Informatics 1

Functional Programming Lecture 8

Tuesday 22 October 2013

# Lambda expressions, functions and binding

Don Sannella

University of Edinburgh

# Tutorials—revision tutorials

Every Monday 1–2pm and Wednesday 2–3pm
Appleton Tower, Computer Lab West (5.05)

Attempt the revision tutorial exercise *in advance*.
*Print out* and bring your solutions.

# Part I

# Lambda expressions

# A failed attempt to simplify

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map sqr (filter pos xs))
  where
  sqr x  =  x * x
  pos x  =  x > 0
```

The above *cannot* be simplified to the following:

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (x * x) (filter (x > 0) xs))
```

# A successful attempt to simplify

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map sqr (filter pos xs))
  where
  sqr x  =  x * x
  pos x  =  x > 0
```

The above *can* be simplified to the following:

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
           (map (\x -> x * x)
              (filter (\x -> x > 0) xs))
```

# Lambda calculus

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
            (map (\x -> x * x)
               (filter (\x -> x > 0) xs))
```

The character `\` stands for $\lambda$, the Greek letter *lambda*.

Logicians write

`\x -> x > 0` as $\lambda x.\, x > 0$

`\x -> x * x` as $\lambda x.\, x \times x$.

Lambda calculus is due to the logician *Alonzo Church* (1903–1995).

# Evaluating lambda expressions

```
    (\x -> x > 0) 3
=
    let x = 3 in x > 0
=
    3 > 0
=
    True

   (\x -> x * x) 3
=
   let x = 3 in x * x
=
   3 * 3
=
   9
```

# Lambda expressions and currying

```
    (\x -> \y -> x + y) 3 4
=
    ((\x -> (\y -> x + y)) 3) 4
=
    (let x = 3 in \y -> x + y) 4
=
    (\y -> 3 + y) 4
=
    let y = 4 in 3 + y
=
    3 + 4
=
    7
```

# Evaluating lambda expressions

The general rule for evaluating lambda expressions is

$$(\lambda x.\, N)\; M$$

$$=$$

$$(\texttt{let}\; x = M \;\texttt{in}\; N)$$

This is sometimes called the $\beta$ rule (or beta rule).

# Part II

# Sections

# Sections

`(> 0)` is shorthand for `(\x -> x > 0)`

`(2 *)` is shorthand for `(\x -> 2 * x)`

`(+ 1)` is shorthand for `(\x -> x + 1)`

`(2 ^)` is shorthand for `(\x -> 2 ^ x)`

`(^ 2)` is shorthand for `(\x -> x ^ 2)`

# Sections

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
           (map (\x -> x * x)
             (filter (\x -> x > 0) xs))

f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))
```

# Part III

# Composition

# Composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)
```

# Evaluating composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)

sqr :: Int -> Int
sqr x  =  x * x

pos :: Int -> Bool
pos x  =  x > 0

  (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

# Compare and contrast

```
possqr :: Int -> Bool
possqr x  =  pos (sqr x)

  possqr 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

```
possqr :: Int -> Bool
possqr =  pos . sqr

  possqr 3
=
  (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

# Composition is associative

```
(f . g) . h = f . (g . h)
```

```
    ((f . g) . h) x
=
    (f . g) (h x)
=
    f (g (h x))
=
    f ((g . h) x)
=
    (f . (g . h)) x
```

# Thinking functionally

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))

f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)
```

# Applying the function

```
f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)

   f [1, -2, 3]
=
   (foldr (+) 0 . map (^ 2) . filter (> 0)) [1, -2, 3]
=
   foldr (+) 0 (map (^ 2) (filter (> 0) [1, -2, 3]))
=
   foldr (+) 0 (map (^ 2) [1, 3])
=
   foldr (+) 0 [1, 9]
=
   10
```

# Part IV

# Variables and binding

# Variables

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

# Part V

# Lambda expressions explain binding

# Lambda expressions explain binding

A variable binding can be rewritten using a lambda expression and an application:

$$
\begin{aligned}
& (N \;\texttt{where}\; x = M) \\
= \;& \\
& (\lambda x.\, N)\; M \\
= \;& \\
& (\texttt{let}\; x = M \;\texttt{in}\; N)
\end{aligned}
$$

A function binding can be written using an application on the left or a lambda expression on the right:

$$
\begin{aligned}
& (M \;\texttt{where}\; f\, x = N) \\
= \;& \\
& (M \;\texttt{where}\; f = \lambda x.\, N)
\end{aligned}
$$

# Lambda expressions and binding constructs

```
    f 2
    where
    f x  =  x+y*y
         where
         y = x+1
=
    f 2
    where
    f  =  \x -> (x+y*y where y = x+1)
=
    f 2
    where
    f  =  \x -> ((\y -> x+y*y) (x+1))
=
    (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

# Evaluating lambda expressions

```
     (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
=
     (\x -> ((\y -> x+y*y) (x+1))) 2
=
     (\y -> 2+y*y) (2+1)
=
     (\y -> 2+y*y) 3
=
     2+3*3
=
     11
```