PROGRAMMING IN HASKELL

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:

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

Note:

- In Haskell, conditional expressions *must* 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 <u>otherwise</u> is defined in the prelude by otherwise = True.

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. 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:

- 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 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 []
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.

```
pred      :: Int → Int
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.

Note:

- The symbol λ is the Greek letter lambda, and 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 λ symbol comes from the lambda calculus, the theory of functions on which Haskell is based.

Why Are Lambdas Useful?

Lambda expressions can be used to give a formal meaning to functions defined using <u>currying</u>.

For example:

```
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 <u>functions as results</u>.

For example:

```
const    :: a → b → a
const x _ = x
```

is more naturally defined by

```
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]
```

Sections

An operator written <u>between</u> its two arguments can be converted into a curried function written <u>before</u> 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 ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called <u>sections</u>.

Why Are Sections Useful?

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

`(1+)`	- successor function
`(1/)`	- reciprocation function
`(*2)`	- doubling function
`(/2)`	- halving function

Exercises

(1) Consider a function <u>safetail</u> 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] $\rightarrow$ 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
```