Programming In Haskell Chapter 4

CS 1JC3

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
```

or

```
abs   :: Int -> Int
abs n = if n >= 0
           then n
           else -n
```

Conditional Expressions

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

```haskell
abs   :: Int -> Int
abs n = if n >= 0 then n else -n
```

or

```haskell
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

Nested Expressions

Conditional expressions can be nested (put inside another conditional expression).

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

Nested Expressions

Conditional expressions can be nested (put inside another conditional expression).

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

signum takes an integer n, and returns -1 if n is negative, 0 if n is equal to zero, and 1 if n is positive.

Note: In Haskell, conditional expressions must always have an else branch

Guarded Equations

As an alternative to the if expression, functions can also be defined using guarded equations.

```
abs n | n >= 0     = n
      | otherwise = -n
```

abs does the same thing as was previously defined, but is more neat looking. It's important to make your code easy to read.

Guarded Equations

Guarded equations can make definitions involving multiple conditions much easier to read.

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

Note: The catch condition otherwise is defined in the prelude as 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
```

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 is defined in the standard prelude. It changes False to True and True to False.

Pattern Matching

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
```

Pattern Matching

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
```

This can be defined more compactly using the wildcard operator (_), which means any value.

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

Pattern Matching

- Patterns are matched in order. For example, the following definition returns False no matter what.

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

Pattern Matching

- Patterns are matched in order. For example, the following definition returns False no matter what.

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

- Patterns may not repeat variables. The following gives an error:

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

List Patterns

Internally, every non empty list is constructed by use of an operator (:) called cons that adds an element to the start of a list.

```
[1,2,3,4] == 1:(2:(3:(4:[])))
```

List Patterns

Internally, every non empty list is constructed by use of an operator (:) called cons that adds an element to the start of a list.

```
[1,2,3,4] == 1:(2:(3:(4:[])))
```

Functions can be defined using $x : xs$ patterns:

```
head        :: [a] -> a
head (x:_) = x

tail        :: [a] -> [a]
tail (_:xs) = xs
```

Note: $x : xs$ patterns need to be put in brackets

List Patterns

- Note: $x : xs$ patterns don't work on empty lists

```
head []
  Error
```

List Patterns

- Note: $x : xs$ patterns don't work on empty lists

```
head []
  Error
```

- We can fix this using pattern matching

```
head     :: [a] -> a
head []     = []
head (x:_) = x
```

Integer Patterns

As in mathematics, functions on integers can be defined using n+k patterns, where n is an integers variable and $k > 0$ is an integer constant.

```
pred        :: Int -> Int
pred (n+1) = n
```

Integer Patterns

As in mathematics, functions on integers can be defined using n+k patterns, where n is an integers variable and $k > 0$ is an integer constant.

```
pred        :: Int -> Int
pred (n+1) = n
```

Note: n+k patterns only work with $k > 0$.

```
pred 0
  Error
```

Lambda Expressions

Functions can be constructed without a name by using lamda expressions

```
\x -> x + x
```

Lamda expressions are useful when defining functions that return a function as a result, and giving more formal meaning to currying, for example:

```
add x y = x+y
```

Lambda Expressions

Functions can be constructed without a name by using lamda expressions

```
\x -> x + x
```

Lamda expressions are useful when defining functions that return a function as a result, and giving more formal meaning to currying, for example:

```
add x y = x+y
```

means

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

Map and Lambda Expressions

The standard Prelude defines a function called map for working on lists. It takes a function and a list and applies that function on each element in a list, returning another list. For example:

```
add1      :: Int -> Int
add1 x    = x + 1

sum1      :: [Int] -> [Int]
sum1 xs = map add1 xs
```

So,

```
sum1 [1,2,3] == ...
```

Map

The standard Prelude defines a function called map for working on lists. It takes a function and a list and applies that function on each element in a list, returning another list. For example,

```
add1      :: Int -> Int
add1 x   = x + 1

sum1      :: [Int] -> [Int]
sum1 xs = map add1 xs
```

So,

```
sum1 [1,2,3] == [2,3,4]
```

Map and Lambda Expressions

Defining a function only to be used in another function seems a little unnecessary. This is where lambda expressions come in. sum1 can be more efficiently defined as:

Map and Lambda Expressions

Defining a function only to be used in another function seems a little unnecessary. This is where lambda expressions come in. sum1 can be more efficiently defined as:

```
sum1 xs = map (\x -> x + 1) xs
```

This way, we don't have to define add1, provided it is only referenced once.

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
```

We can use sections to create functions concisely. For example, the add1 function can be redefined as:

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
```

We can use sections to create functions concisely. For example, the add1 function can be redefined as:

```
add1 = (1+)
```

Note: specifying that add1 takes an argument isn't necessary, because $(+1)$ returns a function that takes an argument.

Creating Functions With Sections

Useful functions constructed in a simple way using sections.

```haskell
succs    :: Integer -> Integer
succs    = (1+)      -- successor function

recipr   :: Float -> Float
recipr   = (1/)      -- reciprocation function

double   :: Integer -> Integer
double   = (*2)      -- doubling function

halve    :: Float -> Float
halve    = (/2)      -- halving function
```

Exercise 1

Consider a function safetail that behaves in the same way as tail, except that safetail doesn't give an error when passed an empty list, it just returns the empty list. Define safetail using

- a conditional expression
- guarded equations
- pattern matching

Hint: the library function null :: [a] − > Bool can be used to test if a list is empty, and you can use tail.

Solution 1

```haskell
-- Conditional Expression
safetail    :: [a] -> [a]
safetail xs = if null xs
                 then xs
                 else drop 1 xs
```

Solution 1

```haskell
-- Conditional Expression
safetail    :: [a] -> [a]
safetail xs = if null xs
                 then xs
                 else drop 1 xs

-- Guarded Equations
safetail    :: [a] -> [a]
safetail xs | null xs    = xs
            | otherwise = drop 1 xs
```

Solution 1

```haskell
-- Conditional Expression
safetail     :: [a] -> [a]
safetail xs = if null xs
                 then xs
                 else drop 1 xs

-- Guarded Equations
safetail     :: [a] -> [a]
safetail xs | null xs    = xs
            | otherwise = drop 1 xs

-- Pattern Matching
safetail     :: [a] -> [a]
safetail []      = []
safetail (_:xs) = xs
```

Exercise 2

Give two more possible definitions for the logical operator (||) using pattern matching.

```
-- Definition 1
False || False = False
False || True  = True
True  || False = True
True  || True  = True
```

(||) represents a Boolean OR. If either of the arguments passed to it equal True, it returns True. So only False OR False equals False.

Solution 2

```
-- Definition 2
False || False = False
_     ||  _    = True
```

Solution 2

```haskell
-- Definition 2
False || False = False
_     ||  _    = True

-- Definition 3
False || b     = b
True  || _     = True
```

Exercise 3

Redefine the following version of (&&) using conditionals rather than patterns:

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

(&&) represents a Boolean AND. It only returns True if both arguments are True. So True AND True equals True, and every other combination equals False.

Solution 3

```
a && b = if a
            then if b then True else False
            else False
```

Exercise 4

Redefine the following function mult using lambda expressions.
Include it's new type signature with brackets.

```
mult     :: Integer -> Integer -> Integer -> Integer
mult x y z = x * y * z
```

Solution 4

```
mult     :: Integer -> (Integer -> (Integer -> Integer))
mult = \x -> (\y -> (\z -> x * y * z))
```

Exercise 5

Define the following functions in a file with their type signatures

- stack takes the first element of a list and puts it on the end of a list
- range takes a numerical value and checks to see if it is between 0 and 10, returns True if it is False otherwise
- addc takes a Char and a String and adds the Char to the beginning of the String
- halves takes a list and divides each element in the list by two

Solution 5

```
stack  :: [a] - > [a]
stack (x:xs) = xs ++ [x]
```

Solution 5

```
stack  :: [a] - > [a]
stack (x:xs) = xs ++ [x]

range :: (Num a,Ord a) => a -> Bool
range x = x >= 0 && x <= 10
```

Solution 5

```
stack  :: [a] - > [a]
stack (x:xs) = xs ++ [x]

range :: (Num a,Ord a) => a -> Bool
range x = x >= 0 && x <= 10

addc :: Char -> [Char] -> [Char]
addc c ss = c:ss
```

Solution 5

```
stack  :: [a] - > [a]
stack (x:xs) = xs ++ [x]

range :: (Num a,Ord a) => a -> Bool
range x = x >= 0 && x <= 10

addc :: Char -> [Char] -> [Char]
addc c ss = c:ss

halves :: (Fractional a) => [a] -> [a]
halves xs = map (/2) xs
```