CENG242 - Haskell Recitation *

Deniz Sayın

2020
February

1 Introduction to Haskell

Features:

- **Purely functional**, every function is a single expression with no side effects, its result depends entirely on its arguments. Values are immutable. Also, functions are first-class members; i.e. functions can be taken as arguments or returned by other functions.
- **Lazy**, values are not evaluated until forced (e.g. by printing). This results in an increase in expressivity, as we can work with types such as infinite lists, or infeasibly large data structures.
- **Statically typed**, the types of every declaration are known at compile time.
- **Type inference**, when no types are specified, the compiler perfoms bidirectional unification to fully infer the types of declarations.

Note: The standard extension for Haskell source files is `.hs`

2 GHC: Glasgow Haskell Compiler

2.1 Introduction

- GHC is a state-of-the-art & open source compiler and interactive environment for Haskell.
- GHC is the compiler and GHCi is the interactive environment.
- The interactive environment functions similarly to the shell (as well as python's interpreter interface, which you already know about). Bindings can be declared, and expressions can be evaluated and printed.
- Installation instructions can be found at `https://www.haskell.org/ghc/download`. Simplest method to just get ghc: `sudo apt install ghc`

*This material is an updated summary composed from haskell.org and learnyouahaskell.com, and my own understanding

- When launched, the environment automatically loads the `Prelude` module, which contains definitions of standard basic data types, typeclasses and functions. Get familiar with it! Check out the documentation at `https://hackage.haskell.org/package/base-4.12.0.0/dotexttt/Prelude.html` once you're comfortable with Haskell.
- GHC/GHCi will be used when grading the code your write during homeworks/lab exams.

2.2 GHCi Basics

- Run with `ghci`. By default, the prompt shows the names of loaded modules:

```
shell-prompt$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

- Can be used for basic arithmetic, like many other environments. `^` is for exponentiation with integer exponents, while `**` is for floating point exponentiation. Note that operators are implemented as functions with infix notation, thus negative values need to be paranthesised when used in expressions:

```
Prelude> 3 * 5 + 7 * 8
71
Prelude> 3 * 2^5
96
Prelude> 3.0 ** 0.5
1.7320508075688772
Prelude> -7
-7
Prelude> 3 + (-7)
-4
Prelude> 3 + -7

<interactive>:2:1: error:
    Precedence parsing error
        cannot mix `+' [infixl 6] and prefix `-' [infixl 6]
in the same infix expression
```

- Comparisons are similar to other languages, except for inequality:
  - `>`, `>=`: greater than, greater than or equal to
  - `<`, `<=`: less than, less than or equal to
  - `==`, `/=`: equal, not equal
- And and or are represented by `&&` and `||` like in C/C++/Java. `not` replaces the `!` operator:

```
Prelude> 3 > 5
False
Prelude> 3 > 5 || 3 < 5
True
Prelude> 1 < 2 && 2 > 5
False
Prelude> not (8 /= 9)
False
```

- Contains commands related to the `ghci` environment (not the Haskell language itself), prefixed with the colon character `:`. Some examples:
  - `:?` shows help
  - `:type` or `:t` shows the type of an expression
  - `:load` or `:l` loads functions from a Haskell source file
  - `:reload` or `:r` reloads a previously loaded source file
  - `:info` or `:i` shows information about a name
  - `:set` sets environment options. e.g. `:set +s` will show the time elapsed and number of bytes allocated while evaluating an expression.
  - `:sprint` shows how much of a value has been evaluated
  - `:q` quits `ghci`
- Bindings can be made directly in the interactive environment [1] .

```
Prelude> a = 3
Prelude> b = 7
Prelude> diff = a - b
Prelude> diff
-4
Prelude> diff + b
3
```

- Bindings can also be loaded from a source file with the `:load` command. Running `ghci` with a source file as a command line argument will load it automatically.

source.hs

```
x = 3.0 -- also, here is single line comment!
y = 8.7
```

[1] This is a new feature in GHC 8. Previously, introducing a binding directly like '`x = 3`' would result in a parse error, and the `let` keyword had to be used as in '`let x = 3`'. This has to do with `ghci` essentially acting like an IO `do` block, which is information outside the scope of CENG242. New versions of `ghci` act as if `let` is present before the binding even when it is not. See here for more details.

```
shell-prompt$ ghci source.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( source.hs, interpreted )
Ok, modules loaded: Main.
*Main> x
3.0
*Main> x + y
11.7
```

Note: Haskell is case-sensitive! Value names (including functions) **must** start with an underscore or a lowercase letter. Conversely, type names and value constructors (to be covered soon) **must** start with an uppercase letter. Anything else is an error.

3 Basic Data Types

- `Bool`: Boolean values, either `True` or `False`.
- `Char`: Characters with unicode values. e.g. `'a'`, `'?'`. Note that unicode is an extension of ASCII, thus characters which are already in ASCII retain their ASCII values (e.g. 'a' is 97).
- `Int`: Bounded signed integer values, at least 30 bits. Usually 64 bits.
- `Word`: Unsigned integer values, same size as `Int`.
- `Integer`: Arbitrary-precision integers.
- `Float`: Single precision floating point numbers.
- `Double`: Double precision floating point numbers.
- `Rational`: Arbitrary-precision rational numbers, represented with two `Integer` values, one for the numerator and the other for the denominator.

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t True
True :: Bool
```

More basic types can be found in the `Prelude`'s documentation.

Note:

Note that unlike `Char` and `Bool`, numeric values defined in in `ghci` will have a polymorphic type (more on this later). You can force them to have a concrete type using a type specification with the `::` keyword:

```
Prelude> x = 3
Prelude> :t x
x ::  Num t => t
Prelude> x = 3 ::  Int
Prelude> :t x
x ::  Int
Prelude> x
3
Prelude> x = 3 ::  Float
Prelude> x
3.0
```

3.1 Defining Functions

Now that we know about basic types, we can move on to defining functions with them. A simplified syntax for defining functions [2] would be as follows:

`<function-name> <param1-name> {<param2-name> ...} = <result-expression>`

Here are some example functions along with how to call them in `ghci`. Note that unlike popular languages you might have used so far (i.e. C/C++/Java/Python), function definition/application does not require parantheses in Haskell and has the highest precedence: [3]

funcs.hs

```haskell
f x = x^2 + 3*x + 7              -- polynomial
g x y = x * y + x                -- multivariate polynomial
sumOfThree a b c = a + b + c     -- sums its three arguments
isA c = c == 'a' || c == 'A'     -- checks whether the argument is an a
```

[2] This syntax disregards the existence of pattern matching (detailed in 4.1)

[3] We will define our functions in separate source files, but they can also be defined on the fly in `ghci` like any other binding.

```
[1 of 1] Compiling Main             ( funcs.hs, interpreted )
Ok, modules loaded: Main.
*Main> f 0
7
*Main> f(3)
25
*Main> f ((3))
25
*Main> f 0 + 10 -- parsed as (f 0) + 10 due to high precedence
17
*Main> g 3 5
18
*Main> sumOfThree 1 2 3
6
*Main> isA 'c'
False
*Main> isA 'A'
True
*Main> :t isA
isA :: Char -> Bool
```

As we have said before, Haskell is a statically typed language with type inference. Since we did not provide types for the functions we defined in `funcs.hs`, GHC inferred the most generic type possible for the functions. For example, the type of the `isA` function turns out to be `Char -> Bool`. This means that the function takes a `Char` and returns a `Bool`. The type of the other functions is more complicated as they can operate on any numeric type. [4] For functions that take multiple arguments, we add more arrows. For example, `Char -> Char -> Int -> Bool` would be the type of a function taking two `Char`s and an `Int` and returning a `Bool`. Why more arrows rather than something that clearly separates the arguments and the result, such as `{Char, Char, Int} -> Bool`? This is due to a feature of Haskell called *currying*, which we cover in 5.1.

We can set types of functions once again with the `::` operator. This is seen as good practice because it clarifies the parameters and the function's purpose. The source file `typed-funcs.hs` given below shows the previous definitions with provided types. Note that a type that cannot be resolved by the compiler will cause compilation errors. For example, setting the type of `isA` to be `Bool -> Bool` will cause an error because the function will try to compare the input `Bool` value with the `Char` values 'a' and 'A', which is not defined.

[4] In this case, this means any type which is an instance of the `Num` typeclass. For example, the type of `f` would be `(Num a) => a -> a`. The compiler infers this type due to the use of arithmetic operators in `f`.

typed-funcs.hs

```haskell
f :: Integer -> Integer
f x = x^2 + 3*x + 7

g :: Double -> Double -> Double
g x y = x * y + x

sumOfThree :: Int -> Int -> Int -> Int
sumOfThree a b c = a + b + c

isA :: Char -> Bool
isA c = c == 'a' || c == 'A'
```

Another important property of functions in Haskell is that they can be applied 'partially', without applying all of their arguments, which results in a new function equivalent to the previous function with some leading arguments already set. This is best shown with an example over the previously defined `sumOfThree` function:

```
[1 of 1] Compiling Main             ( typed-funcs.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t sumOfThree
sumOfThree :: Int -> Int -> Int -> Int
*Main> sumOfTwo = sumOfThree 0 -- sumOfTwo x y = sumOfThree 0 x y
*Main> :t sumOfTwo
sumOfTwo :: Int -> Int -> Int
*Main> sumOfTwo 5 6
11
*Main> addEight = sumOfTwo 8   -- addEight x = sumOfTwo 8 x = sumOfThree 0 8 x
*Main> :t addEight
addEight :: Int -> Int
*Main> addEight 15
23
```

Note:

The operators we have used so far (`+`, `*`, `<`, `&&` etc.) are also functions. Since these functions are defined with symbols, they can be used directly with infix notation. They have to be put between parentheses to be used as a prefix. Similarly, any function we define with alphanumeric characters can be used with prefix notation directly, but needs to be put between backticks (‘, ASCII 96) to be used as an infix:

```
Prelude> 7*2 + 3*5
29
Prelude> (+) ((*) 7 2) ((*) 3 5)
29
Prelude> p x y = sqrt (x - y)
Prelude> p 3 1
1.4142135623730951
Prelude> 3 `p` 1
1.4142135623730951
```

It is possible to modify the precedence and associativity of your own functions. See Fixity Declarations.

3.2 Lists

3.2.1 Introduction

The primary sequential data structure in Haskell is the list, which is essentially a singly-linked list. The empty list is denoted as `[]`. Note that a list is not a type but a type constructor. This means that a value cannot have the type of 'list', but the type 'list of something'. Actual types could be a list of booleans `[Bool]`, a list of lists of integers `[[Int]]` etc. This also shows that lists are homogeneous, i.e. the elements of a list all share the same type. Lists can be defined using commas (just like in Python), e.g. `[3, 42, 24, 59]`.

Note:

The `String` type is defined as `[Char]` by default.

3.2.2 A Few Basic Functions

The most basic operation on a list is appending to the beginning, with the `(:)` function (called cons, which is the name of the same function in Lisp derivatives): [5].

[5] The definition of lists using commas is actually just syntactic sugar for multiple applications of the `(:)` operator. e.g. `[4, 8, 15, 17]` stands for `4:8:15:17:[]`

```
Prelude> 1:[2, 3]
[1,2,3]
Prelude> 4:5:6:7:[]
[4,5,6,7]
Prelude> [0]:[[1, 2], [3, 4, 5]]
[[0],[1,2],[3,4,5]]
Prelude> 'c':"eng"
"ceng"
```

You can decompose a list into the first element and the rest using the `head` and `tail` functions. You can also decompose it into the initial elements and the final element using `init` and `last`, and the `length` function can be used to find the length of a list. Note that the whole list has to be traversed for these final three, while the first two are constant time operations.

```
Prelude> myList = [17, 21, 3, 99, 44]
Prelude> head myList
17
Prelude> tail myList
[21,3,99,44]
Prelude> init myList
[17,21,3,99]
Prelude> last myList
44
Prelude> length myList
5
```

The `take` and `drop` functions can be used to make a new list from the first k elements of a list, or a list excluding the first k elements. `elem` checks whether a value is inside a list.

```
Prelude> take 4 "ceng242"
"ceng"
Prelude> drop 4 "ceng242"
"242"
Prelude> elem 3 [1, 5, 0, 3, 7]
True
```

Lists can be concatenated with the `(++)` function and indexed with the `(!!)` function. Once again, note that concatenating two lists requires the first list to be traversed and that indexing the kth element also requires traversing the first k elements. If you find yourself making use of these a lot when writing low-level list processing functions, you are probably using the wrong approach.

```
Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> ['b', 'o'] ++ ['o']
"boo"
Prelude> [7, 8, 15, 23, 38, 61] !! 0
7
Prelude> [7, 8, 15, 23, 38, 61] !! 3
23
Prelude> [7, 8, 15, 23, 38, 61] !! 100
*** Exception: Prelude.!!: index too large
```

Infinite lists can be constructed using **`repeat`**, **`cycle`**. **`repeat`** repeats the same value forever, **`cycle`** repeats a list forever . While infinite lists allow for nice abstractions, remember that when working with them in practice you have to cut them off at some point.

```
Prelude> ones = repeat 1 -- no problem, the list is not evaluated (laziness!)
Prelude> take 10 ones    -- get the first ten elements of the list, nice
[1,1,1,1,1,1,1,1,1,1]
Prelude> ones            -- forces evaluation of the whole list, not useful
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1... <goes on until interrupted>
Prelude> take 10 (cycle [3, 11, 2])
[3,11,2,3,11,2,3,11,2,3]
```

3.2.3 Ranges

Some data types [6] can be used to define ranges on lists. The syntax: **`[<start>{,<next>}..{<end>}]`** is for defining ranges with parts enclosed in curly braces denoting optional parts. The step size (increment) is assumed to be 1 if left undefined. Observe that the end of the range is also optional, which means that we can define infinite ranges!

```
Prelude> [3..9]
[3,4,5,6,7,8,9]
Prelude> [3,5..16]
[3,5,7,9,11,13,15]
Prelude> [3,5..17]
[3,5,7,9,11,13,15,17]
Prelude> [10,9..(-1)]
[10,9,8,7,6,5,4,3,2,1,0,-1]
Prelude> [10..(-1)]
[]
Prelude> take 5 [1..]
[1,2,3,4,5]
```

[6] Types which are instances of the **`Enum`** typeclass

3.2.4 List Comprehensions

Haskell also includes list comprehensions (just like Python, and previous functional languages which influenced both Haskell and Python). The syntax is as follows:
`[<expr> | <name1> <- <list1> {, <name2> <- <list2> ...} {, <filter-expr>}]`

Essentially, values taken from one or more lists (multiple lists correspond to the cartesian product, like nested loops) are used to build a new list using the expression provided at the start of the list comprehension. Optionally, a boolean expression can be included after all the lists that will filter out certain values.

Some examples in `ghci` to help with understanding the syntax and workings:

```
Prelude> [x^2 | x <- [0..10]]
[0,1,4,9,16,25,36,49,64,81,100]
Prelude> [x^2 | x <- [0..10], x > 5]
[36,49,64,81,100]
Prelude> [x^2 | x <- [0..10], odd x]
[1,9,25,49,81]
Prelude> [[x, y] | x <- [0,1,2], y <- [3, 4, 5]]
[[0,3],[0,4],[0,5],[1,3],[1,4],[1,5],[2,3],[2,4],[2,5]]
Prelude> [[x, y] | x <- [0,1,2], y <- [3, 4, 5], x * y < 5]
[[0,3],[0,4],[0,5],[1,3],[1,4]]
Prelude> [x | x <- [0,1,2], y <- [1..5]]
[0,0,0,0,0,1,1,1,1,1,2,2,2,2,2]
```

3.3 Tuples

Another important composite data type in Haskell is the tuple. Unlike lists, tuples can be composed of multiple types and have a fixed length. Tuples are defined using parentheses like in Python: `({<item1>, <item2> {, <item3> ...}})`. The type of a tuple is determined by the types of its elements:

```
Prelude> myTuple = ("hello", True)
Prelude> :t myTuple
myTuple :: ([Char], Bool)
Prelude> mySecondTuple = ("a", [False, True], 'b', True, "ccccc")
Prelude> :t mySecondTuple
mySecondTuple :: ([Char], [Bool], Char, Bool, [Char])
```

As you can see from the examples, tuples can have as many elements as you like, and have types depending on the types of the arguments. Tuples have a fixed length, unlike lists. This means that you cannot add more values to the front/back, concatenate them etc. It is of course possible to have lists of tuples, however the tuples have to have the same type since lists are homogeneous. For example, you could have a list produced from a string which holds how many of each characters are present in the string, its type could be `[(Char, Int)]`. This list will only hold tuples of that type.

Two basic functions that work on two element tuples (pairs) are `fst` and `snd`. They simply return the first and second element of the pair, respectively.

Before moving on, make sure you distinguish between lists and tuples:

- **Lists** are homogeneous and of arbitrary length. Useful for collecting data of the same type and operating on it.
- **Tuples** can be heterogeneous and have fixed length. Useful for gluing related data together in a lightweight manner.

Note: Haskell includes a zero element tuple, `()`, which is its own type. However, there is no single element tuple by default.

3.4 Type Variables

Functions do not necessarily have to work on a concrete type. As an example, consider the `head` function which returns the first element of a list. What could the type of this function be? This would actually depend on the type of the list. If it only worked on lists of integers, `[Int]`, it would be `[Int] -> Int`. If this was the case for a list of characters, it would be `[Char] -> Char`. However, the function actually works on lists of all types. It takes a list of some type and returns a value of the same type; the actual operation is the same no matter the type of the value that is in the list.

For cases like the `head` function, Haskell allows us to work with polymorphic functions that work on many different types. When stating the type of these functions, we use **type variables**, which stand in for any type. A type variable can be any identifier starting with a lowercase letter (since actual types start with uppercase letters). Usually, single characters like `a`, `b`, `c` or `t` are used. Two example definitions that use type variables are given below in the `typevars.hs` file. More examples with some functions we introduced previously follow. Make sure you also experiment on your own later on to gain a better understanding!

typevars.hs

```haskell
identity :: someType -> someType
identity x = x -- returns its argument

-- we let GHC infer the type of this function
itsgonnabeq x = 'q' -- returns 'q' no matter the argument
```

```
*Main> :t identity
identity :: someType -> someType
*Main> :t itsgonnabeq
itsgonnabeq :: t -> Char
*Main> :t head
head :: [a] -> a
*Main> :t tail
tail :: [a] -> [a]
*Main> :t fst
fst :: (a, b) -> a
*Main> :t take
take :: Int -> [a] -> [a]
```

4 Syntactic Constructs in Haskell

4.1 Pattern Matching

A very useful feature of Haskell when defining functions is the ability to make different definitions for different patterns. This leads to elegant code that appears to be quite readable. Pattern matching can be applied to anything. Integers, characters, lists, tuples, and even your own data types later on. Here's an example:

match.hs

```
commentOnGrade :: String -> String
commentOnGrade "AA"  = "You rock, friend!"
commentOnGrade "BA"  = "Ah, must have been bad luck!"
commentOnGrade other = "You missed the Haskell recitation, didn't you?"
```

When calling the `commentOnGrade` function, the argument will be checked against the patterns starting from the top and continuing towards the bottom. In our case, the string `"AA"` will match the first definition, while `"BA"` will match the second definition. Anything else will be captured and bound to the name `other` by the third definition, since there is no pattern defined there.

If the patterns do not cover everything, which would be the case if we removed the third definition, anything that does not match the provided patterns will cause an error due to "non-exhaustive patterns in function". Also keep in mind that the matching is done in order. If we took the third definition and put it before the first one, it would match any pattern and the definitions made for the `"AA"` and `"BA"` patterns would be unreachable.

Let's also have an example for integers by defining our own factorial function:

fact.hs

```
fact :: Integer -> Integer
fact 0 = 1 -- base case, should be the first!
fact n = n * fact (n - 1) -- recursive case
```

And an example for tuples that returns the third element of a triplet along with the whole triplet. We also introduce the _ syntax, which captures a value but does not bind it to a name (similar to Python, where _ still binds but is used for unused variables by convention), which you can use for values that you don't care about. Also, you can use the `@` character to bind a whole pattern to a name:

trd.hs

```
trd :: (a, b, c) -> ((a, b, c), c)
trd triplet@(_, _, z) = (triplet, z) -- don't care about the first two
```

```
*Main> trd ('a', 1, True)
(('a',1,True),True)
*Main> trd ("first", "second", "third")
(("first","second","third"),"third")
```

For lists, we can pattern match with the cons `(:)` function to separate the list into its head and its tail as a pattern. Note that a list needs to have at least one element to match that pattern. As an example, we can write our own function to join a list of strings using another string:

join.hs

```haskell
join :: String -> [String] -> String
join _ [] = "" -- empty list case, an empty string
join _ [str] = str -- single string case, only that string
join joinStr (firstStr:rest) = firstStr ++ joinStr ++ join joinStr rest
```

```
*Main> join " " ["in", "the", "beginning"]
"in the beginning"
*Main> join "-" ["not much"]
"not much"
*Main> join "->" ["follow", "the", "arrows!"]
"follow->the->arrows!"
```

4.2 if-then-else

We've been putting this one off long enough! Like most other languages, Haskell also has an if-else construct. Unlike in imperative languages however, the else part is mandatory because the construct is an expression; it must have a value whether the predicate holds or not. As an example, we can rewrite the factorial function with it and add a fancy one which causes an error when a negative value is input:

ifact.hs

```haskell
ifact :: Integer -> Integer
ifact n = if n == 0 then 1 else n * ifact (n - 1) -- one line

ifactChecked :: Integer -> Integer
ifactChecked n = if n < 0 -- broken over multiple lines
                    then error "factorial of a negative value"
                    else ifact n
```

```
*Main> ifactChecked 0
1
*Main> ifactChecked 5
120
*Main> ifactChecked (-10)
*** Exception: factorial of a negative value
CallStack (from HasCallStack):
  error, called at ifact.hs:6:26 in main:Main
```

4.3 Guards

With pattern matching, we learned to match on structures of values, by putting in literals (`0`, `"AA"` etc.) or constructors as in the case of lists (`[]`, `(x:xs)` etc.). With guards, we instead check whether values themselves satisfy certain constraints or not. This works just like a chain of if-then-else constructs, but looks a lot nicer. Here's an example for classifying ranges of values with strings:

classify.hs

```haskell
classify :: Double -> String
classify x
    | x < 1e-10 = "That's tiny! Just like a pebble."
    | x < 1e-5  = "Almost negligible."
    | x < 1e5   = "A pretty normal value, if I've ever seen one."
    | x < 1e10  = "Not bad, some pretty good stuff you've got there."
    | otherwise = "BigValue (TM)"
```

The predicates are checked one after another starting from the top, just like in pattern matching. The only constraint to satisfy is that predicates have to `Bool` values of course. The final case is marked with an `otherwise` [7] Hitting a missing case will cause a "non-exhaustive patterns in function" error, as with pattern matching.

4.4 let-in

Sometimes we are faced with the need to compute a value once and reuse it multiple times, or chain lots of function calls together. Being able to name such intermediate values helps in terms of both readability, especially when faced with lots of/long function calls, and performance because function calls are not repeated [8]. The construct which lets us do this in Haskell is the `let-in` construct. Here's an example:

[7] Tidbit: `otherwise` is simply defined as `otherwise = True`

[8] This can easily be optimized away by a compiler, especially with the side-effectless functions of Haskell, but the readability argument alone is enough.

firstNLast.hs

```haskell
firstNLast :: String -> String
firstNLast fullName = let nameList = words fullName -- splits on spaces
                          firstName = head nameList
                          lastName = last nameList
                      in [head firstName, '.', head lastName, '.']
```

```
*Main> firstNLast "Haskell Curry"
"H.C."
*Main> firstNLast "Marcus Ulpius Traianus"
"M.T."
```

4.5 where

We introduced the `let-in` expression which allows us to name and reuse values before an expression. Now, we introduce the `where` construct which does the same thing, but after an expression is finished. Just like a `let-in` expression, it has access to all the bindings of its enclosing scope. Here's the same example we gave for `let-in`, but using `where` instead:

firstNLastwhere.hs

```haskell
firstNLast :: String -> String
firstNLast fullName = [head firstName, '.', head lastName, '.']
  where nameList = words fullName -- splits on spaces
        firstName = head nameList
        lastName = last nameList
```

Remember that `where` is not an expression like `let-in`, but a different block, thus it can span across guards. However, it cannot span across pattern matches.

Note: Remember that bindings are not restricted to values. It is perfectly valid to define functions (even including different pattern matches!) in `let-in` `where`.

4.6 case-of

Pattern matching is quite nice, isn't it? It's almost too bad that we can only use the top-to-bottom match style in function definitions. Well, fear not! The `case-of` construct is the expression version of pattern matching [9], which you can use to pattern match anywhere. Here's an example that determines how 'lucky' a list of integers is, we also mix in a let to cram in even more expressions:

[9] *Ackchyually*, pattern matching is synctactic sugar for a case-of expression

luck.hs

```haskell
luck :: [Int] -> Int
luck [] = 0
luck (x:xs) = let luckIncrement = case x of 7 -> 1    -- lucky!
                                           13 -> -1  -- unlucky :(
                                           _ -> 0    -- who cares?
              in luckIncrement + luck xs
```

5 Higher Order Functions

5.1 Currying

We were previously left wondering why functions with multiple arguments had type signatures like `X -> Y -> Z -> T`. Now, it is time to demistify this by explaining the technique called *Currying*, which is also present in Haskell. It is a method by which functions taking multiple arguments are transformed into multiple function applications, each taking a single argument.

The gist of it is that functions in Haskell only ever take a single parameter and return a single value. The `->` operator is right-associative, so the previous type signature actually corresponds to this: `X -> (Y -> (Z -> T))`. What does this mean exactly? Let us consider the application of a simple function with two arguments, and see how currying works:

- Let us define the function `f`:
  ```
  f ::  Int -> Int -> Int
  f x y = x * x + y
  ```
- Now, as an example, let's evaluate `f 3 5`. This actually corresponds to two function applications in the following way: `(f 3) 5`.
- First, `f` is applied on `3`, and returns a a function of type `Int -> Int`, which internally is something like this: `f3 y = 3 * 3 + y`.
- Then, the resulting function is applied to the second argument `5`, which results in the `Int` expression `3 * 3 + 5`.

It is easy to see how this extends to cases with multiple arguments. In the end, this is equivalent to having functions with multiple arguments, so we will still call functions with multiple arguments as such. This mechanism is what allows seamless partial function application in Haskell.

5.2 Higher Order Functions over Lists

The `Prelude` and `Data.List` modules contain some higher order functions that work on lists that we have not introduced yet, but are very useful. The most important pair are arguably the `map` and `filter` functions.

The `map` function simply takes a function and applies it to every element of a list. Let's see how it works in practice:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> map (/5) [1, 3, 5, 18, 24, 111]
[0.2,0.6,1.0,3.6,4.8,22.2]
Prelude> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]
Prelude> map (++ "!!!") ["hello", "why", "am I", "so excited"]
["hello!!!","why!!!","am I!!!","so excited!!!"]
```

The next important function is `filter`. It takes a predicate (a function returning `Bool`) and list, and returns a new list only including elements of the list for which the predicate is `True`. Here are some examples:

```
Prelude> :t filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude> filter (>7) [1, 4, 19, 2, 8, 13, 0, 22]
[19,8,13,22]
Prelude> filter odd [0..15]
[1,3,5,7,9,11,13,15]
Prelude> filter not [False, False, True, False]
[False,False,False]
```

You may think that the same functionality can be achieved with list comprehensions. That is correct. However, when working with a single function, using `map` and `filter` is much more readable. e.g. `filter f xs` would be `[x | x <- xs, f x]` using a list comprehension. In some cases, a list comprehension might be more readable; the choice is up to you.

These two functions will be your bread and butter when working with lists. Their compositions also work quite nicely. For example, if we had a nested list (two level, like `[[Int]]`) and wanted to double each element without changing the structure, we could simply nest a map call in the following manner: `map (map (*2)) myNestedList`.

Before moving on let us define them on our own to gain insight and see how simple they are:

mapfilt.hs

```haskell
map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs

filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs) = let rest = filter' p xs
                   in if p x
                         then x : rest
                         else rest
```

Another great function for working with lists is `zipWith`. This function takes a function and two lists, and combines elements of the two lists using the provided function:

```
ghci> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
ghci> zipWith (+) [1, 10, 100] [2, 3, 8]
[3,13,108]
ghci> zipWith (+) [1, 10, 100, 10, 1] [2, 44, 57, 11] -- short cutoff
[3,54,157,21]
ghci> zipWith (:) [1, 2, 3] [[5, 7], [6, 9, 10], [3..7]]
[[1,5,7],[2,6,9,10],[3,3,4,5,6,7]]
ghci> zipWith (++) ["haskell", "so", "fun"] [" is", " much", ", right?"]
["haskell is","so much","fun, right?"]
```

There is also `takeWhile` and `dropWhile`, which is like `take` and `drop`, but instead of taking or dropping a certain number of elements, they take or drop until the provided predicate is no longer satisfied.

```
ghci> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
ghci> takeWhile (<= "Birkan") ["Ahmet", "Ali", "Beril", "Mehmet"]
["Ahmet","Ali","Beril"]
ghci> takeWhile (<10) [1..]
[1,2,3,4,5,6,7,8,9]
ghci> dropWhile (<10) [1..30]
[10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
```

Finally, there is `iterate`, which is another function to create infinite lists by starting from a value and repeatedly applying a function:

```
ghci> :t iterate
iterate :: (a -> a) -> a -> [a]
ghci> take 10 (iterate (*2) 3)
[3,6,12,24,48,96,192,384,768,1536]
```

5.3 Lambdas

Lambdas are anonymous functions; essentially they are nameless functions that can be defined locally, an expression that can define a function without binding it to a name. Their definition has the following syntax: `\<parameters> -> <body>`. As an example, let us define a function to multiply three values using a lambda, with both explicit and implicit currying:

multhree.hs

```haskell
multhree :: Int -> Int -> Int -- implicit currying
multhree = \x y z -> x * y * z

multhree' :: Int -> Int -> Int -- explicit currying
multhree' = \x -> (\y -> (\z -> x * y * z))
```

These are very useful for working with higher order functions, for using functions that you cannot get with partial application without binding them to a name. You can see two examples

below, in the first one we zip two lists with a non-trivial two argument function and in the second one we extract the third element from a list of three tuples:

```
ghci> zipWith (\x y -> x^3 + x^2*y + x*y^2) [1, 2, 3] [-1, -2, -3]
[1,8,27]
ghci> map (\(_, _, z) -> z) [('a', 'b', 'c'), ('x', 'y', 'z'), ('p', 'q', 'r')]
"czr"
```

5.4 Composition

Chaining the application of functions is another one of the things you may wish to do often in functional programming. We could define it on our own like this [10], and use it as infix for readability:

```
ghci> comp f g = \x -> f (g x)
ghci> ((+5) `comp` (*2)) 10
25
ghci> ((+5) `comp` (*2) `comp` (/10)) 10
7.0
ghci> (\x -> ((x / 10) * 2) + 5) 10 -- same, but with a lambda
7.0
```

As you can see, this is useful for rendering successive function applications more readable. The `comp` function we defined is still a bit clunky with the backticks and all, so thankfully there is a built-in function for this which is the dot `(.)`. Let's have one more example with it, where we attempt to get lists having an odd number of elements less than five from a list of lists:

```
ghci> filter (\xs -> odd (length (filter (<5) xs))) [[0..5], [1..5], [10..100]]
[[0,1,2,3,4,5]]
ghci> filter (odd . length . filter (<5)) [[0..5], [1..5], [10..100]]
[[0,1,2,3,4,5]]
```

5.5 Application

The final useful built-in we introduce in this section the `($)` operator, which applies its first argument to the second. This seems redundant because application happens by default, e.g. the `f x` expression will apply `f` to `x`. However, since the application operator has a very low precedence and is right associative it becomes useful for avoiding lots of nested parentheses, which makes your code more readable *and* more writable.

To illustrate, let us slightly modify the example lambda function we had above. We will try to determine if the number of elements less than seven in a given list after all its values have been doubled is not odd [11]. Here we go:

[10] This definition is not entirely valid as infixing is left-associative by default while function composition is right-associative

[11] Not being odd is of course being even, but hey... What could you do if I, as the programmer, simply thought that 'not odd' is more readable?

```
ghci> xs = [-1..10]
ghci> not (odd (length (filter (<7) (map (*2) xs)))) -- is this Lisp?
False
ghci> not $ odd $ length $ filter (<7) $ map (*2) xs -- pleasant 8)
False
```

It is also possible to use this operator to force application where it will not happen on its own:

```
ghci> zipWith ($) [(+1), (*7), (/40), (\x -> x*x - x)] [1, 2, 3, 4]
[2.0,14.0,7.5e-2,12.0]
```

6 Typeclasses

Typeclasses are another important concept in Haskell. They are somewhat similar to Java's interfaces. Just as a class can implement an interface, types can be instances of typeclasses. A typeclass is simply a set of functions. If a type is an instance of a typeclass, that means those functions declared by the typeclass have been implemented for that type and will work on it. A great example is the Prelude's `Num` typeclass which brings together numeric types. Let's get some information about it from `ghci`:

```
Prelude> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
         -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

That's quite a lot of information! Here's an approximate Haskell to English translation of the information:

- If a type `a` is in the `Num` typeclass...
- ... it implements `(+)`, `(-)` and `(*)`, which take two values of type `a` and return another value of type `a`...
- ... it implements `negate`, `abs` and `signum`, which take a value of type `a` and also return a value of type `a`...
- ... and it implements `fromInteger`, which takes an `Integer` value and returns an `a`.
- The next line which starts with `MINIMAL` is a pragma, i.e. a compiler directive for GHC,

and not directly related to Haskell. [12]

- The final lines show that the basic types `Word`, `Integer`, `Int`, `Float` and `Double` are all instances of the `Num` typeclass.

In the end, this simply means that we expect a numeric type to implement addition, subtraction and multiplication; as well as negation, absolute value and signum operations. We also expect a function to transform `Integer`s to that value. Of course, these definitions do not have to make sense as long as the type constraints are satisfied, we could define a type `WeirdInt` and implement addition as exponentiation. But, when used sensibly, typeclasses are a powerful mechanism for gathering similar types together.

Let's introduce some of the already defined basic typeclasses that you will come across quite often:

- **Eq**: values of types belonging to this class can be checked for equality. Instances implement the `==` and `/=` functions. Most standard types (except for functions) are instances of this typeclass.
- **Ord**: this typeclass is for types whose values can be ordered. This means that they support comparison with the standard comparison functions, and their values relate to each other as less than, equal or greater than.
- **Show**: members of the typeclass can be represented with strings. Most standard types are an instances of this typeclass, just like `Eq`. `ghci` uses functions of this typeclass when printing evaluation results. The backbone of this typeclass is `show`, which converts values to their string representation:

```
Prelude> show 77
"77"
Prelude> show [1..5]
"[1,2,3,4,5]"
Prelude> show 'a'
"'a'"
Prelude> show True
"True"
Prelude> show head

<interactive>:24:1: error:
    • No instance for (Show ([a0] -> a0)) arising from a use of ‘show’
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: show head
      In an equation for ‘it’: it = show head
```

- **Read**: is like the opposite of `Show`. It is for types that can be parsed from their string representations. The cornerstone of this typeclass is the `read` function, which reads a value from a string. Unlike `show`, `read` cannot deduce its own type when no context is given since its input is always a string:

[12] Actually, it simply shows the minimal definition necessary when instantiating the typeclass. Since there is an `|` between `negate` and `(-)`, we understand that we can get away with only defining one of those. The compiler will define the other one based on the definition of the one we provided. For example, it could define `(-) x y` as `(+) x (negate y)`.

```
Prelude> read "3.14"          -- no type provided, chaos and confusion
*** Exception: Prelude.read: no parse
Prelude> read "3.14" :: Float -- type provided, cool
3.14
Prelude> read "3.14" :: Int   -- type provided, but not valid
*** Exception: Prelude.read: no parse
Prelude> read "3.14" + 7.0    -- type deduced, result is the arg of (+7.0)
10.14
```

- **Num**: for numeric data types, we have already introduced this one.

It is possible to define polymorphic functions that only work on types belonging to a certain typeclass. This is called a **class constraint** and is provided with `=>`, a thick arrow. We can observe this in several standard functions:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
Prelude> :t read
read :: Read a => String -> a
```

Note:

One type can be an instance of multiple typeclasses. For example, see the information about `Bool`:

```
Prelude> :i Bool
data Bool = False | True -- Defined in ‘GHC.Types’
instance Bounded Bool -- Defined in ‘GHC.Enum’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Read Bool -- Defined in ‘GHC.Read’
instance Show Bool -- Defined in ‘GHC.Show’
```

Similarly, multiple class constraints can exist in a function type, such as `myFunction :: (Eq a, Read a, Num b) => b -> a`.

7 Custom Data Types & Typeclasses

7.1 data

The `data` keyword is used to define algebraic data types with the following syntax:
`data <typename> {<type-vars>} = <value-constr1> {| <value-constr2>} {| ...}`
Value constructors can be nullary or have arguments. Here are some examples:

data.hs

```haskell
-- Enumerated (sum) types
data Bool' = T | F
data Color = Red | Green | Blue
-- A product type, like tuples. The type can
-- have the same name as the value constructor.
data Point = Point Double Double
-- A combination of both sum & prod. types
data Pet = Dog String Int | Cat String Int | Butterfly String
-- Some example values:
bestColor = Green
origin = Point 0.0 0.0
yourPets = [Cat "Boncuk" 3, Cat "Prenses" 7, Butterfly "Pırpır"]
```

The way to define functions that work with algebraic data types is through pattern matching:

mario.hs

```haskell
data Character = Mario | Luigi | Yoshi

describe :: Character -> String
describe Mario = "It's-a me, Mario!"
describe Luigi = "The ultimate in second choice."
describe Yoshi = "Open the door, get on the floor..."
```

So far, the left hand side of our declarations have been concrete types. It is also possible for data declarations to declare type constructors rather than types. Just like value constructors take some values and return a new value, type constructors take some types and return a new type. In fact, you are already familiar with some type constructors. One of them is the list. `[]` is a type constructor (with special syntax). Given a concrete type, such as `Int`, it will produce a new concrete type `[Int]`. Let us define our own, which is essentially a box that may or may not contain a value[13]

schrodinger.hs

```haskell
data SchrödingersBox a = Full a | Empty

checkBox :: Show a => SchrödingersBox a -> String
checkBox (Full sth) = "Aha, I found a " ++ show sth ++ "!"
checkBox Empty = "Oh, the box is empty..."
```

[13] Equivalent to the `Maybe` type constructor from the `Prelude`

```
ghci> checkBox (Full 3)
"Aha, I found a 3!"
ghci> checkBox (Full False)
"Aha, I found a False!"
ghci> checkBox Empty
"Oh, the box is empty..."
```

We can also define recursive types, like a binary tree:

tree.hs

```haskell
data Tree a = Empty | Leaf a | Node a (Tree a) (Tree a)

sizeOf :: Tree a -> Int
sizeOf Empty = 0
sizeOf (Leaf _) = 1
sizeOf (Node _ left right) = 1 + sizeOf left + sizeOf right
```

7.2 deriving

The `deriving` keyword can be used to make types instances of existing typeclasses using default behavior (i.e. checking equality of fields when deriving `Eq`, directly printing value constructor names when deriving `Show` etc.):

```
ghci> data Külüstür = Murat | Serçe | R12
ghci> Murat

<interactive>:2:1: error:
    • No instance for (Show Külüstür) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
ghci> Murat == Serçe

<interactive>:3:1: error:
    • No instance for (Eq Külüstür) arising from a use of ‘==’
    • In the expression: Murat == Serçe
      In an equation for ‘it’: it = Murat == Serçe
ghci> data Külüstür = Murat | Serçe | R12 deriving (Eq, Show)
ghci> Murat
Murat
ghci> Murat == Serçe
False
```

The details of how this functionality is implemented depends on the compiler; only basic typeclasses can be derived from, and making your own typeclasses derivable from is non-standard.

7.3 type and newtype

The `type` declaration can be used to create type synonyms, just like C's `typedef`:

typedecl.hs

```haskell
type ValPair = (Double, Double)
type String' = [Char] -- exactly how String is defined
type AssocList a b = [(a, b)] -- with type variables
```

There also exists a special declaration that can be used for types that only have a single value constructor taking a single argument (i.e. useful for making a new type by wrapping an already existing one), which is the `newtype` declaration. There are some differences between using `newtype` and `data` in such cases, but these are outside our scope[14]:

newtype.hs

```haskell
newtype Point = Point (Double, Double)
```

7.4 Record Syntax

You may have noticed that the value constructors can become a little confusing due to the lack of names of their fields. For example, how can you even guess the parameters of this class?

address.hs

```haskell
data Adress = Info String String String String String String String -- ??
```

It is possible to alleviate this problem by writing your own getters for each field, or creating lots of type synonyms; however this is clearly not very practical. This is why Haskell allows you to create getter functions for the fields on the go, with the added benefit that those become visible as field names when deriving from `Show`. Here's the redefinition of the `Address` type:

recaddr.hs

```haskell
data Adress = Info { -- fields are clarified with record syntax
    name :: String,
    addressLine :: String,
    city :: String,
    county :: String,
    zipCode :: String,
    email :: String,
    mobilePhone :: String
    } deriving Show
```

[14] Short story: `data` is lazy while `newtype` is strict. Long story: feel free to do your own research!

```
ghci> address = Info "Çınar Akçalı" "Etlik Mah. Kıvrımlı Cad. Kanarya Apt. No:75
D:3" "Ankara" "Çankaya" "06010" "cinarakcaliceng242@gmail.com" "571 242 02 20"
ghci> address
Info {name = "\199\305nar Ak\231al\305", addressLine = "Etlik Mah. K\305vr\305ml
\305 Cad. Kanarya Apt. No:75 D:3", city = "Ankara", county = "\199ankaya", zipCo
de = "06010", email = "cinarakcaliceng242@gmail.com", mobilePhone = "571 242 02
20"}
ghci> email address
"cinarakcaliceng242@gmail.com"
ghci> mobilePhone address
"571 242 02 20"
```

7.5 instance

The `instance` keyword is used for making your new type an instance of an existing typeclass. For instance, let us make a new array type over a list and make it an instance of the `Num` typeclass:

valarray.hs

```haskell
data Valarray a = Array [a] deriving Show -- diff. names to confuse less

-- could be defined more easily with common helpers
instance Num a => Num (Valarray a) where
    (+) (Array v1) (Array v2) = Array $ zipWith (+) v1 v2
    (-) (Array v1) (Array v2) = Array $ zipWith (-) v1 v2
    (*) (Array v1) (Array v2) = Array $ zipWith (*) v1 v2
    negate (Array v) = Array $ map negate v
    abs (Array v) = Array $ map abs v
    signum (Array v) = Array $ map signum v
    fromInteger x = Array [fromInteger x]
```

```
ghci> a1 = Array [1, 2, 3, 4, 5]
ghci> a2 = Array [5, 4, 3, 2, 1]
ghci> a1 + a2
Array [6,6,6,6,6]
ghci> a1 * a2
Array [5,8,9,8,5]
ghci> negate a1
Array [-1,-2,-3,-4,-5]
ghci> fromInteger 5 :: Valarray Double
Array [5.0]
```

Note that `instance` needs to be used with a concrete type. In our case, we made the declaration suitable for all `Num a`, but we could also make it work only for a specific concrete type, such as `instance Num (Valarray Double) where`.

7.6 class

The `class` keyword can be used to define your own typeclasses:

polygon.hs

```
class Polygon a where
    points :: a -> [(Double, Double)]
    circumference :: a -> Double
    area :: a -> Double
```

8 Modules

Modules are collections of related functions, types and typeclasses. The standard library contains various useful modules, and a large amount of third party libraries also exist. A great tool for browsing through their documentation is Hackage. For the standard libraries, see this.

8.1 Importing Modules

There are various different ways of importing modules:

modules.hs

```
-- import everything in the module
-- call like any other, e.g. sort [1, 5, 2]
import Data.List

-- import only certain functions
import Data.List (sort, intercalate)

-- import everything except some functions
import Data.List hiding (sort)

-- import with the module name as prefix
-- e.g. Data.List.sort [1, 5, 2]
import qualified Data.List

-- import with whatever name you like
-- e.g. L.sort [1, 5, 2]
import qualified Data.List as L

-- can add (sort), hiding etc. in front
import qualified Data.List as L (sort, intercalate)
```

8.2 Creating Your Own Modules

To make your own module, you simply add a `module (...) where` declaration [15] to the beginning of your module file where you list the functions you wish to export between the parentheses:

Vec3.hs

```
module Vec3 (
    Vec3,
    -- Vec3(..) exports every value constructor when there are multiple
    mag,
    dot,
    cross
    ) where

data Vec3 = Vec3 Double Double Double

mag :: Vec3 -> Double
mag (Vec3 x y z) = sqrt $ x^2 + y^2 + z^2

dot :: Vec3 -> Vec3 -> Double
dot (Vec3 x1 y1 z1) (Vec3 x2 y2 z2) = x1*x2 + y1*y2 + z1*z2

cross :: Vec3 -> Vec3 -> Vec3
cross (Vec3 x1 y1 z1) (Vec3 x2 y2 z2) =
    let cx = y1*z2 - y2*z1
        cy = z1*x2 - z2*x1
        cz = x1*y2 - x2*y1
     in Vec3 cx cy cz
```

You can load your module in `ghci` using `:load`. Other source files in the same directory can also import your module, as long as the file name and module name are the same.

[15] The parentheses can be omitted, which will mean that every top-level definition in the file will be exported.