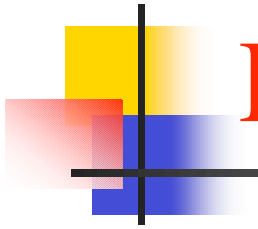
as

```
(define l '(1 2 3 4 5))
(define a (map (lambda (x) (* x x)) l))
```



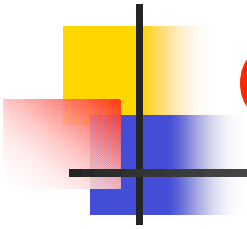
Careful Definition of Syntax

- Official specification of what expressions that use lambda can look like:
  - *exp* = ... | `(lambda` (*var**) *exp*`)`
- Interesting points
  - Can have multiple arguments
  - Can have no arguments
- Application of a function with no arguments
  - `(define blowup (lambda () (/ 1 0)))`
    `(blowup)`



Functions with Zero Arguments?

- We don't see them in math
  - A function with zero arguments would always produce the same result (so, it's just a constant)
- In computing, we see them for several reasons:
  - Encapsulate potential error or divergence.
  - Once we introduce side-effects (destructive modification of data), procedures (the analogs of functions in the world of side effects) of no arguments are common.



Careful Analysis of Analogy

- Recall that:
  `(lambda (x1 ... xn) exp)`
  abbreviates
  ```
  (local ((define (f x1 ... xn) exp))
           f)
  ```
- Is `lambda` as general as `local`? No!
  How do I introduce a recursive function definition using `lambda`?
- It can be done but it involves very deep and subtle use of λ-notation, which is covered in Comp 311.
- You need a name to recur, which `lambda` lacks.

Evaluation of λ-expressions

How do we evaluate a λ-expression

```
(lambda (x1 ... xn) body)
```

It's a value!

What about λ-applications?

```
((lambda (x1 ... xn) body) v1 ... vn)
=> body[x1←v1 ... xn←vn]
```

(called β-reduction)

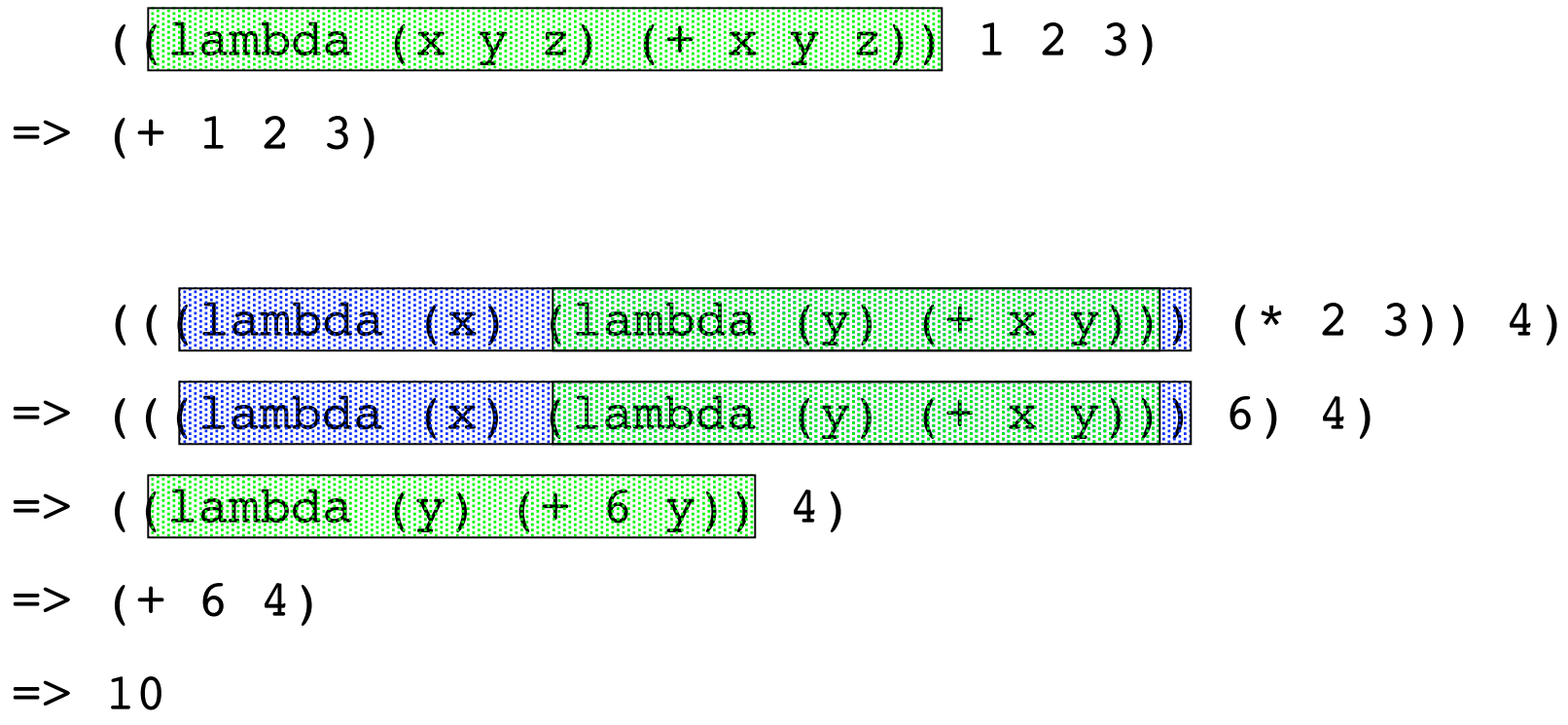
Examples:

```
((lambda (x) (* x 5)) 4) => (* 4 5) => 20

((lambda (x) (x x)) (lambda (x) (x x)))
=> ((lambda (x) (x x)) (lambda (x) (x x)))
=> ...
```

(cool?)

More Examples



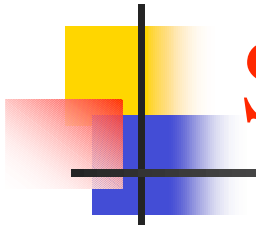
Nesting λ

```
    (lambda (x) (lambda (y) (+ (* x y) (* 4 5)))
=> (lambda (x) (lambda (y) (+ (* x y) (* 4 5)))

    ((lambda (x) (lambda (y) (+ x 1))  5)
=> (lambda (y) (+ 5 1))

    ((lambda (x) (lambda (x) (+ x 1))  5)
=> (lambda (x) (+ x 1))

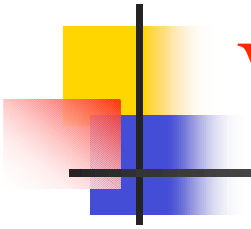
    ((lambda (x) (lambda (y) (y x)))   (lambda (z) (+ y z)))
=> (lambda (y) (y (lambda (z) (+ y z)))))          WRONG!
```



Safe Substitution

- Must rename local variables in the code body that is being modified by the substitution to avoid capturing free variables in the argument expression that is being substituted.

```
   ((lambda (x) (lambda (y) (y x)))   (lambda (z) (+ y z)))
=> ((lambda (x) (lambda (f) (f x)))   (lambda (z) (+ y z)))
=> (lambda (f) (f (lambda (z) (+ y z))))
```



When Should I Use a Lambda?

- It makes sense to use a lambda instead define when
  - the function is not recursive;
  - the function is needed only once; and
  - the function is either
    - being passed to another function, or
    - being returned as the final result (contract returns "->")
- Note: It is hard to read code when lambda is used at the head of an application
  - `((lambda (x) (* x x)) (+ 13 14))`
- We can rewrite this as:
  - ```
    (local ((define x (+ 13 14)))
      (* x x))
    ```
    ```

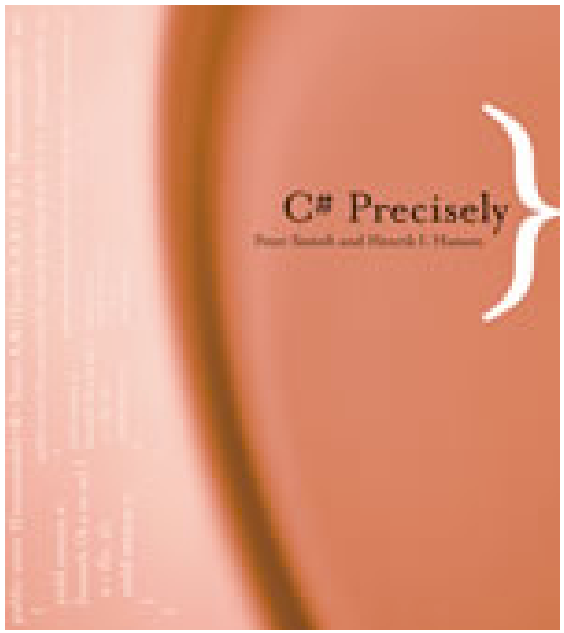
Lambda in Becoming Pervasive in PL

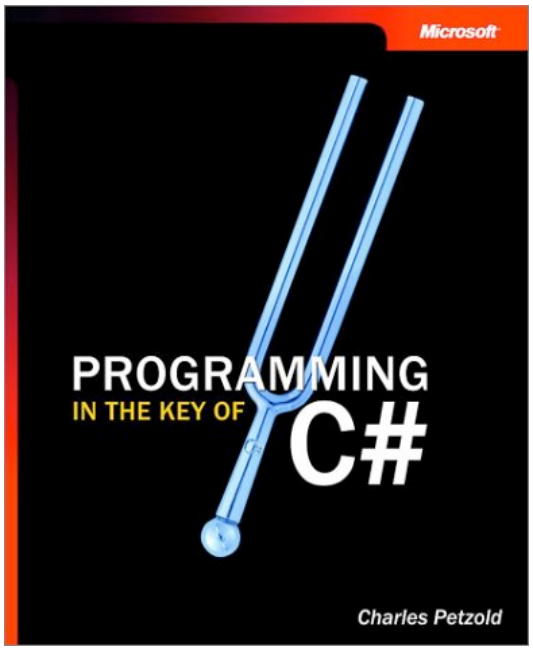
Python

"By popular demand, a few features commonly found in functional programming languages and Lisp have been added to Python [...]"

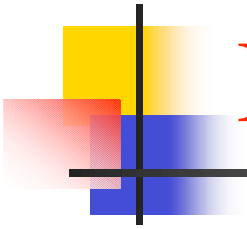
Guido van Rossum,
4.7.4 Lambda Forms,
Python Tutorial

Lambda in C#





"anonymous methods"



For Next Class

- Homework due Monday
- Continue Reading:
  - Ch 21-22: Abstracting designs and first class functions