Chapter 7 - Defining Functions, List Comprehensions
Conditional Expressions

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

```
abs :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.
Conditional expressions can be nested:

\[
\text{signum} \; :: \; \text{Int} \rightarrow \text{Int} \\
\text{signum} \; n = \text{if} \; n < 0 \; \text{then} \; -1 \; \text{else} \\
\hspace{1cm} \text{if} \; n == 0 \; \text{then} \; 0 \; \text{else} \; 1
\]

Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.
Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

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

As previously, but using guarded equations.
Guarded equations can be used to make definitions involving multiple conditions easier to read:

\[
\begin{align*}
\text{signum } n & | n < 0 &= -1 \\
& | n == 0 &= 0 \\
& | \text{otherwise} &= 1
\end{align*}
\]

Note:

- The catch all condition `otherwise` is defined in the prelude by `otherwise = True`. 
Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not False} = \text{True} \\
\text{not True} = \text{False}
\]

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

\[(\&\&) \quad :: \quad \text{Bool} \to \text{Bool} \to \text{Bool}\]

True \&\& True = True
True \&\& False = False
False \&\& True = False
False \&\& False = False

can be defined more compactly by

True \&\& True = True
_ \&\& _ = False
However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

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

Note:

- 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.

\[ [1, 2, 3, 4] \]

Means \( 1:(2:(3:(4:\text{[]}))) \).
Functions on lists can be defined using \( x:xs \) 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.
Note:

- x:xs patterns only match non-empty lists:

  ```
  > head []
  Error
  ```

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

  ```
  head x:_ = x
  ```
Integer Patterns

As in mathematics, functions on integers can be defined using $n+k$ patterns, where $n$ is an integer variable and $k>0$ is an integer constant.

\[
\text{pred} \quad :: \quad \text{Int} \to \text{Int} \\
\text{pred} \ (n+1) = n
\]

pred maps any positive integer to its predecessor.
Note:

- n+k patterns only match integers ≥ k.

> pred 0
Error

- n+k patterns must be parenthesised, because application has priority over +. For example, the following definition gives an error:

  pred n+1 = n
Lambda Expressions

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

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

the nameless function that 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 \.

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?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

\[ \text{add } x \ y = x+y \]

means

\[ \text{add} = \lambda x \rightarrow (\lambda y \rightarrow x+y) \]
Lambda expressions are also useful when defining functions that return functions as results.

For example:

\[
\text{const} :: a \rightarrow b \rightarrow a \\
\text{const } x \_ \_ = x
\]

is more naturally defined by

\[
\text{const} :: a \rightarrow (b \rightarrow a) \\
\text{const } x = \lambda\_ \_ \rightarrow x
\]
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]
```
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 \textbf{sections}. 
Why Are Sections Useful?

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

\[
\begin{align*}
(1+) & \quad - \quad \text{successor function} \\
(1/1) & \quad - \quad \text{reciprocal function} \\
(*2) & \quad - \quad \text{doubling function} \\
(/2) & \quad - \quad \text{halving function}
\end{align*}
\]
(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:

$$\text{True } && \text{True} = \text{True}$$
$$\_ \quad && \_ \quad = \text{False}$$

(4) Do the same for the following version:

$$\text{True } && \text{b} = \text{b}$$
$$\text{False } && \_ \quad = \text{False}$$
Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

\[ \{x^2 \mid x \in \{1\ldots5\}\} \]

The set \(\{1,4,9,16,25\}\) of all numbers \(x^2\) such that \(x\) is an element of the set \(\{1\ldots5\}\).
Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

\[[x^2 \mid x \leftarrow [1..5]]\]

The list [1,4,9,16,25] of all numbers \(x^2\) such that \(x\) is an element of the list [1..5].
The expression \( x \leftarrow [1..5] \) is called a \textbf{generator}, as it states how to generate values for \( x \).

Comprehensions can have \textbf{multiple} generators, separated by commas. For example:

\[
> [(x,y) | x \leftarrow [1,2,3], y \leftarrow [4,5]]
\]

\[
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
\]
Changing the order of the generators changes the order of the elements in the final list:

```plaintext
> [(x, y) | y ← [4, 5], x ← [1, 2, 3]]
[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]
```

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

\[
> [(x, y) \mid y \leftarrow [4, 5], \ x \leftarrow [1, 2, 3]]
\]

\[
[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]
\]

\[
x \leftarrow [1, 2, 3] \text{ is the last generator, so the value of the } x \text{ component of each pair changes most frequently.}
\]
Dependant 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 \(x, y\) 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:

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

For example:

```haskell
> 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.

\[ \{x \mid x \leftarrow [1..10], \text{even } x \} \]

The list \([2,4,6,8,10]\) of all numbers \(x\) such that \(x\) is an element of the list \([1..10]\) and \(x\) is even.
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]
```

For example:

```
> 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:

\[
\text{prime} :: \text{Int} \rightarrow \text{Bool} \\
\text{prime} \ n = \text{factors} \ n \ == \ [1, n]
\]

For example:

```
> prime 15
False
> prime 7
True
```
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]
```

For example:

```haskell
> 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.

\[
\text{zip} :: [a] \to [b] \to [(a,b)]
\]

For example:

\[
> \text{zip} ['a','b','c'] [1,2,3,4] \\
[(‘a’,1),(‘b’,2),(‘c’,3)]
\]
Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

\[
pairs :: [a] \rightarrow [(a,a)]
\]

\[
pairs \; xs = zip \; xs \; (tail \; xs)
\]

For example:

\[
> \text{pairs} \; [1,2,3,4]
\]

\[
[(1,2),(2,3),(3,4)]
\]
Using pairs we can define a function that decides if the elements in a list are sorted:

\[
\text{sorted} \, :: \, \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool} \\
\text{sorted } \text{xs} = \\
\quad \text{and } [x \leq y \mid (x,y) \leftarrow \text{pairs } \text{xs}]
\]

For example:

\[
> \text{sorted } [1,2,3,4] \\
\text{True} \\
> \text{sorted } [1,3,2,4] \\
\text{False}
\]
Using zip we can define a function that returns the list of all positions of a value in a list:

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

For example:

```haskell
> positions 0 [1,0,0,1,0,1,1,0]  
[1,2,4,7]
```
String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means ['a','b','c'] :: [Char].
Because strings are just special kinds of lists, any **polymorphic** function that operates on lists can also be applied to strings. For example:

```
> length "abcde"  
5

> take 3 "abcde"  
"abc"

> zip "abc" [1,2,3,4]  
[('a',1),('b',2),('c',3)]
```
Similarly, list comprehensions can also be used to define functions on strings, such as a function that counts the lower-case letters in a string:

\[
\text{lowers} :: \text{String} \rightarrow \text{Int}
\]

\[
\text{lowers } \text{x} \text{s} = \text{length } [x \mid x \leftarrow \text{x} \text{s}, \text{isLower } x]
\]

For example:

```haskell
> lowers "Haskell 98"
6
```
Exercises

(1) 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

\[
\text{pyths} :: \text{Int} \rightarrow [(\text{Int},\text{Int},\text{Int})]
\]

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

\[
> \text{pyths } 5 \\
[[(3,4,5),(4,3,5)]]
\]
(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

```
perfects :: Int → [Int]
```

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

```
> perfects 500
[6,28,496]
```
The **scalar product** of two lists of integers $xs$ and $ys$ of length $n$ is given by the sum of the products of the corresponding integers:

$$
\sum_{i=0}^{n-1} (xs_i \times ys_i)
$$

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

As we have seen, 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:

\[
\text{factorial 4} = \text{product [1..4]} = \text{product [1,2,3,4]} = 1*2*3*4 = 24
\]
Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

\[
\begin{align*}
\text{factorial } 0 &= 1 \\
\text{factorial } (n+1) &= (n+1) \times \text{factorial } n
\end{align*}
\]

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

\[
\text{factorial 3} = 3 \times \text{factorial 2} = 3 \times (2 \times \text{factorial 1}) = 3 \times (2 \times (1 \times \text{factorial 0})) = 3 \times (2 \times (1 \times 1)) = 3 \times (2 \times 1) = 3 \times 2 = 6
\]
Note:

- 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

Recursion is not restricted to numbers, but can also be used to define functions on lists.

\[
\text{product} :: \text{[Int]} \rightarrow \text{Int} \\
\text{product} \ [] = 1 \\
\text{product} \ (n:ns) = n * \text{product} \ ns
\]

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

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

\[
\text{length} : : [a] \rightarrow \text{Int} \\
\text{length} \ [\] = 0 \\
\text{length} \ (_{\text{:_}}:xs) = 1 + \text{length} \ \text{xs}
\]

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.
For example:

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

\[
\begin{align*}
\text{reverse} & : [a] \to [a] \\
\text{reverse} \ [\] & = [] \\
\text{reverse} \ (x:xs) & = \text{reverse} \ xs \ ++ \ [x]
\end{align*}
\]

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

\[
\begin{align*}
\text{reverse } [1,2,3] \\
&= \\
&= \text{reverse } [2,3] ++ [1] \\
&= (\text{reverse } [3] ++ [2]) ++ [1] \\
&= ((\text{reverse } []) ++ [3]) ++ [2]) ++ [1] \\
&= ([[] ++ [3]) ++ [2]) ++ [1] \\
&= [3,2,1]
\end{align*}
\]
Multiple Arguments

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

- Zipping the elements of two lists:

\[
\begin{align*}
\text{zip} & \quad :: \ [a] \to [b] \to [(a,b)] \\
\text{zip} \; [] \; _ & \quad = \; [] \\
\text{zip} \; _ \; [] & \quad = \; [] \\
\text{zip} \; (x:xs) \; (y:ys) & \quad = \; (x,y) : \text{zip} \; xs \; ys
\end{align*}
\]
Remove the first \( n \) elements from a list:

\[
\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a]
\]

\[
\text{drop 0 } \, \text{xs} = \text{xs}
\]

\[
\text{drop (n+1) } \, \text{[]} = \text{[]}
\]

\[
\text{drop (n+1) } (\_ : \text{xs}) = \text{drop n } \, \text{xs}
\]

Appending two lists:

\[
(++) :: [a] \rightarrow [a] \rightarrow [a]
\]

\[
\text{[]} \, \text{++ ys} = \text{ys}
\]

\[
(x : \text{xs}) \, \text{++ ys} = x \, : \, (\text{xs} \, \text{++ ys})
\]
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 $\geq$ 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:

\[
\text{qsort} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{qsort} \ [\] = []
\]
\[
\text{qsort} \ (x:xs) =
\quad \text{qsort} \ \text{smaller} \ ++ \ [x] \ ++ \ \text{qsort} \ \text{larger}
\]
\[
\text{where}
\]
\[
\text{smaller} = [a \mid a \leftarrow xs, a \leq x]
\]
\[
\text{larger} = [b \mid b \leftarrow xs, b > x]
\]

Note:

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

\[
q \ [3,2,4,1,5] \\
q \ [2,1] \quad ++ \ [3] \quad ++ \quad q \ [4,5] \\
q \ [1] \quad ++ \ [2] \quad ++ \quad q \ [\ ] \\
q \ [\ ] \quad ++ \ [4] \quad ++ \quad q \ [5] \\
[1] \\
[\ ] \\
[\ ] \\
[5]
\]
(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:
  
  \[
  \text{and} :: [\text{Bool}] \rightarrow \text{Bool}
  \]

- Concatenate a list of lists:
  
  \[
  \text{concat} :: [[\text{a}]] \rightarrow [\text{a}]
  \]
- Produce a list with n identical elements:

\[
\text{replicate} :: \text{Int} \to a \to [a]
\]

- Select the nth element of a list:

\[
\text{ (!!!) } :: [a] \to \text{Int} \to a
\]

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

\[
\text{elem} :: \text{Eq a } \Rightarrow a \to [a] \to \text{Bool}
\]
(2) Define a recursive function

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

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

```haskell
> 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:

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