Intro to Haskell Notes: Part 3

Adrian Brasoveanu*

September 30, 2013

Contents

*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, http://learnyouahaskell.com/.

1 Pattern matching

These notes discuss the Haskell syntax for function definitions. Given the central role that functions play in Haskell, these aspects of Haskell syntax are fundamental.

Pattern matching consists of specifying patterns to which some data should conform, 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. You can pattern match on any data type – numbers, characters, lists, tuples, etc. This leads to very expressive code that is also simple and readable.

Let's make a really trivial function that checks if the number we supplied to it is a 7 or not.

```
ghci 1> let {lucky :: (Integral a) ⇒ a → String;
            lucky 7 = "LUCKY NUMBER SEVEN!";
            lucky x = "Sorry, you're out of luck, pal!"}
```

When you call lucky, the patterns will be checked from top to bottom and when the argument of the function conforms to a pattern, the corresponding function body will be used.

The only way a number can conform to the first pattern here is if it is 7. If it's not, it falls through to the second pattern, which matches anything and binds it to *x*.

```
ghci 2> lucky 7
"LUCKY NUMBER SEVEN!"
```

```
ghci 3> lucky 19
"Sorry, you're out of luck, pal!"
```

This function could have also been implemented by using an **if** statement. Optional hw exercise: do that.

But what if we wanted a function that says the numbers from 1 to 5 and says "Not between 1 and 5" for any other number? Without pattern matching, we'd have to make a pretty convoluted **if then else** tree. However, with pattern matching:

```
ghci 4> let {sayMe :: (Integral a) ⇒ a → String;
            sayMe 1 = "One!";
            sayMe 2 = "Two!";
            sayMe 3 = "Three!";
            sayMe 4 = "Four!";
            sayMe 5 = "Five!";
            sayMe x = "Not between 1 and 5"}
```

```
ghci 5> sayMe 1
"One!"
```

```
ghci 6> sayMe 5
"Five!"
```

```
ghci 7> sayMe 6
"Not between 1 and 5"
```

Note that if we moved the last pattern (the catch-all one) to the top, it would always say "Not between 1 and 5", because it would catch all the numbers and they wouldn't have a chance to fall through and be checked for any other patterns.

The Haskell syntax for functions is much cleaner when we're not in `ghci`. For example, you could create a file `sayMe.hs` in your working `ghci` directory (run the command *:!pwd* in `ghci` to determine that directory if you're using a Linux or Mac machine) and type the following code in:

```
sayMe' :: (Integral a) => a -> String
sayMe' 1 = "One!"
sayMe' 2 = "Two!"
sayMe' 3 = "Three!"
sayMe' 4 = "Four!"
sayMe' 5 = "Five!"
sayMe' x = "Not between 1 and 5"
```

Note that we do not use a **let** keyword, semicolons or curly braces; white space (block indentation) is however significant. To load the function in `ghci`, just run the following command:

```
ghci 8> :l sayMe
```

```
ghci 9> sayMe' 5
"Five!"
```

```
ghci 10> sayMe' 6
"Not between 1 and 5"
```

1.1 Implementing *factorial* again

Remember the *factorial* function we implemented previously? We defined the factorial of a number n as *product* $[1..n]$.

```
ghci 11> let factorial' n = product [1..n]
```

```
ghci 12> factorial' 3
6
```

```
ghci 13> factorial' 6
720
```

We can also define a factorial function recursively, the way it is usually defined in mathematics. We start by saying that the factorial of 0 is 1. Then we state that the factorial of any positive integer is that integer multiplied by the factorial of its predecessor.

Here's what that looks like translated in Haskell terms.

```
ghci 14> let {factorial :: (Integral a) => a -> a;
              factorial 0 = 1;
              factorial n = n * factorial (n - 1)}
```

```
ghci 15> factorial 0
1
```

```
ghci 16> factorial 3
6
```

```
ghci 17> factorial 53
427488328406002556429801375338939964969034378836681372467200000000000
```

This is the first time we've defined a function recursively. Recursion is important in Haskell and we'll take a closer look at it later.

But in a nutshell, this is what happens if we try to get the factorial of, say, 3:

- `ghci` tries to compute $3 * factorial\ 2$
- $factorial\ 2$ is $2 * factorial\ 1$, so for now we have $3 * (2 * factorial\ 1)$
- $factorial\ 1$ is $1 * factorial\ 0$, so we have $3 * (2 * (1 * factorial\ 0))$
- now here comes the trick: we've defined $factorial\ 0$ to be just 1 and because it encounters that pattern before the catch-all one, it just returns 1
- so the final result is equivalent to $3 * (2 * (1 * 1))$

Had we written the second pattern on top of the first one, it would catch all numbers, including 0 and our calculation would never terminate.

That's why order is important when specifying patterns and it's always best to specify the most specific ones first and then the more general ones later.

1.2 Pattern matching failures

Pattern matching can also fail. If we define a function like this:

```
ghci 18> let {charName :: Char → String;
          charName 'a' = "Albert";
          charName 'b' = "Broseph";
          charName 'c' = "Cecil"}
```

And then try to call it with an input that we didn't expect, this is what happens:

```
ghci 19> charName 'a'
"Albert"
```

```
ghci 20> charName 'b'
"Broseph"
```

```
ghci 21> charName 'h'
```

`ghci` complains that we have non-exhaustive patterns, and rightfully so. When making patterns, we should always include a catch-all pattern so that our program doesn't crash if we get some unexpected input.

```
ghci 22> let {charName' :: Char → String;
          charName' 'a' = "Albert";
          charName' 'b' = "Broseph";
          charName' 'c' = "Cecil";
          charName' _ = "I don't know this."}
```

```
ghci 23> charName' 'a'
"Albert"
```

```
ghci 24> charName' 'b'
"Broseph"
```

```
ghci 25> charName' 'h'
"I don't know this."
```

2 Pattern matching on tuples

What if we wanted to make a function that takes two vectors in a $2D$ space (that are in the form of pairs) and adds them together?

To add two vectors together, we add their x components and their y components separately.

Here's how we would have done it if we didn't know about pattern matching:

```
ghci 26> let {addVectors' :: (Num a) => (a,a) -> (a,a) -> (a,a);
              addVectors' a b = (fst a + fst b, snd a + snd b)}
```

```
ghci 27> addVectors' (1,2.3) (1,1)
     (2.0,3.3)
```

Well, that works, but there's a better way to do it. Let's modify the function so that it uses pattern matching.

```
ghci 28> let {addVectors :: (Num a) => (a,a) -> (a,a) -> (a,a);
              addVectors (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)}
```

Note that this is already a catch-all pattern. The type of *addVectors* (in both cases) is $addVectors :: (Num\ a) \Rightarrow (a,a) \to (a,a) \to (a,a)$, so we are guaranteed to get two pairs as parameters.

```
ghci 29> :t addVectors
     addVectors :: Num a => (a,a) -> (a,a) -> (a,a)
```

```
ghci 30> addVectors (1,2) (3,4.2)
     (4.0,6.2)
```

2.1 Generalizing *fst* and *snd*

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

```
ghci 31> let {first :: (a,b,c) -> a;
              first (x,_,_) = x;
              second :: (a,b,c) -> b;
              second (_,y,_) = y;
              third :: (a,b,c) -> c;
              third (_,_,z) = z}
```

The _ means the same thing as it does in list comprehensions: we don't care what that part is.

```
ghci 32> first (1,2,3)
     1
```

```
ghci 33> second (1,2,3)
    2
```

```
ghci 34> third (1,2,3)
    3
```

Note the error if we apply these functions to pairs:

```
ghci 35> first (1,2)
```

3 Pattern matching in list comprehensions

```
ghci 36> let xs = [(1,3),(4,3),(2,4),(5,3),(5,6),(3,1)]
```

```
ghci 37> [a+b | (a,b) <- xs]
    [4,7,6,8,11,4]
```

Should a pattern match fail, we just move on to the next element in the input list. Consider this example:

```
ghci 38> let xs = [("abc","def"),("ghi",""),("ABC","DEF"),("","GHI"),("the","end")]
```

```
ghci 39> [[y] ++ ". " ++ [z] ++ "." | ((y:ys),(z:zs)) <- xs]
    ["a. d.","A. D.","t. e."]
```

4 Lists in pattern matching

Lists themselves can be used in pattern matching. We 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:[], we can also use this 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.

The x:xs pattern is used a lot, especially with recursive functions. But patterns that have : in them only match against lists of length 1 or more.

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.

Note: we can't use ++ in pattern matches (ambiguous patterns).

4.1 Implementing the *head* function

Now that we know how to pattern match against lists, let's make our own implementation of the head function.

```
ghci 40> let {head' :: [a] → a;
              head' [] = error "Can't call head on an empty list, dummy!";
              head' (x: _) = x}
```

```
ghci 41> head' [4,5,6]
    4
```

```
ghci 42> head' "Hello"
    'H'
```

```
ghci 43> head' []
    ***Exception: Can't call head on an empty list, dummy!
```

Note that if you want to bind to several variables (even if one of them is just _ and doesn't actually bind at all), we have to put them in parentheses.

Also note the error function that we used: it takes a string and generates a runtime error, using that string as information about what kind of error occurred.

```
ghci 44> :t error
    error :: [Char] → a
```

```
ghci 45> :i error
    error :: [Char] → a    -- Defined in 'GHC.Err'
```

```
ghci 46> :! hoogle    -- info error
    Prelude error :: [Char] -> a
    error stops execution and displays an error message.
    From package base error :: [Char] -> a
```

Note that *error* causes the program to crash, so use it sparingly.

4.2 Another example: pattern matching with lists of different lengths

Let's make a trivial function that tells us some of the first elements of the list in (in)convenient English form.

```
ghci 47> let {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}
```

```
ghci 48> tell []
    "The list is empty"
```

```
ghci 49> tell "h"
    "The list has one element: 'h'"
```

```
ghci 50> tell "he"
    "The list has two elements: 'h' and 'e'"
```

```
ghci 51> tell "hello"
    "This list is long. The first two elements are: 'h' and 'e'"
```

This function is safe because it takes care of the empty list, a singleton list, a list with two elements and a list with more than two elements.

Note that $(x:[])$ and $(x:y:[])$ could be rewriten as $[x]$ and $[x,y]$ – we don't need parentheses in this case.

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

4.3 Implementing *length* again

We already implemented our own *length* function using list comprehension. Now we'll do it by using pattern matching and a little recursion:

```
ghci 52> let {length' :: (Num b) => [a] -> b;
              length' [] = 0;
              length' (_:xs) = 1 + length' xs}
```

This is similar to the factorial function we wrote earlier. First we defined the result of a known input – the empty list. This is also known as the edge condition (a.k.a. the base condition).

Then in the second pattern we take the list apart by splitting it into a head and a tail. We say that the length is equal to 1 plus the length of the tail. We use _ to match the head because we don't actually care what it is.

Also note that we've taken care of all possible patterns of a list: the first pattern matches an empty list and the second one matches anything that isn't an empty list.

```
ghci 53> length' []
    0
```

```
ghci 54> length' "hello"
    5
```

```
ghci 55> length' "hello world"
    11
```

Let's see what happens if we call *length'* on `"ham"`.

```
ghci 56> length' "ham"
    3
```

- first, it will check if it's an empty list; because it isn't, it falls through to the second pattern
- it matches on the second pattern and there it says that the length is $1 + length'$ `"am"`, because we broke it into a head and a tail and discarded the head
- *length'* `"am"` is similarly $1 + length'$ `"m"`; so right now we have $1 + (1 + length'$ `"m"`$)$
- *length'* `"m"` is $1 + length'$ `""` (equivalently, $1 + length'$ $[\,]$), and we've defined *length'* $[\,]$ to be 0
- so in the end we have $1 + (1 + (1 + 0))$

4.4 Implementing *sum*

Assume that the sum of an empty list is 0. We write that down as a pattern. And we also know that the sum of a list is the head plus the sum of the rest of the list.

```
ghci 57> let {sum' :: (Num a) => [a] -> a;
              sum' [] = 0;
              sum' (x:xs) = x + sum' xs}
```

```
ghci 58> sum' []
    0
```

```
ghci 59> sum' [1..10]
    55
```

```
ghci 60> sum' [1..100]
    5050
```

5 As-patterns

As-patterns are a useful way of breaking something up according to a pattern and binding it to names while still keeping a reference to the whole thing. We do that by putting a name and an @ in front of a pattern.

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

```
ghci 61> let {firstLetter :: String → String;
              firstLetter "" = "Empty string, whoops!";
              firstLetter all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]}
```

```
ghci 62> firstLetter "Dracula"
    "The first letter of Dracula is D"
```

```
ghci 63> firstLetter "Tonsil"
    "The first letter of Tonsil is T"
```

```
ghci 64> firstLetter "tonsil"
    "The first letter of tonsil is t"
```

```
ghci 65> firstLetter ""
    "Empty string, whoops!"
```

```
ghci 66> firstLetter []
    "Empty string, whoops!"
```

6 Guards

While patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether an argument (or several arguments) satisfy a property or not.

This is very similar to an **if** statement, but guards are a lot more readable when we have several cascaded conditions we want to check. And they play really nicely with patterns.

6.1 A BMI function

We're going to make a simple function that lists the range of a particular BMI (body mass index). The BMI is weight (in kg) divided by height (in m) squared. If a BMI is less than 18.5, it's in the underweight range. If it's anywhere between 18.5 to 25, it's in the normal range. 25 to 30 is overweight and more than 30 is obese.

```
ghci 67> let {bmiTell :: (Floating a, Ord a) ⇒ a → String;
              bmiTell bmi
                  | bmi ⩽ 18.5 = "underweight range"
                  | bmi ⩽ 25.0 = "normal range"
                  | bmi ⩽ 30.0 = "overweight range"
                  | otherwise = "obese range"}
```

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. Note that there's no = right after the function name and its parameters, before the first guard. Haskell newbies get syntax errors because they sometimes put it there.

One way to remember that the =, i.e., the specification of the function value, follows the guard is to think of the guard as a presupposition that the argument of the function needs to satisfy *before* anything gets computed, i.e., before the function is actually applied to that argument (or arguments, as the case may be). If the presuppositional requirement / guard is satisfied, we can go ahead and compute the value of the function, i.e., we can go ahead and assign a semantic value to the functional expression.

A guard is 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.

```
ghci 68> bmiTell 24.3
    "normal range"
```

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.

```
ghci 69> bmiTell 34.0
    "obese range"
```

This is very reminiscent of a big **if then else** tree in imperative languages, only it is more readable. While big **if else** trees are usually frowned upon, sometimes a problem is defined in such a discrete way that you can't get around them. Guards are a nice alternative for this.

Many times, the last guard is *otherwise*, which is just another word for *True* (it's defined as *otherwise* = *True*) and therefore catches everything.

Thus, guards are very similar to patterns, only patterns check if the input has a particular form while guards check if the input satisfies 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 work together. If no suitable guards or patterns are found, an error is thrown.

6.2 Guards with multi-argument functions: reimplementing *bmiTell*

We can use guards with functions that take as many parameters as we want. Instead of having the user calculate his own BMI before calling the function, let's modify this function so that it takes a height and weight and calculates it for us.

```
ghci 70> let {bmiTell :: (RealFloat a) ⇒ a → a → String;
              bmiTell weight height
                  | weight / height ↑ 2 ⩽ 18.5 = "underweight range"
                  | weight / height ↑ 2 ⩽ 25.0 = "normal range"
                  | weight / height ↑ 2 ⩽ 30.0 = "overweight range"
                  | otherwise = "obese range"}
```

6.3 Implementing the *max* function

Another very simple example: let's implement our own *max* function. If you remember, it takes two things that can be compared and returns the larger of them. This is how we can define it with guards:

```
ghci 71> let {max' :: (Ord a) ⇒ a → a → a;
              max' a b
                  | a > b = a
                  | otherwise = b}
```

```
ghci 72> max' 2 5
    5
```

Guards can also be written inline, but the definition is less readable. Here's an example:

```
ghci 73> let {max'' :: (Ord a) ⇒ a → a → a; max'' a b | a > b = a | otherwise = b}
```

```
ghci 74> max'' 2 5
    5
```

6.4 Implementing the *compare* function

```
ghci 75> let {myCompare :: (Ord a) ⇒ a → a → Ordering;
              a `myCompare` b
                  | a > b = GT
                  | a ≡ b = EQ
                  | otherwise = LT}
```

```
ghci 76> 3 `myCompare` 2
    GT
```

Not only can we call functions in infix form with backticks, we can also define them using backticks if that's more readable.

7 where bindings

In the previous section, we defined a BMI calculator function in which we repeated the expression *weight / height* ↑ 2 three times. It would be better if we could calculate it once, bind it to a name and then use that name instead of the expression. We can modify our function like this:

```
ghci 77> let {bmiTell :: (Floating a, Ord a) ⇒ a → a → String;
              bmiTell weight height
                  | bmi ⩽ 18.5 = "underweight range"
                  | bmi ⩽ 25.0 = "normal range"
                  | bmi ⩽ 30.0 = "overweight range"
                  | otherwise = "obese range"
                where bmi = weight / height ↑ 2}
```

```
ghci 78> bmiTell 85 1.90
    "normal range"
```

We put the keyword **where** after the guards (when you're not using `ghci`, indent it as much as the pipes are indented) and then we define names or functions. These names / function are visible across the guards.

Now we don't have to repeat ourselves, which improves readability. Moreover, if we decide that we want to calculate BMIs a bit differently (e.g., using pounds and inches), we only have to change it once. Finally, this can make our program faster since our *bmi* function is calculated only once.

We could go a bit overboard and present our function like this:

```
ghci 79> let {bmiTell :: (Floating a, Ord a) ⇒ a → a → String;
              bmiTell weight height
                  | bmi ⩽ skinny = "underweight range"
                  | bmi ⩽ normal = "normal range"
                  | bmi ⩽ fat = "overweight range"
                  | otherwise = "obese range"
                where {
                  bmi = weight / height ↑ 2;
                  skinny = 18.5;
                  normal = 25.0;
                  fat = 30.0} }
```

```
ghci 80> bmiTell 85 1.90
    "normal range"
```

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 becoming part of the namespace of other functions. But overusing **where** bindings might decrease the readability of our function instead of increasing it – it's just like overusing footnotes / endnotes in a paper.

Importantly, **where** bindings aren't shared across function bodies of different patterns. If we want several patterns of one function to access some shared name, we have to define it globally with a **let**

binding (see next section).

Note that when we don't work in `ghci`, all the names declared in a **where** block have to be aligned in the exact same way. If we don't align them, Haskell gets confused because it doesn't know they're all part of the same block.

7.1 Pattern matching in where bindings

We can also use pattern matching in **where** bindings. For example, we could have rewritten the **where** section of our previous function as:

```
ghci 81> let {bmiTell :: (Floating a, Ord a) ⇒ a → a → String;
              bmiTell weight height
                  | bmi ⩽ skinny = "underweight range"
                  | bmi ⩽ normal = "normal range"
                  | bmi ⩽ fat = "overweight range"
                  | otherwise = "obese range"
                where {
                  bmi = weight / height ↑ 2;
                  (skinny, normal, fat) = (18.5, 25.0, 30.0)}}
```

```
ghci 82> bmiTell 85 1.90
    "normal range"
```

7.2 Another example: extracting initials

Let's make another fairly trivial function where we get a first and a last name and give someone back their initials.

```
ghci 83> let {initials :: String → String → String;
              initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
                where {(f: _) = firstname; (l: _) = lastname}}
```

```
ghci 84> initials "John" "Doe"
    "J. D."
```

We could have done this pattern matching directly in the function's parameters – it's shorter and clearer actually, see below. But we wanted to show that it's possible to do it in **where** bindings as well.

```
ghci 85> let {initials' :: String → String → String;
              initials' firstname@(f: _) lastname@(l: _) = [f] ++ ". " ++ [l] ++ "."}
```

```
ghci 86> initials' "John" "Doe"
    "J. D."
```

We can even drop the as-patterns without any serious loss in readability:

```
ghci 87> let {initials'' :: String → String → String;
              initials'' (f: _) (l: _) = [f] ++ ". " ++ [l] ++ "."}
```

```
ghci 88> initials'' "John" "Doe"
    "J. D."
```

7.3 Defining functions in where bindings

Just like we've defined constants in **where** blocks, we can also define functions. Let's make a function that takes a list of weight-height pairs and returns a list of BMIs.

```
ghci 89> let {calcBmis :: (Floating a) ⇒ [(a,a)] → [a];
              calcBmis xs = [bmi w h | (w,h) ← xs]
                where bmi weight height = weight / height ↑ 2}
```

```
ghci 90> calcBmis [(80,1.75),(75,1.80)]
    [26.122448979591837,23.148148148148145]
```

The reason we had to introduce *bmi* as a function in this example is because we can't just calculate one BMI from the function's parameters. We have to examine the list passed to the function and there's a different BMI for every pair in there.

Finally, note that **where** bindings can also be nested. It's a common idiom to make a function and define some helper function in its **where** clause and then to also give that function a helper function in its own **where** clause.

8 let bindings

A **let** binding is very similar to a **where** binding. A **where** binding is a syntactic construct that binds variables at the end of a function and the whole function (or a whole pattern-matching subpart) can see these variables, including all the guards.

A **let** binding binds variables anywhere and is an expression itself, but its scope is tied to where the **let** expression appears. So if it's defined within a guard, its scope is local and it will not be available for another guard. But it can also take global scope over all pattern-matching clauses of a function definition if it is defined at that level.

The form is **let** *bindings* **in** *expression*. The names that you define in the **let** part are accessible to the expression after the **in** part. For example, this is how we could define a function that gives us a cylinder's surface area based on its height and radius:

```
ghci 91> let {cylinder :: (Floating a) => a -> a -> a;
              cylinder r h =
                let sideArea = 2 * pi * r * h; topArea = pi * r ↑ 2
                in sideArea + 2 * topArea}
```

```
ghci 92> cylinder 2 7
    113.09733552923255
```

The names should be aligned in the same way when we do not use `ghci`, i.e., when we do not add the curly braces and the semicolon to explicitly indicate the **let** bindings block.

We could have also defined the *cylinder* function with a **where** binding. So what are the main differences between **let** and **where** bindings?

- **let** puts the bindings first and the expression that uses them later, whereas **where** is the other way around
- more importantly, **let** bindings are expressions themselves while **where** bindings are just syntactic constructs that cannot be interpreted on their own

Recall that when we discussed the **if** statement, we said that an **if then else** statement is an expression and it can occur almost anywhere, for example:

```
ghci 93> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
    ["Woo","Bar"]
```

```
ghci 94> 4 * (if 10 > 5 then 10 else 0) + 2
    42
```

Since **let** bindings are expressions too, we can do the same with them. For example:

```
ghci 95> 4 * (let a = 9 in a + 1) + 2
    42
```

They can also be used to introduce functions in embedded expressions (in which case the function names have local scope):

```
ghci 96> [let square x = x * x in (square 5, square 3, square 2)]
    [(25,9,4)]
```

If we want to bind several variables inline, we can separate them with semicolons.

```
ghci 97> let a = 100; b = 200; c = 300 in a * b * c
    6000000
```

```
ghci 98> let foo = "Hey "; bar = "there!" in foo ++ bar
"Hey there!"
```

We don't have to put a semicolon after the last binding, but we can.

```
ghci 99> let a = 100; b = 200; c = 300; in a * b * c
6000000
```

```
ghci 100> let foo = "Hey "; bar = "there!"; in foo ++ bar
"Hey there!"
```

Similarly, we can but often do not have to enclose the bindings with curly braces (although we've been doing this up until now).

8.1 Pattern matching with let bindings

Just like any construct in Haskell that is used to bind values to names, we can pattern match with **let** bindings. E.g., we can dismantle a tuple into components and bind the components to names.

```
ghci 101> let (a, b, c) = (1, 2, 3)
```

```
ghci 102> a
1
```

```
ghci 103> b
2
```

```
ghci 104> c
3
```

We can do all this inside another expression since **let** itself is an expression and it can be freely embedded.

```
ghci 105> (let (a, b, c) = (1, 2, 3) in a + b + c) * 100
600
```

8.2 let bindings in list comprehensions

We can also put **let** bindings inside list comprehensions. Let's rewrite our previous example of calculating lists of weight-height pairs to use a **let** inside a list comprehension instead of defining an auxiliary function with a **where**.

```
ghci 106> let {calcBmis :: (Floating a) ⇒ [(a,a)] → [a];
           calcBmis xs = [bmi | (w,h) ← xs, let bmi = w / h↑2]}
```

```
ghci 107> calcBmis [(80,1.87),(63,1.62)]
      [22.877405702193368,24.005486968449926]
```

We include a **let** inside a list comprehension much like we would a predicate – only it doesn't filter the list, it just introduces a new binding. The names defined in a **let** inside a list comprehension are visible to the output function (the part before the |) and all predicates and sections that come after of the binding.

So we could make our function return only BMIs in the overweight and obese ranges:

```
ghci 108> let {calcBmis :: (Floating a) ⇒ [(a,a)] → [a];
           calcBmis xs = [bmi | (w,h) ← xs, let bmi = w / h↑2, bmi ⩾ 25.0]}
```

We omit the **in** part of a **let** binding when we use it in a list comprehension because the visibility of the binding is already predefined there (but we could use a **let in** binding inside a predicate in a list comprehension and the names would only be visible to that predicate).

The **in** part can also be omitted when defining functions and constants directly in `ghci`, like we've been doing all this time. When we do that, the names are visible throughout the entire interactive session.[1]

```
ghci 109> let zoot x y z = x * y + z
```

```
ghci 110> zoot 3 9 2
      29
```

```
ghci 111> let boot x y z = x * y + z in boot 3 4 2
      14
```

```
ghci 112> boot
```

Should we use **let** bindings all the time instead of **where** bindings? (there are functional programming languages that have only **let** constructs). There are two kinds of situations in which **where** bindings are preferable:

- **where** bindings have scope across guards (but inside a pattern-matching clause) automatically

[1] There's a good reasons for the similar behavior of **let** in list comprehensions and `ghci`: in both cases, we introduce **let** bindings in a monadic, 'sequential' environment – the *list* monad and the *IO* monad, respectively. So **let** bindings have the same scopal behavior as (global) variable assignments in imperative languages.

- **where** bindings define helper names / functions *after* the main function, so the main function's body is closer to its name and type declaration, which increases readability

9 case expressions

Many imperative languages have **case** syntax: we take a variable and execute blocks of code for specific values of that variable. We might also include a catch-all block of code in case the variable has some value for which we didn't set up a case.

But Haskell takes this concept and generalizes it: **case** constructs are expressions, much like **if** expressions and **let** bindings. And we can do pattern matching in addition to evaluating expressions based on specific values of a variable.

Speaking of pattern matching: we already saw this when we discussed function definitions. Well, that's actually just syntactic sugar for **case** expressions. The two pieces of code below do the same thing and are interchangeable:

```
ghci 113> let {head' :: [a] -> a;
               head' [] = error "No head for empty lists!";
               head' (x: _) = x}
```

```
ghci 114> head' "whassup"
    'w'
```

```
ghci 115> head' ""
    *** Exception: No head for empty lists!
```

```
ghci 116> let {head'' :: [a] -> a;
               head'' xs = case xs of {
                 [] -> error "No head for empty lists!";
                 (x: _) -> x}}
```

```
ghci 117> head'' "whassup"
    'w'
```

```
ghci 118> head'' ""
    *** Exception: No head for empty lists!
```

Thus, the syntax for **case** expressions is as follows:

- **case** *expression* **of** *pattern* → *result*
  pattern → *result*
  pattern → *result*
  ...

The *expression* is matched against the patterns. The pattern matching action is what we expect: the first pattern that matches the expression is used. If we fall through the whole **case** expression and no suitable pattern is found, a runtime error occurs.

Note that if we use both guards and **case** expressions in function definitions, the guards cannot appear inside **case** expressions, they have to take scope over them.

This goes well with the informal characterization of guards as presuppositions: they need to be specified before the function application is computed (i.e., the functional expression is evaluated / assigned a semantic value), hence they need to be specified before any specification of the function value. In contrast, **case** expressions are just a way to specify actual function values, i.e., what should get computed assuming the guards / presuppositions are satisfied.

```
ghci 119> let {lessThanTwo :: (Integral a) ⇒ a → String;
          lessThanTwo x
            | x < 2 = case x of {
              0 → "zero";
              1 → "one";
              x → "negative number"}
            | otherwise = "two or more"}
```

```
ghci 120> lessThanTwo 0
     "zero"
```

```
ghci 121> lessThanTwo 1
     "one"
```

```
ghci 122> lessThanTwo (−5)
     "negative number"
```

```
ghci 123> lessThanTwo 5
     "two or more"
```

9.1 Embedding 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, they are useful for pattern matching against something in the middle of an expression:

```
ghci 124> let {describeList :: [a] → String;
          describeList xs = "The list is " ++ case xs of {
            [] → "empty.";
            [x] → "a singleton list.";
            xs → "a longer list."}}
```

```
ghci 125> describeList []
    "The list is empty."
```

```
ghci 126> describeList [1]
    "The list is a singleton list."
```

```
ghci 127> describeList [1..5]
    "The list is a longer list."
```

Alternatively, we could have used a **where** binding and a function definition like so:

```
ghci 128> let {describeList' :: [a] → String;
          describeList' xs = "The list is " ++ what xs
            where {
              what [] = "empty.";
              what [x] = "a singleton list.";
              what xs = "a longer list."}}
```

```
ghci 129> describeList' []
    "The list is empty."
```

```
ghci 130> describeList' [1]
    "The list is a singleton list."
```

```
ghci 131> describeList' [1..5]
    "The list is a longer list."
```

But remember that a function definition with pattern matching is just syntactic sugar for a **case** expression, so using a **where** binding and a function definition like we did above is just a roundabout way of saying what we said more concisely with a **case** expression the first time around.

9.2 Improving readability: case expressions vs. where bindings

In this particular situation, going for a **case** expression directly improves readability because the **case** expression appears at the end of the main function definition. Using **where** just adds more words without improving readability.

But there are cases in which **where** bindings are more readable, e.g., if the **case** expression would have to appear in the middle of the definition of the main function, or we would have to use multiple large **case** expressions etc.