Haskell
Syntax in Functions

http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

September 26, 2019

Syntax in Functions

Pattern Matching

Pattern Matching

Pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns.

When defining functions, you can define separate function bodies for different patterns.

This leads to really neat code that's simple and readable.

You can pattern match on any data type: numbers, characters, lists, tuples, etc.

Pattern Matching

```haskell
lucky :: (Integral a) => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

```
ghci> lucky 7
"LUCKY NUMBER SEVEN!"
ghci> lucky 1
"Sorry, you're out of luck, pal!"
ghci> lucky 2
"Sorry, you're out of luck, pal!"
ghci>
```

Pattern Matching

```haskell
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe x = "Not between 1 and 3"
```

```
ghci> sayMe 1
"One!"
ghci> sayMe 2
"Two!"
ghci> sayMe 3
"Three!"
ghci> sayMe 4
"Not between 1 and 3"
ghci>
```

Pattern Matching

```haskell
sayMe' :: (Integral a) => a -> String
sayMe' x = "Not between 1 and 3"
sayMe' 1 = "One!"
sayMe' 2 = "Two!"
sayMe' 3 = "Three!"
```

```
ghci> sayMe' 1
"Not between 1 and 3"
ghci> sayMe' 2
"Not between 1 and 3"
ghci> sayMe' 3
"Not between 1 and 3"
ghci> sayMe' 4
"Not between 1 and 3"
ghci>
```

Pattern Matching

```haskell
sayMe' :: (Integral a) => a -> String
sayMe' x = "Not between 1 and 3"
sayMe' 1 = "One!"
sayMe' 2 = "Two!"
sayMe' 3 = "Three!"
```

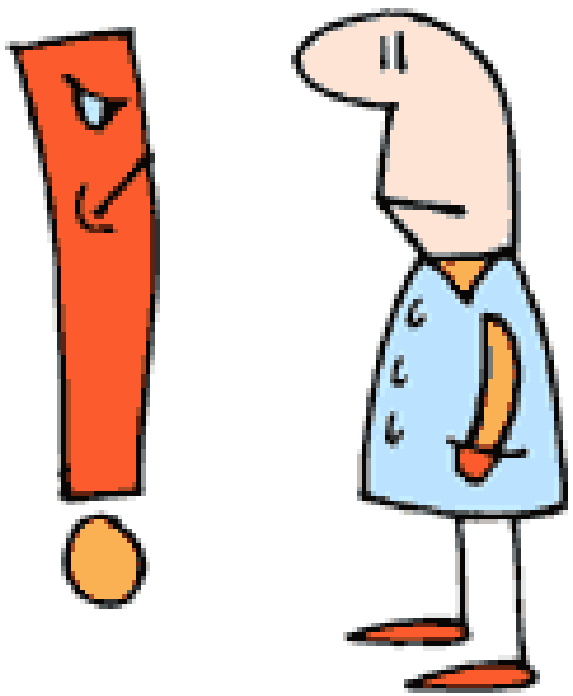
```
ghci> :l "sayme"
[1 of 1] Compiling Main ( sayme.hs, interpreted )

sayme.hs:8:1: Warning:
    Pattern match(es) are overlapped
    In an equation for 'sayMe'':
        sayMe' 1 = ...
        sayMe' 2 = ...
        sayMe' 3 = ...
Ok, modules loaded: Main.
ghci>
```

Pattern Matching

Defining `factorial` recursively

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```



Pattern Matching

"Before you examine me, promise me nothing's wrong."

Pattern Matching

Failed!

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'c'
"Cecil"
ghci> charName 'd'
"*** Exception: charName.hs:(2,1)-(4,22):
Non-exhaustive patterns in function charName
ghci>
```

Pattern Matching

Tuples

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors' :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors' (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

The type of `addVectors` (in both cases) is

```
addVectors :: (Num a) => (a, a) -> (a, a) - > (a, a)
```

so we are guaranteed to get two pairs as parameters.

Pattern Matching

Tuples

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors' :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors' (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

```
ghci> addVectors (1,2) (3,4)
(4,6)
ghci> addVectors' (1,2) (3,4)
(4,6)
ghci> addVectors' (1,2)
<interactive>:49:1:
    No instance for (Show ((a0, a0) -> (a0, a0)))
      arising from a use of 'print'
    In a stmt of an interactive GHCi command: print it
ghci>
```

Pattern Matching

One, two, ...

`fst` and `snd` extract the components of pairs. But what about triples? Well, there are no provided functions that do that but we can make our own.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Pattern Matching

Pattern Match in List Comprehensions

```
ghci> let xs = [(1,2), (3,4), (5,6), (7,8)]
ghci> [a+b | (a,b) <- xs]
[3,7,11,15]
ghci> let xs = [(1,2,3), (4,5,6), (7,8,9)]
ghci> [c-a+b | (a,b,c) <- xs]
[4,7,10]
ghci> let xs = [(1,(2,3)),(4,(5,6)),(7,(8,9))]
ghci> [(a,(b+c)) | (a,(b,c)) <- xs]
[(1,5),(4,11),(7,17)]
ghci> let xs = [((1,2),(3,4)),((5,6),(7,8))]
ghci> [(a+b,c+d) | ((a,b),(c,d)) <- xs]
[(3,7),(11,15)]
ghci>
```

Pattern Matching

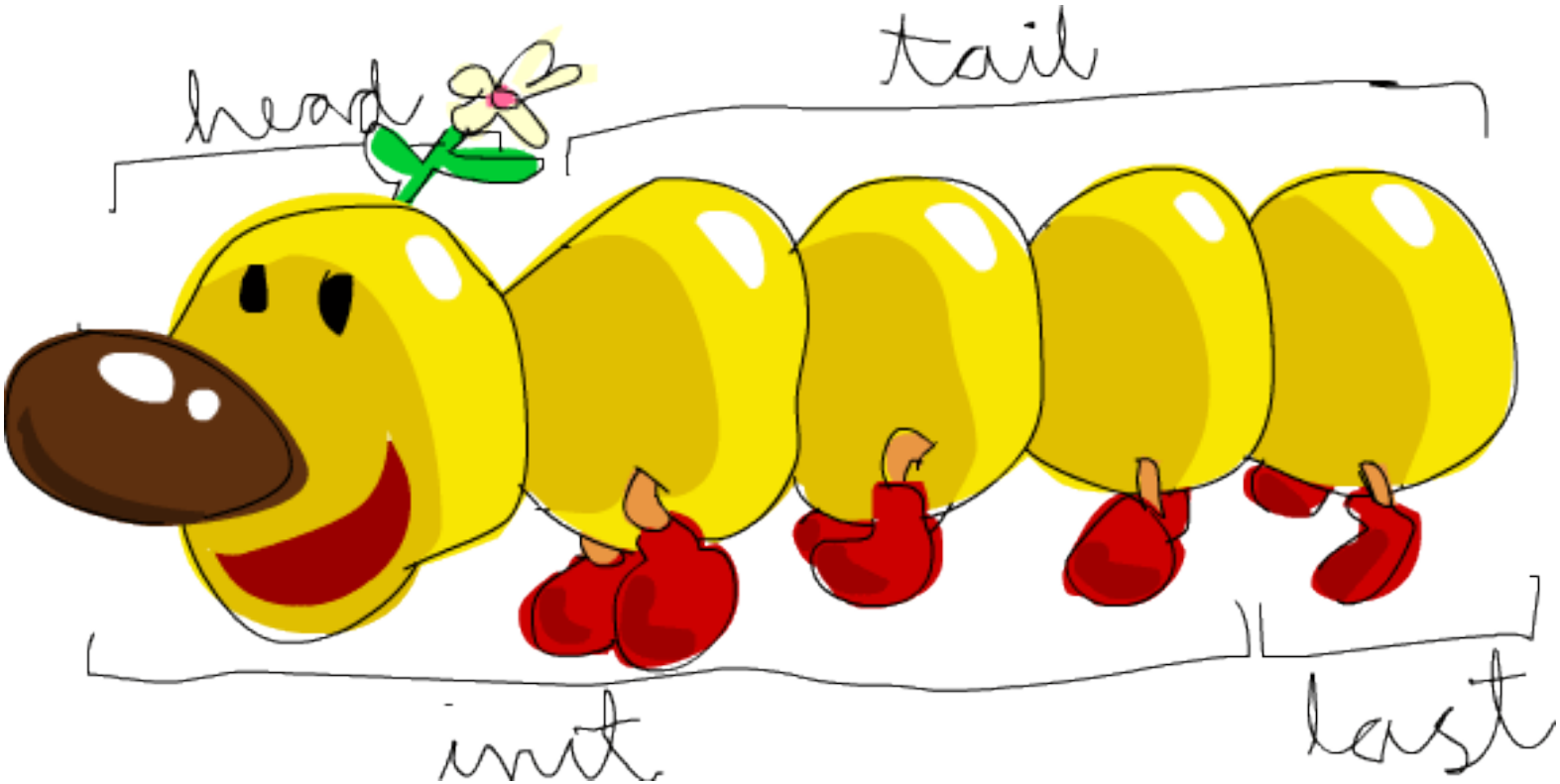
Pattern Match in List Comprehensions

Should a pattern match fail, it will just move on to the next element.

```
Prelude> let xs = [(1,'a'),(2,'b'),(1,'c'),(3,'d'),(1,'e')]
Prelude> [y | (1, y) <- xs]
"ace"
Prelude>
```

Pattern Matching

Pattern Match in Lists



Pattern Matching

Pattern Match in Lists

Lists themselves can also be used in pattern matching.

You can match with the empty list `[]` or any pattern that involves `:` and the empty list.

But since `[1,2,3]` is just syntactic sugar for `1:2:3:[]`, you can also use the former pattern.

A pattern like `x:xs` will bind the head of the list to `x` and the rest of it to `xs`, even if there's only one element so xs ends up being an empty list.

If you want to bind, say, the first three elements to variables and the rest of the list to another variable, you can use something like `x:y:z:zs`. It will only match against lists that have three elements or more.

Pattern Matching

Pattern Match in Lists

```haskell
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

```
ghci> head' []
*** Exception: Can't call head on an empty list, dummy!
ghci> head' [1]
1
ghci> head' [1,2]
1
ghci> head' [1,2,3]
1
ghci> head' "Hello!"
'H'
```

Pattern Matching

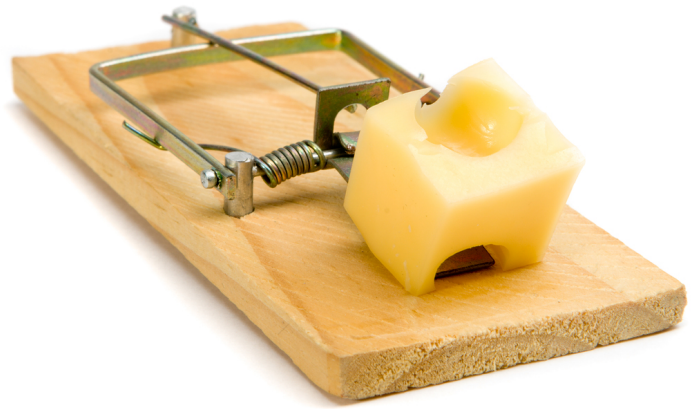
Pattern Match in Lists

```
tell :: (Show a) => [a] -> String
tell []       = "The list is empty"
tell (x:[])   = "The list has one element: " ++
                show x
tell (x:y:[]) = "The list has two elements: " ++
                show x                        ++
                " and "                       ++
                show y
tell (x:y:_)  = "This list is long. "           ++
                "The first two elements are: " ++
                show x                        ++
                " and "                       ++
                show y
```

Pattern Matching

Pattern Match in Lists

Note that `(x:[])` and `(x:y:[])` could be rewriten as `[x]` and `[x,y]` (because its syntatic sugar, we don't need the parentheses).



We can't rewrite `(x:y:_)` with square brackets because it matches any list of length 2 or more.

Pattern Matching

Pattern Match in Lists

Pattern matching with a little recursion.

```
length' :: (Num b) => [a] -> b
length' []     = 0
length' (_:xs) = 1 + length' xs


sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

Pattern Matching

As-patterns

Patterns are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing.

You do that by putting a name and an `@` in front of a pattern.

For instance, the pattern `xs@(x:y:ys)`.

This pattern will match exactly the same thing as `x:y:ys` but you can easily get the whole list via `xs` instead of repeating yourself by typing out `x:y:ys` in the function body again.

Pattern Matching

As-patterns

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++
                     " is "                 ++ [x]
```

```
ghci> capital "Haskell rocks!"
"The first letter of Haskell rocks! is H"
```

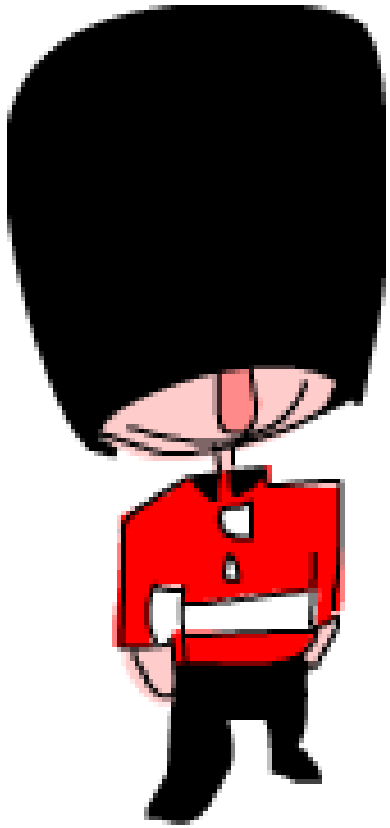
Pattern Matching

you can't use `++` in pattern matches.

If you tried to pattern match against xs `++` ys—, what would be in the first and what would be in the second list? It doesn't make much sense.

It would make sense to match stuff against `(xs ++ [x,y,z])` or just `(xs ++ [x])`, but because of the nature of lists, you can't do that.

Guards, Guards!



Guards, Guards!

Whereas patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether some property of a value (or several of them) are true or false.

That sounds a lot like an if statement and it's very similar.

The thing is that guards are a lot more readable when you have several conditions and they play really nicely with patterns.

Guards, Guards!

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. " ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Guards, Guards!

Guards are indicated by pipes that follow a function's name and its parameters.

Usually, they're indented a bit to the right and lined up.

A guard is basically a boolean expression. If it evaluates to `True`, then the corresponding function body is used. If it evaluates to `False`, checking drops through to the next guard and so on.

If we call this function with 24.3, it will first check if that's smaller than or equal to 18.5. Because it isn't, it falls through to the next guard. The check is carried out with the second guard and because 24.3 is less than 25.0, the second string is returned.

Guards, Guards!

Many times, the last guard is `otherwise`.

`otherwise` is defined simply as `otherwise = True` and catches everything.

This is very similar to patterns, only they check if the input satisfies a pattern but guards check for boolean conditions.

If all the guards of a function evaluate to `False`(and we haven't provided an `otherwise` catch-all guard), evaluation falls through to the next pattern.

That's how patterns and guards play nicely together. If no suitable guards or patterns are found, an error is thrown.

Guards, Guards!

Of course we can use guards with functions that take as many parameters as we want.

```
bmiTell' :: (RealFloat a) => a -> a -> String
bmiTell' weight height
    | weight / height^2 <= 18.5 = "You're underweight, "
                                  "you emo, you!"
    | weight / height^2 <= 25.0 = "You're supposedly normal.'
                                  "Pffft, I bet you're ugly!"
    | weight / height^2 <= 30.0 = "You're fat! Lose some "
                                  "weight, fatty!"
    | otherwise                 = "You're a whale, "
                                  "congratulations!"
```

Guards, Guards!

Of course we can use guards with functions that take as many parameters as we want.

```
bmiTell'' :: (RealFloat a) => a -> a -> String
bmiTell'' weight height
    | bmi <= 18.5 = "You're underweight, "       ++
                    "you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal." ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some "     ++
                    "weight, fatty!"
    | otherwise   = "You're a whale, "           ++
                    "congratulations!"
    where
        bmi = weight / height^2
```

Guards, Guards!

```
max' :: (Ord a) => a -> a -> a
max' a b
     | a > b      = a
     | otherwise = b
```

Guards can also be written inline, although I'd advise against that because it's less readable, even for very short functions.

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Guards, Guards!

```haskell
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
```

```
ghci> 2 `myCompare` 3
LT
ghci> 3 `myCompare` 3
EQ
ghci> 3 `myCompare` 2
GT
ghci>
```

Where!?

```
bmiTell'' :: (RealFloat a) => a -> a -> String
bmiTell'' weight height
    | bmi <= skinny = "You're underweight, "       ++
                      "you emo, you!"
    | bmi <= normal = "You're supposedly normal." ++
                      "Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some "    ++
                      "weight, fatty!"
    | otherwise     = "You're a whale, "          ++
                      "congratulations!"
    where
        bmi = weight / height^2
        skinny = 18.5
        normal = 25.0
        fat    = 30.0
```

Where!?

The names we define in the where section of a function are only visible to that function, so we don't have to worry about them polluting the namespace of other functions

Notice that all the names are aligned at a single column. If we don't align them nice and proper, Haskell gets confused because then it doesn't know they're all part of the same block.

Where!?

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where
      (f:_) = firstname
      (l:_) = lastname


calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where
      bmi weight height = weight / height ^ 2
```

Let It Be

Very similar to `where` bindings are `let` bindings.

`where` bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards.

`let` bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards.

Just like any construct in Haskell that is used to bind values to names, let bindings can be used for pattern matching.

Let It Be

```haskell
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r^2
    in  sideArea + 2 * topArea
```

Let It Be

The difference between `let` and `where` bindings is that `let` bindings are expressions themselves.

`where` bindings are just syntactic constructs.

Let It Be

```
ghci> let i = 3 in i+1
4
ghci> 10 + (let i = 3 in i+1)
14
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Let It Be

`let` bindings inside list comprenhension

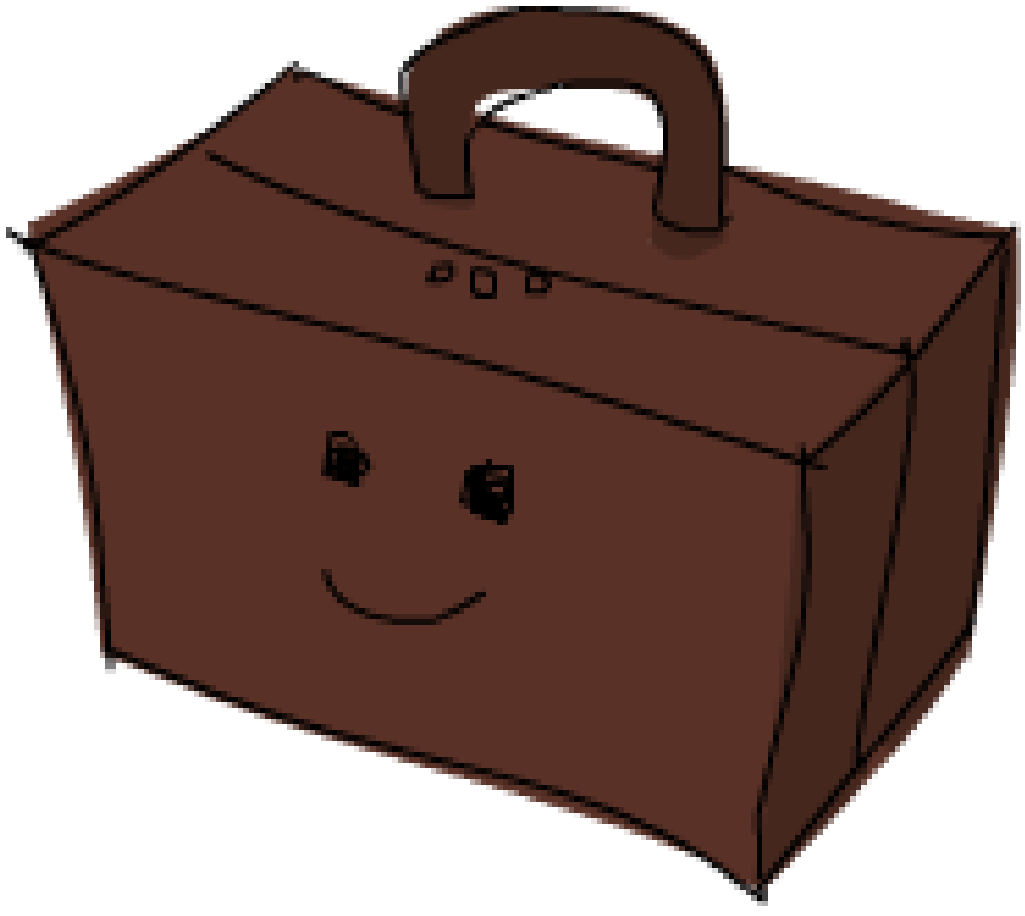
```
calcBmis' :: (RealFloat a) => [(a, a)] -> [a]
calcBmis' xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Return only the BMIs of fat people:

```
calcBmis'' :: (RealFloat a) => [(a, a)] -> [a]
calcBmis'' xs = [bmi | (w, h) <- xs
                     , let bmi = w / h ^ 2
                     , bmi >= 25.0]
```

We can't use the `bmi` name in the `(w, h) <- xs` part because it's defined prior to the `let` binding.

Case expressions



Case expressions

Many imperative languages (C, C++, Java, etc.) have case syntax and if you've ever programmed in them, you probably know what it's about.

It's about taking a variable and then executing blocks of code for specific values of that variable and then maybe including a catch-all block of code in case the variable has some value for which we didn't set up a case.

Case expressions

Haskell takes that concept and one-ups it.

Like the name implies, case expressions are, well, expressions, much like `if else` expressions and `let` bindings.

Not only can we evaluate expressions based on the possible cases of the value of a variable, we can also do pattern matching.

Hmmm, taking a variable, pattern matching it, evaluating pieces of code based on its value, where have we heard this before?

Oh yeah, pattern matching on parameters in function definitions! Well, that's actually just syntactic sugar for case expressions.

Case expressions

These two pieces of code do the same thing and are interchangeable:

```haskell
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```haskell
head' :: [a] -> a
head' xs = case xs of
             []    -> error "No head for empty lists!"
             (x:_) -> x
```

Case expressions

The syntax for case expressions is pretty simple:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` is matched against the patterns.

The pattern matching action is the same as expected: the first pattern that matches the expression is used.

If it falls through the whole case expression and no suitable pattern is found, a runtime error occurs.

Case expressions

Whereas pattern matching on function parameters can only be done when defining functions, case expressions can be used pretty much anywhere.

For instance:

```
describeList :: [a] -> String
describeList xs = "list is " ++ case xs of
                                  []  -> "empty."
                                  [x] -> "a singleton."
                                  xs  -> "a long list."
```

Case expressions

They are useful for pattern matching against something in the middle of an expression. Because pattern matching in function definitions is syntactic sugar for case expressions, we could have also defined this like so:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```