Chapter 4 - Defining Functions
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:

```haskell
signum :: Int → Int
signum n = if n < 0 then -1 else
            if n == 0 then 0 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.

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

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

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

Note:

- The catch all condition \textbf{otherwise} is defined in the prelude by \textbf{otherwise} = True.
Pattern Matching

Many functions have a particularly clear definition using \texttt{pattern matching} on their arguments.

\begin{verbatim}
not :: Bool → Bool
not False = True
not True  = False
\end{verbatim}

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

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

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>False</td>
</tr>
</tbody>
</table>

can be defined more compactly by

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>_</td>
<td>True</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>False</td>
</tr>
</tbody>
</table>
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:

\[
_ \quad \&\& \quad _ \quad = \quad \text{False}
\]

\[
\text{True} \quad \&\& \quad \text{True} \quad = \quad \text{True}
\]

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

\[
b \quad \&\& \quad b \quad = \quad b
\]

\[
_ \quad \&\& \quad _ \quad = \quad \text{False}
\]
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.

\textbf{head} :: [a] \to a  
\textbf{head} (x:_) = x

\textbf{tail} :: [a] \to [a]  
\textbf{tail} (\_:xs) = xs

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

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

```haskell
> head []
*** Exception: empty list
```

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

```haskell
head x:_ = x
```
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 \textit{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 \textit{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:

\[
\text{odds } n = \text{map } f \ [0..n-1] \\
\text{where} \\
f \ x = x^2 + 1
\]

can be simplified to

\[
\text{odds } n = \text{map } (\lambda x \to x^2 + 1) \ [0..n-1]
\]
Operator 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
> (+) 1 2
```

3
3
This convention also allows one of the arguments of the operator to be included in the parentheses. 

For example:

\[
\begin{align*}
> & \ (1+) \ 2 \\
> & \ 3 \\
> & \ (+2) \ 1 \\
> & \ 3
\end{align*}
\]

In general, if $\odot$ is an operator then functions of the form $(\odot)$, $(x\odot)$ and $(\odot y)$ 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 `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:

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

(4) Do the same for the following version:

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