If you have your notebook computer with you (and have Haskell Platform installed), start ghci and try the following tasks.

List Deconstruction

1. (a) What is the type of the function head? Use the command :t to find out the type of a value.
   (b) Since the input type of head is a list \([a]\), let us try it on some input.
      i. head \([1, 2, 3]\) =
      ii. head "abcde" =
      iii. head [] =
   (c) In words, what does the function head do?

2. (a) What is the type of the function tail?
   (b) Try tail on some input.
      i. tail \([1, 2, 3]\) =
      ii. tail "abcde" =
      iii. tail [] =
   (c) In words, what does the function tail do?

3. (a) What is the type of the function last?
   (b) Try last on some input. Think about some input yourself.
      i. last =
      ii. last =
      iii. last =
   (c) In words, what does the function last do?

4. (a) What is the type of the function init?
   (b) Try init on some input. Think about some input yourself.
      i. init =
      ii. init =
      iii. init =
   (c) In words, what does the function init do?
   (d) What property does init and last jointly satisfy?

5. (a) What is the type of the function null?
(b) Try *init* on some input. Think about some input yourself.
   i. `null` =
   ii. `null` =
   iii. `null` =

(c) Can you write down a definition of `null`, by pattern matching?

**List Generation**

1. What are the results of the following expressions?
   (a) `[0..25]` =
   (b) `[0, 2..25]` =
   (c) `[25..0]` =
   (d) `['a'..'z']` =
   (e) `[1..]` =

2. What are the results of the following expressions?
   (a) `[x | x <- [1..10]]` =
   (b) `[x * x | x <- [1..10]]` =
   (c) `[(x, y) | x <- [0..2], y <- "abc"]` =
   (d) What is the type of the expression above?
   (e) `[x * x | x <- [1..10], odd x]` =

3. What are the results of the following expressions?
   (a) `[(a, b) | a <- [1..3], b <- [1..2]]` =
   (b) `[(a, b) | b <- [1..2], a <- [1..3]]` =
   (c) `[(i, j) | i <- [1..4], j <- [(i + 1)..4]]` =
   (d) `[(i, j) | i <- [1..4], even i, j <- [(i + 1)..4], odd j]` =
   (e) `['a'|i <- [0..10]]` =

**Combinators on Lists**

1. (a) What is the type of the function `!!` (two exclamation marks)?
   (b) Try `!!` on some input. Think about some input yourself. Note that `!!` is an infix operator.
   i. `[1,2,3]!!1` =
   ii. `!!` =
   iii. `!!` =
   (c) In words, what does the function `!!` do?

2. (a) What is the type of the function `length`?
   (b) Try `length` on some input.
1. length =
2. length =
(c) In words, what does the function length do?

3. (a) What is the type of the function +? (In ASCII one types ++.)
(b) Try + on some input. Think about some input yourself. Note that + is an infix operator.
   i.
   ii.
(c) In words, what does the function + do?
(d) Wait a minute... Both : and + appear to add elements to a list. How are they different?

4. (a) What is the type of the function concat?
(b) Try concat on some input.
   i. concat =
   ii. concat =
(c) In words, what does the function concat do?

5. (a) What is the type of the function take?
(b) Try take on some input. Since take expects an integer and list, try it on some extreme cases. For example, when the integer is zero, negative, or larger than the length of the list.
   i. take =
   ii. take =
   iii. take =
(c) In words, what does the function take do?

6. (a) What is the type of the function drop?
(b) Try drop on some input. Like take, try it on some extreme cases.
   i. drop =
   ii. drop =
   iii. drop =
(c) In words, what does the function drop do?
(d) Does take, drop, and (+) together satisfy some properties?

7. (a) What is the type of the function map?
(b) Try map on some input. It is a little bit harder, since map expects a functional argument.
   i. map square [1, 2, 3, 4] =
   ii. map (1+) [1, 2, 3, 4] =
   iii. map (const 'a') [1..10] =
(c) In words, what does the function map do?
(d) Is (1+) a function? Try it.

8. (a) What is the type of the function map?
(b) Try map on some input. It is a little bit harder, since map expects a functional argument.
i. map square [1, 2, 3, 4] =
ii. map (1+) [1, 2, 3, 4] =
iii. map =
(c) In words, what does the function map do?

(d) Is (1+) a function? Try it.
i. (1+) 2 =
ii. ((1+) · (1+) · (1+)) 0 =
where (·) is function composition.

Sectioning

- Infix operators are curried too. The operator (+) may have type Integer → Integer → Integer.
- Infix operator can be partially applied too.

\[
(x \oplus) y = x \oplus y \\
(\oplus y) x = x \oplus y
\]

- (1 +) :: Integer → Integer increments its argument by one.
- (1.0 /) :: Float → Float is the “reciprocal” function.
- (/ 2.0) :: Float → Float is the “halving” function.

1. Define a function doubleAll :: [Integer] → [Integer] that doubles each number of the input list. E.g.
• doubleAll [1, 2, 3] = [2, 4, 6].

- How do you define a new function? I’d suggest you to
  (a) create a new text file (using your favourite editor) in your current working directory (the directory you executed ghci). The file should have extension .hs.
  (b) Type your definitions in the file.

(c) Load the file into ghci by the command:

:1 <filename>

2. Define a function quadAll :: [Integer] → [Integer] that multiplies each number of the input list by 4. Of course, it’s cool only if you define quadAll using doubleAll.

λ Abstraction

- Every once in a while you may need a small function which you do not want to give a name to. At such moments you can use the λ notation:

  - map (λx → x × x) [1, 2, 3, 4] = [1, 4, 9, 16]
  - In ASCII λ is written \.

- λ is an important primitive notion. We will talk more about it later.

1. What is the type of (λx → x + 1)?
2. (λx → x + 1) 2 =
3. What is the type of (λx y → x + 2 × y)?
4. (λx y → x + 2 × y) 1 2 =
5. Define doubleAll :: [Integer] → [Integer] again. This time using a λ expression.

6. Pattern matching in λ. To extract, for example, the two components of a pair

  (a) What is the type of (λ(x, y) → (y, x))? 

  (b) (λ(x, y) → (y, x)) (1, ’a’) =

  (c) Alternatively, try

      (λp → (snd p, fst p)) (1, ’a’) =
Back to Lists

1. (a) What is the type of the function \texttt{filter}?

(b) Try \texttt{filter} on some input.
   i. \texttt{filter even [1..10]} =
   ii. \texttt{filter (> 10) [1..20]} =
   iii. \texttt{filter (\lambda x \rightarrow x \mod 3 = 1) [1..20]} =

(c) In words, what does the function \texttt{filter} do?

(d) Define a function \texttt{squaresUpto :: Integer \rightarrow [Integer]} such that \texttt{squaresUpto n} is the list of all positive square numbers that are at most \texttt{n}. For some examples,
   • \texttt{squaresUpto 10} = [1, 4, 9].
   • \texttt{squaresUpto (-1)} = []

2. (a) What is the type of the function \texttt{zip}?

(b) Try \texttt{zip} on some input.
   i. \texttt{zip [1..10] "abcde"} =
   ii. \texttt{zip "abcde" [0..]} =
   iii. \texttt{zip} =

(c) In words, what does the function \texttt{zip} do?

(d) Define \texttt{positions :: Char \rightarrow String \rightarrow Int}, such that \texttt{positions x xs} returns the positions of occurrences of \texttt{x} in \texttt{x}. E.g.
   • \texttt{positions ‘o’ "roodo"} = [1, 2, 4].

Check the handouts if you just cannot figure out how.

(e) What if you want only the position of the \textit{first} occurrence of \texttt{x}? Define \texttt{pos :: Char \rightarrow String \rightarrow Int}, by reusing \texttt{positions}.

Morals of the Story

• Lazy evaluation helps to improve modularity.
  − List combinators can be conveniently reused. Only the relevant parts are computed.

• The combinator style encourages “wholemeal programming”.
  − Think of aggregate data as a whole, and process them as a whole!

Fold on Lists

Now we’ve finally come to the most important function on list we will introduce: the fold.

1. What is the type of the function \texttt{foldr}?

2. Try the following:
   (a) \texttt{foldr (+) 0 [1..10]} =
   (b) \texttt{foldr (\times) 1 [1..10]} =

One way to look at \texttt{foldr (\oplus) e} is that it replaces \texttt{[]} with \texttt{e} and (:) with (\oplus):

\[
\texttt{foldr (\oplus) e [1, 2, 3, 4]} = \texttt{foldr (\oplus) e (1 : (2 : (3 : (4 : []))))) = 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))))}.
\]

1. Define \texttt{prod :: [Integer] \rightarrow Integer}, which computes the product of a list of numbers, using \texttt{foldr}. E.g.
   • \texttt{prod [2, 3, 4]} = 24.
2. (a) Try the following
   i. 3 `max` 5 =
   ii. 5 `max` 3 =

(b) Define \textit{myMaximum} :: [Integer] \rightarrow Integer
that returns the maximum element in a
list, using \textit{max} and \textit{foldr}. (Hint: the
largest Integer is denoted by \textit{maxBound}
in Haskell.) (I want you to define
\textit{myMaximum}, because there is already a
\textit{maximum} doing the same job.)

3. What does \textit{foldr} (:) [] do?

4. Define \textit{myLength} :: [a] \rightarrow Int that computes the
same function as \textit{length}, using \textit{foldr}. Check the
handouts if you just cannot figure out how.

5. Define \textit{myMap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
computes the same function as \textit{map}, using \textit{foldr}. Check
the handouts if you just cannot figure out how.

6. Define \textit{append} :: [a] \rightarrow [a] \rightarrow [a] such that
\textit{append} \textit{x} \textit{ys} is the same as \textit{x} \textit{++} \textit{ys}. Of course,
do not use \textit{++} but use \textit{foldr}. Check the handouts
if you just cannot figure out how.

In fact, any function that takes a list as its input
can be written in terms of \textit{foldr} — although it might
not be always practical.