Lambda the Ultimate

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University

Function filter2: variant of filter1 function from last lecture

```
;; filter2 : test lon ->  lon
;; to construct a list of those numbers n
;; in alon such that (test n) is true
(define (filter2 test alon)
  (cond [(empty? alon) empty]
        [else
          (cond [(test (first alon))
                 (cons (first alon)
                       (filter2 test (rest alon)))]
                [else (filter2 test (rest alon) t)])]))
```

Function filter2 takes function test as a “data” input

How can we pass different test functions as data for filter2?

1) By introducing a top-level definition

```
(define (Positive? n) (> n 0))
(check-expect
  (filter2 Positive? '(-1 2 0 3))
  '(2 3))
```

2) By using a local expression to avoid cluttering top-level definitions --- how?

How can we pass different test functions as data for filter2?

2) By using a local expression to avoid cluttering top-level definitions …

```
(check-expect
 (filter2
    (local ((define (Positive? n) (> n 0)))
           Positive?)
    '(-1 2 0 3))
 '(2 3))
```

… at the expense of cluttered local expressions

Motivation for λ-notation

- It is sometimes convenient to build new functions in the middle of a computation without having to name them
  - Often, these functions are used only once
  - There may even be an unbounded number of such functions created at run-time
- Examples: arguments to functions like
  - **`map`**,
  - **`filter`**,
  - **`fold`**,
  - and many more "higher-order" functions

λ-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 $\lambda x\,.\,x^2 + 1$ evolved instead of $\mathrm{x} \rightarrow \mathrm{x}^2 + 1$

- In Scheme, we write **`(lambda (x) (+ (* x x) 1))`** instead of $\lambda x\,.\,x^2 + 1$

How can we pass different test functions as data for filter2?

3) By using lambda expressions …

```
(check-expect
 (filter2
     (lambda (n) (> n 0))
     '(-1 2 0 3))
 '(2 3))
```

… note that lambda expressions are anonymous!

Lambda Expression

- A *lambda expression* consists of
  - The keyword lambda
  - A list of variable names denoting arguments
  - An expression denoting a function of the arguments
- (**lambda** (var_1 var_2 … var_n) *exp)*
  - *exp* is an arbitrary expression
  - var_i is a variable (argument) that is only available for use within *exp*
- A lambda-expression is just a value with a type that happens to be a function

Function definitions revisited

- A function definition basically defines a function value for a given name, just like any other variable definition defines a value for a name
- `(define (f x) (+ (* x x) 1))` is just short hand for `(define f (lambda (x) (+ (* x x) 1)))`

- Question: when is the divide-by-zero error encountered in the following top-level definitions?
- `(define x (/ 1 0))`
- `(define f (lambda (n) (/ n 0)))`

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 Corky's 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 in holes introduced by local definitions in *`f`*).
- Inside
  `(local [(define` f_1 rhs_1`) ... (define` f_n rhs_n`))` *`body`*`)`

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

Example

Now we can write the following program

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

concisely as

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

`lambda` vs. `local`

- Recall that:

```
(lambda (x1 ... xn) exp)
```

is equivalent to

```
(local ((define (f x1 ... xn) exp)) f)
```

- Is **`lambda`** as general as **`local`**? No!
  How do I introduce a recursive function definition using **`lambda`** alone?
  - It can be done but it involves deep, subtle, and messy use of λ-notation (topic in Comp 311).
  - Direct formulations of recursion rely on the name of the defined function, which **`lambda`** lacks.

Evaluation of λ-expressions

How do we evaluate a λ-expression

(lambda (*x_1* . . . *x_n*) *body*) ?

It's a value --- no further reduction can be performed!

What about λ-applications?

β-reduction rule

((lambda (*x_1* … *x_n*) *body*) V_1 … V_n) => *body*[*x_1*:=V_1 … x_n:=V_n]

where V_1,...,V_n are values and ***body*[*x_1*:=V_1 . . . x_n:=V_n]** means ***body*** with *x_1* replaced by V_1, . . . , x_n replaced by V_n.

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)))
 => ...  ???
```

More Examples

```
    ((lambda (x y z) (+ x y z)) 1 2 3)
=>  (+ 1 2 3)

    (((lambda (x) (lambda (y) (+ x y))) (* 2 3)) 4)
=>  (((lambda (x) (lambda (y) (+ x y))) 6) 4)
=>  ((lambda (y) (+ 6 y)) 4)
=>  (+ 6 4)
=>  10
```

Fine Points of Substitution

- Only the *free* occurrences of a variable are replaced. A variable occurrence **`v`** in an expression **`E`** is *free* iff it does not refer to a variable bound in **`E`**. A *non-free* (*bound*) variable occurrence **`v`** in expression **`E`** must be embedded in a **`local`** scope (defined by a **`lambda`** or a **`local`**) within **`E`**.
- Examples:
  - Neither occurrence of **`x`** is free in **`(lambda (x) x)`**
  - Neither occurrence of **`x`** is free in **`(local [(define x 12)] x)`**
  - **`x`** is free in **`(+ y x)`**
  - **`x`** is free in **`(lambda (y) (+ y x))`**
  - Only the first occurrence of **`x`** is free in
    `((+ x (local [(define x 12)] (* x 13))`

Example: Lambda expression as a return value

```
(define (gen-add-by n)
        (lambda (x) (+ x n)) )
```

What do the following expressions evaluate to?

```
(gen-add-by 1)
((gen-add-by 1) 1)
((gen-add-by 2) ((gen-add-by 1) 1))
```

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))

;; NOTE: the red y is a free variable
    ((lambda (x) (lambda (y) (y x))) (lambda (z) (+ y z)))
=>  (lambda (y) (y (lambda (z) (+ y z)))))
```

which is WRONG (but this case can't arise in legal Scheme evaluations)

Safe Substitution

To salvage the correctness of β-reduction in the general case, we must stipulate that the rule uses safe substitution, where safe substitution renames 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))))
```

We will not hold you responsible on exams for understanding either safe substitution or the subtleties of β-reduction when the argument expressions contain free variables.

When Should I Use a Lambda?

- It makes sense to use a lambda instead of 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 is 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

Examples of lambda in Python

```
identityFunc = lambda x: x
print identityFunc(2) # prints 2

sumFunc = lambda a, b: a + b
print sumFunc(2, 3) # prints 5

squareFunc = lambda a: a * a
cubeFunc = lambda a: a * a * a
rootFunc = lambda a: a ** 0.5

compose = lambda f, g: lambda x: f(g(x))
print compose(squareFunc, rootFunc)(4.0) # prints 4.0
print compose(rootFunc, cubeFunc)(4.0)   # prints 8.0
```

Examples of lambda in F# (anonymous functions)

```
(fun n -> n * 2 ) // anonymous function (a lambda expr)
```

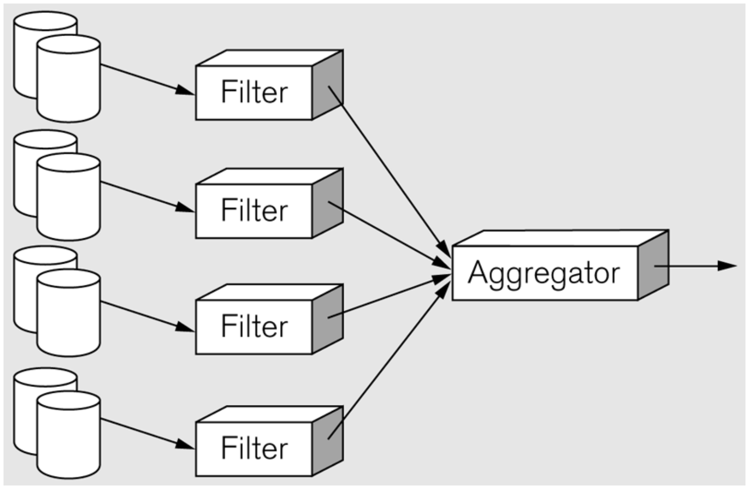
```
List.map (fun n -> n * 2) [1;2;3] // returns [2;4;6]
```

"Lambda expressions are especially useful when you want to perform operations on a list or other collection and want to avoid the extra work of defining a function. Many F# library functions take function values as arguments, and it can be especially convenient to use a lambda expression in those cases."

Source: http://msdn.microsoft.com/en-us/library/dd233201(VS.100).aspx

Use of Functions as Data in Google's Map-Reduce Framework

- Application of functional programming concepts to data-center-scale distributed systems
- Filter = computation specified by map function
- Aggregator = computation specified by reduce (fold) function



Reference: "MapReduce: Simplified Data Processing on Large Clusters", J. Dean & S. Ghemawat, OSDI 2004. http://labs.google.com/papers/mapreduce.html

Source for figure: Figure 10.6 from "Principles of Parallel Programming" by C.Lin & L. Snyder