Haskell Tutorial: Introduction

September 23, 2021

[2]:
```
:opt no-lint
```

1 Introduction

Haskell is a statically typed, purely functional programming language with type inference and lazy evaluation. The first version of Haskell was defined in 1990 and the definition of the Haskell language is a maintained by the Haskell Committee. Haskell 1998 appeared in 1998 and Haskell 2010 was published in July 2010. Haskell is used widely in academia and (to a lesser extend) in the industry. Pandoc, a popular tool to convert between different markup formats, is written in Haskell.

So why is it a neat idea to learn Haskell? There are a couple of possible answers:

- A functional language expands the way you think about programming.
- Haskell makes it easy to reason about programs.
- Haskell is a safe language.
- Haskell is a pure language avoiding side effects.

A popular implementation is the Glasgow Haskell Compiler. It comes with a number of tools. The most important for beginners are:

- `ghci` - an interactive Haskell interpreter
- `ghc` - a Haskell compiler translating Haskell into native machine code
- `runghc` - a program to run Haskell code as scripts

A tool that can be quite helpful in particular for beginners is `hlint`. HLint suggests possible improvements to Haskell source code, often providing suggestions how to simplify code. Another package we like to use is HUnit, a framework for writing unit test cases.

1.1 Expressions

The easiest way to get started with Haskell is to launch `ghci` and to type expressions into the interactive Haskell interpreter. Below are some simple arithmetic expressions. Haskell so far works as a simple calculator. Note that it can do arbitrary precision integer arithmetic.

[3]:
```
2 + 5
```

7

[4]:
```
2 + 5 * 3
```

```
17
```

[5]:
```
(2 + 5) * 3
```

```
21
```

[6]:
```
2^123
```

```
10633823966279326983230456482242756608
```

Haskell is a functional language and that means that pretty much everything in Haskell are functions. Even the simple arithmetic expressions above can be seen as function calls written in *infix notation*. The + operator, for example, is just a syntactic shorthand refering to a function that takes two arguments and returns the sum of them. We can actually write the above expressions in a pure functional representation using *prefix notation*:

[7]:
```
(+) 2 5
```

```
7
```

[8]:
```
(+) 2 ((*) 5 3)
```

```
17
```

[9]:
```
(*) ((+) 2 5) 3
```

```
21
```

[10]:
```
(^) 2 123
```

```
10633823966279326983230456482242756608
```

Of course, writing arithmetic expressions in prefix notation usually does not make the expressions more readable and hence nobody would do this in production code unless there is a very specific reason. At the end, the goal of almost every programming effort is to produce code that is *easy to understand* and *easy to reason about*.

Haskell is also a typed language. So far, we only used integral numbers. Lets see what happens if we devide two numbers.

[11]:
```
6/2
```

```
3.0
```

Apparently, this returns a floating point number. If we want integer division, we have to use the `div` function. Note that in Haskell we write the function name followed by its arguments. If necessary, parenthesis are placed before the function name and after the last argument. This is different from other programming languages where parenthesis are required to enclose the arguments of a function. In C or C++, one would write `div(6, 2)` but in Haskell this is simply `div 6 2`. If parenthesis are necessary in an expression, the function call would be `(div 6 2)`.

[12]:
```
div 6 2
```

```
3
```

Since `div` is a function that takes two arguments and produces a single result value, we can write this expression as well in infix notation by enclosing the function name in backticks.

[13]:
```
6 `div` 2
```

```
3
```

There are predefined functions to test whether a number is `even` or `odd` and we can use the == operator to test whether two values are equal. In a similar way, we can compare numbers. The infix operator notation uses the operators < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), and /= (not equal).

[14]:
```
odd 1
```

```
True
```

[15]:
```
even 1
```

```
False
```

[16]:
```
1 == 2
```

```
False
```

The results show that the comparison operators (functions) return truth values (`True` or `False`).

[17]:
```

```

1.2 Lists

Lists are the most fundamental data type in Haskell. A list is represented by a comma-separated sequence of elements surrounded by square brackets. All elements of a list must be of the same type, i.e., they must all be numbers or they must all be characters and so on. An empty list is represented by opening and closing square brackets with nothing inbetween.

[18]:
```
[]
```

```
[]
```

[19]:
```
[0,1,2,1,3]
```

```
[0,1,2,1,3]
```

Haskell supports an enumeration notation, which is a conveninent way to create longer list (and even infinite lists, as we will see later).

[20]:
```
[1..10]
[0,5..20]
[10,9..1]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
[0,5,10,15,20]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

More complicated lists can be created using *list comprehensions*. List comprehensions mimic what we know from mathematics. In mathematics, if we want to define the set of odd numbers between 1 and 10, we could write $\{x | x \in \{1, .., 10\} \text{ and } x \text{ is odd}\}$. This is how this idea can be translated into Haskell:

[21]:
```
[ x | x <- [1..10], odd x]
```

```
[1,3,5,7,9]
```

If we want the list of squares of all odd numbers between 1 and 10, we can simply apply a function. In mathematics, we would write $\{x^2 | x \in \{1, .., 10\} \text{ and } x \text{ is odd}\}$. For readability, we use the infix operator notation. (As an exercise, rewrite this example in prefix notation.)

[22]:
```
[x^2 | x <- [1..10], odd x]
```

```
[1,9,25,49,81]
```

It is possible to create more complex list comprehensions with variables ranging over multiple lists:

[23]:
```
[ x + y | x <- [1..5], y <- [1..3], x == y]
```

```
[2,4,6]
```

Lists can be concatenated using the ++ operator.

[24]:
```
[1..5] ++ [6..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

The elements of a list have a defined order and a elements can appear multiple times in a list. Hence, lists should not be confused with sets. Since lists maintain an order, it makes sense to talk about the first element of a list or the last element of a list. The `head` function returns the first element of a list while the `tail` function returns the list without its first element. The `last` function returns the last element of a list while the `init` function returns the list without the last element.

[25]:
```
head [1..10]
```

```
1
```

[26]:
```
tail [1..10]
```

```
[2,3,4,5,6,7,8,9,10]
```

[27]:
```
last [1..10]
```

```
10
```

[28]:
```
init [1..10]
```

```
[1,2,3,4,5,6,7,8,9]
```

The `:` operator (often called the cons operator) prepends a head element to a list. In other words, the `(:)` function takes as first argument an element and as second argument a list and it returns the list with the element prepended. Do not confuse this with the `(++)` function, which takes two lists and returns the concatenation of these two lists.

[29]:
```
head [1..10] : tail [1..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

Note that the `:` operator is right associative. This makes it easy to use the cons operator repeatedly to create a list.

[30]:
```
1 : 2 : 3 : []
```

```
[1,2,3]
```

The `null` function returns `True` if a list is empty and `False` otherwise. This gives us a convenient (and fast) way to test whether a list is empty. The `length` function returns the number of elements in a list.

[31]:
```
null [1..10]
```

```
False
```

[32]:
```
length [1..10]
```

```
10
```

Since Haskell allows us to create infinite lists (yes it does!), it is strongly recommended to use `null` to test whether a list is empty. The evaluation of the expression `length [1..] == 0` will not terminate since the calculation of the length of an infinite list will not terminate.

[33]:
```
null [1..]
```

```
False
```

[34]:
```
null [x | x <- [1..10], x == 2^x]
```

```
True
```

There are more predefined useful list functions. Some frequently used functions are:

- The `concat` function takes a list of lists and concatenates them
- The `reverse` function takes a list and reverses all elements in it
- The `take` function takes a list and a number `n` and returns the first `n` elements (or fewer if the list has less than `n` elements)
- The `drop` function takes a list and a number `n` and returns the tail after removing the first `n` elements (or an empty list if the list has less than `n` elements)
- The `elem` function returns `True` if a given value is in the list and `False` otherwise
- The `map` function takes a function and a list and applies the function to all elements of the list
- The `filter` function takes a function returning either `True` or `False` and a list and returns all elements of the list for which the function applied to the list element returns `True`

The last two functions are examples of higher order functions. Higher order functions take functions as arguments or return functions. Programming with higher order functions is extremely powerful. Higher order functions often replace simple loops that you may know from imperative programming languages.

[35]:
```
concat [[1..3],[4..7],[8..10]]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

[36]:
```
reverse [1..10]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

[37]:
```
take 5 [1..10]
```

```
[1,2,3,4,5]
```

[38]:
```
drop 5 [1..10]
```

```
[6,7,8,9,10]
```

[39]:
```
elem 5 [1..10]
```

```
True
```

[40]:
```
map even [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

[41]:
```
filter odd [1..10]
```

```
[1,3,5,7,9]
```

[42]:
```
filter (< 7) [1..10]
```

```
[1,2,3,4,5,6]
```

The last example shows something rather special. We want to select all numbers from the list that are less than 7. To do this, we take the operator < and we fix the second argument to the constant 7. This gives us a function that has only a single argument and we apply this new function to all list elements in order to produce the result. The technique applied here is called currying and is rather fundamental in Haskell. Currying encourages programmers to solve generic problems and then the generic solution can be tailored easily to solve a given task at hand. In the example, we take the generic "less than" operator and we curry it down to the much more specific "less than 7" function.

We finish this section with introducing the indexing operator `!!`. The indexing operator takes two arguments, the first argument is a list and the second argument is an index number. The operator returns the element of the list at the index position. (The first element of a list has the index position 0.) An index number that is not present in the list leads to an error. Hence, seasoned Haskell programmers often try to avoid the indexing operator in order to write pure functions that have no side effects.

[43]:
```
[1..] !! 9
```

```
10
```

1.3 Characters and Strings

A character value is defined by writing the character surrounded by single quotes. Hence, `'a'` represents the character a. Haskell supports unicode and it is no problem to write funny characters in Haskell code. (Whether doing so is a good idea is a different discussion.) Special characters like newlines are represented by escape sequences that start with a backslash. For example, the escape sequence `'\n'` represents a newline character while the escape sequence `'\\'` represents the backslash character.

[44]:
```
'a'
```

```
'a'
```

[45]:
```
'\n'
```

```
'\n'
```

A string is simply a list of characters. Since string literals appear quite frequently, there is a special notation for string literals (lists of characters). A string literal is a possibly empty sequence of characters surrounded by double quotes.

[46]:
```
"this is a string"
```

```
"this is a string"
```

Since a string literal defines a list of characters, we can use list functions on strings.

[47]:
```
length "this is a string"
```

```
16
```

[48]:
```
"this " ++ "is " ++ "a " ++ "string"
```

```
"this is a string"
```

[49]:
```
null ""
```

```
True
```

[50]:
```
head "foo"
tail "foo"
last "foo"
init "foo"
```

```
'f'
```

```
"oo"
```

```
'o'
```

```
"fo"
```

Comparison operators are defined for characters and strings. On strings, the operators do lexicographic comparison.

[51]:
```
'a' < 'b'
'b' < 'a'
"aa" < "ab"
"21" < "111"
21 < 11
```

```
True
```

```
False
```

```
True
```

```
False
```

```
False
```

1.4 Tuples

Tuples are another way to pack multiple values together. Tuples are represented by the values contained in the tuple separated by comma and enclosed in parenthesis. There are some key differences between lists and tuples: - Tuples have a fixed number of values and they are immutable. It is not possible to add values to a tuple or to remove values from a tuple. Tuples are useful in situations where the number of elements is fixed. For example, an edge in a graph can be represented by the two nodes connected by the edge. - The values of a tuple can be of different types. A tuple can easily contain a number, a string, another tuple, and a list.

[52]:
```
(42, "answer", ('a', "string"), [1..10])
```

```
(42,"answer",('a',"string"),[1,2,3,4,5,6,7,8,9,10])
```

Pairs are tuples that have two elements. For pairs, we have predefined functions that can be used to access the first (`fst`) and the second (`snd`) element of a pair.

[53]:
```
fst ("one", 2)
```

```
"one"
```

[54]:
```
snd ("one", 2)
```

```
2
```

A useful function to create lists of pairs is `zip`. The `zip` function takes two lists and returns a list of pairs: The first element of the first list is paired with the first element of the second list, the second element of the first list is paired with the second element of the second list and so on. This continues until one of the lists has been exhausted.

[55]:
```
zip [1..] ["apple", "peach", "pear"]
```

```
[(1,"apple"),(2,"peach"),(3,"pear")]
```

Lists of tuples can also be created using list comprehensions. Lets create a list of pairs where the second element is a square of the first element and each element is in the range [1..10].

[56]:
```
[(a,b) | a <- [1..10], b <- [1..10], b == a^2]
```

```
[(1,1),(2,4),(3,9)]
```

1.5 Types

Haskell is a strongly and statically typed programming language. This means that every value has an associated type. Think of types as a set of values with defined operations on them. For example, take the set of integral numbers. It is natural to think of these numbers as a type.

- *strongly typed*
  - Haskell does not automatically cast a value from one type into a different type
  - By avoiding automatic type conversions, some programming errors can be found before they cause problems
- *statically typed*
  - Types are known at compilation type by the compiler / interpreter
  - Static typing, in combination with strong typing, makes type errors impossible to occur at runtime
- *type inference*
  - Haskell can infer type information and hence the programmer does not have to specify types explicitely for all values (or functions)

Even though Haskell can infer types, we will always specify the type signatures of functions explicitely as this is good practice and captures the intention of the programmer. Some of the basic Haskell types are:

- `Char`: characters (unicode)
- `Bool`: one of the two boolean values `True` and `False`
- `Int`: small signed integer number, usually restricted to 64 bits or 32 bits (platform specific)
- `Integer`: integer numbers of arbitrary precision (well, bounded by the available memory)
- `Double`: floating point numbers, usually 64-bits wide
- `Rational`: rational numbers

We can be explicit that a certain value (or expression) has a certain type. The `::` operator indicates that the value on the left side of the operator is to be understood as having the type defined on the right side of the operator.

[57]:
```
42 :: Double
```

```
42.0
```

[58]:
```
3 / 2 - 1 / 4 :: Rational
```

```
5 % 4
```

The last example says that we want the expression to return a rational number. The Haskell notation `5 % 4` represents the rational number commonly written as the fraction $\frac{5}{4}$ in mathematics. While we can define the type of values and expressions, we usally do not do this and rely on Haskell's type inference to determine the type of values and expressions.

In order to get used to types, it is useful to inspect what Haskell knows about types. In `ghci` or this notebook, it is possible to enquire the runtime system about the type information of an expression. This is done by sending the `:type` command to the runtime (the command may be abbreviated to `:t` as long as there is no other runtime command that starts with a t).

[59]:
```
:type 'a'
```

```
Char
```

[60]:
```
:type "hello"
```

```
[Char]
```

[61]:
```
:type []
```

```
forall a. [a]
```

[62]:
```
:type head
```

```
forall a. [a] -> a
```

[63]:
```
:type map
```

```
forall a b. (a -> b) -> [a] -> [b]
```

The last example says that map is a function that takes a function mapping from type `a` to type `b` and a list of values of type `a` and it returns a list of values of type `b`. Since `a` and `b` are not constrained, the map function can be used for arbitrary types `a` and `b`.

1.6 Functions

Since Haskell is a functional language, the programmer primarily defines functions. Defining a function can be very simple. Lets define a function that takes a single number and returns the square of the number, i.e., $f(x) = x^2$.

[64]:
```
f x = x^2
```

[65]:
```
f 4
```

```
16
```

[66]:
```
f 25
```

```
625
```

We can use the newly defined function to define more functions. Lets define $g(x) = f(x) - 8$.

[67]:
```
g x = f x - 8
```

[68]:
```
g 4
```

```
8
```

While this all works as one would expect, we never really specified that the argument of `f` must be a number. Still, if we try to invoke `f` on the character `c`, Haskell provides us with a type error. What happens here is that Haskell has inferred that the function `f` is not defined for characters, because the function definition uses an expression that is not defined for characters. The error message may not be readable yet but the point here is that this is an error that is thrown at compile time and before execution starts. This means that the programmer is required to fix the problem in the code before it can be run and do harm.

[69]:
```
f 'c'
```

```
No instance for (Num Char) arising from a use of `f'
Possible fix: add an instance declaration for (Num Char)
In the expression: f 'c'
In an equation for `it': it = f 'c'
```

It is good practice to not rely on Haskell's type inference for functions and to define the *type signature* of a function explicitly. This way, you also help Haskell's type inference since you declare the intended type of a function. Here is an example how you define an explicit type signature for a function.

[70]:
```
f :: Integer -> Integer
f x = x^2
```

[71]:
```
f 1234
```

```
1522756
```

The type signature of `f` now says: `f` has a type that takes a value of type `Integer` and returns a value of type `Integer`. While we have defined `f` for `Integer` numbers above, it might be useful to define `f` also for `Int` numbers or all possible numbers. We can do this easily in Haskell.

[72]:
```
f :: Num a => a -> a
f x = x^2
```

[73]:
```
f 5
```

```
25
```

[74]:
```
f 5.0
```

```
25.0
```

[75]:
```
f (5 :: Rational)
```

```
25 % 1
```

The type signature now says that `f` receives a value of a numeric type `a` and it returns a value of the same numeric type `a`. The `a` in the type signature is a type variable; it holds the name of a numeric type. Our new definition of `f` is a *polymorphic* function since it can be applied to arguments with different types. As the examples above show, the type of the argument of the function `f` determines the type of the value returned by the function: `f` is either a function receiving an integer number and returning an integer number or it is a function receiving a floating point number and returning

a floating point number, or it is a function receiving a rational number and returning a rational number.

Since we now know how to define functions, we are essentially ready to start functional programming. But hey, what about all those things that are common in other programming languages like variables, conditional statements, loops? Well, you do not need them because there are functional equivalents (such as pattern matching, guards, recursion, higher order functions, monads) that provide you with a rich toolbox to implement arbitrary algorithms as functions in Haskell. If you are new to programming, you might find that learning functional programming is not very difficult. If, however, you have already some experience with imperative programming languages, then learning Haskell may become a certain challenge since you have to learn to look at programming from a somewhat different perspective. While things may appear initially in a sense weird or difficult, you will hopefully soon start to realize the power of a functional programming style and carry it over even when you write code in an imperative programming language. Many imperative programming languages have been recently extended to better support functional programming. The recent interest in functional programming is also driven by the fact that functional programs are much easier to turn into concurrent programs than imperative programs since functional programs avoid mutable state and functions are pure, i.e., they do not cause side effects.

Here are a few tips on how to think as a functional programmer:

- Do not think about a program as a sequence of operations
- Try to think about the relationship between input and output
- Try to drive simplicity to a maximum
- Think in terms of composition and not in terms of inheritance
- Think about side-effects and how to separate them from the functional core of a program

Haskell Tutorial: Functions

September 24, 2021

[1]:
```
:opt no-lint
```

1 Functions

We have already seen how we can define simple functions. To define a function, we usually define the type signature (although this is optional) and afterwards we define the function itself. This is strongly resembling how you define a function in math.

[2]:
```
sq :: Double -> Double
sq x = x * x

sq 5
map sq [1..10]
```

```
25.0
```

```
[1.0,4.0,9.0,16.0,25.0,36.0,49.0,64.0,81.0,100.0]
```

Note the `sq` function is defined for floating point numbers. If we want a more generic `sq` function that works for all numberic types, we can use the `Num` type class.

[3]:
```
sq :: Num a => a -> a
sq x = x * x

sq 5.0
map sq [1..10]
```

```
25.0
```

```
[1,4,9,16,25,36,49,64,81,100]
```

This function now returns a floating point number if we call it with a floating point number as an argument and it will return an integer if we call it with an integer argument. The function signature defines that `sq` takes as input a value of type `a` and it returns a value of type `a` where `a` belongs to the `Num` typeclass. We will talk more about typeclasses later.

Lets define a function to calculate the Euclidian distance between two points $(x1, y1)$ and $(x2, y2)$. We represent the points naturally as tuples.

[4]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist (x1, y1) (x2, y2) = sqrt (sq (x2 - x1) + sq (y2 - y1))

dist (0,0) (4,3)
```

```
5.0
```

1.1 Pattern Matching

While defining functions, it is often necessary to distinguish difference cases. There are several ways to achieve this in Haskell. A simple but also very powerful mechanism for this is pattern matching: We match the arguments passed to a function against pattern and we apply the first matching function definition. Note that the order of the pattern matters (since we apply the definition of the first matching pattern) and that the set of pattern should catch all possibilities.

Lets define a function `vowel`, which returns `True` if an English language character is a vowel and `False` otherwise. Since the set of vowels is small, we can write a function definition for each vowel and one for the case where the character is not a vowel.

[5]:
```
vowel :: Char -> Bool
vowel 'a' = True
vowel 'e' = True
vowel 'i' = True
vowel 'o' = True
vowel 'u' = True
vowel _ = False

filter vowel "Hello World"
filter (not . vowel) "Hello World"
```

```
"eoo"
```

```
"Hll Wrld"
```

In the above example, we match the character passed to the function `vowel` against constants. The last clause uses the underscore _, a pattern used in situations where one wants to match anything but not use the matched value. The `vowel` function can be used as a filter function to extract all vowels of a string (which is a list of characters). By composing the `vowel` function with the `not` function, we can also easily extract all non-vowels of a string. We will come back to function composition later.

Here is another pattern matching example demonstrating how patterns can be used to match lists. Lets assume we want to define a function `elems` to obtain a string that says “no elements” if a list is empty, “one element” if a list has exactly one element, and “multiple elements” if the list has more than one element. We can achieve this using pattern matching by first testing the special cases and having the catch all case last.

[6]:
```
elems :: [a] -> [Char]
elems []    = "no elements"
elems ([_]) = "one element"
```

```
elems _     = "multiple elements"

"The list has " ++ elems []
"The list has " ++ elems [[]]
"The list has " ++ elems [[],[]]
```

```
"The list has no elements"
```

```
"The list has one element"
```

```
"The list has multiple elements"
```

The first pattern tests whether the list is empty. The second pattern tests whether the list contains a head element followed by an empty list. The third patter matches all arguments and it serves as a catch-all pattern. Note that this definition works with infinite lists. An implementation that counts the list elements would be inferior.

Pattern matching can also be used to extract specific elements of a tuple. Lets define fst', snd', and trd' for 3-tuples. The following definitions are polymorphic, they work regardless which data types are present in a 3-tuple.

[7]:
```
fst' :: (a, b, c) -> a
fst' (x, _, _) = x

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

trd' :: (a, b, c) -> c
trd' (_, _, z) = z

fst' ("lost", 42, "number")
trd' (True, 1, "yes")
```

```
"lost"
```

```
"yes"
```

1.2 Recursion

Compared to imperative programming languages such as C, C++, or Java, a pure functional language like Haskell does not have constructs to program loops. Instead, we use *recursion* to implement loops. Recursion essentially means that the definition of a function may refer to itself. The idea is that a given problem is transferred into a slightly simpler problem until the problem is so simple that it can be directly solved. Hence, a recursive function definition typically consists of a simple case (or multiple simple cases) where the result can be directly provided plus one or more complex cases that can be transformed into simpler cases. This, however, requires that we need a mechanism to distinguish the simple case from the more complex one. Luckily, pattern matching enables us to do this.

Number sequences are classic examples for recursive functions and the probably most famous recursively defined number sequence in computer science textbooks is the fibonacci series. Fi-

bonacci numbers can be defined using the function $fib : \mathbb{N} \to \mathbb{N}$:

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

We can translate this definition (almost) directly into Haskell code:

[8]:
```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

map fib [0..10]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

Recursion is particularly interesting for inductively defined data structures such as lists. A function to sum up the numbers in a list of numbers could be written as follows. (You will later see that this can be done even simpler.) The basic idea is to think of this function as defined as follows:

$$sum'(l) = \begin{cases} 0 & \text{if } l \text{ is the empty list} \\ head(l) + sum'(tail(l)) & \text{otherwise} \end{cases}$$

Below is the definition of sum' in Haskell. Note how pattern matching is used to separate the head from the tail of the list in the general case. This is a very common pattern when functions process lists.

[9]:
```
sum' :: Num a => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs

sum' [1..10]
```

```
55
```

The standard `map` function applies a function to all elements of a list. Lets define our own `map'` function which does the same. The simple case is an empty list where we return an empty list. The general case is a list where we apply the function to the first element and construct a new list out of the new value returned by the function and the application of the function to the tail of the list.

[10]:
```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = (f x) : map' f xs

map' fib []
map' fib [0..10]
```

```
[]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

As another example, lets define another function that takes two lists and mixes them by always taking the head element of the first list, the head element of the second list and so on until a list is empty or both lists are empty.

[11]:
```
mix :: [a] -> [a] -> [a]
mix [] [] = []
mix [] ys = ys
mix xs [] = xs
mix (x:xs) (y:ys) = [x,y] ++ mix xs ys

mix "hlowrd" "el ol"
```

```
"hello world"
```

1.3 Guards

While pattern are already a convenient mechanism, they have their limitations. If you want to test whether a more general condition is true or false, you can use guards. Guards are essentially expressions returning a boolean value that are evaluated in order of their appearance. The first expression evaluating to `True` determines the function definition that is used.

We can rewrite the function producing fibonacci numbers using guards as follows:

[12]:
```
fib' :: Integer -> Integer
fib' n
    | n == 0    = 0
    | n == 1    = 1
    | n >  1     = fib' (n-1) + fib' (n-2)
    | otherwise = error "fib' not defined for negative numbers"

map fib' [0..10]
fib' (-1)
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

```
fib' not defined for negative numbers
```

We can now handle the situation where the argument is negative. Instead of going into an infinite recursion, we call the `error` function. Note that `error` should only be used in situations where an error indicates a programming error. The `error` function should not be used to signal runtime errors. In fact, a good Haskell programmer prefers pure functions that have no side effects. Calling `error` is a pretty strong side effect since the call will abort the program execution. We will later learn about better mechanisms to handle runtime errors.

Since pattern are often more concise and thus more readable than guards, it is sometimes a good idea to mix guards and pattern. Here is another attempt to define a function producing fibonacci numbers. This time, instead of calling the error function, we use the predefined function `undefined` to indicate that the fibonacci function is not defined for negative integers.

[13]:
```
fib'' :: Integer -> Integer
fib'' 0 = 0
fib'' 1 = 1
fib'' n
    | n > 1     = fib'' (n-1) + fib'' (n-2)
    | otherwise = undefined

map fib'' [0..10]
fib'' (-1)
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

```
Prelude.undefined
```

Lets look at some more examples. Here are two functions that compute the greates common divisor (gcd) of two integers.

[14]:
```
-- greatest common divisor (Euclid)
gcdEuclid :: Integer -> Integer -> Integer
gcdEuclid n m
    | mod m n == 0 = n
    | n > m        = gcdEuclid m n
    | otherwise    = gcdEuclid n (m `mod` n)

-- greatest common divisor (dijkstra)
gcdDijkstra :: Integer -> Integer -> Integer
gcdDijkstra n m
    | n == m = n
    | m > n  = gcdDijkstra (m-n) n
    | m < n  = gcdDijkstra m (n-m)

gcdEuclid   54 24
gcdDijkstra 54 24
```

```
6
```

```
6
```

And here is a function that calculates binomial coefficients $\binom{n}{k}$.

[15]:
```
-- binomial coefficient (n choose k)
binom :: Integer -> Integer -> Integer
binom n k
    | k == 0 = 1
    | n == k = 1
    | otherwise = binom (n-1) k + binom (n-1) (k-1)

binom 4 2
```

```
6
```

1.4 Where Bindings

Where bindings can be used in function definitions to bind names to constants, to define local helper functions, or to perform pattern matching. Lets again look at calculating the distance between two points. This time, we use where bindings to introduce constants that are used in the function definition.

[16]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist (x1, y1) (x2, y2) = sqrt (dx^2 + dy^2)
    where dx = x2 - x1
          dy = y2 - y1

dist (0,0) (4,3)
```

```
5.0
```

It is also possible to do pattern matching in where bindings.

[17]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist x y = sqrt ((x2 - x1)^2 + (y2 - y1)^2)
    where (x1, y1) = x
          (x2, y2) = y

dist (0,0) (4,3)
```

```
5.0
```

Finally, we can define helper functions in where bindings that are not accessible outside the function definition. While where bindings may improve readability and may help to reduce duplication of code, they may also reduce readability if they are used too much. Remember, the fine art of computer programming is to write easy to read code.

[18]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist (x1, y1) (x2, y2) = sqrt (sq dx + sq dy)
    where sq x = x * x
          dx = x2 - x1
          dy = y2 - y1

dist (0,0) (4,3)
```

```
5.0
```

1.5 Let Bindings

Let bindings are similar to where bindings. Since let bindings are expressions, they can appear anywhere where expressions can appear. However, different to where bindings, let bindings cannot be referenced in guards.

[19]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist (x1, y1) (x2, y2) =
    let dx = x2 - x1
```

```
        dy = y2 - y1
    in sqrt (dx^2 + dy^2)

dist (0,0) (4,3)
```

```
5.0
```

Let bindings can also do pattern matching.

[20]:
```
dist :: (Double, Double) -> (Double, Double) -> Double
dist x y =
    let (x1, y1) = x
        (x2, y2) = y
    in sqrt ((x2 - x1)^2 + (y2 - y1)^2)

dist (0,0) (4,3)
```

```
5.0
```

1.6 Case Expressions

Case expressions are expressions where the expression evaluated is selected by a value (yielded by another expression) matched against a sequence of pattern. Lets look at an example. The value returned by the expression `n` is matched against the pattern `0`, `1`, and `otherwise` and depending on the match, either the expression `0`, `1`, or `fib (n-1) + fib (n-2)` is evaluated.

[21]:
```
fib n = case n of 0 -> 0
                  1 -> 1
                  otherwise -> fib (n-1) + fib (n-2)

map fib [0..10]
```

```
[0,1,1,2,3,5,8,13,21,34,55]
```

We can also rewrite the `sum'` function from above using a case expression.

[22]:
```
sum' :: Num a => [a] -> a
sum' xs = case xs of [] -> 0
                     (x:xs) -> x + sum' xs

sum' [1..10]
```

```
55
```

Case expressions tend to be relatively verbose and hence most Haskell programmers try to avoid them if other syntactic constructs are sufficient and lead to shorter and more elegant and readable code. However, there are situations where resorting to case expressions is a benefit.

1.7 Function Composition

It is possible to create new functions by combining existing functions. Function composition is denoted using the `.` operator. (We have already used function composition once at the very beginning of this notebook.) In general, function composition means that the function on the right hand side of the `.` operator is applied first and then the function on the left hand side of the `.` operator. In other words, `f . g` results in the application of `g` followed by the application of `f`. Obviously, the types of the functions need to match. The type of the `.` operator is `(.) :: (b -> c) -> (a -> b) -> a -> c`.

[23]:
```
filter (not . vowel) "Hello World"
```

```
"Hll Wrld"
```

1.8 Lambda Functions

Lambda functions, also called anonymous functions, are helper functions without a name. Haskell's foundation is a universal model of computation called Lambda Calculus. A backslash character is commonly used to define lambda functions because of its visual resemblance with the Greek letter lambda (λ).

[24]:
```
(\x -> x*x) 2
```

```
4
```

Lambda functions behave like regular functions with names. However, a lambda function can only have a single clause in its definition. Hence, we must be sure that our pattern covers all cases, otherwise runtime errors will occur. For example, the following definition of `tail'` will be unsafe for an empty list.

[25]:
```
tail' :: [t] -> [t]
tail' = \(_:xs) -> xs

tail' [1..10]
```

```
[2,3,4,5,6,7,8,9,10]
```

Lambda functions are useful in situation where a function is needed that can be easily defined inline. For example, lets multiply all integers of a list by two and add one.

[26]:
```
map (\x -> x * 2 + 1) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

A lambda function gives us a very concise and readable solution without having to define a helper function and finding a good name for it. Of course, we could solve the same task using other means, for example, using function composition.

[27]:
```
map ((+ 1) . (* 2)) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

1.9 Partial Functions and Currying

In Haskell, all functions take only a single argument. This may sound like a contradiction since we are meanwhile used to functions will multiple arguments. To resolve what may appear as a contradiction, we need to look at type signatures again.

[28]:
```
take' :: Int -> [a] -> [a]
take' 0 _      = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

The function `take'` has in its signature only one parameter (i.e., an `Int`). After the `Int` argument has been applied, a function is returned that takes a list of some type `a` as the argument and which returns a list of the same type `a`. We can make use of this when we need a function `take3` that takes the first three elements of a list. We define `take3` to be the `take` function with the first argument fixed to 3.

[29]:
```
take3 :: [a] -> [a]
take3 = take' 3

take3 [1..10]
```

```
[1,2,3]
```

We have already used currying when we created a `+1` function or a `*2` function.

[30]:
```
map ((+ 1) . (* 2)) [1..10]
```

```
[3,5,7,9,11,13,15,17,19,21]
```

From a mathematical point of view, currying a function expression means successively splitting away arguments from the right to the left. In the general case of currying an m-ary function expression: $f(x_1, x_2, \ldots, x_m) = f^{m-1}(x_1)(x_2)\ldots(x_m)$.

Haskell Tutorial: Higher Order Functions

October 8, 2020

[1]:
```
:opt no-lint
```

1 Higher Order Functions

Functions are first class citizens in Haskell. They can be passed as arguments, they can be returned as results, and they can be constructed from other functions (e.g., via function composition or currying). In the following, we will look at some of the higher order functions that are used very frequently.

1.1 Mapping with `map`

The perhaps most basic function that takes a function as an argument is `map`, which applies a function given as the first argument to all elements of a list given as the second argument.

[2]:
```
map even [1..10]
map (*2) [1..10]
map (\x -> x*x) [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
[1,4,9,16,25,36,49,64,81,100]
```

We could implement our own version of `map` in the following way:

[3]:
```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = (f x) : (map' f xs)

map' even [1..10]
map' (*2) [1..10]
map' (\x -> x*x) [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
[1,4,9,16,25,36,49,64,81,100]
```

1.2 Filtering with `filter`

A filter is a function that takes a predicate and a list and returns the list of elements that satisfy the predicate. A predicate is a function returning a boolean value indicating whether an element should pass the filter or not.

[4]:
```
filter even [1..10]
filter (<6) [1..10]
```

```
[2,4,6,8,10]
```

```
[1,2,3,4,5]
```

We could implement our own version of `filter` in the following way:

[5]:
```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs)
    | p x       = x : filter' p xs
    | otherwise =     filter' p xs

filter' even [1..10]
filter' (<6) [1..10]
```

```
[2,4,6,8,10]
```

```
[1,2,3,4,5]
```

1.3 Zipping with `zip` and `zipWith`

Sometimes values of two lists have to be zipped together by building a list of tuples where the first tuple contains the first list elements, the second tuple the second list elements and so forth. This is what the `zip` function does. The `zipWith` function generalizes this idea by allowing you to provide a function that should be applied to the matching list elements. Note that the zipping stops when one of the list runs out of elements.

[6]:
```
zip [1..] "hello"
zipWith (\a b -> (a, b)) [1..] "hello"
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

The `zip` and `zipWith` functions can be easily implemented.

[7]:
```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f _ [] = []
zipWith' f [] _ = []
zipWith' f (x:xs) (y:ys) = (f x y) : zipWith' f xs ys

zip' = zipWith' (\a b -> (a, b))

zip' [1..] "hello"
zipWith' (\a b -> (a, b)) [1..] "hello"
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

With the `zipWith` function, we can implement a function generating the Fibonacci sequence in a rather efficient way. Apparently, Haskell's lazyness allows us to process lists while they are being constructed.

[8]:
```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 20 fib
```

```
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181]
```

1.4 Reducing with `foldl` and `foldr`

Sometimes it is necessary to reduce the elements of a list to a single value. The fold functions are often an effective means to do that. An example is the calculation of the sum of all elements in a list of numbers.

[9]:
```
foldl (+) 0 [1..3]
```

```
6
```

The above call to `foldl` does a left associative addition of all list elements: $(((0+1)+2)+3 = 6$. The first parameter of `foldl` is a step function and the second argument initializes an accumulator. Subsequently, the step function is called with the accumulator and the next list element until all list elements have been processed.

[10]:
```
foldr (+) 0 [1..3]
```

```
6
```

The above call to `foldr` does a right associative addition of all list elements: $0 + (1 + (2 + 3)) = 6$. Since the addition is associative, both folds produce the same result. (An operation is associative if within an expression containing two or more occurrences of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed.)

For operations that are not associative, the difference between `foldl` and `foldr` matters. Lets look at substraction, which is not associative:

[11]:
```
foldl (-) 0 [1..3]
foldr (-) 0 [1..3]
```

```
-6
```

```
2
```

The left fold calculates $(((0 - 1) - 2) - 3 = -6$. The right fold, however, calculates $0 - (1 - (2 - 3)) = 2$.

The `foldl` and `foldr` functions require that a starting value is provided. In some situations, it is possible to use the first (or last) element of a list as a starting value. In such situations, the `foldl1` and `foldr1` functions can be used if the list is guaranteed to have at least one element. (If the list has only a single element, then `foldl1` and `foldr1` return that element.)

It is relatively easy to implement the fold functions:

[12]:
```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' f z []     = z
foldr' f z (x:xs) = f x (foldr' f z xs)

foldr' (-) 0 [1..3]
```

```
2
```

[13]:
```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z []     = z
foldl' f z (x:xs) = foldl' f (f z x) xs

foldl' (+) 0 [1..3]
```

```
6
```

The fold functions can be used to implement many standard utility functions. Here are some examples:

[14]:
```
sum' :: Num a => [a] -> a
sum' = foldl (+) 0
sum'' = foldl1 (+)

sum' [1..5]
sum'' [1..5]
sum' []
```

```
15
```

```
15
```

```
0
```

[15]:
```
product' :: Num a => [a] -> a
product' = foldl (*) 1
product'' = foldl1 (*)

product' [1..5]
product'' [1..5]
product' []
```

```
120
```

```
120
```

```
1
```

[16]:
```
maximum' :: Ord a => [a] -> a
maximum' = foldl1 (\x acc -> if x > acc then x else acc)

maximum' [1,3,5,2,3,4]
maximum' "hello world"
```

```
5
```

```
'w'
```

1.5 Example: Sorting using quicksort

The basic idea of the quicksort algorithm is to sort a list by picking an element and and then sorting all elements smaller than the chosen element and sorting all elements larger than the chosen element and combining the results. Obviously, the lists to be sorted get smaller and smaller until

we reach the trival case where the list to be sorted is an empty list. This can be translated directly into Haskell. (The typeclass `Ord` is for totally ordered datatypes. The `compare` function compares two values and returns whether the first is less than (LT), greater than (GT) or equal (EQ) to the second.)

[17]:
```
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort smaller ++ [x] ++ quickSort larger
    where
        smaller = filter (< x) xs
        larger  = filter (>= x) xs
```

[18]:
```
quickSort [1..3]
quickSort [1,4,2,3,1,2,3]
quickSort "Hello World"
```

```
[1,2,3]
```

```
[1,1,2,2,3,3,4]
```

```
" HWdellloor"
```

Note that this implementation of quicksort can be improved since it does multiple passes over the to be sorted list in each iteration and it does not perform an in place sort, i.e., it creates new temporary lists. You will learn more about sorting algorithms and quicksort in the Algorithms and Data Structures module.

To avoid the two passes over the list to calculate `smaller` and `lesser`, we can write our own function of simply use the `partition` function provided by `Data.List`.

[19]:
```
import Data.List

quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort smaller ++ [x] ++ quickSort larger
    where
        (smaller, larger) = partition (< x) xs
```

[20]:
```
quickSort [1..3]
quickSort [1,4,2,3,1,2,3]
quickSort "Hello World"
```

```
[1,2,3]
```

```
[1,1,2,2,3,3,4]
```

```
" HWdellloor"
```

The quickSort function is nice but kind of limited since it always sorts the data according to the native order of the elements. We can make our sorting function more general by adding an extra argument that provides a comparison function.

[21]:
```
quickSort' :: (Ord a) => (a -> a -> Ordering) -> [a] -> [a]
quickSort' _ [] = []
quickSort' c (x:xs) = quickSort' c smaller ++ [x] ++ quickSort' c larger
    where
        smaller = filter (\y -> c y x == LT) xs
        larger  = filter (\y -> c y x /= LT) xs
```

[22]:
```
quickSort' compare [1,4,2,3,9,8,7,6,5]
quickSort' (flip compare) [1,4,2,3,9,8,7,6,5]
```

```
[1,2,3,4,5,6,7,8,9]
```

```
[9,8,7,6,5,4,3,2,1]
```

The flip function flips the arguments of a function, which causes the comparison to be done in the different direction (descending order instead of ascending order).

Note that we can easily derive a quicksort function using the native sort order out of the more generic quicksort function via currying.

[23]:
```
quickSortNative = quickSort' compare

quickSortNative [1,4,2,3,9,8,7,6,5]
```

```
[1,2,3,4,5,6,7,8,9]
```

Haskell Tutorial: Datatypes

October 29, 2020

[1]:
```
:opt no-lint
```

1 Datatypes

Sometimes we may want to introduce new data types for specific purposes.

1.1 Employee

We introduce the mechanisms that can be used to define our own datatypes by defining a datatype representing information about an employee.

[2]:
```
data EmployeeInfo = Employee String Int Double [String]
    deriving (Show)

p = Employee "Joe Sample" 22 186.3 []
print p
```

```
Employee "Joe Sample" 22 186.3 []
```

To define a new datatype, use the keyword `data`, which is followed by the name of our new type (also called a type constructor), in our example `EmployeeInfo`. The type constructor is followed by the data constructor `Employee`. The data constructor is used to create a value of the `EmployeeInfo` type. Both the type constructor and the data constructor must start with a capital letter. The `Int`, `Double`, `String` and `[String]` (that follow after the data constructor `Employee`) are the components/fields of the type. (If you are familiar with an object-oriented language, these components serve the same purpose as fields for a class).

In this particular example, the name of the type constructor (`EmployeeInfo`) and the value/data constructor (`Employee`) use a different name for you to see which is which. It is common practice to give them both the same name. There is no ambiguity because the type constructor's name is used only in type declaration / type signatures and the value constructor's name is used in the actual code (in writing expressions).

The `deriving (Show)` part asks Haskell to make sure that the new type belongs to the `Show` typeclass, which implies to derive a `show` function that renders an instance of our newly defined datatype into a string. This `show` function is used by the `print` function. If there would be no

`deriving (Show)`, Haskell would not be able to print an instance of our new datatype. We will talk more about typeclasses later, so lets not dig deeper here.

Consider the fields from the example above. It is not very clear what the `Int`, `Double`, `String` and `[String]` fields mean. To make things as clear as possible, it is possible to introduce synonyms for existing types at any time. Such synonyms give a type a more descriptive name.

[3]:
```
type Age = Int
type Salary = Double
type Name = String
type Manager = [Name]
data EmployeeInfo = Employee Name Age Salary Manager
    deriving (Show)

p = Employee "Joe Sample" 22 86123.35 ["Lucy Boss"]
print p
```

```
Employee "Joe Sample" 22 86123.35 ["Lucy Boss"]
```

Data type definitions can be nested. We improve our example by replacing the `Age` field with a `Birthday` field. Using a `Birthday` in our `Employee` type has the advantage that the birthday is immutable while the age changes once every year. Since a date consists of a year, a month, and a day, we construct another data type to represent dates. (In real projects, you likely want to use a pre-defined date type.)

[4]:
```
type Day = Int
type Month = Int
type Year = Int
data Date = Date Year Month Day
    deriving (Show)

type Name = String
type Birthday = Date
type Salary = Double
type Manager = [Name]

data Employee = Employee Name Birthday Salary Manager
    deriving (Show)

p = Employee "Joe Sample" (Date 1983 06 17) 86123.35 ["Lucy Boss"]
print p
```

```
Employee "Joe Sample" (Date 1983 6 17) 86123.35 ["Lucy Boss"]
```

New data types can also be alternatives of types. The simplest examples are enumerations. We can use alternative types to define the sex of a person.

[5]:
```
data Sex = Female | Male | Diverse
    deriving (Show)
```

The Sex datatype has three data constructors, Female, Male and Diverse, separated by a bar symbol (the bar symbol can be read as "or"). The data constructors are commonly referred to as alternatives or cases. The data constructors can take zero or more arguments.

[6]:
```
data Employee = Worker  Name Birthday Sex Salary Manager
              | Manager Name Birthday Sex Salary
    deriving (Show)

w = Worker "Joe Sample" (Date 1983 06 17) Diverse 86123.35 ["Lucy Boss"]
print p
m = Manager "Lucy Boss" (Date 1967 02 20) Female 113213.23
print m
```

```
Employee "Joe Sample" (Date 1983 6 17) 86123.35 ["Lucy Boss"]


Manager "Lucy Boss" (Date 1967 2 20) Female 113213.23
```

Data type definitions can be recursive, giving us the option to have inductively defined data types. Lets make the manager of an empoyee another employee so that we can represent a whole employee hierarchy.

[7]:
```
data Employee = Worker Name Birthday Sex Salary Employee
              | Manager Name Birthday Sex Salary
    deriving (Show)

lucy :: Employee
lucy = Manager "Lucy Boss" (Date 1967 02 20) Female 113213.23

joe :: Employee
joe = Worker "Joe Sample" (Date 1983 06 17) Diverse 86123.35 lucy

print joe
```

```
Worker "Joe Sample" (Date 1983 6 17) Diverse 86123.35 (Manager "Lucy Boss" (Date 1967 2 20) Fema
```

1.2 Maybe

Type definitions can be parameterized by introducing a type parameter in the type definition. A common example is the definition of Maybe in the prelude.

[8]:
```
data Maybe a = Nothing | Just a
    deriving (Show)
```

The definition of `Maybe` has a type as its argument, i.e., the type is parametrized. The `Maybe` type can be used to create types that contain a certain typed value or nothing. This helps in situation where some value is optional or not always defined. Lets take a short excursion by looking at the standard `tail` function: it fails if called with a list that has no tail.

[9]:
```
tail [1..3]
tail [1]
tail []
```

```
[2,3]
```

```
[]
```

```
Prelude.tail: empty list
```

Throwing an error is a pretty bad side effect. With `Maybe`, we have a tool that we can use to write a safe tail function, which returns `Nothing` or `Just` a list.

[10]:
```
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

safeTail [1..3]
safeTail [1]
safeTail []
```

```
Just [2,3]
```

```
Just []
```

```
Nothing
```

1.3 Employee (continued)

With this definition of `Maybe`, we can further improve our `Employee` type by enabling managers to be managed by other managers but top-level managers are not managed by anyone. And at this point, we may want to give up the distinction between managers and workers (since an employee is a manager once she manages someone).

[11]:
```
data Employee = Employee Name Birthday Sex Salary (Maybe Employee)
    deriving (Show)

lucy = Employee "Lucy Boss" (Date 1967 02 20) Female 113213.23 Nothing

joe = Employee "Joe Sample" (Date 1983 06 17) Diverse 86123.35 (Just lucy)
```

```
print joe
```

```
Employee "Joe Sample" (Date 1983 6 17) Diverse 86123.35 (Just (Employee "Lucy Boss" (Date 1967 2
```

The interpretation is that the manager is either `Just Employee` or `Nothing`. The `Maybe` type is very widely used to indicate missing values in Haskell.

The last improvement we want to make is to use the record syntax. With the record syntax, we can give the various fields of our type names. Haskell will then automatically generate functions that can be used to access the different fields of a type. The record syntax also changes how data instances are rendered using the `show` function.

[12]:
```
type Day = Int
type Month = Int
type Year = Int
data Date = Date { year :: Year, month :: Month, day :: Day }
    deriving (Show)

type Name = String
type Euro = Double      -- never represent money using floating point types in␣
 ↪real code

data Sex = Female | Male | Diverse
    deriving (Show)

data Employee = Employee { name :: Name
                         , birthday :: Date
                         , sex :: Sex
                         , salary :: Euro
                         , manager :: Maybe Employee }
    deriving (Show)

lucy = Employee { name = "Lucy Boss"
                , birthday = Date { year = 1967, month = 02, day = 20 }
                , sex = Female
                , salary = 113213.23
                , manager = Nothing }

joe = Employee { name = "Joe Sample"
               , birthday = Date { year = 1983, month = 06, day = 17 }
               , sex = Diverse
               , salary = 86123.35
               , manager = Just lucy }

print joe
print (manager joe)
```

```
print lucy
print (manager lucy)
```

```
Employee {name = "Joe Sample", birthday = Date {year = 1983, month = 6, day = 17}, sex = Diverse
```

```
Just (Employee {name = "Lucy Boss", birthday = Date {year = 1967, month = 2, day = 20}, sex = Fe
```

```
Employee {name = "Lucy Boss", birthday = Date {year = 1967, month = 2, day = 20}, sex = Female,
```

```
Nothing
```

1.4 Binary Tree

As another example, let us define a parametric type for a binary tree in order to practice what we have learned.

[13]:
```
data Tree a = Empty
            | Leaf a
            | Branch a (Tree a) (Tree a)
            deriving (Eq, Show)

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print t

:type t
```

```
Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
```

```
Tree Char
```

Haskell has derived that `t` is of type `Tree Char`. In the definition of `Tree`, we ask Haskell to derive the typeclasses `Eq` and `Show`. We do this to obtain a `show` function that takes a `Tree` value and converts it into a string so that we can print values. Lets see what happens if we create a Tree of numbers.

[14]:
```
t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print t

:type t
```

```
Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
```

```
forall a. Num a => Tree a
```

Haskell derived that `t` is of type `Num a => Tree a`, i.e., it is a tree of a type that is constrained to the `Num` typeclass.

Lets now define a function `values :: Tree a -> [a]` that takes a tree and returns all values stored in the tree as a list. Note that this function works for any type that can be used with the tree type. We can use this function with a tree of strings or a tree of numbers, i.e., this function is polymorphic.

[15]:
```
values :: Tree a -> [a]
values Empty = []
values (Leaf x) = [x]
values (Branch x l r) = values l ++ [x] ++ values r

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ values t

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ values t
```

```
"abcdef"


[1,2,3,4,5,6]
```

The `values` function can be implemented in different ways since the tree may be traversed in three different ways: * pre-order (NLR): 1. Return the value of the current node (N) 1. Return the values of the left subtree (L) 1. Return the values of the right subtree (R) * in-order (LNR): 1. Return the values of the left subtree (L) 1. Return the value of the current node (N) 1. Return the values of the right subtree (R) * post-order (LRN): 1. Return the values of the left subtree (L) 1. Return the values of the right subtree (R) 1. Return the value of the current node (N)

Perhaps we should have three traversal functions.

[16]:
```
preOrderValues :: Tree a -> [a]
preOrderValues Empty = []
preOrderValues (Leaf x) = [x]
preOrderValues (Branch x l r) = [x] ++ preOrderValues l ++ preOrderValues r

inOrderValues :: Tree a -> [a]
inOrderValues Empty = []
inOrderValues (Leaf x) = [x]
inOrderValues (Branch x l r) = inOrderValues l ++ [x] ++ inOrderValues r

postOrderValues :: Tree a -> [a]
postOrderValues Empty = []
postOrderValues (Leaf x) = [x]
postOrderValues (Branch x l r) = postOrderValues l ++ postOrderValues r ++ [x]
```

```
t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ preOrderValues t
print $ inOrderValues t
print $ postOrderValues t
```

```
"dbacfe"


"abcdef"


"acbefd"
```

We can also define a function `map :: (a -> b) -> Tree a -> Tree b` applying a function to all values stored in leaves of our tree.

[17]:
```
map :: (a -> b) -> Tree a -> Tree b
map f Empty = Empty
map f (Leaf x) = Leaf (f x)
map f (Branch x l r) = Branch (f x) (map f l) (map f r)

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ map (+1) t
```

```
Branch 5 (Branch 3 (Leaf 2) (Leaf 4)) (Branch 7 (Leaf 6) Empty)
```

In a similar way, we can define a `foldr` function that takes a function to reduce a tree value and an already reduced value, a zero value, and a tree and returns the reduced value.

[18]:
```
foldr :: (a -> b -> b) -> b -> Tree a -> b
foldr _ z Empty = z
foldr f z (Leaf x) = f x z
foldr f z (Branch x l r) = foldr f (f x (foldr f z r)) l

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ foldr (+) 0 t
print $ foldr (*) 1 t
```

```
21


720
```

Note that our in-order `values` function is a special case of a fold over the tree.

[19]:
```
values :: Tree a -> [a]
values = foldr (:) []

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ values t
```

```
"abcdef"
```

Since `Tree` is also an instance of the `Eq` typeclass, we can test trees for equality.

[20]:
```
t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
r = map (+1) t
print $ t == r
print $ t /= r
```

```
False


True
```

[30]:
```
draw :: Show a => Tree a -> String
draw t = pfxDraw "" t where
    pfxDraw p Empty          = p ++ "+:\n"
    pfxDraw p (Leaf x)       = p ++ "+-" ++ show x ++ "\n"
    pfxDraw p (Branch x l r) = center ++ left ++ right where
        center = p ++ "+-" ++ show x ++ "\n"
        left   = pfxDraw (p ++ "  ") l
        right  = pfxDraw (p ++ "  ") r

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
putStr $ draw t
```

```
+-4
  +-2
    +-1
    +-3
  +-6
    +-5
    +:
```

Haskell Tutorial: Typeclasses

November 5, 2020

[1]:
```
:opt no-lint
```

1 Typeclasses

Typeclasses describe a set of types that have a common interface and behaviour. We have already seen typeclasses such as `Ord` or `Num` or `Show`. In a nutshell, a typeclass defines common behaviour of types that belong to the typeclass. A type belonging to a typeclass implements the functions and behaviour defined by the typeclass. Note that the notion of a typeclass is very different from the concept of classes in object-oriented programming languages like C++ or Java. The closest concepts in popular object-oriented programming languages are probably Java interfaces.

The perhaps simplest standard typeclass is `Eq`. Types that are *instances* of the `Eq` typeclass implement an operator (a function) that can be used to determine whether two values of the type are equal. Note that the way equality is determined is type specific. For numeric types, equality is defined by comparing two numerical values. For lists, equality is defined by comparing all elements of two lists. There are also types where equality is not defined for all values or where it would be infeasible to decide equality of two values. The `Eq` typeclass can be defined in Haskell as follows:

[2]:
```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

The definition says that type `a` is an instance of the class `Eq` if there are (overloaded) operations `==` and `/=`, of the appropriate type, defined on it. A type implementing the `Eq` typeclass, i.e., a type that is an instance of the `Eq` typeclass, must implement these operation with the given type signature and the associated semantics. The functions (operations) of a typeclass are also called *methods* of the typeclass. We can go even further and provide default implementations in case this is possible. Note that in this particular example, the default implementations refer to each other, which means that a type instance of `Eq` only needs to define one of the two methods.

[3]:
```
class Eq a where
    (==) :: a -> a -> Bool
    (==) a b = not (a /= b) -- default implementation
    (/=) :: a -> a -> Bool
    (/=) a b = not (a == b) -- default implementation
```

Lets define a simple enumeration type for weekdays. We can make our `Weekday` type an instance

of the `Eq` typeclass by defining the methods of the `Eq` typeclass. Since we have default implementations for the methods, it is sufficient to define only one of the methods. The other method will then be available immediately due to its default implementation.

[4]:
```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

instance Eq Weekday where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _   == _   = False

Sun == Sun
Sun /= Sat
```

```
True
```

```
True
```

The next example shows how we can define lists of a given type to be an instance of `Eq`. We assume that the type of the list elements already is an instance of `Eq`.

[5]:
```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False

[Sun, Mon, Tue] == [Sun, Mon, Tue]
```

```
True
```

When we write generic (polymorphic) code, we often refer to types that belong to certain type classes, i.e., to all types that have a certain common behaviour. We can express that a type should be an instance of a certain typeclass by expressing a constraint on a type, which we call a *context*. The expression `Eq a` says that we constrain the type `a` to those types that are instances of the `Eq` typeclass.

The following example shows how our `Eq` typeclass can be used. In order to determine whether a certain element is in a list, we test whether the first element is equal to the element we are looking for or we test whether the element we are looking for is in the tail of the list. Our definition of the `elem` function can be used with all list element types that are instances of the `Eq` typeclass. This is expressed using the `Eq a` context in the type signature of our function. Since numeric types are instances of the `Eq` typeclass, we can determine whether a number is an element of a list of numbers. Similarly, since characters are instances of the `Eq` typeclass, we can determine whether a character is an element of a list of characters (a string).

[1]:
```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem y (x:xs) = y == x || elem y xs
```

It is possible to extend typeclasses. The `Ord` typeclass defines order relations. The `Ord` typeclass can be defined as an extension of the `Eq` typeclass. We say that `Eq` is a *superclass* of `Ord` or that `Ord` is a *subclass* of `Eq`. The definition below also provides default method implementations.

[6]:
```
data Ordering = LT | EQ | GT

instance Eq Ordering where
    LT == LT = True
    EQ == EQ = True
    GT == GT = True
    _  == _  = False

class (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    compare              :: a -> a -> Ordering
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ  -- default implementation
                | x <= y    = LT
                | otherwise = GT

    x <= y = compare x y /= GT  -- default implementation
    x < y  = compare x y == LT  -- default implementation
    x >= y = compare x y /= LT  -- default implementation
    x > y  = compare x y == GT  -- default implementation

    max x y | x <= y    =  y  -- default implementation
            | otherwise =  x
    min x y | x <= y    =  x  -- default implementation
            | otherwise =  y
```

An interesting typeclass is the `Functor` typeclass. It generalizes the `map` function we already know from lists to any parametric types that can contain values. The Functor typeclass is defined as follows:

[7]:
```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Note that the type variable `f` is applied to other types in `f a` and `f b`.

[8]:
```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)

instance Functor Tree where
    fmap f (Leaf x)     = Leaf (f x)
```

```
    fmap f (Branch l r) = Branch (fmap f l) (fmap f r)

t = Branch (Leaf "hello") (Leaf "world")
print t
print $ fmap reverse t
```

```
Branch (Leaf "hello") (Leaf "world")
```

```
Branch (Leaf "olleh") (Leaf "dlrow")
```

The type `Maybe` is an instance of the `Functor` typeclass as is the standard list type. Hence we can apply functions to `Maybe` values in a list or `Maybe` values in a tree; the `Functor` typeclass nicely hides the differences between a list and tree.

[15]:
```
-- instance Functor Maybe where
--     fmap f Nothing  = Nothing
--     fmap f (Just x) = Just (f x)

-- instance Functor [] where
--     fmap = map

print $ fmap (*2) (Just 1)

l = [Just 1, Just 2, Just 3, Nothing]
t = Branch (Branch (Leaf (Just 1)) (Leaf (Just 2))) (Leaf Nothing)

print $ fmap (fmap (*2)) l
print $ fmap (fmap (*2)) t
```

```
Overlapping instances for Functor Maybe arising from a use of `fmap'
Matching instances:
  instance Functor Maybe -- Defined at <interactive>:1:10
  instance Functor Maybe -- Defined at <interactive>:1:10
In the second argument of `($)', namely `fmap (* 2) (Just 1)'
In the expression: print $ fmap (* 2) (Just 1)
In an equation for `it': it = print $ fmap (* 2) (Just 1)
```

[10]:
```
```

Haskell Tutorial: Monads

November 26, 2020

[1]:
```
:opt no-lint
```

1 Monads

Functions in Haskell are pure. They do not have side effects, they do not have state. They are composable. And because of this purity, functional code is easier to analyze and verify. Real-world computing, however, requires to interact with a world that is full of state and side effects. Even in pure math functions, we can hit situations where results are undefined (e.g., division by zero or taking the square root of a negative real number). Monads provide a solution for dealing with such situations. While Monads seem a bit difficult to understand, you have already used Monads. (For the mathematically inclined students, Monads come out of category theory, a relatively recent branch of mathematics that started to evolve in the 1940s.)

1.1 Maybe Monad

We have already mentioned that `Maybe` can be used to handle situations where a value may not be present. `Maybe` is actually defined as a monad in Haskell. Lets look at an example and more specifically at the `head` and `tail` functions of lists. Both functions fail if the list is empty. Using `Maybe`, we can define safe `head` and `tail` functions:

[2]:
```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (x:xs) = Just xs

safeHead []
safeTail []
```

```
Nothing
```

```
Nothing
```

Lets assume that we want to combine our safe functions to obtain the third element of a list. Given a list, we could take twice the tail of it and then extract the head element. If we get `Nothing` at any point in time, we return `Nothing`. Otherwise, we return `Just a`.

[3]:
```
third :: [a] -> Maybe a
third xs =
    case safeTail xs of
      Nothing -> Nothing
      Just a -> case safeTail a of
                  Nothing -> Nothing
                  Just b -> safeHead b

third ""
third "1234"
```

```
Nothing
```

```
Just '3'
```

`Maybe` is implemented as a monad in Haskell, i.e., it is an instance of the `Monad` type class, which requires that `Maybe` implements a bind and a unit operator. The bind operator is written as `(>>=)` and the unit operator `return`. The bind operator for a monad `m` has the signature `(>>=) :: m a -> (a -> m b) -> m b`. The first argument of the bind operator is a monadic value of type `a` and the second argument is a function that takes a value of type `a` and returns a monadic value of type `b`. The bind operator itself returns a monadic value of type `b`. The `return` operator takes a value of type `a` and returns the value as a monadic value of type `a`. With the help of the bind operator, we can write our `third` function in a more compact way:

[4]:
```
third' :: [a] -> Maybe a
third' xs = safeTail xs >>= (\a -> safeTail a >>= (\b -> safeHead b))

third' ""
third' "1234"
```

```
Nothing
```

```
Just '3'
```

One way to look at this is that the `Maybe` monad boxes the values returned and that the bind operator allows us to apply a function to a boxed value, returning another boxed value. Since the bind operator is associative, we can remove parenthesis and we get an even shorter version:

[5]:
```
third' :: [a] -> Maybe a
third' xs = safeTail xs >>= \a -> safeTail a >>= \b -> safeHead b

third' ""
third' "1234"
```

```
Nothing
```

```
Just '3'
```

The bind operator for the `Maybe` monad is defined such that it returns `Nothing` if the first argument is `Nothing`. In other words, the lamda functions are only applied if the monad has a proper value.

Haskell provides the `do` notation in order to make longer sequences of bind operations easier to read:

[6]:
```
third'' :: [a] -> Maybe a
third'' xs = do
    a <- safeTail xs
    b <- safeTail a
    safeHead b

third'' ""
third'' "1234"
```

```
Nothing
```

```
Just '3'
```

This version is much easier to read and write since the handling of the mondic values is hidden in the bind operator of the `Maybe` monad.

1.2 IO Monad

Anything of type `IO` is called an action. An action is something we do that modifies the state of the real world, e.g., print something to the terminal, read input from the console, establish a network connection. The `IO` monad helps to separate the side effects from the pure functional code.

[7]:
```
isRaining :: IO Bool
isRaining = do
    putStrLn "Is it raining outside? (y/n)"
    input <- getChar
    return (input == 'y')
```

[8]:
```
isRaining
```

```
Is it raining outside? (y/n)
False
```

The `isRaining` function defines an action that returns an `IO Bool`. The `putStrLn` function takes a string and returns an IO action (putStrLn :: String -> IO ()). The `getChar` function is an IO action returning a char (getChar :: IO Char) and the <- operator runs the IO action and binds the result to `input`. The `return` operator finally encapsulates the boolean value returned by the comparison operator in an `IO Bool`. Note that we are sequencing IO actions here and finally use the return operator to return the result as an IO action. This allows us to use `isRaining` to compose more complex IO actions. Also note that `return` is simply turning a `Bool` value into an `IO Bool` value. Do not confuse the `return` operator of a Haskell monad with return statements of other programming languages that you may be familiar with.

[9]:
```
hello :: IO String
hello = do
    putStrLn "Please enter your name"
    name <- getLine
```

```
    putStrLn ("Hello " ++ name ++ "!")
    return name
```

[10]:
```
hello
```

```
Please enter your name
Hello !
""
```

As an exercise, try to convert the `do` notation used in the example above into the underlying sequence of bind operators (»=).

[11]:
```
dialog :: IO ()
dialog = do
    name <- hello
    rain <- isRaining
    putStrLn ("Hi " ++ name ++ ", I have heard that it is " ++ (if rain then ""␣
 ↪else "not ") ++ "raining")
```

[12]:
```
dialog
```

```
Please enter your name
Hello Lena!
Is it raining outside? (y/n)
Hi Lena, I have heard that it is not raining
```

[13]:
```

```

Haskell Tutorial: Functors, Applicatives, Monads

December 4, 2020

[1]:
```
:opt no-lint
import Control.Applicative
import Control.Monad
```

1 Functors

A very basic concept of Haskell is the application of a function to a value.

[2]:
```
(*5) 2
```

```
10
```

Sometimes, we have values that are contained in a certain context. For example, `Just 2` is the value 2 contained in the context `Just`. Another example is the value 2 contained in a list: `[2]`. We cannot directly apply the function `(*5)` to `Just 2` or `[2]` since we first have to obtain the value 2 from its context. You can think of the context as a box that contains a value. For lists, we already know that we can apply a function to all list elements by using the `map` function. A generalization of `map` is the function `fmap :: (a -> b) -> f a -> f b`.

[3]:
```
map  (*5) [2]
fmap (*5) [2]
fmap (*5) (Just 2)
fmap (*5) Nothing
```

```
[10]
```

```
[10]
```

```
Just 10
```

```
Nothing
```

Types that implement `fmap` are called functors. A functor is formally defined as a type class:

[4]:
```
-- class Functor f where
--     fmap :: (a -> b) -> f a -> f b
```

Types implementing the `Functor` type class, i.e., types that are instances of the `Functor` type class, implement `fmap` in a way that is consistent with the nature of the context.

[5]:
```
fmap' :: (a -> b) -> [a] -> [b]
fmap' _ [] = []
fmap' f (x:xs) = f x : fmap' f xs

fmap' :: (a -> b) -> Maybe a -> Maybe b
fmap' f (Just x) = Just $ f x
fmap' f Nothing = Nothing
```

Haskell commonly defines the special operator <$> for `fmap`.

[6]:
```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap

fmap (*5) [2]
(*5) <$> [2]
fmap (*5) (Just 2)
(*5) <$> (Just 2)
```

```
[10]
```

```
[10]
```

```
Just 10
```

```
Just 10
```

Interestingly, functions are functors as well. Hence, we can apply `fmap` to functions, giving us function composition.

[7]:
```
f = fmap (*5) (+5)
g = (*5) <$> (+5)
h = (*5) . (+5)

f 0
g 0
h 0
```

```
25
```

```
25
```

```
25
```

Types implementing the `Functor` type class have to implement `fmap` such that the following functor laws hold (`id` is the identity function):

[8]:
```
-- fmap id = id
-- fmap (f . g) = fmap f . fmap g
```

2 Applicative

The `Applicative` type class extends the `Functor` type class. While the `Functor` type class assumes that the values are wrapped in a context (a box), `Applicative` also supports functions wrapped in a context (a box). The `Applicative` type class is defined as follows:

[9]:
```
-- class Functor f => Applicative f where
--    pure  :: a -> f a
--    (<*>) :: f (a -> b) -> f a -> f b
```

The function `pure` takes a value and puts it into the context (a box). The operator `(<*>)` applies a function in a context to a value in a context and it returns a value in a context.

[10]:
```
Just (*5) <*> Just 2
[(*5)] <*> [2]

[(*1),(*2),(*3)] <*> [1..3]
```

```
Just 10
```

```
[10]
```

```
[1,2,3,2,4,6,3,6,9]
```

The following example combines `<$>` (fmap) with `<*>`. The first parenthesis applies the `(*)` function to `Just 5`, which givs us the function `Just (*5)`. This function is then applied to `Just 2`, which gives us `Just 10`.

[11]:
```
((*) <$> (Just 5)) <*> (Just 2)
```

```
Just 10
```

Types implementing the `Applicative` type class have to implement the `<*>` operator and the function `pure` such that the following applicative laws hold (`id` is the identity function):

[12]:
```
-- pure id <*> v = v
-- pure f <*> pure x = pure (f x)
-- u <*> pure y = pure ($ y) <*> u
-- u <*> (v <*> w) = pure . <*> u <*> v <*> w
```

The last applicative law says that `<*>` is associative.

3 Monads

The `Monad` type class extends the `Applicative` type class. It defines a function `return` that puts a value into a monad and a function `(>>=)` called *bind* that takes a value in a monad (a box), a function that takes a value and returns a value in a possibly different monad, and returns the later monadic value.

[13]:
```
-- class Applicative m => Monad m where
--     return :: a -> m a
--     (>>=)  :: m a -> (a -> m b) -> m b
```

To illustrate this idea, we use the type `Maybe`, which happens to be an instance of the Monad type class. Lets start with a function that takes an `Integer` and returns a `Maybe Integer`.

[14]:
```
half :: Integer -> Maybe Integer
half x = if even x
         then Just (x `div` 2)
         else Nothing

half 8
```

```
Just 4
```

Unfortunately, we cannot compose `half` with itself since `half` takes an `Integer` but returns a boxed value. This is where the bind operator can help. (Note that `return` is a rather confusing name for what the function does.) There is another operator `(=<<)`, which swaps the first two arguments, which is sometimes handy. Note that the operators kind of indicate how the value is flowing through a sequence of functions.

[15]:
```
Just 8 >>= half
Just 4 >>= half
Just 2 >>= half
Just 1 >>= half

return 8 >>= half >>= half >>= half >>= half
half =<< half =<< half =<< half =<< return 8
```

```
Just 4
```

```
Just 2
```

```
Just 1
```

```
Nothing
```

```
Nothing
```

```
Nothing
```

Types implementing the `Monad` type class have to implement the `>>=` operator and the function `return` such that the following monad laws hold (`id` is the identity function):

[16]:
```
-- return a >>= f = f a
-- m >>= return = m
-- (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Monds play an important role in Haskell since they can be used to encapsulate side effects. For example, the `IO` Monad takes care of input and output operations. The `getLine :: IO String`

function takes no arguments and returns an IO action to read a string from the input. The `putStrLn :: String -> IO String` takes a string and returns an IO action to print it. These functions can be chained together:

[17]:
```
getLine >>= putStrLn
```

```
some input typed in here
```

In case the result of a chained function is not needed, one can use the *then* (`>>`) operator.

[18]:
```
putStr "Hello" >> putStr " " >> putStrLn "World"
```

```
Hello World
```

Haskell has a special notation for monads, the `do` notation. Using the do notation, code manipulating monads starts to look like imperative code. Hence, some people call monads *programmable semicolons* since semicolons are often used to sequence statements in imperative languages. However, in Haskell, the behaviour of monads is programmable.

[19]:
```
do
    putStr "Hello"
    putStr " "
    putStrLn "World"

do
    line <- getLine
    putStrLn line
```

```
Hello World
```

```
some more input typed in here
```

4 Summary

Below is a summary of the operators defined by the type classes discussed here. Recall that a `Monad` type is also `Applicative` and that an `Applicative` type is also a `Functor`.

[20]:
```
-- (<$>) ::   (a -> b) -> f a -> f b    -- Functor
-- (<*>) :: f (a -> b) -> f a -> f b    -- Applicative
-- (=<<) :: (a -> m b) -> m a -> m b    -- Monad
-- (>>=) :: m a -> (a -> m b) -> m b    -- Monad
```

Haskell Tutorial: Sets

November 6, 2020

[1]:
```
:opt no-lint
```

1 Ordered Sets

Haskell has support for typed ordered sets with fast insert, delete, and lookup operations. It is highly recommended to use the set implementation instead of abusing lists to represent ordered sets. To use the set functions, you have to import `Data.Set`. It is a recommended to use a qualified import in order to handle name clashes with functions defined by the prelude. And using a qualified import may also help with readability.

[2]:
```
import qualified Data.Set as Set
```

A common way to create sets is to create them from a list. But there are also convenient ways to create an empty set or a set has a single member, called a *singleton*.

[3]:
```
s0 = Set.empty
s1 = Set.singleton 'c'
s2 = Set.fromList "Hello World"
s3 = Set.fromList ['a'..'z']
```

The `null` function can be used to test whether a set is empty while the `size` function returns the number of elements of a set.

[4]:
```
map Set.null [s0,s1,s2,s3]
map Set.size [s0,s1,s2,s3]
```

```
[True,False,False,False]
```

```
[0,1,8,26]
```

The `toAscList` function returns the elements of a set as a list in ascending order. Similarly, `toDescList` returns the elements of a set as a list in descending order.

[5]:
```
Set.toAscList s3
Set.toDescList s3
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
"zyxwvutsrqponmlkjihgfedcba"
```

Set `member` function can be used to test whether a value is a member of a given list and the `isSubsetOf` function can be used to test whether the first set is a subset of the second set.

[6]:
```
map (Set.member 'o') [s0,s1,s2,s3]
map (Set.isSubsetOf s2) [s0,s1,s2,s3]
```

```
[False,False,True,True]
```

```
[False,False,True,False]
```

The `insert` and `delete` functions can be used to add and delete elements. There is also a `filter` function that can be used to create a set from all members of a set where a given predicate is true.

[7]:
```
Set.null $ Set.delete 42 $ Set.insert 42 $ Set.insert 42 s0
Set.toAscList $ Set.filter (< 'g') s3
```

```
True
```

```
"abcdef"
```

You can use the standard set operations `union`, `intersection`, and `difference`. The `isSubsetOf` function tests whether the first argument is a subset of the second argument and the `member` function tests whether a member is in a set. The `disjoint` function tests whether two sets are disjoint, i.e., they have no common elements.

[8]:
```
Set.union s2 s3
Set.intersection s2 s3
Set.difference s2 s3
```

```
fromList " HWabcdefghijklmnopqrstuvwxyz"
```

```
fromList "delor"
```

```
fromList " HW"
```

Since the sets have an order, it is possible to split sets at a certain elem. Note that the splitting element is not required to be a member of the set.

[9]:
```
sp = Set.split 'm' s3
Set.elems $ fst sp
Set.elems $ snd sp
```

```
"abcdefghijkl"
```

```
"nopqrstuvwxyz"
```

[10]:
```
Set.split 32 $ Set.fromList [0,10..60]
```

```
(fromList [0,10,20,30],fromList [40,50,60])
```

Many higher order funtions like `map`, `filter` and various folds are implemented for sets as well.

[11]:
```
Set.map (succ) $ Set.fromList [1..10]
```

```
fromList [2,3,4,5,6,7,8,9,10,11]
```

[12]:
```
Set.filter even $ Set.fromList [1..10]
```

```
fromList [2,4,6,8,10]
```

[13]:
```
Set.foldr (+) 0 $ Set.fromList [1..10]
Set.foldr (:) [] $ Set.fromList [1..10]
Set.foldl (flip (:)) [] $ Set.fromList [1..10]
```

```
55
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

1.1 Internals

Sets are internally represented as balanced ordered binary trees. It is possible to look at the internal representation by using the `showTree` function. Note that there may be leaf nodes without a value.

[14]:
```
putStrLn $ Set.showTree s3
```

```
'h'
+--'d'
|  +--'b'
|  |  +--'a'
|  |  +--'c'
|  +--'f'
|     +--'e'
|     +--'g'
+--'p'
   +--'l'
   |  +--'j'
   |  |  +--'i'
   |  |  +--'k'
   |  +--'n'
   |     +--'m'
   |     +--'o'
   +--'t'
      +--'r'
      |  +--'q'
      |  +--'s'
      +--'x'
         +--'v'
         |  +--'u'
         |  +--'w'
         +--'y'
```

```
+--|
+--'z'
```

Haskell Tutorial: Maps

September 27, 2019

[1]:
```
:opt no-lint
```

0.1 Maps

Haskell has support for maps that map a key to a value. Mathematically, you may think of a map as a binary relation mapping keys to values. To use the map functions, you have to import `Data.Map`. There are actually two different kinds of maps in the Haskell library: `Data.Map.Strict` provides finite maps for (sometimes called dictonaries) for situations where all values will be forced to exist. If laziness is required, i.e., you want to store something potentially infinite in a map, you should use `Data.Map.Lazy`. Haskell maps are pretty powerful. This document only provides a very basic overview. For a full description, see the official documentation.

[2]:
```
import qualified Data.Map as Map
```

A common way to create maps is to create them from a list of tuples. But there are also convenient ways to create an empty map or a map has a single member, called a *singleton*.

[3]:
```
m0 = Map.empty
m1 = Map.singleton "Eve" 42
m2 = Map.fromList $ zip ["hello", "world"] [1..]
```

The `null` function can be used to test whether a map is empty while the `size` function returns the number of elements of a map.

[4]:
```
map Map.null [m0, m1, m2]
map Map.size [m0, m1, m2]
```

```
[True,False,False]
```

```
[0,1,2]
```

Maps can be converted back to lists using the `toList`, `toAscList`, and `toDescList` functions. The `toAscList`, and `toDescList` functions return the map tuples ordered by the keys.

[5]:
```
Map.toList m2
Map.toAscList m2
Map.toDescList m2
```

```
[("hello",1),("world",2)]
```

```
[("hello",1),("world",2)]
```

```
[("world",2),("hello",1)]
```

The `elems` function returns the list of all values while the `keys` function returns the list of keys.

[6]:
```
Map.elems m2
Map.keys m2
```

```
[1,2]
```

```
["hello","world"]
```

There are many ways to lookup values for key. First, you can use the `member` function to test whether a certain key exists in the map. The `!` operator returns the value of a key that exists in a map or it throws an error. There are other functions that avoid throwing an error. A simple solution is `findWithDefault`, which returns the value of a key in a map or a default value.

[7]:
```
Map.member "hello" m2
m2 Map.! "hello"
Map.findWithDefault 0 "hello" m2
Map.findWithDefault 0 "hello" m1
```

```
True
```

```
1
```

```
1
```

```
0
```

0.1.1 TODO

- explain insert, delete, update
- explain unions, intersection, difference
- explain map, mapWithKey, filter, mapKeys

[8]:
```

```

Haskell Tutorial: Tips and Tricks

October 4, 2020

1 Tips and Tricks

This is a somewhat unstructured collection of notes that may be useful. Consider this more a scratch space instead of a well thought out sequence of thoughts. If you thing more tips and tricks should be included, drop me a note and I will see whether I can add additional material.

1.1 Debugging

Most of the following is true for any programming language. So take the tricks and the advise serious and try to follow them.

Step #0 Compile your code. Code that has never seen a compiler in most cases does not work. There are even online Haskell interpreter that do not require any installation, simply google. There is no excuse for submitting code that has not seen a compiler.

Step #1 Turn on warnings (-Wall in ghci) and pay attention to all warnings and errors. Fix things. Iterate until the compiler is happy. And no, the compiler is not broken. The chances that a beginner finds a serious compiler bug is low.

Step #2 Write test cases (if not done yet). Automate your tests so you can re-run them easily after each and every change. Yes, automate, re-run all tests after each and every change. Ad-hoc testing is a waste of your time. So automate testing, stop wasting time.

Step #3 If your code does not pass your collection of test cases (your test suite) and it is not immediately clear what is wrong, you may need to go and figure what your code actually does:

1. If multiple functions are involved, test them separately. Go back to step 0 for each of them.
2. If you have a toy around, explain your code to the toy. (If no toy is available, use a student instead.) Explaining your code to someone (even to a toy) often reveals the bug. I am serious, try it, it often works.
3. If still no clue, it may help to obtain trace information. The `Debug.Trace.trace :: String -> a -> a` function may help. Consider this example:

[1]:
```
sum' :: [Integer] -> Integer
sum' [] = 0
sum' (x:xs) = x + sum' (tail xs)
```

```
sum' [1..10] == 55
```

```
False
```

Assuming you do not spot the error immediately, you can add the following import and another definition of sum' to generate nice traces.

[2]:
```
import Debug.Trace

sum' :: [Integer] -> Integer
sum' xs | trace ("trace: sum' " ++ show xs) False = undefined
sum' [] = 0
sum' (x:xs) = x + sum' (tail xs)

sum' [1..10] == 55
```

```
False
```

This will show on the error output the following trace:

```
trace: sum' [1,2,3,4,5,6,7,8,9,10]
trace: sum' [3,4,5,6,7,8,9,10]
trace: sum' [5,6,7,8,9,10]
trace: sum' [7,8,9,10]
trace: sum' [9,10]
trace: sum' []
```

Now it should be easy to spot why the function returns wrong results.

1.2 Unit Testing

Unit testing is a software testing method by which individual units (colletions functions or modules) are tested to determine whether they are working properly for the test cases. Unit tests are extremely valuable since they automate ad-hoc testing and they can be executed again after each code change. There are unit testing frameworks for almost any programming language. For Haskell, there is HUnit. It is best explained by an example.

[3]:
```
import Test.HUnit

fib = 0 : 1 : zipWith (+) fib (tail fib)

tests = TestList
        [ TestCase ([0,1,1,2,3] @=? take 5 fib)
        , TestCase (354224848179261915075 @=? fib !! 100)
        ]

runTestTT tests
```

```
Counts {cases = 2, tried = 2, errors = 0, failures = 0}
```

To use HUnit, you have to import `Test.HUnit`. The function we want to test is the really fancy version of the function producing Fibonacci numbers. To do this, we create a `TestList`, which contains a collection of test cases. Our first test case intents to check the start of the sequence. The `@=?` operator has the expected value on the left side and the code we want to test on the right side. In other words, we compare the result of `take 5 fib` against the known expected result `[0,1,1,2,3]`. In the second test case, we check whether the 100th fibonacci number is calculated correctly. To run our tests, we can use the `runTestTT` function, which produces a test-based report in the terminal.

The code above is an example. In practice, you will likely put your test cases into a separate file. If you use a build system like Cabal, you can declare that you have test cases and then `cabal test` will run the tests.

1.3 Property Testing

Unit testing uses test cases that have been carefully defined during the different phases of a software project. It is always recommended to create examples while trying to understand a problem and before searching for an algorithm and finally writing a programm. These examples are natural test cases. Likewise, when bugs are found, it is good practice to write tests that trigger the buggy behaviour before fixing the bug.

Property based testing follows a different idea: Instead of working out specific test cases, the programmer defines properties that should generally hold for the functions that have been implemented. A test engine is then going to generate random test cases and verfiying that the properties hold for these test cases. Haskell has a QuickCheck package that does this very nicely. Lets again look at an example. We are trying to reimplement the `take` function and we want to ensure it does the same as the `take` function provided by Haskell.

[4]:
```haskell
import Test.QuickCheck

take' :: Int -> [a] -> [a]
take' 0 _ = []
take' n (x:xs) = x : take' (n-1) xs

prop_takeapp :: Int -> [Int] -> Bool
prop_takeapp n xs = take' n xs == take n xs

quickCheck prop_takeapp
```

```
*** Failed! (after 1 test):
Exception:
  <interactive>:(2,1)-(3,35): Non-exhaustive patterns in function take'
1
[]
```

We define the property `prop_takeapp` that compares the output produced by our `take'` function

with the output of Haskell's `take` function. We then pass our property to QuickCheck and quickly finds a test case where things break (`tail' 1 []`). Obviously, we are not handling the case where the list is shorter than the first argument.

[5]:
```
import Test.QuickCheck

take' :: Int -> [a] -> [a]
take' 0 _ = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs

prop_takeapp :: Int -> [Int] -> Bool
prop_takeapp n xs = take' n xs == take n xs

quickCheck prop_takeapp
```

```
*** Failed! Falsifiable (after 6 tests and 5 shrinks):
-1
[0]
```

Apparently we have another problem. If the first argument is negative, it makes sense to return an empty list. Our code, however, interprets a negative number pretty much as positive number. This is simple to fix now that we know about it.

[6]:
```
import Test.QuickCheck

take' :: Int -> [a] -> [a]
take' n _
    | n <= 0 = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs

prop_takeapp :: Int -> [Int] -> Bool
prop_takeapp n xs = take' n xs == take n xs

quickCheck prop_takeapp
```

```
+++ OK, passed 100 tests.
```

Our `tail'` definition now handles negative numbers like the original `tail` definition and QuickCheck does not find any test cases where the two functions produce different results. Of course, this does not prove that our definition of `take` is correct, but we know at least that it is correct for 100 test cases.