Chapter 4 - Defining Functions
Conditional Expressions

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

\[
\text{abs :: Int} \rightarrow \text{Int} \\
\text{abs } n = \text{if } n \geq 0 \text{ then } n \text{ else } -n
\]

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

\[
\text{signum} \; :: \; \text{Int} \rightarrow \text{Int} \\
\text{signum} \; n = \begin{cases} 
-1 & \text{if } n < 0 \\
0 & \text{if } n == 0 \\
1 & \text{else}
\end{cases}
\]

Note:

In Haskell, conditional expressions must \underline{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.

| abs n | n ≥ 0     = n |
|       | otherwise = -n |

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n &lt; 0 )</td>
<td>-1</td>
</tr>
<tr>
<td>( n = 0 )</td>
<td>0</td>
</tr>
<tr>
<td>otherwise</td>
<td>1</td>
</tr>
</tbody>
</table>

Note:

\( \xi \) 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.

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

\[
\begin{align*}
\text{not False} &= \text{True} \\
\text{not True} &= \text{False}
\end{align*}
\]

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

```
(&&) :: Bool → Bool → 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:

ζ 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 \( 1:(2:(3:(4:[]))) \).
Functions on lists can be defined using \texttt{x:xs} patterns.

\begin{minipage}{\textwidth}
\begin{minipage}{.5\textwidth}
\begin{align*}
\text{head} & :: [a] \to a \\
\text{head } (\text{x:}_\text{)} &= \text{x} \\
\text{tail} & :: [a] \to [a] \\
\text{tail } (_\text{:xs}) &= \text{xs}
\end{align*}
\end{minipage}
\begin{minipage}{.5\textwidth}
\begin{itemize}
\item head and tail map any non-empty list to its first and remaining elements.
\end{itemize}
\end{minipage}
\end{minipage}
Note:

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

> head []
Error

ζ 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.

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

pred maps any positive integer to its predecessor.
Note:

ζ n+k patterns only match integers \( \geq k \).

> pred 0
Error

ζ 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 \texttt{lambda}, and is typed at the keyboard as a backslash \( \backslash \).

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

In Haskell, the use of the \( \lambda \) symbol for nameless functions comes from the \texttt{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

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

For example:

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

is more naturally defined by

\[
\text{const} :: a \to (b \to a)
\]
\[
\text{const } x = \lambda _\to x
\]
Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:

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

can be simplified to

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```
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
- reciprocation function
- doubling function
- halving function
Exercises

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

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

Hint: the library function \texttt{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:

<table>
<thead>
<tr>
<th>True &amp;&amp; True = True</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ &amp;&amp; _ = False</td>
</tr>
</tbody>
</table>

(4) Do the same for the following version:

<table>
<thead>
<tr>
<th>True &amp;&amp; b = b</th>
</tr>
</thead>
<tbody>
<tr>
<td>False &amp;&amp; _ = False</td>
</tr>
</tbody>
</table>