Informatics 1
Functional Programming Lecture 5

Function properties

Don Sannella
University of Edinburgh
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"

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

\[(++): \{a\} \to \{a\} \to \{a\}\]
\[[] ++ ys \quad = \quad ys\]
\[(x:xs) ++ ys \quad = \quad x : (xs ++ ys)\]

"abc" ++ "de"

= 
lea : ("bc" ++ "de")
= 
lea : (lea : ("c" ++ "de"))
= 
lea : (lea : (lea : ("" ++ "de")))
= 
lea : (lea : (lea : "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 operators

• 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.
Properties of append

\[\text{prop\_append\_assoc :: } [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}\]
\[\text{prop\_append\_assoc } xs \ ys \ zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)\]

\[\text{prop\_append\_ident :: } [\text{Int}] \rightarrow \text{Bool}\]
\[\text{prop\_append\_ident } xs = xs ++ [] == xs \&\& xs == [] ++ xs\]

\[\text{prop\_append\_cons :: } \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}\]
\[\text{prop\_append\_cons } x \ xs = [x] ++ xs == x : xs\]
Efficiency


time

(++) :: [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)

\[
\left( (xs_1 ++ xs_2) ++ xs_3 \right) ++ xs_4
\]

with computing (associated to the right)

\[
xs_1 ++ (xs_2 ++ (xs_3 ++ xs_4))
\]

where \( n_1, n_2, n_3, n_4 \) are the lengths of \( xs_1, xs_2, xs_3, xs_4 \).

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 \( m^2n \) 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)

\[ ((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8)) \]

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!
BUT:

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!