Haskell Functions
Outline

- Defining Functions
- List Comprehensions
- Recursion
Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions:

```plaintext
if cond then e1 else e2
```

- $e_1$ and $e_2$ must be of the same type
- else branch is always present

```plaintext
abs :: Int -> Int
abs n = if n >= 0 then n else -n

max :: Int -> Int -> Int
max x y = if x <= y then y else x

take :: Int -> [a] -> [a]
take n xs = if n <= 0 then []
    else if xs == [] then []
    else (head xs) : take (n-1) (tail xs)
```
Guarded Equations
As an alternative to conditionals, functions can also be defined using guarded equations.

Prelude:

\[
\text{abs } n \mid n \geq 0 \quad = n \\
| \quad \text{otherwise} = -n
\]

Guarded equations can be used to make definitions involving multiple conditions easier to read:

\[
\text{signum } n \mid n < 0 \quad = -1 \\
| \quad n == 0 \quad = 0 \\
| \quad \text{otherwise} = 1
\]

compare with …

\[
\text{signum } n = \text{if } n < 0 \text{ then } -1 \text{ else } \\
\quad \text{if } n == 0 \text{ then } 0 \text{ else } 1
\]
Guards and patterns can be freely mixed, the first equation whose pattern matches and guard is satisfied is chosen.

```
take :: Int -> [a] -> [a]
take n xs | n <= 0 = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```
Pattern Matching

- Many functions are defined using pattern matching on their arguments.

```haskell
not :: Bool -> Bool
not False = True
not True  = False
```

- Pattern can be a constant value, or include one or more variables.

not maps False to True, and True to False.
Functions can be defined in many different ways using pattern matching. For example

\[ (\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \]

\[
\begin{align*}
\text{True} & \; \&\& \; \text{True} = \text{True} \\
\text{True} & \; \&\& \; \text{False} = \text{False} \\
\text{False} & \; \&\& \; \text{True} = \text{False} \\
\text{False} & \; \&\& \; \text{False} = \text{False}
\end{align*}
\]

can be defined more compactly by

\[ \begin{align*}
\text{True} \; \&\& \; \text{True} & = \text{True} \\
_ & \; \&\& \; _ & = \text{False}
\end{align*} \]

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

\[ \begin{align*}
\text{False} \; \&\& \; _ & = \text{False} \\
\text{True} \; \&\& \; \text{b} & = \text{b}
\end{align*} \]

• The underscore symbol _ is a **wildcard** pattern that matches any argument value.
Patterns are matched in order. For example, the following definition always returns False:

```
_ && _ = False
True && True = True
```

Patterns may not repeat variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```
List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called “cons” that adds an element to the start of a list.

[1, 2, 3, 4] \rightarrow \text{Means } 1 : (2 : (3 : (4 : []))).

Functions on lists can be defined using \textit{x:xs} patterns.

\begin{align*}
\text{head} & :: [a] \rightarrow a \\
\text{head (x:_)} & = x \\
\text{tail} & :: [a] \rightarrow [a] \\
\text{tail (::_xs)} & = xs
\end{align*}

head and tail map any non-empty list to its first and remaining elements.

is this definition complete?
Note:

- x:xs patterns only match **non-empty** lists:
  ```haskell
  > head []
  Error
  ```

- x:xs patterns must be parenthesised, because application has priority over (:) - For example, the following definition gives an error:
  ```haskell
  head x:_ = x
  ```

- Patterns can contain arbitrarily deep structure:
  ```haskell
  f ((_: (_, x):_)) = x
  g [[_]] = True
  ```
Totality of Functions

- **(Total) function** maps every element in the function’s domain to an element in its codomain.
- **Partial function** maps zero or more elements in the function’s domain to an element in its codomain, and can leave some elements undefined.
- Haskell functions can be partial. For example:

```haskell
head (x:_) = x

> head []
*** Exception: Prelude.head: empty list

> "10elements" !! 10
*** Exception: Prelude.(!): index too large
```
Lambda Expressions

Functions can be constructed *without naming* the functions by using lambda expressions.

\[ \lambda x \rightarrow x + x \]

- This nameless function takes a number \( x \) and returns the result \( x + x \).

- The symbol \( \lambda \) is the Greek letter *lambda*, and is typed at the keyboard as a backslash \( \backslash \).

- In mathematics, nameless functions are usually denoted using the \( \mapsto \) symbol, as in \( x \mapsto x + x \).

- In Haskell, the use of the \( \lambda \) symbol for nameless functions comes from the *lambda calculus*, the theory of functions on which Haskell is based.
Why are Lambda's Useful?

1. Lambda expressions can be used to give a formal meaning to functions defined using currying. For example:

\[\begin{align*}
\text{add } x \ y &= x + y \\
\text{square } x &= x \times x
\end{align*}\]

means

\[\begin{align*}
\text{add } &= \ \lambda x \rightarrow (\lambda y \rightarrow x + y) \\
\text{square } &= \ \lambda x \rightarrow x \times x
\end{align*}\]
2. Lambda expressions can be used to avoid naming functions that are only referenced once. For example:

```haskell
odds n = map f [0..n-1]
    where
    f x = x * 2 + 1
```

can be simplified to

```haskell
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

3. Lambda expressions can be bound to a name (function argument)

```haskell
incrementer = \x -> x + 1
add (incrementer 5) 6
```
Case Expressions

Pattern matching need not be tied to function definitions; they also work with case expressions. For example:

1. \( \text{take } m \text{ ys} = \text{case } (m, \text{ ys}) \text{ of} \)
   \[
   (n, \_\_) \mid n \leq 0 \rightarrow [] \\
   (\_, []\) \rightarrow [] \\
   (n, x:\text{xs}) \rightarrow x : \text{take } (m-1) \text{ xs}
   \]

2. \( \text{length } [] = 0 \)
   \( \text{length } (\_:\text{xs}) = 1 + \text{length } \text{xs} \)

using a case expression and a lambda:

\( \text{length } = \lambda \text{ls} \rightarrow \text{case } \text{ls} \text{ of} \)
\[
[] \rightarrow 0 \\
(\_:\text{xs}) \rightarrow 1 + \text{length } \text{xs}
\]
Let and Where

The let and where clauses are used to create a local scope within a function. For example:

```
(1) reserved s = -- using let
    let keywords = words “if then else for while”
    relops = words “== != < > <= >=”
    elemInAny w [] = False
    elemInAny w (l:ls) = w `elem` l || elemInAny w ls
    in elemInAny s [keywords, relops]

reserved s = -- using where
    elemInAny s [keywords, relops]
    where keywords = words “if then else for while”
      ...
      elemInAny w (l:ls) = w `elem` l || elemInAny w ls

(2) unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([], [])
unzip ((a, b):rest) =
  let (as, bs) = unzip rest
  in (a:as, b:bs)
```
Let vs. Where

The `let ... in ...` is an expression, whereas `where` blocks are declarations bound to the context. For example:

```haskell
f x  -- using where block
    | cond1 x  = a
    | cond2 x  = g a
    | otherwise = f (h x a)
where a = w x
```

```haskell
f x  -- using let-in expression
    = let a = w x
      in case () of
        _ | cond1 x  -> a
        _ | cond2 x  -> g a
        _ | otherwise -> f (h x a)
```
Sections

An operator written between its two arguments can be converted into a *curried* function written before its two arguments by using parentheses. For example:

```
> 1 + 2
3
```

```
> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses. For example:

```
> (1+) 2
3
```

```
> (+2) 1
3
```

In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called *sections*. 

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

- successor function \((\text{\textbackslash }y \rightarrow 1 + y)\)
- reciprocal function \((1/)\)
- doubling function \((\ast 2)\)
- halving function \((/2)\)

Sometimes it is convenient or necessary to pass an operator as a parameter or when stating its type.
Exercises

(1) Consider a function `safetail` that behaves in the same way as `tail`, except that `safetail` maps the empty list to the empty list, whereas `tail` gives an error in this case. Define `safetail` using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function `null :: [a] → Bool` can be used to test if a list is empty.
(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:

\[
\begin{align*}
\text{True} & \land \text{True} = \text{True} \\
_ & \land _ = \text{False}
\end{align*}
\]

(4) Do the same for the following version:

\[
\begin{align*}
\text{True} & \land b = b \\
\text{False} & \land _ = \text{False}
\end{align*}
\]
Outline

- Defining Functions
- List Comprehensions
- Recursion
List Comprehensions

- A convenient syntax for defining lists
- Set comprehension - In mathematics, the comprehension notation can be used to construct new sets from old sets. E.g.,
  \[ \{(x^2,y^2) \mid x \in \{1,2,...,10\}, \ y \in \{1,2,...,10\}, \ x^2+y^2 \leq 101\} \]
- Same in Haskell: new lists from old lists
  
  \[ [(x^2, \ y^2) \mid x \gets [1..10], \ y \gets [1..10], \ x^2 + y^2 \leq 101] \]

  generates:

  \[
  [(1,1), (1,4), (1,9), (1,16), (1,25), (1,36), (1,49), (1,64), (1,81), (1,100), (4,1), (4,4), (4,9), (4,16), (4,25), (4,36), (4,49), (4,64), (4,81), (9,1), (9,4), (9,9), (9,16), (9,25), (9,36), (9,49), (9,64), (9,81), (16,1), (16,4), (16,9), (16,16), (16,25), (16,36), (16,49), (16,64), (16,81), (25,1), (25,4), (25,9), (25,16), (25,25), (25,36), (25,49), (25,64), (36,1), (36,4), (36,9), (36,16), (36,25), (36,36), (36,49), (36,64), (49,1), (49,4), (49,9), (49,16), (49,25), (49,36), (49,49), (49,64), (64,1), (64,4), (64,9), (64,16), (64,25), (64,36), (81,1), (81,4), (81,9), (81,16), (108,1)]
\]
List Comprehensions: Generators

- The expression `x <- [1..10]` is called a **generator**, as it states how to generate values for `x`.
  - generators can be infinite, e.g.,

```
> take 3 [x | x <- [1..]]
[1,2,3]
```

- Comprehensions can have **multiple** generators, separated by commas. For example:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Multiple generators are like **nested loops**, with later generators as more deeply nested loops whose variables change value more frequently.
For example:

\[
\begin{align*}
&> [(x,y) \mid y \leftarrow [4,5], \ x \leftarrow [1,2,3]] \\
&\quad [(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
\end{align*}
\]

\(x \leftarrow [1,2,3]\) is the last generator, so the value of the \(x\) component of each pair changes most frequently.
Dependent Generators

Later generators can depend on the variables that are introduced by earlier generators.

\[ \{ (x,y) \mid x \leftarrow [1..3], \ y \leftarrow [x..3] \} \]

The list \([(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]\) of all pairs of numbers \((x,y)\) such that \(x\) and \(y\) are elements of the list \([1..3]\) and \(y \geq x\).

Using a dependant generator we can define the library function that concatenates a list of lists:

\[
\text{concat} \quad :: \quad [[a]] \rightarrow [a] \\
\text{concat } xss = [x \mid xss <- xss, x <- xss]
\]

\[ \text{> concat } [[[1,2,3],[4,5]],[6]] \]

\[ [1,2,3,4,5,6] \]
Guards
List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

list all numbers x s.t. x is an element of the list [1..10] and x is even

Example: Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

> factors 15

[1,3,5,15]
A positive integer is **prime** if its only factors are 1 and itself. Hence, using **factors** we can define a function that decides if a number is prime:

```haskell
prime :: Int -> Bool
prime n = factors n == [1,n]
```

Using a guard we can now define a function that returns the list of **all primes** up to a given limit:

```haskell
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

```haskell
> primes 15
False
> primes 7
True
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```
The Zip Function

A useful library function is `zip`, which maps two lists to a list of pairs of their corresponding elements.

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

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Using `zip` we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..n], x == x']
    where n = length xs - 1
```

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```
Using zip we can define a function that returns the list of all pairs of adjacent elements from a list:

```
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x ≤ y | (x,y) <- pairs xs]
```

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
> sorted [1,2,3,4]
True
> sorted [1,3,2,4]
False
```
A triple \((x,y,z)\) of positive integers is called **pythagorean** if \(x^2 + y^2 = z^2\). Using a list comprehension, define a function

```haskell
pyths :: Int -> [(Int,Int,Int)]
```

that maps an integer \(n\) to all such triples with components in \([1..n]\). For example:

```haskell
> pyths 5
[(3,4,5),(4,3,5)]
```

```haskell
pyths n = [(x,y,z)| x<-[1..n],
y<-[1..n],
z<-[1..n],
x^2+y^2==z^2]
```
A positive integer is **perfect** if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```haskell
perfects :: Int -> [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```haskell
> perfects 500
[6,28,496]
```
The **scalar product** of two lists of integers \( \text{xs} \) and \( \text{ys} \) of length \( n \) is given by the sum of the products of the corresponding integers:

\[
\sum_{i=0}^{n-1} (\text{xs}_i \times \text{ys}_i)
\]

Using a list comprehension, define a function that returns the scalar product of two lists.

```haskell
scalarProduct :: [Int] -> [Int] -> Int
scalarProduct xs ys =
    sum[x*y|(x,y) <- zip xs ys]
```