Informatics 1

Functional Programming Lecture 5

Tuesday 23 September 2014

More fun with recursion

Don Sannella

University of Edinburgh

Tutorials

Attendance is compulsory.

Tuesday/Wednesday	Computation and Logic
Thursday/Friday	*Functional Programming*

You *must* do each week's tutorial exercise! Do it *before* the tutorial!

Bring a *printout* of your work to the tutorial!

You may *collaborate*, but you are responsible for knowing the material.

Mark of 0% on tutorial exercises means you have no incentive to *plagiarize*.

But *you will fail the exam if you don't do the tutorial exercises!*

Start work on the tutorial as *early* as possible.

Required text and reading

Haskell: The Craft of Functional Programming (Third Edition),
Simon Thompson, Addison-Wesley, 2011.

or

Learn You a Haskell for Great Good!
Miran Lipovača, No Starch Press, 2011.

Reading assignment

Monday 15 September 2014	Thompson: parts of Chap. 1–3 and 5
	Lipovača: parts of intro, Chap. 1–2
Monday 22 September 2014	Thompson: parts of Chap. 3–7
	Lipovača: parts of Chap. 1, 3–4

The assigned reading covers the material very well with plenty of examples.

There will be no lecture notes, just the books. *Get one of them and read it!*

Part I

Booleans and characters

Boolean operators

```
not :: Bool -> Bool
(&&), (||) :: Bool -> Bool -> Bool

not False  =  True
not True   =  False

False && False  =  False
False && True   =  False
True  && False  =  False
True  && True   =  True

False || False  =  False
False || True   =  True
True  || False  =  True
True  || True   =  True
```

Defining operations on characters

```
isLower :: Char -> Bool
isLower x  =  'a' <= x && x <= 'z'

isUpper :: Char -> Bool
isUpper x  =  'A' <= x && x <= 'Z'

isDigit :: Char -> Bool
isDigit x  =  '0' <= x && x <= '9'

isAlpha :: Char -> Bool
isAlpha x  =  isLower x || isUpper x
```

Defining operations on characters

```
digitToInt :: Char -> Int
digitToInt c | isDigit c  =  ord c - ord '0'

intToDigit :: Int -> Char
intToDigit d | 0 <= d && d <= 9  =  chr (ord '0' + d)

toLower :: Char -> Char
toLower c | isUpper c  =  chr (ord c - ord 'A' + ord 'a')
          | otherwise  =  c

toUpper :: Char -> Char
toUpper c | isLower c  =  chr (ord c - ord 'a' + ord 'A')
          | otherwise  =  c
```

These rely on the conversion functions:

```
ord :: Char -> Int     -- same as: fromEnum :: Char -> Int
chr :: Int -> Char     -- same as: toEnum :: Int -> Char
```

Part II

Conditionals and Associativity

Conditional equations

```
max :: Int -> Int -> Int
max x y | x >= y    =  x
        | y >= x    =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z   =  x
           | y >= x && y >= z   =  y
           | z >= x && z >= y   =  z
```

Conditional equations with otherwise

```
max :: Int -> Int -> Int
max x y | x >= y      =  x
        | otherwise   =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | otherwise         =  z
```

Conditional equations with otherwise

```
max :: Int -> Int -> Int
max x y | x >= y      =  x
        | otherwise   =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | otherwise         =  z

otherwise :: Bool
otherwise =  True
```

Conditional expressions

```
max :: Int -> Int -> Int
max x y  =  if x >= y then x else y

max3 :: Int -> Int -> Int -> Int
max3 x y z  =  if x >= y && x >= z then x
               else if y >= x && y >= z then y
               else z
```

Another way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  if x >= y then
                  if x >= z then x else z
               else
                  if y >= z then y else z
```

Key points about conditionals

- As always: write your program in a form that is easy to read. Don't worry (yet) about efficiency: premature optimization is the root of much evil.
- Conditionals are your friend: without them, programs could do very little that is interesting.
- Conditionals are your enemy: each conditional doubles the number of test cases you must consider. A program with five two-way conditionals requires $2^5 = 32$ test cases to try every path through the program. A program with ten two-way conditionals requires $2^{10} = 1024$ test cases.

A better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  max (max x y) z
```

An even better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  x `max` y `max` z

max :: Int -> Int -> Int
max x y | x >= y      =  x
        | otherwise   =  y
```

An even better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  x `max` y `max` z

max :: Int -> Int -> Int
x `max` y | x >= y      =  x
          | otherwise   =  y
```

`x + y` *stands for* `(+) x y`

`x >= y` *stands for* `(>=) x y`

`` x `max` y `` *stands for* `max x y`

Associativity

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
  (x `max` y) `max` z  ==  x `max` (y `max` z)
```

It doesn’t matter where the parentheses go with an associative operator, so we often omit them.

Associativity

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
  (x `max` y) `max` z  ==  x `max` (y `max` z)
```

It doesn't matter where the parentheses go with an associative operator, so we often omit them.

Why we use infix notation

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
  max (max x y) z  ==  max x (max y z)
```

This is much harder to read than infix notation!

Key points about associativity

- There are a few key properties about operators: *associativity*, *identity*, *commutativity*, *distributivity*, *zero*, *idempotence*. You should know and understand these properties.
- When you meet a new operator, the first question you should ask is “Is it associative?” The second is “Does it have an identity?”
- Associativity is our friend, because it means we don’t need to worry about parentheses. The program is easier to read.
- Associativity is our friend, because it is key to writing programs that run twice as fast on dual-core machines, and a thousand times as fast on machines with a thousand cores. We will study this later in the course.

Part III

Append

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

  "abc" ++ "de"
=
  ('a' : ('b' : ('c' : []))) ++ ('d' : ('e' : []))
=
  'a' : (('b' : ('c' : [])) ++ ('d' : ('e' : [])))
=
  'a' : ('b' : (('c' : []) ++ ('d' : ('e' : []))))
=
  'a' : ('b' : ('c' : ([] ++ ('d' : ('e' : [])))))
=
  'a' : ('b' : ('c' : ('d' : ('e' : []))))
=
  "abcde"
```

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

   "abc" ++ "de"
=
   'a' : ("bc" ++ "de")
=
   'a' : ('b' : ("c" ++ "de"))
=
   'a' : ('b' : ('c' : ("" ++ "de")))
=
   'a' : ('b' : ('c' : "de"))
=
   "abcde"
```

Properties of append

```
prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  (xs ++ ys) ++ zs  ==  xs ++ (ys ++ zs)

prop_append_ident :: [Int] -> Bool
prop_append_ident xs  =
  xs ++ [] == xs  &&  xs == [] ++ xs

prop_append_cons :: Int -> [Int] -> Bool
prop_append_cons x xs  =
  [x] ++ xs  ==  x : xs
```

Efficiency

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

  "abc" ++ "de"
=
  'a' : ("bc" ++ "de")
=
  'a' : ('b' : ("c" ++ "de"))
=
  'a' : ('b' : ('c' : ("" ++ "de")))
=
  'a' : ('b' : ('c' : "de"))
=
  "abcde"
```

Computing `xs ++ ys` takes about n steps, where n is the length of `xs`.

A useful fact

```
-- prop_sum.hs
import Test.QuickCheck

prop_sum :: Integer -> Property
prop_sum n  =  n >= 0 ==>  sum [1..n]  ==  n * (n+1) `div` 2
```

```
[melchior]dts: ghci prop_sum.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
*Main> quickCheck prop_sum
+++ OK, passed 100 tests.
*Main>
```

Associativity and Efficiency: Left vs. Right

Compare computing (associated to the left)

$$((\mathtt{xs}_1 \mathbin{+\!+} \mathtt{xs}_2) \mathbin{+\!+} \mathtt{xs}_3) \mathbin{+\!+} \mathtt{xs}_4$$

with computing (associated to the right)

$$\mathtt{xs}_1 \mathbin{+\!+} (\mathtt{xs}_2 \mathbin{+\!+} (\mathtt{xs}_3 \mathbin{+\!+} \mathtt{xs}_4))$$

where n_1, n_2, n_3, n_4 are the lengths of $\mathtt{xs}_1, \mathtt{xs}_2, \mathtt{xs}_3, \mathtt{xs}_4$.
Associating to the left takes

$$n_1 + (n_1 + n_2) + (n_1 + n_2 + n_3)$$

steps. If we have m lists of length n, it takes about $m^2 n$ steps.
Associating to the right takes

$$n_1 + n_2 + n_3$$

steps. If we have m lists of length n, it takes about mn steps.

When $m = 1000$, the first is a thousand times slower than the second!

Associativity and Efficiency: Sequential vs. Parallel

Compare computing (sequential)

$$x_1 + (x_2 + (x_3 + (x_4 + (x_5 + (x_6 + (x_7 + x_8))))))$$

with computing (parallel)

$$((x_1 + x_2) \ + \ (x_3 + x_4)) \quad + \quad ((x_5 + x_6) \ + \ (x_7 + x_8))$$

In sequence, summing 8 numbers takes 7 steps.

If we have m numbers it takes $m - 1$ steps.

In parallel, summing 8 numbers takes 3 steps.

$$x_1 + x_2 \text{ and } x_3 + x_4 \text{ and } x_5 + x_6 \text{ and } x_7 + x_8$$
$$(x_1 + x_2) + (x_3 + x_4) \text{ and } (x_5 + x_6) + (x_7 + x_8),$$
$$((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$$

If we have m numbers it takes $\log_2 m$ steps.

When $m = 1000$, the first is a hundred times slower than the second!