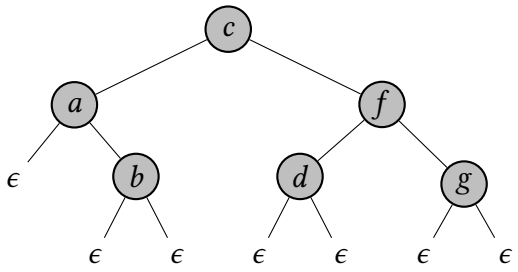
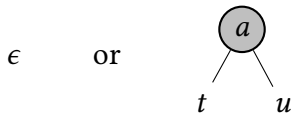


4 Algebraic datatypes

Exercise 4.1 (Warm-up: data structures, `BinaryTree.lhs`). Binary trees are probably the second most popular data structure after lists and arrays. Binary trees are everywhere: expression trees, pedigrees, and tournament trees are examples of binary trees; binary trees can be used to implement sets, finite maps, and priority queues.

A *binary tree* is either

- empty, or
- a node that consists of a left tree, an element, and a right tree.



The recursive definition introduces *binary* trees as each node features exactly *two sub-trees*.

In computer science, trees typically grow downwards; that is, trees are drawn with the root at the top. The terminology surrounding trees is partly inspired by biological trees (root, leaf etc) and partly by pedigrees (child, parent etc). Consider the binary tree shown above.

- The node c is the *root*; the empty sub-trees are also known as *leaves*. (A tree with n nodes has $n + 1$ leaves. Would you agree?)
- All nodes, with the exception of the root, have a *parent*: d's parent is f; f's parent is c; c has no parent as it is the root.
- Nodes may or may not have children: f has *children* d and g; d has no children; a has one child.
- The nodes a and f are *siblings*; d and g are *siblings*.

The recursive definition of trees can be easily transliterated into a Haskell datatype definition.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
deriving (Show)
```

Like the type of lists, `Tree elem` is a *container type*: a binary tree contains elements of type `elem`. Thus, we have a tree *of* strings, a tree *of* lists *of* characters, or a tree *of* integers, etc.

1. Capture the binary tree shown above as a Haskell expression of type `Tree Char`.

2. Conversely, picture the Haskell expressions below.

```
Node Empty 4711 (Node Empty 0815 (Node Empty 42 Empty))
Node (Node (Node Empty "Frits" Empty) "Peter" Empty) "Ralf" Empty
Node (Node Empty 'a' Empty) 'k' (Node Empty 'z' Empty)
```

3. We have emphasized in the lectures that every datatype comes with a pattern of definition. Write down the "tree design pattern". Apply the design pattern to define a function `size :: Tree elem → Int` that calculates the number of elements contained in a given tree. The size of the tree shown in above is 6.

4. Define functions `minHeight, maxHeight :: Tree elem → Int` that calculate the length of the shortest and the length of the longest path from the root to a leaf. The minimum height of our running example above is 2; the maximum height is 3.

5. What is the relation between the size, the minimal, and the maximal height of a tree?

6. Define a function `member :: (Eq elem) ⇒ elem → Tree elem → Bool` that determines whether a specified element is contained in a given binary tree.

Exercise 4.2 (Programming, `BinaryTree.lhs`).

1. Define functions `preorder, inorder, postorder :: Tree elem → [elem]` that return the elements contained in a tree in pre-, in-, and post-order, respectively e.g.

```
⋙ preorder abcdfg
"cabfdg"
⋙ inorder abcdfg
"abcdfg"
⋙ postorder abcdfg
"badgfc"
```

   where `abcdfg` represents the tree shown at the top of the previous page (see Exercise 4.1). What is the running time of your programs?

2. Define a function `layout :: (Show elem) ⇒ Tree elem → String` that turns a tree into a string, showing one element per line and emphasizing the structure through indentation e.g. `putStr (layout abcdfg)` produces (turn your head by 90° to the left):

```
    / 'a'
        \ 'b'
- 'c'
        / 'd'
    \ 'f'
        \ 'g'
```

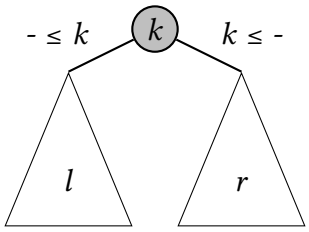
3. *Optional:* Define a function that generates suitable LaTeX-code for typesetting binary trees. (There are a variety of packages for typesetting trees that you may want to use as a starting point.)

Exercise 4.3 (Programming, `BinaryTree.lhs`).

1. Define a function `build :: [elem] → Tree elem` that constructs a binary tree from a given list of elements such that `inorder ∘ build = id`. The shape of the tree does not matter. (Apply the list design pattern.)
2. Define a function `balanced :: [elem] → Tree elem` that constructs a balanced tree from a given list of elements, i.e. for each node the size of the left and the size of the right sub-tree should differ by at most one. The order of elements, however, does not matter. (Again, apply the list design pattern.)
3. Harry Hacker claims that his function `create :: Int → Tree ()` can construct a tree of size `n` in logarithmic time i.e. `create n` takes $\Theta(\log n)$ steps. Genius or quacksalver?

Exercise 4.4 (Programming: data structures, `BinarySearchTree.lhs`). A *binary search tree* is a binary tree such that

- the left sub-tree of every node only contains elements less than or equal to the element in the node;
- the right sub-tree of every node only contains elements greater than or equal to the element in the node.



In other words, an inorder traversal of the tree yields a non-decreasing sequence of elements. For example, `registry` defined below is a binary search tree.

```
registry  ::  Tree String
registry  =  Node (Node (Node Empty "Frits" Empty) "Peter" Empty) "Ralf" Empty
```

1. Define a function `member :: (Ord elem) :: elem → Tree elem → Bool` that determines whether a specified element is contained in a given binary search tree. What's the difference to Exercise 4.1.6?
2. Define a function `insert :: (Ord elem) ⇒ elem → Tree elem → Tree elem` that inserts an element into a search tree e.g.

```
 ⋙ insert "Nienke" registry
Node (Node (Node Empty "Frits" (Node Empty "Nienke" Empty))
  "Peter" Empty) "Ralf" Empty
```

   (Just in case you have implemented binary search trees in an imperative or object-oriented language: is there a fundamental difference between the implementations?)

3. Define a function `delete :: (Ord elem) ⇒ elem → Tree elem → Tree elem` that removes an element from a binary search tree e.g.

```
⋙ delete "Frits" registry
Node (Node Empty "Peter" Empty) "Ralf" Empty
```

   If the element is not contained in the tree, the input tree is simply returned unchanged.

4. Use the library of Exercise 3.5 to test your code. In particular, define a function

```
isSearchTree :: (Ord elem) ⇒ Tree elem → Bool
```

   that checks whether a given binary tree satisfies the search tree property. The most difficult part is to define a function that generates binary *search* trees. One approach is to program a function `trees :: [elem] → Probes (Tree elem)` that generates *all* trees whose inorder traversal yields the given list of elements: `and [ inorder t == xs | t ← trees xs ]`. Applied to an ordered list, `tree` will then generate search trees. (For the mathematically inclined: which sequence does `[ length (trees [1 .. i]) | i ← [0 .. ]]` generate? If you are clueless, `oeis.org` is your friend.)

Exercise 4.5 (Worked example: red-black trees, `RedBlackTree.lhs`). The running time of `member`, `insert`, and `delete` is bounded by the height of the binary search tree. Sadly, in the worst case the height is proportional to the size. To improve linear to logarithmic running time, a multitude of balancing schemes have been proposed: 2-3 trees, AA-trees, 2-3-4 trees, red-black trees, AVL-trees, 1-2 brother trees, B-trees, (a, b)-trees, size-balanced trees, splay trees, etc. This exercise discusses a fairly popular balancing scheme: red-black trees.

A red-black tree is a binary tree whose nodes are coloured either red or black.

```
data RedBlackTree elem
  = Leaf
  | Red   (RedBlackTree elem) elem (RedBlackTree elem)
  | Black (RedBlackTree elem) elem (RedBlackTree elem)
  deriving (Show)
```

Elements of this type are required to satisfy:

Red-condition: Each red node has a black parent.

Black-condition: Each path from the root to a leaf contains exactly the same number of black nodes—this number is called the tree's *black height*.

The conditions ensure that the height of a red-black tree of size n is bounded by $\Theta(\log n)$. Do you see why? Note that the red-condition implies that the root of a red-black tree is black.

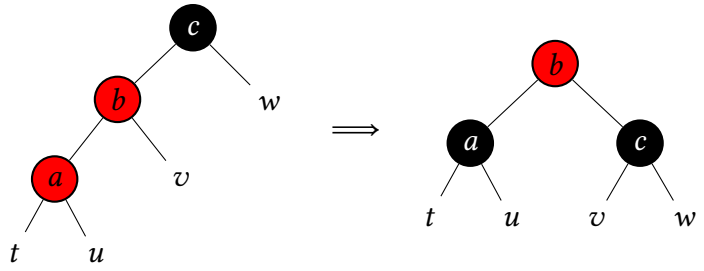
1. Adapt the membership test to red-black trees.

```
member :: (Ord elem) ⇒ elem → RedBlackTree elem → Bool
```

2. Adapt the insertion algorithm to red-black trees. (Do not worry about red- and black-conditions initially.)

```
insert :: (Ord elem) ⇒ elem → RedBlackTree elem → RedBlackTree elem
```

Let's agree that we always create a red node for the to-be-inserted element. Consequently, the black-condition stays intact. Only the red-condition is possibly violated. The idea is to repair violations through *local* transformations along the search path. The diagram below shows a tree shape that violates the red-condition and illustrates how to repair the defect.



There are three other illegal tree shapes that can occur after inserting a red node. Which ones?

Now introduce a *smart constructor*

```
black :: RedBlackTree elem → elem → RedBlackTree elem → RedBlackTree elem
```

that implements these local transformations, and replace the actual constructor `Black` by the smart constructor in the code for `insert`. What happens if a red node percolates to the top?

3. Again, test the code using the library of Exercise 3.5. Define a function

   ```
   isRedBlackTree :: RedBlackTree elem → Bool
   ```

   that checks whether a tree is indeed a proper red-black tree. To exercise `insert` you also need to write a generator for red-black trees. Can you adopt the function `trees` of Exercise 4.4.4 to this setting?

Exercise 4.6 (Programming and mathematics, `Calculus.lhs`). Lisa Lista's younger brother just went through differential calculus at high school. She decides to implement derivatives in Haskell to be able to easily double-check his homework solutions. To this end she introduces a datatype of primitive functions and a datatype of compound functions:

```
data Primitive
  = Sin                         -- trigonometric: sine
  | Cos                         -- cosine
  | Exp                         -- exponential
  deriving (Show)

infixl 6 :+:
infixl 7 :*:
infixr 9 :.:

data Function
  = Const Rational          -- constant function
  | Id                      -- identity
```

```
  |  Prim Primitive          -- primitive function
  |  Function :+: Function   -- addition of functions
  |  Function :*: Function   -- multiplication of functions
  |  Function :.: Function   -- composition of functions
deriving (Show)
```

The idea is that each element of `Primitive` and `Function` *represents* a function over the reals. The following table shows some functions and their representations.

function	representation
$f(x) = c$	`Const c`
$f(x) = x$	`Id`
$f(x) = 2x^2 + 5x$	`Const 2 :*: Id :*: Id :+: Const 5 :*: Id`
$f(x) = 2\sin(x) + x$	`Const 2 :*: Prim Sin :+: Id`
$f(x) = x\cos(x^2)$	`Id :*: Prim Cos :.: (Id :*: Id)`

1. Define a function `apply :: Function → (Double → Double)` that applies the representation of a function to a given value. In a sense, `apply` maps syntax to semantics: the representation of a function is mapped to the actual function.

2. Define a function `derive :: Function → Function` that computes the derivative of a function.

3. After Lisa has captured the rules of derivatives as a Haskell function (see Part 3), she tests the implementation on a few simple examples. The initial results are not too encouraging:

   ```
   ⋙ derive (Const 1 :+: Const 2 :*: Id)
   Const (0 % 1) :+: Const (0 % 1) :*: Id :+: Const (2 % 1) :*: Const (1 % 1))
   ⋙ derive (Id :.: Id :.: Id)
   Const (1 % 1) :.: (Id :.: Id) :*: (Const (1 % 1) :.: Id :*: Const (1 % 1))
   ```

   Implement a function `simplify :: Function → Function` that simplifies the representation of a function using the laws of algebra. (This is a lot harder than it sounds!) *Hint:* use smart constructors.

Exercise 4.7 (ADT Notations). A number of algebraic types are defined below. If possible, give three *different* expressions for each of these types. Examples of expressions of type `Int` are `1`, `2` and `1 + 1`.

```
data Day      = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
data Nat      = Nat Int
data PosNat   = Pos Int | NotAPosNat
data Number   = Whole Nat | Decimal Real
data Void     = Void
```

Exercise 4.8 (Type inference and ADTs). Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions, using the definitions of exercise 4.7. Please note that you can use GHCi to check your answer.

```
f1 Saturday  = True
f1 Sunday    = True
f1 _         = False

f2 Monday    = 1
f2 Tuesday   = 2
f2 Wednesday = 3
f2 Thursday  = 4
f2 Friday    = 5
f2 Saturday  = 6
f2 Sunday    = 7

f3 x y       = f2 x == f2 y

f4 x         = Nat x

f5 (Nat x)   = x

f6 x
| x > 0      = Pos x
| otherwise  = NotAPosNat

f7 (Pos x) (Pos y) = Pos (x+y)
f7 _ _             = NotAPosNat

f8 (Pos x) (Pos y)
| x > y            = Pos (x-y)
| otherwise        = NotAPosNat
```

Exercise 4.9 (Record Notations). A number of record and algebraic data types are defined below. Give one expression for each of the *record* types.

```
data Month  = JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC
data Date   = Date {day :: Int, month :: Month, year :: Int}
data Name   = Name {first :: String, last :: String, prefix :: String}
data Person = Person {name :: Name, birth :: Date, father :: Name, mother :: Name}
```

Exercise 4.10 (Day). Module `Day` defines the following algebraic data type:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Write the following functions:

1. `friday_on_my_mind :: Day → Bool` which yields `True` only if the argument is `Friday`.
2. `is_weekend :: Day → Bool` which yields `True` only if the argument is `Saturday` or `Sunday`.
3. `on_my_mind :: Day → Day → Bool` which yields `True` only if the two arguments are equal. For instance, `Friday `on_my_mind` d` is `True` only if d = `Friday`.

4. `yesterday :: Day → Day` and `tomorrow :: Day → Day` that return the previous day and next day of their argument respectively.

Exercise 4.11 (Date). Module `Date` defines the following types:

```
data Date  = Date { year :: Int, month :: Month, day :: Int }
data Month = January | February  | March   | April    | May | June | July
           | August  | September | October | November | December
```

Implement the following functions on these types:

1. `is_leap_year y` determines whether year y is a leap year. In a leap year, month february has 29 days instead of the usual 28. A year is a leap year if it is dividable by 4, except if it is a multiple of 100 that is not dividable by 400. For instance, 1600 is a leap year, but 1700 is not a leap year.

2. `no_of_days y m` determines the number of days of month m in year y (taking leap years into account).

3. `yesterday` and `tomorrow` that return the date of the previous day and next day respectively (taking leap years into account).

Exercise 4.12 (Fractions). Choose a suitable representation for the type `Q` in module `Q.hs` to implement rational numbers $\mathbb{Q}$, also known as fractions. The following operations on `Q` values should be supported:

- The functions `equalQ` and `smallerQ` test whether two fractions are equal or smaller respectively.
- The operations `plusQ`, `decrementQ`, `timesQ`, and `divideQ` implement the mathematical operations $+$, $-$, $\cdot$, and $\div$.
- Just like integers and reals, fractions are signed numbers (can be negative, positive, or zero). `absoluteQ` takes the absolute value, `signOfQ` returns the sign of the argument and `negateQ` flips the sign.
- `isIntQ` tests whether the argument corresponds to a whole number, and yields `True` only in that case. `IQ` creates a fractional value of its `Int` argument. `QR` creates the floating point value that approximates the fractional `Q` argument.

In your implementation you can use the `Int` instance of the function `gcd`, which calculates the *greatest common divisor* of two positive integers. This function is defined in the `GHC.Real` module, which is included in `Prelude`.

Exercise 4.13 (Card). A standard card deck consists of 52 cards. Every card has a *suit* and a *value*. The suits are *heart* (♥), *diamond* (♦), *spade* (♠), and *club* (♣). The possible values are the numbers 2 up to and including 10, and *jack*, *queen*, *king*, and *ace*. Implement the following components and use algebraic data types and records where possible.

1. **Representation** Design suitable data structures to represent a deck of cards. The types representing a card, suit, and value should be named `Card`, `Suit`, and `Value` respectively. You *must* choose an algebraic type *or* a record type to create `Card`. You are free to choose your representation for `Suit` and `Value`.

2. **Equality of cards** Write the function `equalCard` that returns `True` only if the two arguments are the same card.

3. **Printing and parsing** Write the functions `showCard` and `parseCard`. The `parseCard` function should be the inverse of the `showCard` instance.

   $\forall c$ `::` `Card`: `parseCard (showCard` c`)` $= c$.

   Can you also make sure that the following holds for every `String` s?

   `let c :: Card; c = parseCard` s `in showCard c` $= s$?

Hints to practitioners 4. Inventing names is hard. In Haskell, we introduce names for values including functions, types and type variables, datatypes and their constructors, type classes and their methods (see §6), and modules (see Appendix **??**). As a rule of thumb, the wider the scope of an entity, the more care should be exercised in choosing a suitable identifier.

For example, the argument of a function only scopes over the function body i.e. the right-hand side of an equation in definitional style.

```
swap :: (a, b) → (b, a)
swap (x, y)  =  (y, x)
```

Hence it is perfectly fine to use short names such as `x` and `y`, which are, of course, not very telling. Or would you prefer the definition below?

```
swap' (firstComponent, secondComponent)
=  (secondComponent, firstComponent)
```

Likewise, the type variables `a` and `b` only scope over the signature of `swap`. Again, it acceptable to use short names. On the other hand, the signature of `swap` may appear in the interface documentation of a module. But would you prefer the signature below?

```
swap' ::  (typeOfFirstComponent, typeOfSecondComponent)
       → (typeOfSecondComponent, typeOfFirstComponent)
```

By contrast, the identifier `swap` scopes over the entire module where its definition resides. (And beyond, if `swap` is exported by the module.) Hence the name should be chosen with care. (Are you happy with `swap` or would you prefer `swapTheComponentsOfAPair`?) The same rule applies to names of datatypes, constructors, and modules, all of which are potentially globally visible.

Tastes differ, of course. If you intensively dislike the name of a library function or type, you are free to introduce synonyms e.g.

```
type Z  =  Integer
sort  =  insertionSort
```

(As an aside, we can also rename module names in qualified imports e.g.
`import qualified BinarySearchTree as Set`.) However, we cannot introduce synonyms for constructors (unless you use a recent extension of Haskell called pattern synonyms) or class and method names.