Functions continued
Outline

- Defining Functions
- List Comprehensions
- Recursion
A Function without Recursion

Many functions can naturally be defined in terms of other functions.

factorial :: Int → Int
factorial n = product [1..n]

factorial maps any integer n to the product of the integers between 1 and n

Expressions are evaluated by a stepwise process of applying functions to their arguments. For example:

factorial 4
= product [1..4]
= product [1,2,3,4]
= 1*2*3*4
= 24
Recursive Functions

Functions can also be defined in terms of themselves. Such functions are called **recursive**.

\[
\text{factorial } 0 = 1 \\
\text{factorial } n = n \times \text{factorial } (n-1)
\]

factorial 3 = 3 \times \text{factorial } 2

= 3 \times (2 \times \text{factorial } 1)

= 3 \times (2 \times (1 \times \text{factorial } 0))

= 3 \times (2 \times (1 \times 1))

= 3 \times (2 \times 1)

= 3 \times 2

= 6

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.
Note:

- The base case factorial $0 = 1$ is appropriate because $1$ is the identity for multiplication: $1 \times x = x = x \times 1$.

- The recursive definition **diverges** on integers $< 0$ because the base case is never reached:

```plaintext
> factorial (-1)
Error: Control stack overflow
```
Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.

- As we shall see, however, many functions can naturally be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.
Recursion on Lists

Lists have naturally a recursive structure. Consequently, recursion is used to define functions on lists.

```
product :: [Int] → Int
product [] = 1
product (n:ns) = n * product ns
```

Product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

```
= 2 * (3 * product [4])
= 2 * (3 * (4 * product []))
= 2 * (3 * (4 * 1))
= 24
```
Using the same pattern of recursion as in product we can define the \textbf{length} function on lists.

\[
\begin{align*}
\text{length} & : [a] \rightarrow \text{Int} \\
\text{length} \; [] & = 0 \\
\text{length} \; (_\!:xs) & = 1 + \text{length} \; xs
\end{align*}
\]

\text{length} \; [1,2,3] \\
= \; 1 + \text{length} \; [2,3] \\
= \; 1 + (1 + \text{length} \; [3]) \\
= \; 1 + (1 + (1 + \text{length} \; [])) \\
= \; 1 + (1 + (1 + 0)) \\
= \; 3
Using a similar pattern of recursion we can define the `reverse` function on lists.

```
reverse :: [a] → [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

reverse [1,2,3]

= reverse [2,3] ++ [1]

= (reverse [3] ++ [2]) ++ [1]

= ((reverse [3] ++ [2]) ++ [1]

= (reverse [] ++ [3]) ++ [2]) ++ [1]

= (([ ] ++ [3]) ++ [2]) ++ [1]

= [3,2,1]
Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

  \[
  \text{zip} \quad :: \quad [a] \to [b] \to [(a,b)] \\
  \text{zip} \ [\] \ _ \ = \ [\] \\
  \text{zip} \ _ \ [\] \ = \ [\] \\
  \text{zip} \ (x:xs) \ (y:ys) \ = \ (x,y) : \text{zip} \ xs \ ys
  \]

- Remove the first \(n\) elements from a list:

  \[
  \text{drop} \quad :: \quad \text{Int} \to [a] \to [a] \\
  \text{drop} \ n \ xs \mid n \leq 0 \ = \ xs \\
  \text{drop} \ _ \ [\] \ = \ [\] \\
  \text{drop} \ n \ (_:xs) \ = \ \text{drop} \ (n-1) \ xs
  \]

-Appending two lists:

  \[
  (++) \quad :: \quad [a] \to [a] \to [a] \\
  [\] \ ++ \ ys \ = \ ys \\
  (x:xs) \ ++ \ ys \ = \ x : (xs \ ++ \ ys)
  \]
Laziness Revisited

Laziness interacts with recursion in interesting ways. For example, what does the following function do?

```haskell
numberList xs = zip [0..] xs
```

```
> numberList "abcd"
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```
Laziness with Recursion
Recursion combined with lazy evaluation can be tricky; stack overflows may result in the following example:

```
expensiveLen [] = 0
expensiveLen (_:xs) = 1 + expensiveLen xs
```

```
stillExpensiveLen ls = len 0 ls
  where len z [] = z
       len z (_:xs) = len (z+1) xs
```

```
cheapLen ls = len 0 ls
  where len z [] = z
       len z (_:xs) = let z' = z+1
                     in z' `seq` len z' xs
```

> expensiveLen [1..10000000] -- takes quite a while
> stillExpensiveLen [1..10000000] -- also takes a long time
> cheapLen [1..10000000] -- less memory and time
The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $> \text{the head}$, and then appending the resulting lists on either side of the head value.
Using recursion, this specification can be translated directly into an implementation:

```haskell
qsort  :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

Note:
- This is probably the simplest implementation of quicksort in any programming language!
For example (abbreviating qsort as q):

\[ q \{3,2,4,1,5\} \]

\[ q \{2,1\} ++ [3] ++ q \{4,5\} \]

\[ q \{1\} ++ [2] ++ q \{\} \]

\[ q \{\} ++ [4] ++ q \{5\} \]

\[ [1] \]

\[ [\] \]

\[ [\] \]

\[ [5] \]
(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:
  
  ```
  and :: [Bool] → Bool
  and [] = True
  and (b:bs) = b && and bs
  ```

- Concatenate a list of lists:
  
  ```
  concat :: [[a]] → [a]
  ```
- Produce a list with n identical elements:
  
  ```haskell
  replicate :: Int -> a -> [a]
  ```

- Select the nth element of a list:
  
  ```haskell
  (!! :: [a] -> Int -> a
  ```

- Decide if a value is an element of a list:
  
  ```haskell
  elem :: Eq a => a -> [a] -> Bool
  ```
(2) Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

that merges two sorted lists of integers to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```
Define a recursive function

\[
\text{msort} :: [\text{Int}] \rightarrow [\text{Int}]
\]

that implements \textit{merge sort}, which can be specified by the following two rules:

1. Lists of length \( \leq 1 \) are already sorted;
2. Other lists can be sorted by sorting the two halves and merging the resulting lists.
Exercises + Some answers

(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:
  \[\text{and} :: [\text{Bool}] \rightarrow \text{Bool}\]
  \[
  \begin{align*}
  \text{and} \; [] & = \text{True} \\
  \text{and} \; (b:b:s) & = b \land \text{and} \; s
  \end{align*}
  \]

- Concatenate a list of lists:
  \[\text{concat} :: [[\text{a}]] \rightarrow \text{[a]}\]
  \[
  \begin{align*}
  \text{concat} \; [] & = [] \\
  \text{concat} \; (x:xss) & = x + \text{concat} \; xss
  \end{align*}
  \]
- Produce a list with n identical elements:

\[
\text{replicate} :: \text{Int} \to a \to [a]
\]
\[
\text{replicate} \ 0 \_ = []
\]
\[
\text{replicate} \ n \ x = x : \text{replicate} \ (n-1) \ x
\]

- Select the nth element of a list:

\[
(!!) :: [a] \to \text{Int} \to a
\]
\[
(!!) \ (x:_:\_) \ 0 = x
\]
\[
(!!) \ (_:xs) \ n = (!!) \ xs \ (n-1)
\]

- Decide if a value is an element of a list:

\[
\text{elem} :: \text{Eq} \ a \Rightarrow a \to [a] \to \text{Bool}
\]
\[
\text{elem} \ x \ [\] = \text{False}
\]
\[
\text{elem} \ x \ (y:ys) | x==y = \text{True}
\]
\[
\text{otherwise} = \text{elem} \ x \ ys
\]
(2) Define a recursive function

```
define merge :: [Int] -> [Int] -> [Int]
```
(3) Define a recursive function

\[
\text{msort} :: \text{[Int]} \rightarrow \text{[Int]}
\]

that implements \textit{merge sort}, which can be specified by the following two rules:

1. Lists of length \( \leq 1 \) are already sorted;
2. Other lists can be sorted by sorting the two halves and merging the resulting lists.

\[
\begin{align*}
\text{halves } xs &= \text{splitAt (length } xs \text{ `div` } 2) \text{ } xs \\
\text{msort } [] &= [] \\
\text{msort } [x] &= [x] \\
\text{msort } xs &= \text{merge (msort } ys) \text{ (msort } zs) \\
& \quad \text{where } (ys,zs) = \text{halves } xs
\end{align*}
\]