Defining Functions

COMP2209 - Programming III

Dr Julian Rathke

These slides use modified content from Graham Hutton’s slides for the module textbook Programming in Haskell
Writing a new function - first steps

- The basic building blocks of Functional Programming are functions.
- We create new functions using combinations of existing functions.
- When defining a function in Haskell, the first consideration should always be what type the function should be.
- Take care to consider class constraints for polymorphic functions (do you use, +, ==, > etc)?
- Within a script, good practice dictates that we specify the type of every function.
  - This provides a form of documentation for your code.
Basic Form

We have already seen the basic form of a function definition:

```
function-name arg1 ... argN = body-expression
```

For example:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

And we could figure out that we can define functions by cases using conditionals:

```
abs :: (Ord a, Num a) => a -> a
abs n = if n >= 0 then n else -n
```
Guarded Equations

- Conditional statements can get very messy, especially when nested.
- A far far preferable way to define functions by cases on their values is to use **guarded equations**
- These allow you to define functions directly by listing predicates for when each case is to be applied.

\[
abs :: (\text{Ord} \ a, \text{Num} \ a) \Rightarrow a \rightarrow a \\
abs \ n \mid n \geq 0 \ = \ n \\
\mid \text{otherwise} \ = \ -n
\]

You are allowed as many guards as you like.

Guards are evaluated top-down

\mid \text{is pronounced "such that"
Error Cases

Haskell, like Java, features exceptions and exception handling but makes a distinction between an error that is not expected to be handled and an exception proper.

This is similar to the distinction between Checked and Unchecked Exceptions in Java.

For now, we’ll just use the function

\[
\text{error} :: \text{String} \to a
\]

as a simple means of reporting an error. This is not an exception and should not be caught. Note the type here. An error is a value of any type.
Example: quadratic roots

We all know the simple formula for calculating roots of a quadratic equation:

\[ \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

Let’s write a function

```
quadroots :: Float → Float → Float → String
```

that returns a string message containing the roots. There are a few cases to consider:

When a is 0 there is no solution.
When \( b^2 - 4ac \) is 0 there is only one root
Otherwise there are two roots
Example: quadratic roots - guarded

Obviously this is screaming "guarded equations" at us.

quadroots :: Float → Float → Float → String
quadroots a b c | a == 0 = ...
                | b*b-4*a*c == 0 = ...
                | otherwise = ...

- So we just need to decide what to do in each case.
  - For a == 0 clearly we can return an error.
  - For the second case we use show from the Show class to convert the single root value to a string along with a message.
  - For the final case we do the same but convert both roots.
Example: quadratic roots

```haskell
quadroots :: Float -> Float -> Float -> String
quadroots a b c | a == 0         = error "Not quadratic"
                | b*b-4*a*c == 0 = "Root is " ++ show (-b/2*a)
                | otherwise    = "Upper Root is "
                               ++ show ((-b + sqrt(abs(b*b-4*a*c)))/2*a))
                               ++ "and Lower Root is "
                               ++ show ((-b - sqrt(abs(b*b-4*a*c)))/2*a))
```

This isn’t particularly nice. We are using the same calculation expressions repeatedly. Apart from being inefficient, it’s hard to read.

Let’s tidy it up by using a where clause.
Example: quadratic roots with local defns

```
quadroots :: Float -> Float -> Float -> String
quadroots a b c | a==0             = error "Not quadratic"
    | discriminant==0 = "Root is " ++ show centre
    | otherwise       = "Upper Root is " ++ show (centre + offset)
          ++ " and Lower Root is " ++ show (centre - offset)
    where discriminant = abs(b*b-4*a*c)
          centre = -b/2*a
          offset = sqrt(discriminant/2*a)
```

Local definitions using where clauses help tidy the code but can also aid readability by providing meaningful names for subexpressions.
Example: let local definitions

An alternative to the where clause is to use a local let block. The general form for these is:

```
let name = expression in expression
```

To write the above example this way we could use:

```haskell
quadroots a b c =
  let discriminant = abs(b^2-4*a*c) in
  let centre = -b/2*a in
  let offset = sqrt (discriminant/2*a) in
  let quadcases a b c | a==0             = error "Not quadratic"
  quadcases a b c | discriminant==0  = "Root is " ++ show centre
  quadcases a b c | otherwise = "Upper Root is "
                   ++ show (centre + offset)
                   ++ " and Lower Root is "
                   ++ show (centre - offset)
  in quadcases a b c
```

This is perhaps less nice than previous slide
Pattern Matching

• Guarded equations provide a means of defining functions in very specific cases based on predicates over the argument values to a function.

• A simpler form of this is to consider cases where there is a single particular value supplied as an argument, or where the value has a certain shape irrespective of the specific values making up that shape
  • e.g. a List with at least three elements

• For cases such as these we should define our functions using Pattern Matching

• This allows for very direct definitions of functions by simple cases.
Pattern Matching Examples

not :: Bool \rightarrow Bool
not False = True
not True = False

Basic patterns: fixed values, **unique** variables, wildcard

nonzero :: Int \rightarrow Bool
nonzero 0 = False
nonzero _ = True

This is a really direct definition that simply states what the function does on each element in Bool

This uses a wildcard pattern in the second case. Patterns are evaluated top-down.

This uses a variable pattern in the first case. When applied, if the first argument evaluates to True, the second argument will be bound to b
Structure Patterns

- For compound types where the values are built using value constructors such as Lists and Tuples we can pattern match using those same constructors.
- The pattern `(x, y)` matches any pair of values.
- The pattern `(x, True, _)` matches any 3-tuple whose middle value is the boolean value True, the first value of the matching 3-tuple is then bound to `x`.
- For example:

  
  ```
  fst :: (a,b) -> a
  fst (x,_) = x
  
  snd :: (a,b) -> b
  snd (_,y) = y
  ```

  We can also nest tuple patterns e.g. `((_, 4), (True, z))`
List Patterns

- Similarly to tuples we can use the list constructor (:) to build patterns to match lists
- The pattern [] matches the empty list
- The pattern (x : y : []) matches a list with exactly two elements
- The pattern (x : xs) matches a list with at least one element
- You’ll notice that I use parentheses for List patterns. This is necessary as application has priority over (:
- e.g. head x:_ = x does not parse in GHC
- For example:

```haskell
last [] = error "Empty List"
last (x:[]) = x
last (x:xs) = last xs
```
Composite Patterns

Where pattern matching gets really powerful is where you combine patterns between multiple arguments.

```haskell
fetch :: Int -> [a] -> a
fetch _ [] = error "Empty List"
fetch 0 (x:_):xs = x
fetch n (_:xs) = fetch (n-1) xs
```

This really relies on the top-down evaluation order of the patterns. Be careful with this as it is easy to miss cases. When in doubt, list all cases explicitly. e.g.

```haskell
fetch 0 [] = error "Empty List"
fetch n [] = error "Empty List"
fetch 0 (x:_):xs = x
fetch n (_:xs) = fetch (n-1) xs
```
String Patterns

- Strings are just Lists of Char values
- As such, List patterns may be used to match String values also
- The pattern ( _ : ’a’ : _ ) matches Strings whose second character is ’a’.
- The pattern [ ] matches the empty String
- For example

```haskell
toDoubleStr :: String → String
toDoubleStr [ ] = [ ]
toDoubleStr (c:cs) = toUpper c : toUpperStr cs
```
Lambda Expressions

• When discussing Curried functions above we noted that functions may return functions as values.
• For example, add 5 returns a function that adds 5 to whatever it is applied to.
• It is useful to think of this function as a value in its own right but how do we refer to it?
• A lambda expression does the job. We can think of these as anonymous functions.
• The unnamed add 5 function is represented as

$$\lambda x \rightarrow \text{add } 5 \ x$$

The $\lambda$ signifies that this is a function value and the $x$ is a named argument to the function the expression to the right of $\rightarrow$ is the body of the function.
What’s that funny symbol?

• That is the greek letter (lower-case) lambda: \( \lambda \)

• It comes from the mathematical calculus called Lambda calculus that forms the backbone of Haskell

• You can use it in Haskell programs by simply using the backslash character followed by a variable name

• For example, a Haskell rendering of the (curried) add function is

\[
\text{add} :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a \\
\text{add} = \backslash x \rightarrow (\backslash y \rightarrow x + y)
\]

• Lambda expressions are really useful when you want to define a function that returns a function as a result.

\[
\text{const} :: a \rightarrow b \rightarrow a
\]

\[
\text{const } x \_ = x
\]

is better as

\[
\text{const} :: a \rightarrow b \rightarrow a
\]

\[
\text{const } x = \lambda \_ \rightarrow x
\]
Operator Sections

• We have already seen that an operator written between its two arguments can be converted into a curried function by using parentheses.
  • e.g. $5 + 7$ can equivalently be written as $(+)(5, 7)$
• This is a Haskell convention. Interestingly, this allows one of the arguments to be included within the parentheses.
  • e.g. $(5+)(7)$ is the “add 5” function applied to 7 and is equivalent to $5 + 7$
  • e.g. $(+)(7)(5)$ is the “add 7” function applied to 5 and is equivalent to $5 + 7$
• In general, for any operator $\circ$ we can form the functions $(\circ)(\cdot)$, $(\cdot)(\circ)$ and $(\circ)(\cdot)$ - these are known as sections.
Quick Quiz - Common Sections

Think of informative names for these sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Name?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+)</td>
<td>???</td>
</tr>
<tr>
<td>(*3)</td>
<td>???</td>
</tr>
<tr>
<td>(/2)</td>
<td>???</td>
</tr>
<tr>
<td>(2^)</td>
<td>???</td>
</tr>
<tr>
<td>(1/)</td>
<td>???</td>
</tr>
</tbody>
</table>
YOUR QUESTIONS

Next Lecture:
List Comprehensions