

# Defining Functions

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
```

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

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`.

# 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, as it avoids evaluating the second argument if the first argument is False:

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

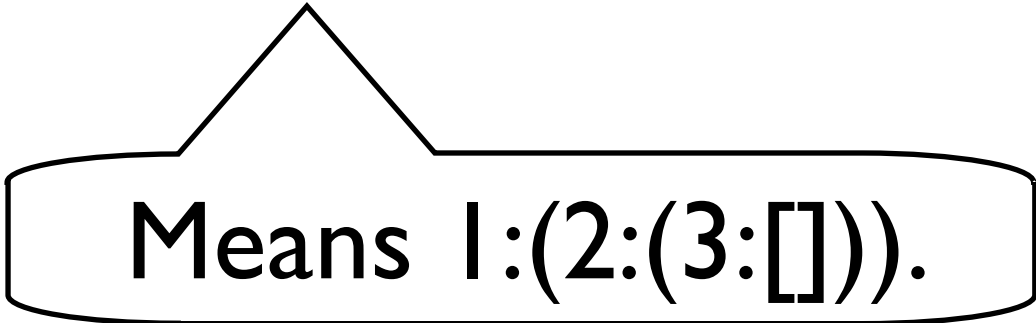
The underscore symbol `_` is the wildcard pattern that matches any argument value.



# List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called “cons” that adds a new element to the start of a list.

[1, 2, 3]



Means 1:(2:(3:[])).

The cons operator can also be used in patterns, in which case it destructs a non-empty list.

```
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.

# Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression.

$\lambda x \rightarrow x+1$

$\backslash x \rightarrow x+1$

Haskell syntax

The nameless function that takes a number  $x$  and returns the result  $x+1$ .

# 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,

$$\text{compose } f \ g \ x = f \ (g \ x)$$

is more naturally defined by

$$\text{compose } f \ g = \lambda x \rightarrow f \ (g \ x)$$

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:

- (i) a conditional expression;
- (ii) guarded equations;
- (iii) pattern matching.

Hint:

The prelude function  $\text{null} :: [a] \rightarrow \text{Bool}$  can be used to test if a list is empty.

```
safeTail : [a] -> [a]
```

```
safeTail xs = if null xs then [ ] else  
tail xs
```

```
safeTail xs | null xs = [ ]  
            | otherwise = tail xs
```

```
safeTail [ ] = [ ]  
safeTail (x : xs) = xs
```