Informatics 1

Functional Programming Lecture 5

Function properties

Don Sannella

University of Edinburgh

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
```

You've seen these in Inf1-CL. It's very easy to define them - just write out all of the possibilities.
(You can do it in less space by using variables in patterns.)

&& and || are associative and commutative.
True is the identity for && and False is the identity for ||.

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
```

There are 256 characters in the "ASCII" character set.
It is often convenient to define functions using their numerical codes,
using the fact that the order of characters is the same as the order of their codes.

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
```

We can define these functions using arithmetic on character codes.

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
```

Notice the overlap between the guards in max3.

If both apply (e.g. x=y=z) then Haskell takes the first. (In this case it doesn't matter - both give the same result.)

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
```

Because the guards are checked in order, you can replace the last case with "otherwise" (since the last guard is always true if the others don't hold).

Otherwise is just another name for 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
```

You can write the same thing using if-then-else.

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
```

The last version seemed a bit inefficient:
we checked x >= y in the first condition, then y >= x in the second condition.
Here we do the comparison just once, giving one fewer test.
But reasoning about this is a lot harder than the other version.

It's more important to be clear than to be efficient:
- to you, next week or next year
- to people you are working with

Pretend that the next person who reads your code is a dangerous psychopath, and they know where you live.
Make it READABLE.

Making it fast is the LAST thing to do.
Much better:
- get it right, make it readable and easy to understand
- then MEASURE how fast it runs
- if it runs too slow, fix the bottleneck

Premature optimisation is the root of much evil!

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.

So use conditionals (case splitting) but not more than you need to.

A better way to define max3

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

You can define maximum of three numbers using max (of two numbers) twice. Very simple and clear!

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
```

Even clearer: write max as infix - writing backquotes around a function name makes it into an infix operator.

An even better way to define max3

```
max3 :: 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`

Everything is a function in Haskell.

Infix notation is just another way of writing function application.

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.

Does max have an identity? (Yes: negative infinity, called minBound in Haskell)

Part III

Append

Append

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

You've seen ++ in a previous lecture.
Here is the definition.

[a] means "list of a".
a is a TYPE VARIABLE, and can stand for any type.

```
  "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"
```

The definition of ++ is recursive in its first argument.
The computation is hard to read - the parentheses get in the way.

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"
```

Here is the same thing again, using string notation for character lists.
Question: why is recursion in the FIRST argument?
Try doing recursion in the second argument instead, and see what happens.
I don't think it's possible, at least not directly.

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`.

Time is proportional to the length of xs - we say it is "linear in the length of xs". The length of ys doesn't matter. So ++ isn't commutative with respect to time - the order matters.

A useful fact

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

prop_sum :: Int -> 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. (uses the fact on the last page)
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!

So ++ associates to the right in Haskell.

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!

Associative functions are great for parallelising computation!