Syntax in Functions

Pattern matching

This chapter will cover some of Haskell's cool syntactic constructs and we'll start with 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. Let's make a really trivial function that checks if the number we supplied to it is a seven or not.

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

When you call lucky, the patterns will be checked from top to bottom and when it 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. This function could have also been implemented by using an if statement. 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 it:

```
1. sayMe :: (Integral a) => a -> String
2. sayMe 1 = "One!"
3. sayMe 2 = "Two!"
4. sayMe 3 = "Three!"
5. sayMe 4 = "Four!"
6. sayMe 5 = "Five!"
7. sayMe x = "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.

Remember the factorial function we implemented previously? We defined the factorial of a number n as product [1..n]. 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 how that looks like translated in Haskell terms.

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

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. It tries to compute 3 * factorial 2. The factorial of 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 the factorial of 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.

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

```
1. charName :: Char -> String
2. charName 'a' = "Albert"
3. charName 'b' = "Broseph"
4. charName 'c' = "Cecil"
```

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

```
1. ghci> charName 'a'
2. "Albert"
3. ghci> charName 'b'
4. "Broseph"
5. ghci> charName 'h'
6. "*** Exception: tut.hs:(53,0)-(55,21): Non-
   exhaustive patterns in function charName
```

It 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.

Pattern matching can also be used 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 together two vectors, we add their x components separately and then their y components separately. Here's how we would have done it if we didn't know about pattern matching:

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

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

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

There we go! Much better. Note that this is already a catch-all pattern. 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.

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
```

The _ means the same thing as it does in list comprehensions. It means that we really don't care what that part is, so we just write a _.

Which reminds me, you can also pattern match in list comprehensions. Check this out:

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

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

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.

Note: 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.

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

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

Checking if it works:

```
1. ghci> head' [4,5,6]
2. 4
3. ghci> head' "Hello"
4. 'H'
```

Nice! Notice 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 surround them in parentheses. Also notice 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. It causes the program to crash, so it's not good to use it too much. But calling head on an empty list doesn't make sense.

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

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

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] (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.

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

```
1. length' :: (Num b) => [a] -> b
```

```
2. length' [] = 0
3. 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. 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.

Let's see what happens if we call length' on "ham". 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. O-kay. The length' of "am" is, similarly, 1 + length' "m". So right now we have 1 + (1 + length' "m"). length' "m" is 1 + length' "" (could also be written as 1 + length' []). And we've defined length' [] to be 0. So in the end we have 1 + (1 + (1 + 0)).

Let's implement sum. We know 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. So if we write that down, we get:

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

There's also a thing called *as patterns*. Those 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. Here's a quick and dirty example:

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

```
1. ghci> capital "Dracula"
2. "The first letter of Dracula is D"
```

Normally we use as patterns to avoid repeating ourselves when matching against a bigger pattern when we have to use the whole thing again in the function body.

One more thing — 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!

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.

Instead of explaining their syntax, let's just dive in and make a function using guards. We're going to make a simple function that berates you differently depending on your BMI (body mass index). Your BMI equals your weight divided by your height squared. If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25 then you're considered normal. 25 to 30 is overweight and more than 30 is obese. So here's the function (we won't be calculating it right now, this function just gets a BMI and tells you off)

```
1. bmiTell :: (RealFloat a) => a -> String
2. bmiTell bmi
3.     | bmi <= 18.5 = "You're underweight, you emo, you!"
4.     | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"

5.     | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
6.     | otherwise   = "You're a whale, congratulations!"
```

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.

This is very reminiscent of a big if else tree in imperative languages, only this is far better and 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 very nice alternative for this.

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.

Of course 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.

```
1. bmiTell :: (RealFloat a) => a -> a -> String
2. bmiTell weight height
3.     | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
4.     | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I be
   t you're ugly!"
5.     | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"

6.     | otherwise                 = "You're a whale, congratulations!"
```

Let's see if I'm fat ...

```
1. ghci> bmiTell 85 1.90
2. "You're supposedly normal. Pffft, I bet you're ugly!"
```

Yay! I'm not fat! But Haskell just called me ugly. Whatever!

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

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.

```
1. max' :: (Ord a) => a -> a -> a
2. max' a b
3.     | a > b     = a
4.     | otherwise = b
```

Guards can also be written inline, although I'd advise against that because it's less readable, even for very short functions. But to demonstrate, we could write max' like this:

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

Ugh! Not very readable at all! Moving on: let's implement our own compare by using guards.

```
1. myCompare :: (Ord a) => a -> a -> Ordering
2. a `myCompare` b
3.     | a > b     = GT
4.     | a == b    = EQ
5.     | otherwise = LT
```

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

Note: Not only can we call functions as infix with backticks, we can also define them using backticks. Sometimes it's easier to read that way.

Where!?

In the previous section, we defined a BMI calculator function and berator like this:

```
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 be
t you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"

    | otherwise                   = "You're a whale, congratulations!"
```

Notice that we repeat ourselves here three times. We repeat ourselves three times. Repeating yourself (three times) while programming is about as desirable as getting kicked inna head. Since we repeat the same expression three times, it would be ideal if we could calculate it once, bind it to a name and then use that name instead of the expression. Well, we can modify our function like this:

```
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
```

We put the keyword where after the guards (usually it's best to indent it as much as the pipes are indented) and then we define several names or functions. These names are visible across the guards and give us the advantage of not having to repeat ourselves. If we decide that we want to calculate BMI a bit differently, we only have to change it once. It also improves readability by giving names to things and can make our programs faster since stuff like our bmi variable here is calculated only once. We could go a bit overboard and present our function like this:

```
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!"
```

```
7.     where bmi = weight / height ^ 2
8.           skinny = 18.5
9.           normal = 25.0
10.          fat = 30.0
```

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 bindings aren't shared across function bodies of different patterns. If you want several patterns of one function to access some shared name, you have to define it globally.

You can also use where bindings to *pattern match*! We could have rewritten the where section of our previous function as:

```
1. ...
2. where bmi = weight / height ^ 2
3.       (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

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

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

We could have done this pattern matching directly in the function's parameters (it would have been shorter and clearer actually) but this just goes to show that it's possible to do it in where bindings as well.

Just like we've defined constants in where blocks, you can also define functions. Staying true to our healthy programming theme, let's make a function that takes a list of weight-height pairs and returns a list of BMIs.

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

And that's all there is to it! 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.

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 give those functions helper functions as well, each with its own *where* clause.

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's see them in action! This is how we could define a function that gives us a cylinder's surface area based on its height and radius:

```
1. cylinder :: (RealFloat a) => a -> a -> a
2. cylinder r h =
3.     let sideArea = 2 * pi * r * h
4.         topArea = pi * r ^2
5.     in  sideArea + 2 * topArea
```

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. As you can see, we could have also defined this with a *where* binding. Notice that the names are also aligned in a single column. So what's the difference between the two? For now it just seems that *let* puts the bindings first and the expression that uses them later whereas *where* is the other way around.

The difference is that *let* bindings are expressions themselves. *where* bindings are just syntactic constructs. Remember when we did the if statement and it was explained that an if else statement is an expression and you can cram it in almost anywhere?

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

2. ["Woo", "Bar"]
3. ghci> 4 * (if 10 > 5 then 10 else 0) + 2
4. 42
```

You can also do that with let bindings.

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

```
2. 42
```

They can also be used to introduce functions in a local scope:

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

If we want to bind to several variables inline, we obviously can't align them at columns. That's why we can separate them with semicolons.

```
1. ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "ther
   e!" in foo ++ bar)
2. (6000000,"Hey there!")
```

You don't have to put a semicolon after the last binding but you can if you want. Like we said before, you can pattern match with *let* bindings. They're very useful for quickly dismantling a tuple into components and binding them to names and such.

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

You 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*.

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

We include a *let* inside a list comprehension much like we would a predicate, only it doesn't filter the list, it only binds to names. 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 the BMIs of fat people:

```
1. calcBmis :: (RealFloat a) => [(a, a)] -> [a]
2. 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.

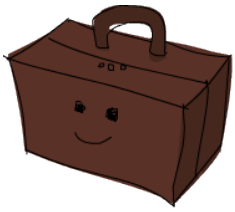
We omitted the *in* part of the *let* binding when we used them in list comprehensions because the visibility of the names is already predefined there. However, we could use a *let in* binding in a predicate and the names defined would only be visible to that predicate. The *in* part can also be omitted when defining functions and constants directly in GHCi. If we do that, then the names will be visible throughout the entire interactive session.

```
1. ghci> let zoot x y z = x * y + z
2. ghci> zoot 3 9 2
```

```
3. 29
4. ghci> let boot x y z = x * y + z in boot 3 4 2
5. 14
6. ghci> boot
7. <interactive>:1:0: Not in scope: `boot'
```

If *let* bindings are so cool, why not use them all the time instead of *where* bindings, you ask? Well, since *let* bindings are expressions and are fairly local in their scope, they can't be used across guards. Some people prefer *where* bindings because the names come after the function they're being used in. That way, the function body is closer to its name and type declaration and to some that's more readable.

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.

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

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

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

As you can see, the syntax for case expressions is pretty simple:

```
1. case expression of pattern -> result
2.                    pattern -> result
3.                    pattern -> result
```

```
4.                    ...
```

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.

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

```
1. describeList :: [a] -> String
2. describeList xs = "The list is " ++ case xs of [] -> "empty."
3.                                                [x] -
   > "a singleton list."
4.                                                xs -> "a longer list."
```

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:

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