Chapter 4 - Defining Functions

Pattern Matching

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

```
not :: Bool →
Bool
not False = True
not True  =
False
```

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching.

```
(&&) :: 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
```

- The underscore symbol _ is a **wildcard** pattern that matches any argument value.

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
```

- 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
```

Tuple Patterns

A tuple of patterns is a pattern.

```
fst :: (a,b) → a
fst (x,_) = x

snd :: (a,b) → a
snd (_,y) = y
```

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:[]))).

```
test :: [Char] → Bool
test [‘a’,_,_] = True
test _ = False
```

```
test (‘a’:_) = True
test _ = False
```

Functions on lists can be defined using **x:xs** patterns.

```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

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

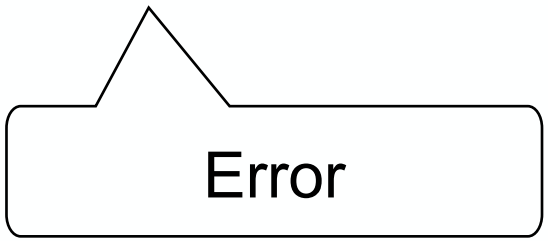
Note:

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

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

- `x:xs` patterns must be **parenthesised**, because application has priority over (:).

```
head x:_ = x
```



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:

```
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**.

```
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:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Note:

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

Lambda Expressions

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

```
λx → x + x
```

the nameless function that takes a number x and returns the result x + x.

Note:

- The symbol λ is typed at the keyboard as a **backslash** \.

- In mathematics, nameless functions are usually denoted using the ↦ symbol, as in x ↦ x + x.

- In Haskell, the use of the λ symbol for nameless functions comes from the **lambda calculus**.

Why Are Lambda's Useful?

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

```
add :: Int → Int → Int
add x y = x + y
```

corresponds to

```
add :: Int → (Int → Int)
add = λx → (λy → x + y)
```

```
\x -> (\y -> x+y)
```

Why Are Lambda's Useful?

Defining functions that return functions as result

```
const :: a -> b -> a
const x _ = x + y
```

corresponds to

```
const :: a → (b → a)
const x = λ_ → 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 (λx → 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.

```
> 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 ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called **sections**.

Why Are Sections Useful?

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

`(1+)` - successor function

`(1/)` - reciprocation function

`(*2)` - doubling function

`(/2)` - halving function