Session 4

Lists

Where we are

this session introducing lists:

- list comprehensions
- some library functions, including polymorphic ones
- case study: developing a small program
- *Aim:* design and write larger programs in Haskell, operating on whole lists at a time where possible.

next session using higher-order functions (functions that take functions as arguments) to create even more concise and powerful programs

The focus here is on using library functions without worrying about how they're implemented. (We'll cover that in sessions 8 and 9.)

Lists

Lists

Zero or more things of the same type

```
                [1,2,3] :: [Int]
['H','e','l','l','o'] :: [Char]
              "Hello" :: [Char]
    ["Hello","world"] :: [[Char]]
        [div,(+),(*)] :: [Int -> Int -> Int]
        [product,sum] :: [[Int] -> Int]
                  "" :: [Char]
                  [] :: [a]
```

What are the rules for comparing lists?

interpreter

```
compare "abc" "ac"
compare "abc" "abd"
```

Lists may even be infinite (more in session 9).

Aside: type synonyms

- You can introduce a new name for a type using a type synonym declaration, like the following:

Prelude
```
type String = [Char]
```

- This is predefined in the Haskell "**Prelude**", a special module containing definitions of the basic and many useful functions. These functions are always available.
- Using a type synonym like **String** is entirely equivalent to using the expanded type.

interpreter
```
:t ['H','e','l','l','o']
:t "Hello"
:t ""
:t ["Hello","world"]
```

Special lists: arithmetic sequences

Making subranges of an arithmetic sequence:

interpreter
```
[1..7]
[1,2..7]
[1,3..11]
[1,4..20]
[12,11..5]
```

Infinite lists (you'll need to interrupt GHCi):

interpreter
```
[1..]
[1,2..]
[1,3..]
[0,-1..]
```

List comprehensions

List comprehensions

Mathematical set notation:

$$\{f(x,y) \mid x \in A \wedge y \in B \wedge x + y \leq 5\}$$

Haskell: **[f x y | x <- xs, y <- ys, x+y <= 5]** except that lists are not sets:

- For each **x** in the list **xs**,
  - for each **y** in the list **ys** such that **x+y <= 5**,
    - include the value **f x y**.

interpreter

```
[n*n | n <- [1..10]]
[(x,c) | x <- [1..3], c <- "abcd"]
[(x,y) | x <- [1..4], y <- [1..3]]
[(x,y) | x <- [1..4], y <- [1..3], x+y <= 5]
[x*y | x <- [1..4], y <- [1..3], x+y <= 5]
```

List comprehension syntax

A list comprehension expression has the form

$$[\ \mathit{expression}\ \texttt{|}\ \mathit{qualifier}_1\texttt{, ..., }\mathit{qualifier}_n\ ]$$

where each qualifier is either

a generator of the form *arg* **<-** *expression*, where the expression denotes a list. This introduces a new variable **x** that may be used in later qualifiers and in the result expression. This variable is bound to each value in the list in turn.

a guard which may be any expression of type **Bool**, and limits the combinations considered.

More examples

Try to predict what these will do before you run them: **interpreter**

```
[n^2 | n <- [1..10]]
[2^n | n <- [1..10]]

[n | n <- [1..100], n*n == 49]
[n | n <- [1..], n*n == 49]

[(x, y) | x <- [1..3], y <- [1..x]]

[c | c <- "Hello world", isLower c]
[ord c | c <- "Hello world", isLower c]
```

Standard functions on lists

Polymorphism

The function **length** works for all types of lists: **interpreter**

```
length [1,2,3,4]
length "abc"
length ["Hello", "world"]
```

In fact its definition is independent of the type of elements in the list (though they must all be of the same type).

This is expressed by the polymorphic type of **length**: **Prelude**

```
length :: [a] -> Int
```

Here **a** is a type variable, standing for an arbitrary type.

Type variables

The function **++** returns the concatenation of two lists of the same type:

interpreter

```
[1,2,3] ++ [4,5]
"abc" ++ "def"
["Goodbye"] ++ ["cruel", "world"]
```

Moreover, the list returned has the same type of elements as the two argument lists, as expressed by the function's type:

Prelude

```
(++) :: [a] -> [a] -> [a]
```

Instances:

Prelude

```
(++) :: [Int] -> [Int] -> [Int]
(++) :: [Char] -> [Char] -> [Char]
(++) :: [[Char]] -> [[Char]] -> [[Char]]
```

Concatenation of several lists

This related function concatenates a list of lists:

Prelude

```
concat :: [[a]] -> [a]
```

For example:

interpreter

```
concat [[1,2,3], [4,5]]
concat ["abc", "defg"]
concat [[4,5], [], [3], [6,7]]
concat [[1,2]]
concat []
```

The type stipulates that all of the lists must be of the same type.

Predefined list functions

Prelude

```
null :: [a] -> Bool
head :: [a] -> a
tail :: [a] -> [a]
last :: [a] -> a
init :: [a] -> [a]
reverse :: [a] -> [a]
```

Try some experiments, including:

interpreter

```
null [1..6]
null [6..1]
head "hello"
tail "hello"
last "hello"
init "hello"
reverse "hello"
```

More predefined list functions

Front and back sublists of a list:

Prelude
```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

Experiments:

interpreter
```
take 3 "hello"
take 20 [1..5]
drop 3 "hello"
drop 20 [1..5]
```

A list of n copies of a value:

Prelude
```
replicate :: Int -> a -> [a]
```

interpreter
```
replicate 5 3
replicate 9 'a'
```

Infinite lists

Infinite lists can be used, as long as we only ask for some front part of the list:

interpreter
```
take 20 [1..]
null [1..]
head [1..]
```

For operations like **drop**, the initial part of the output depends only on the initial part of the input, so these operations are also useful on infinite lists:

interpreter
```
take 20 (drop 30 [1..])
```

But we can't use functions that look at the end of the list, like **length**, **last** or **reverse**, on infinite lists.

Infinite repetition

There are several library functions making infinite lists, e.g.

Prelude
```
repeat :: a -> [a]
```

makes a list with an infinite number of copies of the argument:

repeat x = **[**$x, x, x, \ldots$

We can use these infinite lists, as long as we don't look at the end. For example:

interpreter
```
take 20 (repeat 3)
take 20 (repeat '*')
```

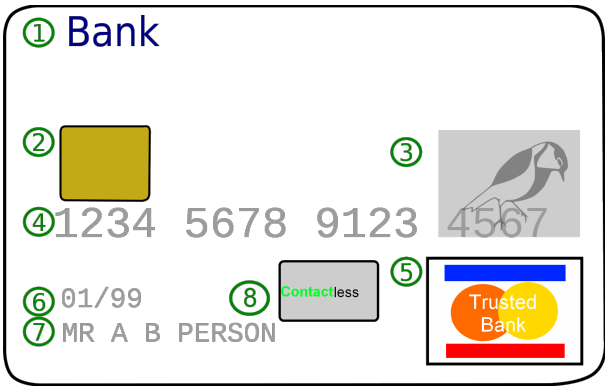
In fact this is how **replicate** is defined:

Prelude
```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Developing a list function

Example: validating payment card numbers

Consider the card number ④ on a payment card:



Not all card numbers are valid, to protect against common errors in entering numbers.

The basic validity check can be done without consulting a database.

Validation method

This is a simple checksum calculation devised by Hans Peter Luhn of IBM in 1954 (US patent 2950048 "Computer for verifying numbers"):

1. Substitute every second number from the back by doubling it, and, if the result is larger than 9, subtracting 9:

1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7
↓		↓		↓		↓		↓		↓		↓		↓	
2		6		1		5		9		4		8		3	

2. Add the unchanged and substituted digits:

$$2+2+6+4+1+6+5+6+9+1+4+3+8+5+3+7 = 74$$

3. The number is valid if the last digit of the sum is 0.

This usually catches common errors such as transposing, doubling or omitting digits.

Haskell implementation

We want to write a function to test for valid numbers:

```
valid :: String -> Bool
valid s = undefined
```

We'll assume that all the characters in the string are digits. We need to convert digit characters to numbers:

```
digits :: String -> [Int]
digits s = [digitToInt c | c <- s, isDigit c]
```

Here **`digitToInt`** from the **`Data.Char`** library is the same as the **`charToNum`** function we wrote in exercises 2.

To get these in reverse order, we can just use **`reverse`**.

Dividing the list in two

Because we're going to treat digits in odd and even positions differently, we'll write functions to pick out each sublist:

```
odds :: [a] -> [a]
odds xs = undefined

evens :: [a] -> [a]
evens xs = undefined
```

Although we only need to work on lists of **Int**s, we've made them polymorphic, because the problem is independent of what's in the lists. We want:

- **odds** "abcde" = "ace"
- **evens** "abcde" = "bd"

There are several ways to do this.

A handy library function

The predefined function **zip** takes two lists, possibly of different types, and returns a list of pairs of corresponding elements:

interpreter

```
zip "abc" [1..4]
zip [1..] "abcdef"
```

Its type signature is

Prelude

```
zip :: [a] -> [b] -> [(a,b)]
```

There is also

interpreter

```
unzip [('a',1),('b',2),('c',3)]
```

with type signature

Prelude

```
unzip :: [(a,b)] -> ([a],[b])
```

Using zip

zip is very useful:

interpreter

```
zip "computer" "science"
[x | (x, y) <- zip "computer" "science", x == y]
```

We can even use it with infinite lists:

interpreter

```
zip [1..] "abcde"
[x | (n, x) <- zip [1..] "abcde", odd n]
[x | (n, x) <- zip [1..] "abcde", even n]
```

So we define

```
odds :: [a] -> [a]
odds xs = [x | (n, x) <- zip [1..] xs, odd n]

evens :: [a] -> [a]
evens xs = [x | (n, x) <- zip [1..] xs, even n]
```

Substitution function

Substitution of a single digit follows the specification:

```
substitute :: Int -> Int
substitute n
  | n*2 > 9 = n*2 - 9
  | otherwise = n*2
```

We can test this on all the digits at once:

interpreter

```
[(n, substitute n) | n <- [0..9]]
```

Putting it all together

Now we can define our top-level function:

```
valid :: String -> Bool
valid s = total `mod` 10 == 0
  where
    total = sum (odds rev_ns) +
            sum [substitute n | n <- evens rev_ns]
    rev_ns = reverse (digits s)
```

- The reversed list of digits is used twice, so we name it in a **`where`** definition.
- The total is only used once, but giving it a meaningful name makes the code clearer.

More library functions

Maximum and minimum

The function **`max`** has the general type

Prelude

```
max :: Ord a => a -> a -> a
```

There is also a **`min`** function:

Prelude

```
min :: Ord a => a -> a -> a
```

and list versions of both functions:

Prelude

```
maximum :: (Ord a) => [a] -> a
minimum :: (Ord a) => [a] -> a
```

```
-- checksum formula used on credit and debit cards
-- US Patent 2950048 "Computer for verifying numbers",
-- Hans Peter Luhn, IBM
-- (filed 6 January 1954, granted 23 August 1960)
module Luhn where

import Data.Char

-- Luhn algorithm for validating card numbers
valid :: String -> Bool
valid s = total `mod` 10 == 0
  where
    total = sum (odds rev_ns) +
            sum [substitute n | n <- evens rev_ns]
    rev_ns = reverse (digits s)

-- the digits in a string
digits :: String -> [Int]
digits s = [digitToInt c | c <- s, isDigit c]

-- elements in even-numbered positions, counting from 1
evens :: [a] -> [a]
evens xs = [x | (i, x) <- zip [1..] xs, even i]

-- elements in odd-numbered positions, counting from 1
odds :: [a] -> [a]
odds xs = [x | (i, x) <- zip [1..] xs, odd i]

-- substitute digit: same as summing digits of n*2
substitute :: Int -> Int
substitute n
  | n*2 > 9 = n*2 - 9
  | otherwise = n*2
```

Figure 4.1: Validity checker for card numbers

Examples:

interpreter

```
maximum [3,1,4,5,9,2,6]
minimum [3,1,4,5,9,2,6]
maximum [6]
maximum []
```

Functions from the `Data.List` module

Sorting and grouping:

Data.List

```
sort :: Ord a => [a] -> [a]
group :: Eq a => [a] -> [[a]]
```

Examples:

interpreter

```
:m + Data.List

sort [3,1,4,5,9,2,6]
sort "Hello world!"
group [1,2,2,2,3,1,1,2]
group "abracadabra"
sort "abracadabra"
group (sort "abracadabra")
```

String processing

String processing

Since strings are lists of characters, general list functions may be used on them, but there are also some special functions on strings:

Prelude

```
lines :: String -> [String]
unlines :: [String] -> String
words :: String -> [String]
unwords :: [String] -> String
```

For example

interpreter

```
lines "two words\nline\n"
unlines ["two words", "line"]
words "two words"
unwords ["two", "words"]
unwords (reverse (words "hello world"))
unwords [reverse w | w <- words "hello world"]
```

Multi-line output

The result of **`unlines`** is a string with embedded newlines:

interpreter

```
unlines ["line one", "line two"]
```

To get the interpreter to display a string, acting on the newlines it contains, we can use **putStr**. interpreter

```
putStr "abc\ndef\n"
putStr (unlines ["line one", "line two"])
putStr (unlines ["(" ++ show n ++ ")" | n <- [1..10]])
putStr (unlines [replicate n '*' | n <- [1..10]])
```

We'll return to **putStr** in session 10, but for now we can only use it in the interpreter.

Next session: higher-order functions

Functions that take functions as arguments, e.g.: interpreter

```
map ord "Hello world"
filter isLower "Hello world"
filter odd [3,1,4,1,5,9,2,6]
takeWhile odd [3,1,4,1,5,9,2,6]
dropWhile odd [3,1,4,1,5,9,2,6]
```

These abstractions allow us to write even more powerful programs.

Exercises

These exercises provide practice in manipulating lists, both using list comprehensions and library functions.

1. In the interpreter, write expressions for
   (a) All the numbers from 1 to 100.
   (b) Squares of the numbers from 1 to 20.
   (c) Divisors of 100.
   Hint: these will be numbers **n** between 1 and 100 such that **100 `mod` n == 0**.

2. This function triples each integer in a list:

   ```
   tripleAll :: [Int] -> [Int]
   tripleAll ns = [3*n | n <- ns]
   ```

   In your source file, give a definition of the similar function

   ```
   squareAll :: [Int] -> [Int]
   ```

   that squares all the elements of a list of integers.

3. (a) Give a definition of the function

   ```
   capitalize :: String -> String
   ```

   that returns the input list with the lower case letters capitalized. You will need to use the function Data.Char

   ```
   toUpper :: Char -> Char
   ```

from the `Data.Char` module. (You'll need `import Data.Char` in your file.)

(b) Generalizing the previous part, write a function

```
capitalizeLetters :: String -> String
```

that does the same, but discards non-letters. You may wish to use the following function from the `Data.Char` module:

`Data.Char`

```
isAlpha :: Char -> Bool
```

4. (a) Write a function

```
backwards :: String -> String
```

that takes a string consisting of words, and returns a string of the same words in the reverse order.

(b) Write a function of the same type that takes a string consisting of words, and returns a string of the same words in the same order, but with each word reversed.

5. Write a function

```
divisors :: Int -> [Int]
```

returning the list of numbers that evenly divide the given number, *e.g.* `divisors 12` should be `[1,2,3,4,6,12]`.

Hint: generalize your answer to 1(c).

6. Write a function

```
average :: [Double] -> Double
```

that computes the average of a list of numbers.

7. A *palindrome* is a word that is the same reversed, *e.g.* "kayak" or "repaper".

(a) Write a function to test whether a word is a palindrome. (There's no need for a list comprehension here.)

(b) Some famous palindromes rely on ignoring non-letters and the distinction between upper and lower case (e.g. "Madam, I'm Adam"). Write another function to test for this. (*Hint:* Try to doing it by calling functions you've already written, instead of copying and tweaking code.)

8. Write a function

```
frequency :: [Char] -> [(Char, Int)]
```

that takes a list of characters and produces a list of unique characters, each paired with the number of times they occur in the original list, e.g.

```
frequency "abracadabra"
```

should return

```
[('a',5), ('b',2), ('c',1), ('d',1), ('r',2)]
```

Hint: use **group** (from **Data.List**) and some other functions mentioned in the lecture.

Your function should not actually require that the elements of the list be characters, so you can rewrite its type as

```
frequency :: Ord a => [a] -> [(a, Int)]
```

Try it on a list of numbers.

9. Consider how you would write a function

```
palindromic :: [Char] -> Bool
```

that returns **True** if some re-arrangement of the input string (assumed to be all lowercase letters) is a palindrome. We could do this by generating all the re-arrangements of the input string, but there will be a lot of them, and there is another approach. For example,

- The function should return **True** for **""**, **"a"**, **"aa"**, **"aaa"**, **"aabba"** and **"ababc"**.
- It should return **False** for **"abc"** and **"aaabbc"**.

That is because while **"ababc"** can be re-arranged as the palindromes **"abcba"** or **"bacab"**, that is not possible for **"abc"** or **"aaabbc"**. All the letters in a palindrome must be paired, with the possible exception of the middle letter. Thus if a letter occurs an odd number of times, one of them must be placed in the middle to form a palindrome. If more that one letter occurs an odd number of times, this is impossible. Use this idea to define the function.

10. Add to your **Geometry** module a function

```
readGrid :: String -> [(Point, Char)]
```

That takes a piece of text and maps it onto the two-dimensional plane, with the first character at the origin. For example, **readGrid "#.#\n###\n#.#\n"** should return **interpreter**

```
[(Point 0 0,'#'),(Point 1 0,'.'),(Point 2 0,'#'),
 (Point 0 (-1),'#'),(Point 1 (-1),'#'),(Point 2 (-1),'#'),
 (Point 0 (-2),'#'),(Point 1 (-2),'.'),(Point 2 (-2),'#')]
```

11. The **Data.List** library module contains two functions **Data.List**

```
tails :: [a] -> [[a]]
inits :: [a] -> [[a]]
```

(a) Try them on various inputs and work out what they do: **interpreter**

```
:m + Data.List

tails [1,2,3,4]
tails "abcd"
```

```
inits [1,2,3,4]
inits "abcd"
```

Do they work on infinite lists?

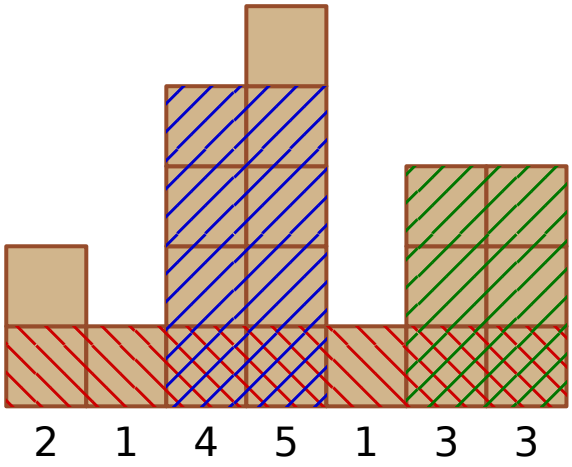
(b) Consider the following functions defined using **tails** and **inits**:

```
foo :: [a] -> [([a], [a])]
foo xs = zip (inits xs) (tails xs)

bar :: [a] -> [[a]]
bar xs = [ys | ts <- tails xs, ys <- inits ts, not (null ys)]
```

Work out what they do, give them descriptive names, and put them in a new module called **ListUtilities**.

12. Consider the problem of finding the area of the largest rectangle that fits under a histogram. For example, several rectangles fit under the histogram below, but a few of the larger ones are marked:



In this case, the largest area of any rectangle is **8**.

Use one of the functions from the previous question to write a function

```
largestRectangle :: [Int] -> Int
```

that takes a list of bar heights and returns the area of the largest rectangle under that histogram. You may assume that the list is non-empty and all the heights are non-negative. For example, **largestRectangle [2,1,4,5,1,3,3]** (the histogram shown above) should return **8**.

(There are faster implementations than this brute-force solution, but they are quite complicated and require language features we have not yet met. Also: (a) in some situations it is more important to minimize programmer time than runtime, and (b) a common strategy to develop efficient algorithms is to start with a simple and obviously correct solution and transform it.)