Pandoc's Haskell

draft version 9

Contents

Chapter 1

Preface

1.1 Goals of the tutorial

1. reach intermediate level Haskell knowledge by guiding through the codebase of Pandoc
2. become acquainted with the codebase of Pandoc

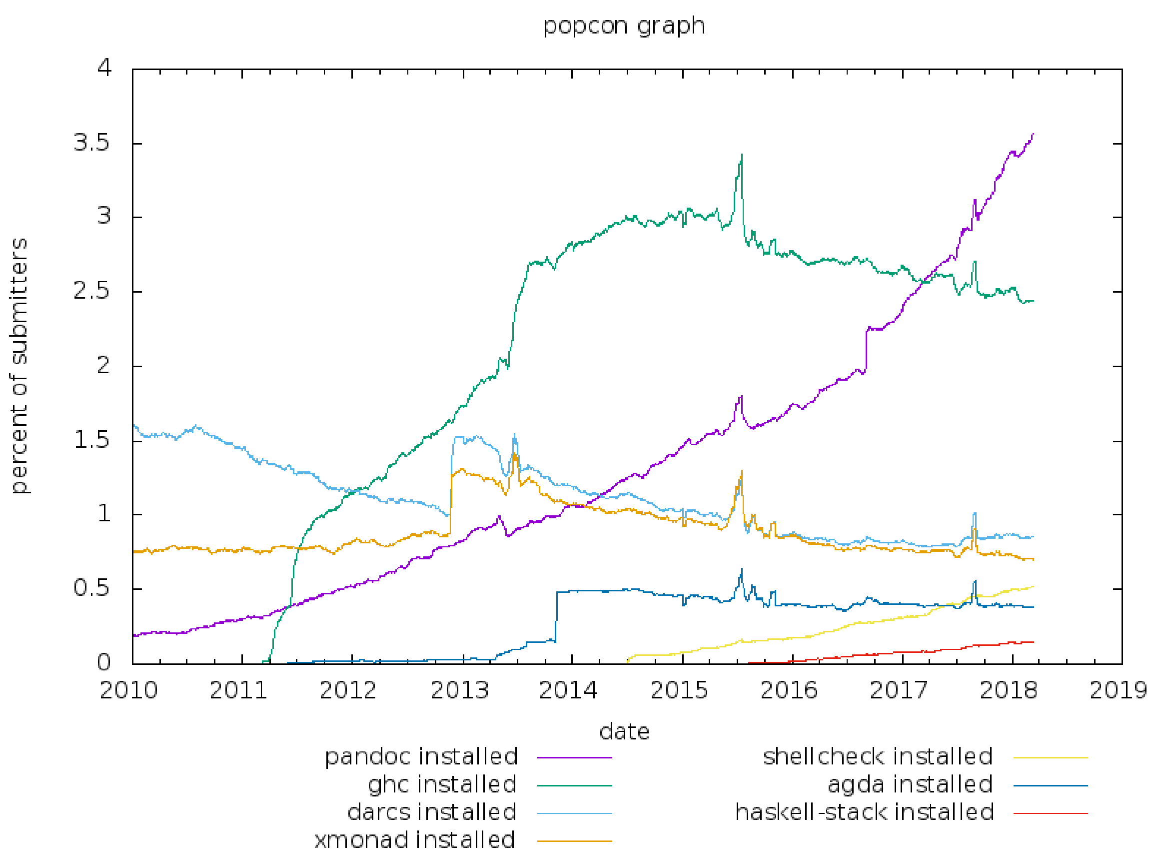

Figure 1.1: Debian Popularity Contest trends for the most popular Haskell packages

1.1.1 Target audience

Basic Haskell knowledge is required; see Basic Haskell language constructs.

The tutorial aims to be self-contained.

1.2 Introduction to Pandoc

Pandoc is a document converter which can convert between several markup formats like Markdown and also support document formats like HTML, PDF, docx and ODT.

According to GitHub and the Debian Popularity Contest, Pandoc is the most popular Haskell application.

1.2.1 Markdown basics

Markdown is designed to be easy to write and read.
A simple example Markdown document is the following:

```
Level-one header
================

First paragraph with a [link](/url) and *emphasis*.

Second paragraph with `verbatim text`.

    -- code block
    not True = False
    not False = True

Level-two header
----------------

- list item
- list item
  - sublist item
  - sublist item
```

Pandoc's Markdown supports a lot more constructs like subscripts, superscripts, math formulas, ordered lists, tables, pictures, footnotes and citations. The User's Guide has all the details.

The markdown source of this tutorial can be found here.

1.2.2 Command line interface of `pandoc`

By default `pandoc` works as a pipe (the command line examples are given in Linux):

```
$ echo 'Hello, *World*!' | pandoc
<p>Hello, <em>World</em>!</p>
```

The target language can be changed with the `--to` option (the default is `html`):

```
$ echo 'Hello, *World*!' | pandoc --to latex
Hello, \emph{World}!
```

The source language can be changed with the `--from` option (the default is `markdown`):

```
$ echo '<p>Hello, <em>World</em>!</p>' | pandoc --from html --to markdown
Hello, *World*!
```

Language extensions can be turned on and off individually (some of them are turned on by default):

```
$ echo ':smile: H~2~O' | pandoc --from markdown
<p>:smile: H<sub>2</sub>O</p>
```

```
$ echo ':smile: H~2~O' | pandoc --from markdown+emoji
<p>😄 H<sub>2</sub>O</p>
```

```
$ echo ':smile: H~2~O' | pandoc --from markdown+emoji-subscript
<p>😄 H~2~O</p>
```

Pandoc can be invoked with input and output files. The source and target language is guessed by the file extensions when it is not given explicitly.

```
$ pandoc inputfile.md -o outputfile.html
```

Multiple input files are concatenated by `pandoc`:

```
$ pandoc chap1.md chap2.md chap3.md
```

Instead of an input file, an absolute URI may be given.
For example, BBC news headlines can be read without styling by

```
$ pandoc http://www.bbc.com/news --from html-native_divs-native_spans \
--to markdown-header_attributes --reference-links | less
```

1.2.3 Standalone documents

By default, `pandoc` produces a fragment in the output format.
For example, the following command produces a fragment of a linux manual page:

```
$ echo 'Hello, *World*!' | pandoc --to=man
.PP
Hello, \f[I]World\f[]!
```

To produce a standalone document, one should add the `--standalone` or `-s` option:

```
$ echo 'Hello, *World*!' | pandoc --to=man -s
.\" Automatically generated by Pandoc 2.1.3
.\"
.TH "" "" "" "" ""
.hy
.PP
Hello, \f[I]World\f[]!
```

Standalone documents contain metadata like title, author and date which can be set with the `--metadata` or `-M` option:

```
$ echo 'Hello, *World*!' | \
pandoc --to=man -s -M title:Hello -M author:'Péter Diviánszky' -M date:07.04.2018
.\" Automatically generated by Pandoc 2.1.3
.\"
.TH "Hello" "" "07.04.2018" "" ""
.hy
.PP
Hello, \f[I]World\f[]!
.SH AUTHORS
Péter Diviánszky.
```

Metadata can be parsed from a separate file.
Suppose that `metadata.yaml` contains

```
---
title: Hello
author:
- Péter Diviánszky
date: 07.04.2018
---
```

then one can invoke `pandoc` like

```
$ pandoc metadata.yaml content.md
```

The content of the metadata file can be included directly in the Markdown source file.

Table of contents can be generated with `--toc`.
Sections can be numbered with `--number-sections`.

The language can be set like `--variable=lang:hu`.

Several output formats allow other useful options like `--css` and `--self-contained` for HTML output.

Chapter 2

Basic Haskell language constructs

This section is a revision of the required Haskell knowledge rather than a full-blown Haskell introduction. Skim through this section to note in which areas need you more preparation for the rest of the tutorial.

2.1 Lexical structure of Haskell modules

2.1.1 Comments

```
x = 1  -- comment begins with two dashes, lasts until the end of the line
y = 2
------------------------ comment with more dashes -------------------------
```

```
{-
multi-line
comments {- may be nested -}
-}
```

2.1.2 Layout

Layout of Haskell code matters:

```
f x
= x         -- *wrong*
```

```
f x
 = x        -- *right*
```

Only the *indentation* of code lines matters, other whitespace does not matter.

Example:

```
module Main where   -- 'where' starts a new block
f x = y  where      -- 'where' starts another block
    y = z
    z =
```

```
        x
w = f 4
```

This is the same as:

```
module Main where {
f x = y  where { y = z; z = x; };
w = f 4;
}
```

Keywords which start a new block: `where`, `of` and `do`.

2.1.3 Literals

Note: There are no negative number literals, see negation syntax.

```
109                     -- a decimal integral
0o155, 0O155            -- the same octal integrals
0x6d, 0x6D, 0X6d, 0X6D  -- the same hexadecimal integrals
```

```
0.314                   -- decimal fractional
3.14e-1, 3.14E-1        -- the same decimal fractionals in scientific notation
```

```
'a', 'á', 'A', '1'      -- some alphanumeric characters
'+', '!', '&'           -- some symbol characters
'\''                    -- the single quote character
'\\'                    -- the backslash character
'\n', '\LF'             -- the line feed character
'\225'                  -- the 'á' character with decimal unicode code
'\xE1'                  -- the 'á' character with hexadecimal unicode code
'\o341'                 -- the 'á' character with octal unicode code
```

```
""                      -- the empty string
"aá\225\\\'\""          -- a non-empty string
"\225\&9"               -- same as "á9"
"multi-line \
   \string"             -- same as "multi-line string"
```

2.1.4 Identifiers

2.1.4.1 Operator identifiers

operator: string of `!?.#$%@&*+-~^/|\<=>` characters or non-ASCII unicode symbols
exceptions: `--`, `::`, `->`, `<-`, `=>`, `=`, `~`, `\`, `|`

alphanumeric identifiers: string of alphanumeric characters or `_'`, beginning with a letter or `_`
exceptions: `_`, `module`, `where`, `import`, `qualified`, `as`, `hiding`, `data`, `type`, `newtype`, `deriving`, `class`, `instance`, `default`, `infix`, `infixl`, `infixr`, `let`, `in`, `case`, `of`, `if`, `then`, `else`, `do`

Examples:

```
f' x = x + 1 : x * 2 : []
-- operators: +, :, *
-- alphanumeric identifiers: f', x
```

The distinction between operators and alphanumeric identifiers matters only syntactically, see application syntax.

2.1.4.2 Variable vs. constructor identifiers

In Haskell, constructors are the building blocks of data types and types. There is a syntactical difference between constructors and variables to help the reader of the source code.

constructor: identifier beginning with an uppercase letter or `:` (colon).
variable: any other identifier

Lexically, *function names* and *function parameters* are variables.

Examples:

```
f' x = x + 1 : x * 2 : []
-- variables: f', x, +, *
-- constructors: :
```

The distinction between variables and constructors matters in pattern matching (in expressions) and quantification (in types).

2.1.4.3 Qualified names

Any identifier can be qualified by prefixing it with one or more qualifier:

```
Prelude.id
Data.Map.toList
Data.Monoid.<>      -- qualified operator
```

The meaning of qualifiers is discussed in Import declarations.

2.1.4.4 Special syntax for some frequently used constructors

The following constructors are not identifiers lexically but can be considered as identifiers with special syntax. See tuple and list syntax and compound types:

```
(->)   -- function type constructor
[]     -- empty list constructor (as an expression) / list type constructor (as a type)
()     -- unit constructor / type constructor
(,)    -- 2-tuple constructor / type constructor
(,,)   -- 3-tuple constructor / type constructor
...
```

2.2 Expressions

2.2.1 Expressions vs. types

In Haskell source code, every word can marked as an ‘expression’ or a ‘type’.

For example, expressions are colored red and types are colored blue here:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt i xs = (take (i :: Int) xs, drop i xs)
```

The general rule is that types comes after `::` until the end of the next language construct.

`::` can be pronounced as 'is a', so `i :: Int` can be pronounced as 'i is an int'.

There are separate namespaces for types and expressions, which means the red `i` and the blue `i` above are different variables.

2.2.2 Application syntax

In Mathematics, $f(x)(y)$ means that f is a function, and f applied on x is also a function and that function is applied on y.

In Haskell the same is written as `f x y` or `(f x) y`.
The two are exactly the same because function application is left-associative.

2.2.2.1 Prefix and infix notation

Prefix notation for alphanumeric identifiers:

```
id 2                        -- identity function applied to 2
div x y                     -- div applied to x and y
Either Int Bool             -- disjunct union type of Int and Bool
Prelude.Either Int Bool     -- how to apply a qualified name
```

Prefix notation for operators:

```
(:) 1 []                    -- (:) applied to 1 and []
(++) xs ys                  -- (++) applied to xs and ys
(Prelude.++) xs ys          -- how to apply a qualified operator
```

Expressions with infix notation:

```
x `div` y                   -- same as  div x y
Int `Either` Bool           -- same as  Either Int Bool
Int `Prelude.Either` Bool   -- same as  Prelude.Either Int Bool
1 : []                      -- same as  (:) 1 []
xs ++ ys                    -- same as  (++) xs ys
xs Prelude.++ ys            -- same as  (Prelude.++) xs ys
```

2.2.2.2 Precedences – order of applications

```
f x + y     -- same as ((f x) + y), prefix application is stronger than infix application
x + y * z   -- same as (x + (y * z)), because (*) is stronger than (+)
x + y + z   -- same as ((x + y) + z), because (+) is left-associative
```

Precedences are declared with fixity declarations.

2.2.2.3 Negation syntax

Negation is the only prefix operation:

```
-1          -- negated 1 (not a literal)
- 1         -- the same negated 1; whitespace does not matter
-x          -- negated x
```

Negation `-` is parsed as if it was `0-` :

```
-x+2        -- same as  0-x+2 = (0-x)+2 = (-x)+2
-x^2        -- same as  0-x^2 = 0-(x^2) = -(x^2)
```

Negation needs parenthesis in several cases:

```
x - (- 2)   -- cannot be written as  x - - 2
x - (-2)    -- the same, cannot be written as  x - -2
```

2.2.2.4 Partial application

The *arity* of a function is the number of its arguments. For example, the arity of `(+)` is 2.

A function/constructor is partially applied if it is applied to less arguments than its arity.

```
(+) 1       -- same as \x -> (+) 1 x, see lambda expression
map f       -- same as \xs -> map f xs
```

2.2.2.5 Overapplication

A function is overapplied if it is applied to more arguments than its arity.

```
head [sin, cos] 1  -- same as (head [sin, cos]) 1
```

Overapplying a constructor yields always a type error.

2.2.3 Lambda expression

```
\x -> x + 1          -- \pattern -> expression
```

```
\x y -> x + 2*y      -- same as  \x -> \y -> x + 2*y
```

2.2.4 Section

Left sections:

```
(+1)                 -- same as  \x -> x + 1
(+1) 15              -- evaluated to 16
(+ 2*a)              -- same as  \x -> x + 2*a
```

Right sections:

```
(2^)                 -- same as  \n -> 2^n
(f 2 ^)              -- same as  \n -> f 2 ^ n
```

2.2.5 Tuple and list syntax

Tuple and list construction has special syntax:

```
('c', 3)             -- same as  (,) 'c' 3
(1, "hello", True)   -- same as  (,,) 1 "hello" True
```

```
[]                   -- empty list
[1,2,3]              -- same as  1 : 2 : 3 : []
```

Dot-dot expressions are just syntactic sugars:

```
[0..]                -- same as  enumFrom 0
[0..10]              -- same as  enumFromTo 0 10
[0,2..]              -- same as  enumFromThen 0 2
[0,2..10]            -- same as  enumFromThenTo 0 2 10
```

```
['a'..'z']           -- another use case for enumFromTo
```

2.2.5.1 List comprehension syntax

```
[2^n | n <- [0..10]]          -- one generator
[2^n | n <- reverse xs]       -- the generator can refer to any list
[2^n | n <- [0..10], even n]  -- one generator, one Boolean guard
[2^n | even n]                -- one Boolean guard (the result is empty or singleton)
[(a, b) | a <- [0..10], b <- [a..10]]    -- two generators
[x | n <- [0..], let x = 2^n, x > 10^9]  -- generator, local declaration, Boolean guard
...
```

2.2.6 Type annotation

With type annotations one can specialize the type of an expression.

```
1 :: Int             -- expression :: type
f 1 :: Int           -- same as  (f 1) :: Int
g (1 :: Int)
```

2.2.7 If expression

```
if a < b then a else b       -- if expression then expression else expression
```

2.2.8 Case expression

```
case as of                --  case expression of
   [] -> True             --     pattern -> expression
   x: xs -> odd x         --     pattern -> expression
```

Case alternatives can also have guards and local definitions.

2.3 Patterns

The meaning of a pattern is given if we know what expressions are *matched* by the pattern and which variables are *bound* to which expressions by a successful match.

2.3.1 Variable pattern

```
v           -- matches everything, binds variable v to the matched expression
```

Each variable can be bound only once:

```
f x x = 3   -- *wrong*, x is bound twice
```

If the variable is not used, it can be prefixed with an underscore:

```
_v          -- matches everything, binds variable _v to the matched expression
            -- but no warning is given if _v is not used
```

Wildcard can be seen as a special case:

```
_           -- matches everything, binds nothing
```

```
f _ _ = 3   -- ok, no multiple binding
```

2.3.2 Literal pattern

Literals can be used in patters, with negation too:

```
f (-1) = 1     -- matches (-1)
f x    = x
```

2.3.3 Constructor pattern

`Bool`, tuple and list patterns has similar syntax as the corresponding expressions:

```
True                -- matches True, (not False), (False || True), ...
(True, False)       -- matches (not False, False), ...; same as ((,) True False)
[]                  -- matches an empty list
_ : _               -- matches any non-empty list; same as ((:) _ _)
True : _            -- matches any non-empty list with a True head
[a, True]           -- same as (a : True : [])
```

2.3.4 Application in patterns

Patterns have more restriction on application than expressions:

```
not True     -- *wrong pattern*, only constructor application is allowed in patterns
(:) True     -- *wrong pattern*, partial application is not allowed in patterns
(:) True []  -- ok, because the arity of (:) is 2
```

Overapplication of constructors are always wrong:

```
(:) a b c    -- *wrong pattern*, *wrong expression*
```

2.3.5 At-pattern

With an at-pattern one can match a pattern and bind a variable to the whole expression at the same time:

```
        -- variable @ pattern
v@(_:t)   -- matches non-empty lists, binds v to the whole list and t to the the tail
```

Example:

```
f v@(_:t) = t ++ v

f [1,2,3] == [2,3,1,2,3]
```

2.4 Types

2.4.1 Simple type

```
Char            -- the type of unicode characters
Bool            -- the type of Booleans
```

2.4.2 Compound type

```
Set Int         -- Set applied on Int; set of integers
```

Compound list and tuple types have special syntax:

```
(Int, Char)     -- same as  (,) Int Char
[Int]           -- same as  [] Int
```

2.4.2.1 Function type

```
Bool -> Bool              -- same as   (->) Bool Bool
Bool -> Bool -> Bool      -- same as   Bool -> (Bool -> Bool)
```

2.4.3 Type variables

```
a -> b -> a
(a, [a])        -- possible representation of non-empty lists of a-s
```

Undefined type variables are implicitly forall-quantified at the beginning of the type.

```
id :: a -> a                 -- id can be used for any type
```

2.4.4 Type class constraints

Type class constraints controls the possible substitutions of type variables

```
Num c => c -> c              -- c can be substituted by Int, Double, ...
Num Int => a                 -- same as  a, because there is  instance Num Int
Num Char => a                -- yields no  instance Num Char  found error
Eq [a] => b                  -- same as  Eq a => b, because  Eq a => Eq [a]
```

Multiple constraints:

```
(Num a, Show a) => a -> a    -- same as  (Show a, Num a) => a -> a
(Eq a, Ord a) => b           -- same as  Ord a => b, because  Eq a => Ord a
```

2.5 Declarations

2.5.1 Function definition

```
double x = 2 * x        -- one argument
```

```
add3 x y z = x + y + z  -- 3 arguments
```

2.5.1.1 Operator definition

```
a *+ b = a * b + b          -- same as   (*+) a b = a * b + b
```

```
(f . g) x = f (g x)         -- same as   (.) f g x = f (g x)
```

```
a `diff` b = abs (a - b)    -- same as   diff a b = abs (a - b)
```

2.5.1.2 Function alternatives

```
not True  = False    -- 1st alternative
not False = True     -- 2nd alternative
```

2.5.1.3 Guards

```
min x y
    | x <= y    =  x
    | otherwise =  y        -- otherwise = True
```

2.5.1.4 Where block

```
f . g = h
  where
    h x = f (g x)
```

2.5.1.5 Recursive function

```
nub [] = []
nub (x: xs) = x: nub [a | a <- xs, a /= x]
```

Mutual recursion:

```
evenLength [] = True
evenLength (_: xs) = oddLength xs

oddLength [] = False
oddLength (_: xs) = evenLength xs
```

2.5.2 Constant definition

```
c = 1 + 4
```

```
[one, two, three, four, five] = [1..5]    -- five constants defined
```

Constant definition used in a where block:

```
distribute []      = ([], [])
distribute (x: xs) = (x: odds, evens)
  where
    (evens, odds) = distribute xs          -- defines evens and odds
```

2.5.2.1 Recursive constant

```
cycle xs = ys
  where
    ys = xs ++ ys       -- recursive constant
```

Mutual recursion:

```
falseTrue = False: trueFalse
trueFalse = True:  falseTrue
```

2.5.2.2 Ad-hoc polymorphic constant

The final type of an ad-hoc polymorphic constant is determined by the context in which it is used.

```
one :: Num a => a
one = 1
```

2.5.3 Type signature

Each type signature should have a corresponding definition.

```
one :: Int          -- variable(s) :: type
```

```
two, three :: Int   -- same as 'two :: Int' and 'three :: Int'
```

Chapter 3

Basic Haskell declarations

3.1 Numbers

Entities in this section are defined in `Prelude` *and* `Data.Complex`*.*

Types

```
Int             --- integer modulo 2^64 (or 2^32)
Integer         --- integer
Rational        --- ratio of two integers
Float           --- single precision floating point number
Double          --- double precision floating point number
Complex Float   --- complex numbers built on Float
Complex Double  --- complex numbers built on Double
```

Type classes

```
Num        =  {Int, Integer, Rational, Float, Double, Complex Float, Complex Double, ...}
Real       =  {Int, Integer, Rational, Float, Double, ...}
Integral   =  {Int, Integer, ...}
Fractional =              {Rational, Float, Double, Complex Float, Complex Double, ...}
RealFrac   =              {Rational, Float, Double, ...}
Floating   =                        {Float, Double, Complex Float, Complex Double, ...}
RealFloat  =                        {Float, Double, ...}
```

Constants

```
pi :: Floating a => a    --- π = 3.14..
```

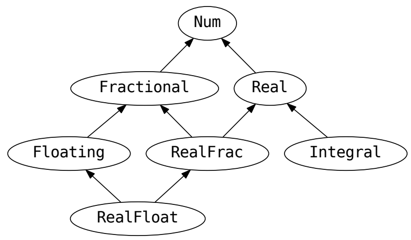

Figure 3.1: Type class hierarchy

Conversions

```
fromIntegral :: (Num b, Integral a) => a -> b       --- integer to any number type
realToFrac   :: (Fractional b, Real a) => a -> b    --- real to fractional
```

Rounding

```
truncate :: (Integral b, RealFrac a) => a -> b    --- round towards zero
round    :: (Integral b, RealFrac a) => a -> b    --- round towards nearest
ceiling  :: (Integral b, RealFrac a) => a -> b    --- round up
floor    :: (Integral b, RealFrac a) => a -> b    --- round down
```

Operators

```
(+)    :: Num a => a -> a -> a    --- addition
(*)    :: Num a => a -> a -> a    --- multiplication
(-)    :: Num a => a -> a -> a    --- subtraction
negate :: Num a => a -> a      --- negation, can be used with prefix operator '-'
(/)    :: Fractional a =>  a -> a -> a    --- division
(^)    :: (Num a, Integral b) =>        a -> b -> a    --- non-negative exponent
(^^)   :: (Integral b, Fractional a) => a -> b -> a    --- integer exponent
(**)   :: Floating a =>                 a -> a -> a    --- floating exponent
```

Functions

```
abs   ::      Num a => a -> a       --- absolute value
sqrt  :: Floating a => a -> a       --- square root
log   :: Floating a => a -> a       --- logarithm to the base of e
exp   :: Floating a => a -> a       --- base e exponential function
sin,   cos,   tan   :: Floating a => a -> a  --- sine, cosine, tangent (argument in radians)
asin,  acos,  atan  :: Floating a => a -> a  --- arcsine, arccosine, arctangent
sinh,  cosh,  tanh  :: Floating a => a -> a  --- hyperbolic sine, cosine, tangent
asinh, acosh, atanh :: Floating a => a -> a  --- hyperbolic arcsine, arccosine, arctangent
quot  :: Integral a => a -> a -> a  --- quotient (multiplicative)
```

```
div   :: Integral a => a -> a -> a  --- quotient (additive)
rem   :: Integral a => a -> a -> a  --- remainder (multiplicative)
mod   :: Integral a => a -> a -> a  --- remainder (additive)
gcd   :: Integral a => a -> a -> a  --- greatest common divisor
```

3.2 Booleans and comparison

Entities in this section are defined in `Prelude`.

Types

```
Bool         --- Boolean value
Ordering     --- result of comparison
```

Constructors

```
False     :: Bool       --- false
True      :: Bool       --- true

GT        :: Ordering   --- greater
LT        :: Ordering   --- less
EQ        :: Ordering   --- equal
```

Type classes

There are `Eq` and `Ord` instances for almost every type but functions.

```
Eq   = {Int, Double, Char, (Int, Char), [Int], ([Int], Char), [[Int]], ...}
Ord  = {Int, Double, Char, (Int, Char), [Int], ([Int], Char), [[Int]], ...}
```

Constants

```
otherwise :: Bool       --- same as True
```

Logical connectives

```
(&&) :: Bool -> Bool -> Bool    --- logical and
(||) :: Bool -> Bool -> Bool    --- logical or
not  :: Bool -> Bool            --- logical negation
```

Operators

```
(==) :: Eq a  => a -> a -> Bool    --- equal
(/=) :: Eq a  => a -> a -> Bool    --- not equal
(<)  :: Ord a => a -> a -> Bool    --- less
(>)  :: Ord a => a -> a -> Bool    --- greater
(<=) :: Ord a => a -> a -> Bool    --- less or equal
(>=) :: Ord a => a -> a -> Bool    --- greater or equal
```

Functions

```
compare :: Ord a => a -> a -> Ordering    --- compare two elements
even :: Integral a => a -> Bool     --- True if even
odd  :: Integral a => a -> Bool     --- True if odd
min  :: Ord a => a -> a -> a        --- minimum of two elements
max  :: Ord a => a -> a -> a        --- maximum of two elements
```

3.3 Tuples

Entities in this section are defined in `Prelude`*.*

Types (`a`, `b`, `c` and `d` are arbitrary types)

```
(a, b)            --- ordered pair (2-tuple)
(a, b, c)         --- ordered triple (3-tuple)
(a, b, c, d)      --- 4-tuple
...
```

Constructors

```
(,)   :: a -> b -> (a, b)                     --- ordered pair (2-tuple) constructor
(,,)  :: a -> b -> c -> (a, b, c)             --- ordered triple (3-tuple) constructor
(,,,) :: a -> b -> c -> d -> (a, b, c, d)   --- 4-tuple constructor
...
```

Functions

```
fst :: (a, b) -> a    --- first element
snd :: (a, b) -> b    --- second element
```

3.4 Lists

Entities in this section are defined in `Prelude` *or* `Data.List`*.*

Notes

- In case of `Foldable t`, replace `t` with the list type constructor. For example,

```
concat :: [[a]] -> [a]
```

Type

```
[a]     --- a-list (a is an arbitrary type)
```

Constructors

```
[]      :: [a]                  --- empty list
(:)     :: a -> [a] -> [a]      --- non-empty list constructor, 1:(2:[]) == [1,2]
```

Generic functions

```
(++)    :: [a] -> [a] -> [a]    --- concatenate two lists
concat  :: Foldable t => t [a] -> [a]    --- concatenate many lists
reverse :: [a] -> [a]           --- reverse order of elements
head    :: [a] -> a             --- first element of a list
last    :: [a] -> a             --- last element of a list
init    :: [a] -> [a]           --- all element but the last
tail    :: [a] -> [a]           --- all element but the first
inits   :: [a] -> [[a]]         --- iterated init
tails   :: [a] -> [[a]]         --- iterated tail
repeat  :: a -> [a]             --- repeat element infinitely
```

3.4.0.1 Functions with `Int`

```
(!!)      :: [a] -> Int -> a          --- element indexed by 0, 1, 2, ...
length    :: [a] -> Int               --- length of a list
take      :: Int -> [a] -> [a]        --- take first n element of a list
drop      :: Int -> [a] -> [a]        --- drop first n element of a list
splitAt   :: Int -> [a] -> ([a], [a]) --- take and drop together
replicate :: Int -> a -> [a]          --- repeat an element n times
```

3.4.0.2 Functions with `Bool`

```
null :: Foldable t => t a -> Bool       --- True if there is no element
and  :: Foldable t => t Bool -> Bool    --- logical and for more values
or   :: Foldable t => t Bool -> Bool    --- logical or for more values
```

Functions with numbers

```
sum     :: (Num a, Foldable t) => t a -> a    --- sum of elements
product :: (Num a, Foldable t) => t a -> a    --- product of elements
```

Functions with tuples

```
zip   :: [a] -> [b] -> [(a, b)]    --- pairing of list elements (zipping)
unzip :: [(a, b)] -> ([a], [b])    --- unzipping of list of pairs
```

3.4.0.3 Functions with `Eq`

```
elem       :: (Eq a, Foldable t) => a -> t a -> Bool    --- is the element in the list?
delete     :: Eq a => a -> [a] -> [a]     --- delete first occurrence of element
nub        :: Eq a => [a] -> [a]          --- delete repeating elements
group      :: Eq a => [a] -> [[a]]        --- group equal attached elements
isPrefixOf :: Eq a => [a] -> [a] -> Bool  --- True if second list starts with first list
```

3.4.0.4 Functions with `Ord`

```
minimum :: (Ord a, Foldable t) => t a -> a    --- minimum element
maximum :: (Ord a, Foldable t) => t a -> a    --- maximum element
insert  :: Ord a => a -> [a] -> [a]           --- insert element into sorted list
sort    :: Ord a => [a] -> [a]                --- sort list (increasing order)
```

3.5 Characters

Entities in this section are defined in `Prelude` *or* `Data.Char`.

Type

```
Char    --- unicode characters
```

Functions

```
ord        :: Char -> Int    --- unicode code
chr        :: Int -> Char    --- character of given unicode code
isSpace    :: Char -> Bool   --- True for ' ', '\t', '\n', ...
isDigit    :: Char -> Bool   --- True for '1', '2', ...
isAlpha    :: Char -> Bool   --- True for 'a', 'A', ...
isUpper    :: Char -> Bool   --- True for 'A', 'B', ...
isLower    :: Char -> Bool   --- True for 'a', 'b', ...
toUpper    :: Char -> Char   --- toUpper 'a' == 'A'
```

```
toLower    :: Char -> Char   --- toLower 'A' == 'a'
digitToInt :: Char -> Int    --- digitToInt '3' == 3
intToDigit :: Int -> Char    --- intToDigit 3 == '3'
```

3.6 Strings

Entities in this section are defined in `Prelude`*.*

Type

```
String         --- type String = [Char]
```

Type class

There are `Show` and `Read` instances for almost every type but functions.

```
Show  = {Int, Double, Char, (Int, Char), [Int], ([Int], Char), [[Int]], ...}
Read  = {Int, Double, Char, (Int, Char), [Int], ([Int], Char), [[Int]], ...}
```

Functions

```
show    :: Show a => a -> String    --- convert to string
read    :: Read a => String -> a
lines   :: String -> [String]    --- split string by newlines
unlines :: [String] -> String    --- concatenate strings with newlines
words   :: String -> [String]    --- split string by spaces
unwords :: [String] -> String    --- concatenate strings with whitespaces
```

3.7 Enumerations

Entities in this section are defined in `Prelude`*.*

Type class

```
Enum = {Int, Integer, Rational, Float, Double, Char, Bool, ...}
```

Dot-dot expressions: lists made by arithmetic sequences

```
enumFrom       :: Enum a => a -> [a]              --- difference = 1
enumFromTo     :: Enum a => a -> a -> [a]         --- difference = 1, with upper bound
enumFromThen   :: Enum a => a -> a -> [a]         ---
enumFromThenTo :: Enum a => a -> a -> a -> [a]    --- with upper bound
```

Syntax

```
[1..]        = enumFrom 1
[1..100]     = enumFromTo 1 100
[1,3..]      = enumFromThen 1 3
[1,3..100] = enumFromThenTo 1 3 100
```

Conversions

```
fromEnum :: Enum a => a -> Int     --- index
toEnum   :: Enum a => Int -> a     --- inverse of fromEnum
```

3.8 Higher-order functions

Entities in this section are defined in `Prelude` *or* `Data.Function`*.*

```
id      :: a -> a                          --- id "anything" == "anything"
const   :: a -> b -> a                     --- const 3 "True" == 3
($)     :: (a -> b) -> a -> b              --- (even $ 1+2) == False
(.)     :: (b -> c) -> (a -> b) -> a -> c  --- (even . (+1)) 2 == False
uncurry :: (a -> b -> c) -> (a, b) -> c    --- uncurry mod (5,2) == 1
curry   :: ((a, b) -> c) -> a -> b -> c    --- inverse of uncurry
flip    :: (a -> b -> c) -> b -> a -> c    --- flip (**) 3 4 == 64.0
on      :: (b -> b -> c) -> (a -> b) -> a -> a -> c   --- ((==) `on` even) 2 10 == True
until   :: (a -> Bool) -> (a -> a) -> a -> a          --- until (> 10) (+ 1) 3 == 11
```

Functions with lists

```
map       :: (a -> b) -> [a] -> [b]        --- map (^2) [1..4] == [1,4,9,16]
iterate   :: (a -> a) -> a -> [a]          --- iterate (+2) 2 == [2,4..]
filter    :: (a -> Bool) -> [a] -> [a]     --- filter (<3) [1..100] == [1,2]
partition :: (a -> Bool) -> [a] -> ([a], [a])    --- partition odd [1..5] == ([1,3,5],[2,4])
takeWhile :: (a -> Bool) -> [a] -> [a]     --- takeWhile (<3) (cycle [1..5]) == [1,2]
dropWhile :: (a -> Bool) -> [a] -> [a]     --- dropWhile (<3) [1..5] == [3,4,5]
span      :: (a -> Bool) -> [a] -> ([a], [a])    --- span (<3) [1..5] == ([1,2],[3,4,5])
any :: Foldable t => (a -> Bool) -> t a -> Bool  --- any (<3) [1,2,4] == True
all :: Foldable t => (a -> Bool) -> t a -> Bool  --- all (<3) [1,2,4] == False
```

Generalized functions

```
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]      --- zipWith (+) [2,3] [2,2] == [4,5]
groupBy   :: (a -> a -> Bool) -> [a] -> [[a]]        --- groupBy ((&&) `on` odd) [1,3,4,5]
sortBy    :: (a -> a -> Ordering) -> [a] -> [a]      --- sort by ordering
maximumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a    --- maximum by ordering
minimumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a    --- minimum by ordering
```

Folds

```
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a        --- foldl1 (**) [2,3,4] == 4096.0
scanl1 ::               (a -> a -> a) -> [a] -> [a]      --- scanl1 (^) [2,3,4] == [2,8,4096]
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a        --- fold from right
scanr1 ::               (a -> a -> a) -> [a] -> [a]      ---   with intermediate values
foldl  :: Foldable t => (b -> a -> b) -> b -> t a -> b   --- foldl (+) 5 [2,3,4] == 14
foldl' :: Foldable t => (b -> a -> b) -> b -> t a -> b   --- strict foldl
scanl  ::               (b -> a -> b) -> b -> [a] -> [b] --- scanl1 with initial value
foldr  :: Foldable t => (a -> b -> b) -> b -> t a -> b   --- foldr1 with initial value
scanr  ::               (a -> b -> b) -> b -> [a] -> [b] --- scanr1 with initial value
```

3.9 Error handling

Entities in this section are defined in `Prelude`*.*

```
undefined :: a          --- abort evaluation
error :: String -> a    --- (error "impossible") aborts evaluation with message "impossible"
```

3.10 Fixity declarations in `Prelude`

```
infixr 9  !!, .
infixr 8  ^, ^^, **
infixl 7  *, /, `rem`, `mod`, `div`, `quot`
infixl 6  -, +
infixr 5  :, ++
infix  4  ==, /=, <, >, <=, >=
infixr 3  &&
infixr 2  ||
infixr 0  $
```

3.11 Type class hierarchy in `Prelude`

Type classes are discussed in `class` definitions.

```
class                                   Show a
class                                   Read a
class                                   Enum a
class                                   Eq a
class                         Eq a =>   Ord a
class                                   Num a
class                        Num a =>   Fractional a
class                 Fractional a =>   Floating a
class               (Num a, Ord a) =>   Real a
class             (Real a, Enum a) =>   Integral a
```

```haskell
class   (Real a, Fractional a) =>  RealFrac a
class (RealFrac a, Floating a) =>  RealFloat a
```

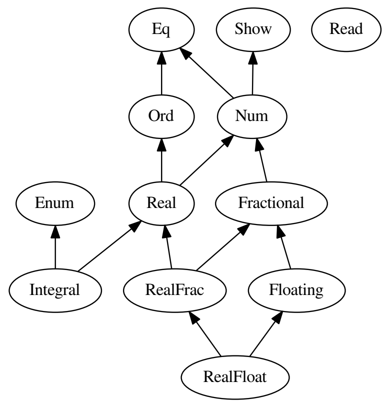

Figure 3.2: Type class hierarchy

Explanation: `Eq` contains `Ord`, i.e. if there is an `Ord` instance for type T, then there is an `Eq` instance for T.

Chapter 4

Advanced Haskell language constructs

4.1 Declarations (2)

4.1.1 Fixity declaration

```
infix  4  ==, /=    -- (==) and (/=) are non-associative
infixl 6  +         -- (+) is left-associative; (+) is stronger than (==)
infixr 0  $         -- ($) is right-associative; 0 is the lowest precedence level
infixr 9  .         --                           9 is the highest precedence level
```

For more examples see fixity declarations in Prelude

4.1.2 `type` – type synonym definition

Definition of type synonyms:

```
type String = [Char]        -- 'type'
```

Note that `String` has a constructor name (begins with an uppercase letter), but `String` is *not* a type constructor; it is a defined constant at the type level similarly to `otherwise = True` at the value level.

4.1.2.1 Type synonyms with parameters

Type synonyms may have 0 or more parameters:

```
type Two a = (a, a)     -- 'Two Int' is equal to '(Int, Int)'
```

The *arity* of the type synonym is the number of its parameters.

Partial application of type synonyms is not allowed:

```
type Tw = Two         -- *wrong*, use 'type Tw a = Two a'
```

4.1.3 `data` – algebraic data type definition

`data` defines a new *algebraic data type (ADT)*, which consists of a type constructor and zero or more (expression) constructors.

For example,

```
data Bool = False | True        -- 2 constructors
                                -- (False and True are in expression namespace)
```

defines

```
Bool                -- type constructor
False :: Bool       -- constructor, so False is a pattern too
True  :: Bool       -- constructor, so True is a pattern too
```

The type constructor and the constructor may have the same name because they are in different namespaces:

```
data Example{-type-} = Example{-expression-} | OtherExampleConstructor
```

4.1.3.1 Constructor fields

Constructors may have zero or more *fields* holding any type of data. For example,

```
data Complex = Pair Double Double       -- Pair has two Double fields
```

defines

```
Complex
Pair :: Double -> Double -> Complex     -- Pair can be used in patterns
```

```
re :: Complex -> Double
re (Pair r _) = r                 -- Pair used in a pattern
```

4.1.3.2 ADTs with parameters

Algebraic data types may have zero or more *parameters*, for example:

```
data Complex a = Pair a a
```

4.1.3.3 Recursive ADTs

Algebraic data types may be recursive, for example:

```
data Nat = Zero | Suc Nat
```

```
data BinaryTree a b
    = Leaf a
    | Node
        (BinaryTree a b)    -- left subtree
        b                   -- value at node
        (BinaryTree a b)    -- right subtree
```

Mutual recursion is also allowed.

4.1.3.4 Records

Fields of constructors can be named.
For example, instead of

```
data Complex = Pair Double Double
```

one can write

```
data Complex = Pair {re :: Double, im :: Double}
```

which has the advantage that it defines also the *accessor functions*

```
re :: Complex -> Double         -- field accessor function
im :: Complex -> Dobble         -- field accessor function
```

Moreover, the following expressions are valid

```
Pair {re = 3, im = 4}           -- record construction, same as 'Pair 3 4'
Pair {im = 4, re = 3}           -- record construction, same as 'Pair 3 4'
p {re = 3}                      -- record update, same as 'Pair 3 (im p)'
p {im = 4}                      -- record update, same as 'Pair (re p) 4'
```

Record syntax is especially useful when the constructor has many fields.

Record syntax is also possible if the ADT has more than one constructor.

Contracted syntax:

```
data Complex = Pair {re, im :: Double}
```

4.1.4 `newtype` definitions

`newtype` is similar to `data` with the following constraints:

- Exactly one constructor is defined
- Exactly one field of the constructor is defined

Advantages of `newtype` over `data`:

- better runtime performance
- the compiler can derive more type class instances

4.1.5 `class` definitions

Example:

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

defines

```
Eq                                 -- type class constructor
(==) :: Eq a => a -> a -> Bool     -- type class method
(/=) :: Eq a => a -> a -> Bool     -- type class method
```

The implementations of `(==)` and `(/=)` are given in the instance definitions.

4.1.5.1 Default methods

Default implementation of methods:

```
class Eq a
  where
    (==) :: a -> a -> Bool
    x == y = not (x /= y)           -- used if the instance does not implement (==)

    (/=) :: a -> a -> Bool
    x /= y = not (x == y)           -- used if the instance does not implement (/=)
```

4.1.5.2 Superclasses

```
class Eq a => Ord a             -- Eq is a superclass of Ord
  where
    (<) :: a -> a -> Bool
```

- an `Ord` instance is definable on a type only if an `Eq` instance is defined already
- the constraint `Ord a` implies `Eq a`, so one can simplify the constraint `(Eq a, Ord a)` to `Ord a`

4.1.6 `instance` definitions

Examples:

```
instance Eq Bool
  where
    True  == True  = True
    False == False = True
    _     == _     = False

    -- -- (/=) is not implemented, so the compiler defines it as:
    -- x /= y = not (x == y)
```

```
instance Eq a => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys       -- the second (==) is recursion
    _      == _      = False
```

```
instance (Eq a, Eq b) => Eq (a, b) where
    (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

These instance definitions are used recursively to solve the following constraints:

```
Eq Bool
Eq [Bool]
Eq (Bool,Bool)
Eq [[Bool]]
Eq [(Bool,Bool)]
Eq ([Bool],[Bool])
```

```
Eq ((Bool,Bool),[Bool])
-- ...
```

Instance definitions are not allowed on type synonyms in Haskell 98.

4.2 What is a Haskell program?

This subsection contains the essential vocabulary which is needed to speak about Haskell programs.

4.2.1 Organization of Haskell source code

Haskell module: Group of Haskell definitions which are stored in a text file like `Example.hs`
Haskell library: Haskell modules in hierarchical file structure
Haskell program (or executable): Haskell modules with a main module
Haskell package: Haskell library and/or a set of Haskell executables

4.2.2 Phases of execution of Haskell programs

Knowing phases of execution helps to understand error messages.

1. *lexical analysis*
   - recognize the beginning and end of "words" and punctuation (these are called *tokens*)
   - recognize layout (indentation matters in Haskell)
   - skip whitespace and comments
2. *parsing*
   - recognize language constructs
3. loading imports
   - (recursively) do these phases until code generation for all imported modules
4. *scope checking*
   - determine the defining location of each identifier
5. reordering
   - recognize hidden parentheses
   - connect function declaration with function definition (can be apart)
6. *type inference*
   - check validity of expressions and declarations
7. *optimization*
   - transform definitions to make execution more time/space efficient
8. *code generation*
   - transform definitions to machine code (maybe for an abstract machine)
9. *linking*
   - compose code with code generated for imported modules
   - compose code with RTS (*Runtime System*: code for builtin definitions, garbage collection, scheduling and profiling)
10. *execution* (called *runtime* when used as an adjective)
    - execute the linked code

Phases 1-9 are called *compilation* (or *compilation time* when used as an adjective).

Compilation is done by the compiler, execution is done by the operating system.

4.2.3 Possible programmer errors

Haskell programmers may cause the following kind of errors:

Compile time / static error: error recognized during phases 1-6 by the compiler
(all compile time errors are caught until the end of type checking)
Runtime error: error during execution, recognized by the runtime system
(all errors are caught by the runtime system and not by the operating system)
Semantic error: error during execution which is not recognized by the runtime system
(may be recognized by testing)
Performance issue: runtime resource usage is not acceptable / not reasonable
(may be recognized by profiling and benchmarks)

4.2.4 Cached intermediate results of compilation

Caching the results of the compilation phases helps to avoid unnecessary recompilation of modules (if only one of the modules changes, for example).

Executable file: cached result of linking
Object file: cached result of code generation, needed for linking
Interface file: cached result of type checking, needed in phases 3-7 for modules importing this one

4.3 Modules

General module structure:

```
{-# LANGUAGE CPP #-}                         -- 0 or more language extension pragmas

module X where                               -- module header (may be missing)

import Prelude                               -- 0 or more import declarations

x = 1                                        -- 0 or more other declarations
```

4.3.1 Module header

Simple module header:

```
module A.B.C where       -- this module should be located at A/B/C.hs in the filesystem
                         --  otherwise it cannot be imported by other modules
```

Modules called `Main` has an extra requirement:

```
module Main where        -- 'main' should be defined in the module
```

No module header is also possible:

```
-- no header --          -- same as  'module Main where',  but 'main' is not required
```

4.3.1.1 Export list

Export list is placed after the module name:

```
module A.B.C          -- What is exported?
    ( A               -- - A from type namespace
                      --   (type synonym / type constructor / type class with instances)
    , B (..)          -- - B from type namespace with all parts
                      --   (constructors, field accessors / methods)
    , C (D, (:+:), e) -- - C from type namespace with listed parts
                      --   (constructors D and (:+:), field accessor / method e)
    , f               -- - f from expression namespace (function / constant)
    , (++)            -- - operator (++)
    , module X        -- - everything which is exported by X
    , module A.B.C    -- - everything which is defined in this module
    ) where
```

Empty export list or no export list is possible:

```
module A.B.C () where   -- nothing is exported
```

```
module A.B.C where      -- same as  'module A.B.C (module A.B.C) where'
                        -- (exports everything which is defined in this module)
```

4.3.2 Import declaration

Examples of import declarations:

```
import A.B.C          -- import everything which A.B.C exports
```

```
import X              -- import from module X:
    ( A               -- - A from type namespace
    , B (..)          -- - B from type namespace with all parts
    , C (D, (:+:), e) -- - C from type namespace with listed parts
    , f               -- - f from expression namespace
    , (++)            -- - operator (++)
    )
```

```
import Y hiding       -- import everything which Y exports, but do not import
    (A, B (..), C (D, (:+:), e), f, (++))                 -- these items
```

`Prelude` is implicitly imported:

```
import Prelude        -- this line is added if 'Prelude' is not imported explicitly
```

4.3.2.1 Qualify imported identifiers

Any imported identifiers can be used qualified in expressions:

```
three = Prelude.id 3  -- 'id' imported from 'Prelude', applied on 3
```

One can change the needed qualification at the import declaration:

```
import A.B.C as ABC   -- 'ABC.id' means now 'A.B.C.id', and 'A.B.C.id' is not in scope
```

Other examples:

```
import A.B.C as ABC (x, y)             -- combination with list of imported items
```

```
import A.B.C as ABC hiding (x, y)      -- combination with hiding
```

These tricks are allowed:

```
import A as C
import B as C    -- same qualifications, especially useful when exporting 'module C'
```

```
import A
import B as A              -- use the same qualification
```

```
import A as B
import B as A              -- swap qualifications (not nice)
```

4.3.2.2 Require qualification of imported identifiers

With keyword `import qualified`, one can prohibit unqualified use of the imported identifiers:

```
import qualified Prelude    -- now 'id' is not in scope, only 'Prelude.id'
```

`import qualified` can be used as `import`.
Examples:

```
import qualified Prelude (id)
```

```
import qualified Prelude hiding (id)
```

```
import qualified Prelude as P hiding (id)
```

4.4 Kinds

4.4.1 What is a kind?

Kinds are the types of type expressions.

If the type `t` has kind `k` then we write `t :: k`.
This syntax is the same as the `x :: t` for expressions and types, see Expressions vs. types.

As an example, `Int :: *`.
`*` is the type of types. `*` can be pronounced as 'type', so `Int :: *` can be pronounced as 'int is a type'.

In `ghci`, one can ask for the kind of a type expression:

```
Prelude> :k Int
Int :: *
```

The fact `Int :: *` cannot be checked in Haskell98, but can be checked in an extension of it:

```
{-# language KindSignatures #-}

x = 1 :: (Int :: *)
```

There are more interesting kinds than `*`. An example is `* -> *` which is the kind of type functions from type to type (for example, `Maybe :: * -> *`).

4.4.2 Kinds of type constructors

```
Int   :: *                -- Int is a type
Char  :: *                -- Char is a type
Bool  :: *                -- Bool is a type
Maybe :: * -> *           -- Maybe is a function from type to type
[]    :: * -> *           -- list is a function from type to type
```

```
(,)   :: * -> * -> *             -- same as  * -> (* -> *)
(,,)  :: * -> * -> * -> *        -- same as  * -> (* -> (* -> *))
(,,,) :: * -> * -> * -> * -> *
...
```

4.4.3 Type application

Type application has the same syntax as function application.

Partial application is possible:

```
(,,)                   :: * -> * -> * -> *    -- no application
(,,) Int               :: * -> * -> *         -- partial application
(,,) Int Char          :: * -> *              -- partial application
(,,) Int Char Bool     :: *                   -- full application; same as (Int, Char, Bool)
```

(Partial application of type synonyms is not allowed, see type synonyms.)

Kind mismatch is rejected by the compiler:

```
(,) Maybe             -- *wrong*, kind mismatch at application
(,) Int Char Bool     -- *wrong*, overapplication of type constructor (,)
Int Char              -- *wrong*, overapplication of type constructor Int
```

4.4.4 Kind of constraints

`Constraint` is the kind of type class constraints.
For example:

```
Num Int               :: Constraint
Num                   :: * -> Constraint
Num a                 :: Constraint           -- if a :: *
(Num Int, Eq Int)     :: Constraint           -- constraints can be tupled together
```

```
Num a => a            :: *      -- as if  (=>) :: Constraint -> * -> *
```

4.4.5 Type expression vs. type vs. type constructor

Type expression or type-level expression is an expression at the type level, like `Maybe Int`.

Type is a type expression with kind `*`.

Sometimes 'type' is said instead of 'type expression' which may cause confusion.

Every expression should have a *type*. For example, this hole cannot be filled: `_ :: Maybe`.

Type constructor is a type expression defined by `data` and `newtype`.
There are some built-in type constructors too: `(->)`, `[]`, `()`, `(,)`, `(,,)`, …

Examples:

```
              -- type expression?    type?    type constructor?
Int           --  yes                 yes      yes
Maybe Int     --  yes                 yes      no
String        --  yes                 yes      no
Maybe         --  yes                 no       yes
(,,) Int      --  yes                 no       no
```

4.4.6 Other kinds

Kinds which are beyond the scope of this tutorial:

- kind variables
- types lifted to the kind level, for example the kind of type-level natural numbers
- The kind of a type may be other than `*`. The kind of a type `T` encodes the calling convention of expressions of type `T`; `*` is the kind of *lifted* types (whose values are accessed by a pointer).

  Remember this when you ask the kind of the function type constructor:

  ```
  > :k (->)
  (->) :: TYPE q -> TYPE r -> *
  ```

4.5 Haskell language extensions

Very brief history of the Haskell language descriptions:

- 1990: The Haskell version 1.0 Report was published
- 1999: The Haskell 98 Report: Language and Libraries was published
- 2002: The Revised Haskell 98 Report: Language and Libraries was published
- Haskell 2010 Language Report

The Haskell language used nowadays is defined either as Haskell98 or as Haskell2010 plus a set of language extensions.

Haskell modules can specify the actually used language extensions in the very beginning of the module (before the module header).

Syntax:

```
{-# LANGUAGE CPP #-}              -- 'language' (with lowercase letters) is also good
{-# LANGUAGE PatternGuards #-}
```

or packed together:

```
{-# LANGUAGE CPP, PatternGuards #-}
```

The language extensions used by Pandoc are discussed at Haskell language extensions.

The following language extensions are used by Pandoc:

- preprocessing: `CPP`
- imports: `NoImplicitPrelude`
- syntactic sugars
  - expression related: `TupleSections`, `MultiWayIf`, `LambdaCase`
  - pattern related: `PatternGuards`, `ViewPatterns`
  - both for expressions and patterns: `OverloadedStrings`
- type class extensions
  - classes: `MultiParamTypeClasses`
  - instances: `FlexibleInstances`, `TypeSynonymInstances`, `IncoherentInstances`, `UndecidableInstances`
    - deriving: `GeneralizedNewtypeDeriving`, `DeriveDataTypeable`, `DeriveGeneric`
  - contexts: `FlexibleContexts`
- types: `ExplicitForAll`, `ScopedTypeVariables`, `RelaxedPolyRec`
- type declaration: `GADTs`
- other: `TemplateHaskell`

4.5.1 CPP

CPP stands for C PreProcessor.

CPP is used for conditional compilation. Typical use cases for conditional compilation:

- backward compatibility for library dependencies
- allow the code to be platform-dependent
- turn features on/off in a library or executable

Syntax: one CPP directive per line, *without indentation*.

CPP directives used in Pandoc:

- `#if` *condition*
- `#ifdef` *macro*
- `#ifndef` *macro*
- `#else`
- `#endif`

`#if`, `#ifdef` and `#ifndef` need a matching `#endif`.

`#else` is optional but should be placed between an `#if…` and an `#endif` pair.

4.5.1.1 Example: `#if` used for backward compatibility

```
#if MIN_VERSION_base(4,8,3)
import System.IO.Error (IOError, isDoesNotExistError)
#else
import System.IO.Error (isDoesNotExistError)
#endif
```

4.5.1.2 Example: `#ifdef` and `#ifndef` used for platform-dependent code

```
#ifndef _WINDOWS
import System.Posix.IO (stdOutput)
import System.Posix.Terminal (queryTerminal)
#endif
```

Later in the same module, in a `do` block:

```
#ifdef _WINDOWS
  let istty = True
#else
  istty <- queryTerminal stdOutput
#endif
```

4.5.1.3 Example: turn a feature on/off

cabal package flags define CPP macros (see later):

```
#ifdef EMBED_DATA_FILES
```

4.5.2 `ViewPatterns`

View patterns allows to apply a function *before* pattern matching.

Example:

```
initLast xs = (init xs, last xs)

reverse (initLast -> (xs, x)) = x: reverse xs
```

Example:

```
greet (words -> ["My","name","is",x]) = "Hello " ++ x
greet (words -> ["I","am",x]) = "Hi " ++ x
greet xs = "I don't understand " ++ show xs
```

is the same as

```
greet xs = case words xs of
    ["My","name","is",x] -> "Hello " ++ x
    ["I","am",x] -> "Hi " ++ x
    _ -> "I don't understand " ++ show xs
```

Chapter 5

Data structures

After going through Haskell language constructs let's see some concrete data structures with their common operations.

5.1 Text representations

We start with text representations.

5.1.1 List of characters

Beginners treat texts as lists of characters.
The type `String` is defined as a type synonym for lists of characters:

```
type String = [Char]          -- defined in Prelude
```

Pros:

- generic list operations can be used (`take`, `drop`, `replicate`, …)
- the head and the tail of the text can be matched with the `(:)` pattern
- automatically imported (defined in `Prelude`)

Cons:

- memory inefficient: each character in the text uses approximately *40 bytes* of RAM on 64-bit architectures
- time inefficient: accessing the nth character takes O(n) time (in the case of accessing individual characters)

5.1.2 Packed unicode texts

Packed unicode text is the best option for most real-world applications.

Pros:

- memory and time efficient
- convenient & fast functions like `takeEnd` and `dropEnd`, see later

Cons:

- string literals work only with the `OverloadedString` language extension
- pattern matching works only with the `ViewPatterns` or `PatternSynonyms` language extension
- `Data.Text` should be imported with `qualified` import to avoid name clashes

Overview of the `Data.Text` API:

```
data Text                           -- name of the data structure

pack   :: String -> Text            -- conversion from String to Text
unpack :: Text -> String            -- conversion from Text to String

instance Eq Text                    -- equality test for Text values
instance Ord Text                   -- ordering for Text values

take :: Int -> Text -> Text
drop :: Int -> Text -> Text
takeEnd :: Int -> Text -> Text      -- takeEnd 2 "hello" == "lo"
dropEnd :: Int -> Text -> Text      -- dropEnd 3 "hello" == "he"
...
```

`Data.Text.IO` exports the following functions:

```
readFile  :: FilePath -> IO Text         -- read the contents of a file as a Text
writeFile :: FilePath -> Text -> IO ()   -- write a Text to a file
...
```

5.1.3 Chunks of packed unicode texts

`Data.Text.Lazy` has the same API as `Data.Text` but it is optimized for streaming large quantities of texts. It is the best choice for writing filters (an application which takes an input text and gives an output text).

Pros (compared to `Data.Text`):

- The whole text can be bigger than the memory; only the part which are worked on should fit into memory.
- some operations may be faster

Cons (compared to `Data.Text`):

- some operations may be slower

Converstion between lazy and strict `Text`s:

```
Data.Text.Lazy.toStrict   :: Data.Text.Lazy.Text -> Data.Text.Text
Data.Text.Lazy.fromStrict :: Data.Text.Text -> Data.Text.Lazy.Text
```

5.1.4 Packed bytes

`ByteString` is a string of bytes.

The type of bytes in Haskell is called `Word8`.

Pros:

- `ByteString` is more efficient than `Text`.

Cons:

- `ByteString` is *not* a proper text representation, it should be used for storing binary data or maybe ASCII texts.

Overview of the `Data.ByteString` API:

```
data ByteString                -- name of the data structure

pack :: [Word8] -> ByteString   -- conversion from list of bytes to a ByteString
unpack :: ByteString -> [Word8] -- conversion from a ByteString to list of bytes

instance Eq ByteString          -- equality
instance Ord ByteString         -- comparision

take :: Int -> ByteString -> ByteString
drop :: Int -> ByteString -> ByteString
...

readFile  :: FilePath -> IO ByteString
writeFile :: FilePath -> ByteString -> IO ()
...
```

`Data.ByteString.Char8` contains conversion functions between `ByteString` and *ASCII only* strings:

```
pack   :: String -> ByteString
unpack :: ByteString -> String
```

5.1.5 Chunks of packed bytes

`Data.ByteString.Lazy` has the same API as `Data.ByteString` but optimized for streaming large quantities of data.

Pros (compared to `Data.ByteString`):

- The whole string can be bigger than the memory; only the part which are worked on should fit into memory.
- some operations may be faster

Cons (compared to `Data.ByteString`):

- some operations may be slower

Conversion between lazy and strict `ByteString`s:

```
Data.ByteString.Lazy.toStrict
    :: Data.ByteString.Lazy.ByteString -> Data.ByteString.ByteString
Data.ByteString.Lazy.fromStrict
    :: Data.ByteString.ByteString -> Data.ByteString.Lazy.ByteString
```

5.2 Data combinators

5.2.1 `Maybe` values

Entities in this section are defined in `Prelude` or `Data.Maybe`.

`(Maybe c)` can be seen as a list of `c`-s with 0 or 1 element.

Type

```
Maybe :: * -> *          --- list with most one value
```

Constructors

```
Nothing   :: Maybe a                --- corresponds to the empty list
Just      :: a -> Maybe a           --- result corresponds to the singleton list
```

Instances

```
instance   Eq a => Eq (Maybe a)
instance  Ord a => Ord (Maybe a)
instance Show a => Show (Maybe a)
instance Read a => Read (Maybe a)
```

Functions

```
isNothing :: Maybe a -> Bool                    --- is empty
isJust    :: Maybe a -> Bool                    --- is non-empty
maybe     :: b -> (a -> b) -> Maybe a -> b      --- eliminator for Maybe
find      :: Foldable t => (a -> Bool) -> t a -> Maybe a   --- first element by condition
lookup    :: Eq a => a -> [(a, b)] -> Maybe b   --- lookup in an association list
```

5.2.2 `Either` – disjoint union

`Either a b` is an `a` or a `b` but not both.

Entities in this section are defined in `Prelude`.

Type

```
Either :: * -> * -> *    --- disjunct union
```

Constructors

```
Left   :: a -> Either a b    --- tag elements of the first type
Right  :: b -> Either a b    --- tag elements of the second type
```

Instances

```
instance     (Eq a, Eq b) => Eq (Either a b)
instance   (Ord a, Ord b) => Ord (Either a b)
instance (Show a, Show b) => Show (Either a b)
instance (Read a, Read b) => Read (Either a b)
```

Functions

```
either :: (a -> c) -> (b -> c) -> Either a b -> c  --- either even not (Left 3) == False
lefts  :: [Either a b] -> [a]     --- lefts  [Left 3, Right 'c', Left 4] == [3, 4]
rights :: [Either a b] -> [a]     --- rights [Left 3, Right 'c', Left 4] == ['c']
```

5.3 Containers

The standard Haskell containers are defined in the containers package.

5.3.1 `Set`

`Set a` is isomorphic to `a -> Bool` but it is more efficient.

There are conversion functions between sets and lists also:

```
toList      :: Set a -> [a]
fromList    :: Ord a => [a] -> Set a
```

Note that sets are not isomorphic to lists, because the ordering and multiplicity of elements does not matter in sets:

```
fromList ['a', 'a']  ==  fromList ['a']
fromList ['a', 'b']  ==  fromList ['b', 'a']
```

Other basic operations:

```
empty       :: Set a
singleton   :: a -> Set a
union       :: Ord a => Set a -> Set a -> Set a
delete      :: Ord a => a -> Set a -> Set a

member      :: Ord a => a -> Set a -> Bool
size        :: Set a -> Int
```

Sets can be used for eliminating duplicate elements from lists:

```
ordNub :: Ord a => [a] -> [a]
ordNub l = go empty l
  where
    go _ [] = []
    go s (x:xs) | member x s = go s xs
                | otherwise  = x : go (insert x s) xs
```

5.3.2 `Map`

`(Map k v)` represents a function from a finite subset of `k` to `v`.
`(Map k v)` is isomorphic to `(k -> Maybe v)`.

There are conversion functions between `(Map k v)` and `[(k, v)]` also:

```
toList      :: Map k v -> [(k, v)]
fromList    :: Ord k => [(k, v)] -> Map k v
```

Note that maps are not isomorphic to these pair lists, because the ordering of pairs does not matter and subsequent pairs with the same first elements take priority over the previous ones:

```
fromList [('a', 1), ('b', 2)]  ==  fromList [('b', 2), ('a', 1)]
fromList [('a', 1), ('a', 2)]  ==  fromList [('a', 2)]
```

Note that `(Set c)` is isomorphic to `(Map c ())`.

Basic operations:

```
empty       :: Map k v                    -- the empty map, i.e. the domain is empty set
singleton   :: k -> v -> Map k v                    -- (singleton 'c' 3) maps 'c' to 3
union       :: Ord k => Map k v -> Map k v -> Map k v   -- left-biased union
delete      :: Ord k => k -> Map k v -> Map k v     -- deletes an element from the domain

lookup      :: Ord k => k -> Map k v -> Maybe v     -- use the map as a function
size        :: Map k v -> Int                       -- size of the domain
```

5.3.3 `Seq`

`Seq c` is isomorphic to `[c]` but the access of elements by position is more efficient.

Basic operations:

```
empty       :: Seq a
singleton   :: a -> Seq a
(><)        :: Ord a => Seq a -> Seq a -> Seq a

splitAt     :: Ord a => Int -> Seq a -> (Seq a, Seq a)

length      :: Seq a -> Int

viewl       :: Seq a -> ViewL a

-- | View of the left end of a sequence.
data ViewL a
    = EmptyL        -- ^ empty sequence
    | a :< Seq a    -- ^ leftmost element and the rest of the sequence

viewr       :: Seq a -> ViewR a

-- | View of the right end of a sequence.
data ViewR a
    = EmptyR        -- ^ empty sequence
    | Seq a :> a    -- ^ the sequence minus the rightmost element,
                    -- and the rightmost element
```

5.4 Generic operations on data structures

5.4.1 `Monoid` type class

A data type has a `Monoid` instance if we pick two operations which behave like `(++)` and `[]` for lists:

```
(xs ++ ys) ++ zs == xs ++ (ys ++ zs)         -- associativity
        [] ++ xs == xs                       -- left unit
        xs ++ [] == xs                       -- right unit
```

The operation which is similar to `(++)` and `[]` are called `mappend` and `mempty`.

Lists are of course monoids:

```
instance Monoid [a]
```

This means the we can use `mappend` instead of `(++)` and `mempty` instead of `[]`.

The advantage of monoids is the possibility to define generic operations on all monoids. Every definition which can be given with `(++)` and `[]` alone can be generalized to data structures with `Monoid` instance.

For example,

```
concat :: [[a]] -> [a]
concat [] = []
concat (x: xs) = x ++ concat xs
```

can be generalized as

```
mconcat :: Monoid m => [m] -> m
mconcat [] = mempty
mconcat (x: xs) = x `mappend` mconcat xs
```

This means that if we define `mempty` and `mappend` for a specific data structure then we get `mconcat` for free for that data structure.

Notable `Monoid` instances defined already:

```
instance Monoid [a]
instance Monoid Text
instance Monoid ByteString
instance Ord a => Monoid (Set a)
instance Ord a => Monoid (Map a b)
instance Ord a => Monoid (Seq a)
```

5.4.1.1 `Sum` monoid

There are several data combinators defined only to override the type class instances on the underlying data structure. These data combinators are usually defined with `newtype`.

For example, `Data.Monoid` defines:

```
newtype Sum c = Sum { getSum :: c }
```

`Sum c` is isomorphic to `c`, but the difference is that `Sum c` has a dedicated `Monoid` instance:

```
instance Num a => Monoid (Sum a)
```

`mappend` and `mempty` behaves like `(+)` and `0` on the underlying data structure.

Usage example:

```
getSum (mconcat $ map Sum [1,2,3,4,5,6,7,8,9,10])  ==  55
```

5.4.1.2 `Endo` monoid

`Endo` is another data combinator which is defined to override the type class instances of a type.

`Endo c` is isomorphic to `(c -> c)`:

```
newtype Endo c = Endo { appEndo :: c -> c }
```

The `Monoid` instance on `Endo c`:

```
instance Monoid (Endo c) where
    mempty = Endo id
    Endo f `mappend` Endo g = Endo (f . g)
```

Usage example:

```
appEndo (mconcat $ replicate 10 $ Endo (*2)) 1  ==  1024
```

5.4.2 `Foldable` type class

Type `t` has a `Foldable` instance if `t` can be seen as a list.
The essence of the `Foldable` class:

```
class Foldable t where
  toList :: t a -> [a]
```

Some instances:

```
instance Foldable []
instance Foldable Set
instance Foldable Maybe   -- can be seen as list with 0 or 1 element
instance Foldable Complex -- the complex number as a 2-elem list
```

Functions which consume lists are generalized to `Foldable`:

```
length  :: (Foldable t) => t a -> Int
null    :: (Foldable t) => t a -> Bool
elem    :: (Foldable t, Eq a => a -> t a -> Bool
sum     :: (Foldable t, Num a => t a -> a
maximum :: (Foldable t, Ord a) => t a -> a
foldr   :: (Foldable t) => (a -> b -> b) -> b -> t a -> b
foldl   :: (Foldable t) => (b -> a -> b) -> b -> t a -> b
...
```

The performance of these functions would drop dramatically if they were defined with `toList`:

```
-- just an example
length  :: (Foldable t) => t a -> Int
length = List.length . toList               -- wrong, List.length is an O(n) operation
```

To fix this performance issue, the `length` method was included in `Foldable`:

```
class Foldable t where      -- t can be seen as a list
  toList :: t a -> [a]
  length :: t a -> Int
```

```haskell
instance Foldable Set where
  length = size    -- O(1)
  ...
```

The similar happened to the other functions, so eventually `Foldable` has got lots of members:

```haskell
class Foldable t where
    toList  :: t a -> [a]
    length  :: t a -> Int
    elem    :: Eq a => a -> t a -> Bool             -- note the extra Eq constraint
    maximum :: Ord a => t a -> a
    minimum :: Ord a => t a -> a
    sum     :: Num a => t a -> a
    product :: Num a => t a -> a
    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldr'  :: (a -> b -> b) -> b -> t a -> b       -- strict foldr
    foldl   :: (b -> a -> b) -> b -> t a -> b
    foldl'  :: (b -> a -> b) -> b -> t a -> b       -- strict foldl
    foldr1  :: (a -> a -> a) -> t a -> a
    foldl1  :: (a -> a -> a) -> t a -> a
    fold    :: Monoid m => t m -> m                 -- general fold
    foldMap :: Monoid m => (a -> m) -> t a -> m
```

This is not a problem in practice however, because most members have a default implementation:

```haskell
class Foldable t where
    ...
    elem :: Eq a => a -> t a -> Bool
    elem = any . (==)
    ...

-- | Determines whether any element of the structure satisfies the predicate.
any :: Foldable t => (a -> Bool) -> t a -> Bool
any p = ... -- defined with foldMap, see later
```

The final result is that `Foldable` instances need to define at least `foldMap` or `foldr` and the other members are automatically defined.

Example instance:

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

instance Foldable Tree where
   foldMap f Empty = mempty
   foldMap f (Leaf x) = f x
   foldMap f (Node l k r) = foldMap f l `mappend` f k `mappend` foldMap f r
```

It is a good exercise to implement `foldMap` and `foldr` with each-other:

```haskell
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f = foldr (mappend . f) mempty
```

```haskell
foldr :: (a -> b -> b) -> b -> t a -> b
foldr f z t = appEndo (foldMap (Endo . f) t) z
```

The following example helps to understand how the `foldMap`-defined `foldr` works:

```
   foldr (+) 0 [1,2,3]
== appEndo (foldMap (Endo . (+)) [1,2,3]) 0
== appEndo (Endo (1+) <> Endo (2+) <> Endo (3+)) 0
== appEndo (Endo ((1+) . (2+) . (3+))) 0
== ((1+) . (2+) . (3+)) 0
== (1+) ((2+) ((3+) 0))
== 6
```

A usage example for `foldMap`:

```
getSum (foldMap Sum [1..10]) == 55
```

5.4.3 `Functor` type class

`Functor` could be called `Mappable` because it is a generalization of the `map` function.

First define `map` for `Tree`s:

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

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty = Empty
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node l k r) = Node (mapTree f l) (f k) (mapTree f r)
```

The general `map` function should be something like:

```
fmap :: Functor t => (a -> b) -> t a -> t b
```

`fmap` has this type indeed. `fmap` is a class member, so the actual definition looks like

```
class Functor t where
  fmap :: (a -> b) -> t a -> t b
```

There is an operator form for `fmap`:

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

infixl 4 <$>
```

Notable `Funtor` instances:

```
instance Functor []         -- Defined in ‘GHC.Base’
instance Functor Maybe      -- Defined in ‘GHC.Base’
instance Functor ((->) r)   -- Defined in ‘GHC.Base’
instance Functor ((,) a)    -- Defined in ‘GHC.Base’
instance Functor (Either a) -- Defined in ‘Data.Either’
```

Usage examples:

```
(even <$> [1,2,3])  ==  [False,True,False]
(even <$> Just 5)   ==  Just False
(even <$> Nothing)  ==  Nothing
(even <$> (+1)) 3   ==  True
(even <$> (1, 2))   ==  (1, True)    -- affects the second element only
```

```
(even <$> Left 1)   ==  Left 1       -- Left is not affected
(even <$> Right 2)  ==  Right True
```

Chapter 6

Computations

A computation is a description how to solve a specific task.

6.1 Computation vs. data

Similiarties between computations and data structures:

- both have types
- both can be combined from smaller parts with different combinators

Differences between computations and data structures:

- data is for inspection; computations are for execution
- (as a consequence) some computations cannot be pattern matched on

Interestingly, `Maybe`, `Either` and `[]` can be seen either as a data structure or as a computation at the same time.

Examples:

- A `Maybe Int` value can be seen as a computation which gives an `Int` but which may fail.
- An `Either String Int` value can be seen as a computation which gives an `Int` but which may fail with a `String` error message.
- An `[Int]` value can be seen as a computation which gives a non-deterministic `Int`.

A bit more detailed example for lists as non-deterministic computations is the following.

There is only syntactic difference between the following definitions from the compiler's view, but the different syntax suggests different interpretations too:

```
-- the list of all Pythagorean triples
pythagoreanTriples :: [(Int, Int, Int)]
pythagoreanTriples =
    [ (a, b, c)            -- each Pythagorean triple has form (a, b, c) where
    | c <- [1..]           -- c is any positive natural number
    , b <- [1..c]          -- b is any natural number between 1 and c
    , a <- [1..b]          -- a is any a natural number between 1 and b
    , a^2 + b^2 == c^2     -- such that a^2 + b^2 == c^2
    ]
```

```haskell
-- a computation which produces a Pythagorean triple
aPythagoreanTriple :: [(Int, Int, Int)]
aPythagoreanTriple = do
    c <- [1..]                  -- let c be a positive natural number
    b <- [1..c]                 -- let b be a natural number between 1 and c
    a <- [1..b]                 -- let a be a natural number between 1 and b
    guard (a^2 + b^2 == c^2)    -- such that a^2 + b^2 == c^2
    return (a, b, c)            -- let the result be (a, b, c)
```

6.2 `IO` actions

`IO` actions are computations which may involve any kind of I/O actions.

6.2.1 The `IO` type constructor

`IO` is a built-in type constructor:

```haskell
IO :: * -> *
```

An `IO a` value can be seen as a *code of an interactive program* which returns an `a` value when the program is executed.

We say **IO action** or just **action** instead of "code of an interactive program".

Examples of types constructed with `IO`, giving an element for each:

```haskell
-- getChar  is the action which waits for a character and returns it
getChar :: IO Char

-- (putChar c) is the action of putting c to the console
putChar :: Char -> IO ()

-- (sequence xs) is the action which performs actions xs and returns their collected results
sequence :: [IO a] -> IO [a]

-- (sequence_ xs) is the action which performs actions xs and returns ()
sequence_ :: [IO a] -> IO ()

-- (pure x) is the action which immediately returns x
pure :: a -> IO a

-- (join x) is the action which first performs x, then performs the action returned by x
join :: IO (IO a) -> IO a
```

Examples of actions constructed from smaller actions:

```haskell
putStr :: String -> IO ()       -- put a string to the console
putStr s = sequence_ (map putChar s)

putStrLn :: String -> IO ()     -- put a string and a newline to the console
putStrLn s = putStr (s ++ "\n")
```

```
getLine :: IO String            -- collects input until the first newline
getLine = join (f <$> getChar)      -- (<$>) is fmap
  where
    f :: Char -> IO String
    f '\n' = pure []
    f c = (c:) <$> getLine
```

You may have noticed that there is a `Functor` instance of `IO`.

6.2.2 Performing actions

Actions i.e. codes of interactive computations are runnable in two ways:

A) Compile & run the code

   1. define `main :: IO ()`
   2. compile the module containing `main`
   3. run the produced executable program (maybe several times)

B) Interpret the code

   - enter the action in the Haskell interpreter

Example A, step 1, definition of `main`:

```
---------- contents of X.hs ----------
main :: IO ()      -- main should have this type
main = join $ f . read <$> getLine
  where
    f :: Int -> IO ()
    -- print "Hello world!" n times
    f n = sequence_ $ replicate n $ putStrLn "Hello world!"
```

Example A, step 2, compiling `main`:

```
$ ghc X
[1 of 1] Compiling Main             ( X.hs, X.o )
Linking X ...
```

Example A, step 3, running the executable (`2` is entered by the user):

```
$ ./X
2
Hello world!
Hello world!
```

Example B:

```
> getLine
hello               -- user input
"hello"             -- result of getLine
```

Actions cannot be performed in Haskell definitions, because this makes Haskell an impure language:

```
-- not allowed in safe Haskell, breaks equational reasoning
unsafePerformIO :: IO a -> a
```

```haskell
-- should be True, but it is mostly False
shouldBeTrue :: Bool
shouldBeTrue = s == s'
  where
    s = [c, c]
      where
        c = unsafePerformIO getChar

    s' = [unsafePerformIO getChar, unsafePerformIO getChar]
```

6.2.3 Elementary actions

Some elementary actions are the following:

```haskell
getChar    :: IO Char
putChar    :: Char -> IO ()
readFile   :: FilePath -> IO String       -- type FilePath = String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
-- + operations on mutable variables and arrays
-- + communication primitives between program threads
-- + database operations
-- + rendering images on the screen
-- + communication between programs, http, ...
```

6.2.4 Combinators for `IO` actions

The set of basic combinators:

```haskell
pure   :: a             -> IO a                -- actions without side effects
```

```haskell
(<$>)  :: (a -> b)      -> IO a -> IO b        -- apply a function on the returned value
```

```haskell
liftA2 :: (a -> b -> c) -> IO a -> IO b -> IO c -- combine two actions with a function
```

```haskell
join   :: IO (IO a) -> IO a                    -- flatten an action-returning action
```

Other combinators can be defined on top of the basic combinators.
For example, `sequence`, `sequence_`, `liftA5` and `join2` can be defined as:

```haskell
sequence :: [IO a] -> IO [a]
sequence [] = pure []
sequence (a: as) = liftA2 (:) a (sequence as)
```

```haskell
(>>) :: IO a -> IO b -> IO b
ia >> ib = liftA2 (\_ b -> b) ia ib         -- or:   (>>) = liftA2 (const id)
```

```haskell
sequence_ :: [IO a] -> IO ()
sequence_ [] = pure ()
sequence_ (a: as) = a >> sequence_ as
```

```haskell
(<*>) :: IO (a -> b) -> IO a -> IO b
iab <*> ia = liftA2 ($) iab ia              -- or:   (<*>) = liftA2 ($)
```

```haskell
infixl 4 <$>, <*>
```

```haskell
liftA5 :: (a -> b -> c -> d -> e -> f) -> IO a -> IO b -> IO c -> IO d -> IO e -> IO f
liftA5 f ia ib ic id ie  =  f <$> ia <*> ib <*> ic <*> id <*> ie
-- f :: a -> (b -> (c -> (d -> (e -> f))))
-- f <$> ia :: IO (b -> (c -> (d -> (e -> f))))
-- (f <$> ia) <*> ib :: IO (c -> (d -> (e -> f)))
-- ((f <$> ia) <*> ib) <*> ic :: IO (d -> (e -> f))
-- (((f <$> ia) <*> ib) <*> ic) <*> id :: IO (e -> f)
-- ((((f <$> ia) <*> ib) <*> ic) <*> id) <*> ie :: IO f
```

```haskell
join2 :: IO (IO (IO a)) -> IO a
join2 iiia = join (join iiia)          -- or: join2 = join . join
```

```haskell
(>>=) :: IO a -> (a -> IO b) -> IO b
ia >>= f = join (f <$> ia)
```

6.2.5 `do` notation

`do` notation is a syntactic sugar to combine actions.
`do` notation is desugared to `(>>)` and `(>>=)` calls.

Example 1:

```haskell
sequence [] = pure []
sequence (ia: ias) = do
    a <- ia
    as <- sequence ias
    return (a: as)
```

```haskell
sequence [] = pure []
sequence (ia: ias) =        -- do desugared
    ia >>= \a ->
    sequence ias >>= \as ->
    return (a: as)
```

Example 2:

```haskell
sequence_ [] = pure ()
sequence_ (ia: ias) = do
    ia
    ias
```

```haskell
sequence_ [] = pure ()
sequence_ (ia: ias) =        -- do desugared
    ia >>
    ias
```

Example 3:

```haskell
getLine = do
    c <- getChar
    case c of
        '\n' -> pure []
        c -> do
```

```
            cs <- getLine
            return (c: cs)
```

```
getLine =                    -- do desugared
    getChar >>= \c ->
    case c of
        '\n' -> pure []
        c ->
            getLine >>= \cs ->
            return (c: cs)
```

6.3 Generalizations of `IO` combinators

6.3.1 Random value generation

`(Gen a)` is the type of random generators of `a`-typed values.
We say just generator instead of random generator.

`Gen` is a type constructor:

```
Gen :: * -> *
```

Examples of types constructed with `Gen`, giving an element for each:
(These functions are defined in `Test.QuickCheck.Gen`. Note that `Gen` values cannot be printed but you can visualize them with `generate`, see the next section.)

```
-- Generates one of the given values.
elements :: [a] -> Gen a
```

```
-- Chooses one of the given generators, with a weighted random distribution.
frequency :: [(Int, Gen a)] -> Gen a
```

```
-- Generates a value that satisfies a predicate.
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
-- Generates a random element with each combinator and collect their result in a list
sequence :: [Gen a] -> Gen [a]
```

```
-- Generates always the given element (not random)
pure :: a -> Gen a
```

```
-- Generates a generator randomly, then generates an element with it
join :: Gen (Gen a) -> Gen a
```

Examples of generators constructed from smaller generators:

```
vectorOf :: Int -> Gen a -> Gen [a]
vectorOf n gen = sequence (replicate n gen)
```

```
infiniteListOf :: Gen a -> Gen [a]
infiniteListOf gen = sequence (repeat gen)
```

```
-- Generates a non-empty list of random length. The maximum length is given explicitly.
listOfSize :: Int -> Gen a -> Gen [a]
listOfSize n gen = join (f <$> elements [0..n])
```

```haskell
  where
    f k = vectorOf k gen
```

You may have noticed that there is a `Functor` instance for `Gen`.

6.3.1.1 Performing random value generation

We can generate one random value with generate:

```haskell
generate :: Gen a -> IO a
```

Example usage:

```
Test.QuickCheck.Gen>  generate $ elements [True, False]
False
```

For convenience, there is a `sample` function which can be used during development:

```haskell
-- Generates some example values and prints them to stdout.
sample :: Show a => Gen a -> IO ()
```

Example usage:

```
Test.QuickCheck.Gen>  sample $ elements [True, False]
True
True
True
False
True
False
True
False
False
True
False
```

```
Test.QuickCheck.Gen>  sample $ vectorOf 20 $ elements  " *"
"  **  ****** *  *  *"
"*   ***  **    *   *"
"  **    ***** ****  "
" *  *  *  *  *    * "
"  *  *** **   **   *"
" *****     **    *  "
" * ****  ** *  *  * "
" ** **  * * ** * ***"
"****  *  * **  *****"
" *      *  *** * ***"
"***    **** * * ** *"
```

6.3.1.2 Basic combinators

There are elementary generators and a set of basic combinators with which any other generators can be constructed.

The four most basic combinators are surprisingly similar to the combinators of `IO` actions:

```
pure   :: a               -> Gen a                    -- give back the given element, no choice

(<$>)  :: (a -> b)        -> Gen a -> Gen b           -- apply a function on the result

liftA2 :: (a -> b -> c) -> Gen a -> Gen b -> Gen c -- combine two generator result

join   :: Gen (Gen a) -> Gen a                        -- flatten an generator-returning generator
```

Definitely, there is a structure here which is worth to be abstracted out.

6.4 Generic combinators for computations

In fact, there are several type classes based on the above functions, each of them has its own merits:

```
(<$>)  :: (a -> b) -> f a -> f b                | | | |Functor     |
liftA2 :: (a -> b -> c) -> f a -> f b -> f c    | | |              |
pure   :: a -> f a                              | | |Applicative   |
join   :: f (f a) -> f a                        | |Monad           -
mzero  :: f a                                   |                  |
mplus  :: f a -> f a -> f a                     |MonadPlus         |Alternative
```

The idea is the following.
`(f t)` denotes a computation which returns a `t` value.

- `(<$>)` applies a pure function on the result of a computation.

  This set of combinators is called `Functor`.

- `liftA2` combines the results of two sub-computations
  `pure` creates a computation which gives back a specific value

  With `liftA2`, `pure` and `(<$>)` one can combine finite many computations arbitrarily, but it is not possible to pattern match on the results of computations.

  This set of combinators is called `Applicative`.

- `join` with `(<$>)` makes possible to pattern match on the result of a computation: If `m` is computation resulting a `t` value and `f` is a function on `t` values resulting computation, then `join (f <$> m)` is a computation which invokes `m` and then invokes `f v` where `v` is the result of `m`. Here `f` is able to pattern match on `v`.

  With `join`, `liftA2`, `pure` and `(<$>)` one can construct dynamic computations, i.e. computations where the choice of the next sub-computation may depend on the result of the previous sub-computations.

  This set of combinators is called `Monad`.

- `mzero` makes possible to finish a computation in the middle.
  `mplus` makes possible to add alternative directions to the computation.

  With `mzero`, `mplus`, `liftA2`, `pure` and `(<$>)` one can construct non-deterministic computations.

  This set of combinators is called `Alternative`.

  With `mzero`, `mplus`, `join`, `liftA2`, `pure` and `(<$>)` one can construct dynamic non-deterministic computations.

This set of combinators is called `MonadPlus`.

6.4.1 `Functor`

Class definition:

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

  -- replace the result of a computation
  (<$) :: a -> f b -> f a
  (<$) = fmap . const               -- default class member
```

You may expect that all `Functor` instance satisfy the following laws. $f \equiv g$ means that f and g have the same behaviour.

```
fmap id ≡ id
fmap (g . h) ≡ fmap g . fmap h
```

Notable definitions:

```
void :: Functor f => f a -> f ()
void x = () <$ x
```

6.4.2 `Applicative`

Class definition:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  infixl 4 <*>, *>, <*
  (<*>) :: f (a -> b) -> f a -> f b

  -- combine two computations and keep the result of the first
  (<*) :: f a -> f b -> f a
  (<*) = liftA2 const

  -- combine two computations and keep the result of the second
  (*>) :: f a -> f b -> f b
  a1 *> a2 = (id <$ a1) <*> a2
```

Notable definitions:

```
-- combine the results of two computations
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

```
-- conditionally do a computation
when :: Applicative f => Bool -> f () -> f ()
```

```
-- opposite of when
unless :: Applicative f => Bool -> f () -> f ()
```

```
-- do several computations and collect their results
sequenceA :: Monad m => [m a] -> m [a]
```

```
-- computation which does nothing interesting and results a () value
unit :: f ()
```

```
-- pair the results of two computations
(**) :: f a -> f b -> f (a, b)
```

Laws (f g means that f and g are isomorphic):

```
unit ** v ≅ v
u ** unit ≅ u
u ** (v ** w) ≅ (u ** v) ** w
```

6.4.3 `Monad`

Class definition:

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b

  return :: a -> m a                    -- same as pure

  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n

  fail   :: String -> m a               -- will be moved to MonadFail
```

Notable definitions:

```
-- effectful fmap
(=<<) :: Monad m => (a -> m b) -> m a -> m b
(=<<) = flip (>>=)
```

```
-- effectful function composition
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
(f <=< g) a = f =<< g a
```

```
-- flipped (<=<)
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(>=>) = flip (<=<)
```

Laws:

```
     pure <=< f  ≡  f                   -- like       id . f ≡ f
     f <=< pure  ≡  f                   -- like       f . id ≡ f
(f <=< g) <=< h  ≡  f <=< (g <=< h)     -- like  (f . g) . h ≡ f . (g . h)
```

6.4.3.1 `State` monad

The simplified interface of `State` monad:

```
State :: * -> * -> *   -- (State Int Char) produces a Char with the help of an Int state
```

```
instance Monad (State s)        -- and also Functor, Applicative
```

```
state :: (s -> (a, s)) -> State s a
runState :: State s a -> s -> (a, s)
```

Example usage:

```
newId :: State Int Int
newId = state $ \s -> (s+1, s+1)

numberLine :: String -> State Int String
numberLine s = do
    i <- newId
    pure $ show i ++ ". " ++ s
```

```
Main>  runState (mapM numberLine ["hello","world"]) 0
(["1. hello","2. world"],2)
```

Useful functions:

```
evalState :: State s a -> s -> a        -- the final state is not needed
evalState m s = fst $ runState m s
```

```
execState :: State s a -> s -> s        -- only the final state is needed
execState m s = snd $ runState m s
```

6.4.3.2 Implementation of `State`

```
{-# language ViewPatterns #-}

newtype State s a = State {runState :: s -> (a, s)}

state :: (s -> (a, s)) -> State s a
state = State

instance Functor (State s) where
    -- fmap :: (a -> b) -> State s a -> State s b
    fmap f (State g) = State $ \(g -> (a, s)) -> (f a, s)

instance Applicative (State s) where
    pure a = State $ \s -> (a, s)

    -- (<*>) :: State s (a -> b) -> State s a -> State s b
    State sf <*> State sa = State $ \(sf -> (f, sa -> (a, s))) -> (f a, s)

instance Monad (State s) where
    return = pure

    -- (>>=) :: State s a -> (a -> State s b) -> State s b
    State f >>= g = State $ \(f -> (a, s)) -> runState (g a) s
```

6.4.3.3 `Maybe` monad

`Maybe a` can be seen as a computation which either succeeds and gives an `x :: a` or fails.

Implementation:

```
instance Functor Maybe where
    fmap f = maybe Nothing (Just . f)

instance Applicative Maybe where
    pure = Maybe
    mf <*> ma = maybe Nothing (\f -> maybe Nothing (\a -> Just (f a))) mf

instance Monad Maybe where
    ma >>= f = maybe Nothing f ma
```

Tests:

```
(+1) <$> Nothing  ==  Nothing
(+1) <$> Just 3  ==  Just 3
Nothing   <*> Nothing  ==  Nothing
Nothing   <*> Just 3   ==  Nothing
Just (+1) <*> Nothing  ==  Nothing
Just (+1) <*> Just 3   ==  Just 3
Nothing  >>=  (\x -> if odd x then Nothing else Just (x + 1)) == Nothing
Just 3   >>=  (\x -> if odd x then Nothing else Just (x + 1)) == Nothing
Just 4   >>=  (\x -> if odd x then Nothing else Just (x + 1)) == Just 5
```

6.4.4 `Alternative`

Class definition:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  some :: f a -> f [a]
  many :: f a -> f [a]

some v = (:) <$> v <*> many v
many v = some v <|> pure []
```

Laws:

```
    empty <|> x ≡ x
    x <|> empty ≡ x
(x <|> y) <|> z ≡ x <|> (y <|> z)
```

Notable definitions:

```
-- (guard b) returns () if b is True, and it is mzero if b is False.
guard :: Alternative f => Bool -> f ()
```

```
-- One or none
optional :: Alternative f => f a -> f (Maybe a)
```

6.4.5 `Traversable`

Motivation: `fmap` is generalized `map`; traverse is generalized `fmap`:

```
map      ::                                   (a ->   b) -> [a] ->    [b]
fmap     ::                     Functor t  => (a ->   b) -> t a ->    t b
traverse :: (Applicative f, Traversable t) => (a -> f b) -> t a -> f (t b)
```

`fmap` maps a *pure* function, `traverse` maps an *effectful* function.

Usage example:

```
-- generalized (zip [0..])
assignIds :: Traversable t => t a -> t (Int, a)
assignIds t = evalState (traverse assignId t) 0
  where
    assignId :: a -> State Int (Int, a)        -- the effect is (State Int)
    assignId x = do
        i <- newId
        pure (i, x)

newId :: State Int Int
newId x = state $ \s -> (s, s+1)
```

```
Main>  assignIds "hello"   -- same as  zip [0..] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
```

```
Main>  assignIds $ Map.fromList [("a",10),("b",4),("c",20)]
fromList [("a",(0,10)),("b",(1,4)),("c",(2,20))]
```

6.5 Monad transformers

Goal: mix different side effects

6.5.1 `StateT s` – adding state `s` to a computation

6.5.1.1 Interface

```
StateT :: * -> (* -> *) -> (* -> *)

instance Functor m => Functor (StateT s m)
instance Monad m => Applicative (StateT s m)
instance Monad m => Monad (StateT s m)

StateT :: (s -> m (a, s)) -> StateT s m a
runStateT :: StateT s m a -> s -> m (a, s)
```

6.5.1.2 Implementation

```
newtype StateT s m a
    = StateT {runStateT :: s -> m (a, s)}

instance Functor m => Functor (StateT s m) where
    -- fmap :: (a -> b) -> StateT s m a -> StateT s m b
    fmap f (StateT g) = StateT $ \s -> first f <$> g s

instance Monad m => Applicative (StateT s m) where
    -- pure :: a -> StateT s m a
    pure a = StateT $ \s -> pure (a, s)

    -- (<*>) :: StateT s m (a -> b) -> StateT s m a -> StateT s m b
    StateT sf <*> StateT sa = StateT $ \s -> do
        (f, s) <- sf s
        (a, s) <- sa s
        pure (f a, s)

instance Monad m => Monad (StateT s m) where
    -- (>>=) :: StateT s m a -> (a -> StateT s m b) -> StateT s m b
    StateT f >>= g = StateT $ \s -> do
        (a, s) <- f s
        runState (g a) s
```

6.5.1.3 Usage example

```
newInt :: Monad m => StateT Int m Int
newInt = StateT $ \s -> pure (s, s+1)

lift :: Monad m => m a -> StateT s m a
lift m = StateT $ \s -> do  a <- m
                            pure (a, s)

numberLine :: String -> StateT Int IO String
numberLine s = do
    c <- lift $ do
        putStr $ "Number this line? " ++ s ++ "\n(y/n) "
        c <- getChar
        putStr "\n"
        pure c
    case c of
        'y' -> do
            i <- newInt
            pure $ show i ++ ". " ++ s
        _ -> pure s

test :: IO ()
test = do
    (xs, _) <- flip runStateT 1 $
```

```
          traverse numberLine ["first","second","third"]
    traverse_ putStrLn xs
```

6.5.1.4 `State` defined with `StateT`

```
type State s = StateT s Identity
```

The definition of the `Identity` monad:

```
newtype Identity a = Identity {runIdentity :: a}

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)
instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity a = Identity (f a)
instance Monad Identity where
    Identity a >>= g = g a
```

6.5.2 `ExceptT e` – add exception handling

Remark: The `Maybe` monad is isomorphic to `ExceptT () Identity`.

6.5.2.1 Interface

```
ExceptT :: * -> (* -> *) -> (* -> *)

ExceptT :: m (Either e a) -> ExceptT e m a
runExceptT :: ExceptT e m a -> m (Either e a)

instance Functor m => Functor (ExceptT e m)
instance Monad m => Applicative (ExceptT e m)
instance Monad m => Monad (ExceptT e m)
```

6.5.2.2 How to throw an error

```
throwError :: Monad m => e -> ExceptT e m a
throwError e = ExceptT $ pure $ Left e
```

The actual `throwError` is more polymorphic (in an ad-hoc way).

Chapter 7

Testing

7.1 QuickCheck

`QuickCheck` is a property based testing library for Haskell, which means that the programmer defined properties of functions are automatically checked for random inputs.

7.1.1 Quickly check properties

The main top-level function of `QuickCheck` is `quickCheck`.
Example usage:

```
Test.QuickCheck>  quickCheck $ \a b c -> (a + b) + c == a + (b + c)
+++ OK, passed 100 tests.
```

Here 100 random integer values was generated for each of `a`, `b` and `c` and `(a + b) + c == a + (b + c)` was evaluated to `True` for all of them, so the test succeeded.

Another example usage:

```
Test.QuickCheck>  quickCheck $ \a -> a == a + a
*** Failed! Falsifiable (after 2 tests):
1
```

Here 2 random integer values was generated for `a` and for the second value, which was 1, `a == a + a` was not evaluated to `True`, so the test failed.

The type of `quickCheck` is:

```
quickCheck :: Testable p => p -> IO ()
```

The `Testable` type class has the following instances, for example:

```
instance Testable Bool
instance (Arbitrary a, Show a, Testable p) => Testable (a -> p)
```

This means that an `(a -> b -> c -> Bool)` function is `Testable` if there are `Arbitrary` and `Show` instances for `a`, `b` and `c`.

The `(Arbitrary a)` constraint means that it is possible to generate random values of the `a` type. The `Arbitrary` class has the following instances, for example:

```
instance Arbitrary Bool
instance Arbitrary Integer
instance Arbitrary a => Arbitrary [a]
instance (Arbitrary a, Arbitrary b) => Arbitrary (Either a b)
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b)
```

So there is an `Arbitrary` instance for `Integer` which yields that a fuction `(Integer -> Integer -> Integer -> Bool)` is testable too.

7.1.2 Changing generator behaviour

A randomly generated `Integer` value can be 0 too, so the following test will fail:

```
Test.QuickCheck>  quickCheck $ \a b -> (a `div` b) * b + a `mod` b == a
*** Failed! Exception: 'divide by zero' (after 1 test):
0          -- a
0          -- b
```

There are sevaral fixes to this problem:

A) filter out 0 values in the generated random values for `b`
B) change the test case such that if `b == 0` then it becomes `True`
C) do not generate 0 values for `b` at all

Solution C) is the best, and luckily there is a nice implementation for it. The trick is that there is a `NonZero` data combinator, which is just a wrapper

```
newtype NonZero a = NonZero {getNonZero :: a}
```

which has an overridden `Arbitrary` instance:

```
instance (Num c, Eq c, Arbitrary c) => Arbitrary (NonZero c) where
  -- here we call arbitrary for c but omit zero values
  arbitrary = NonZero <$> (arbitrary `suchThat` (/= 0))
```

The updated test case which uses `NonZero`:

```
Test.QuickCheck>  quickCheck $ \a (NonZero b) -> (a `div` b) * b + a `mod` b == a
+++ OK, passed 100 tests.
```

Other modifier usage examples:

```
Test.QuickCheck>  quickCheck $ \(NonNegative a) (Positive b) -> a `div` b == a `quot` b
+++ OK, passed 100 tests.
```

```
Test.QuickCheck>  quickCheck $ \(NonEmpty xs) -> xs == head xs: tail xs
+++ OK, passed 100 tests.
```

```
Test.QuickCheck Data.List>  quickCheck $ \(Ordered xs) -> sort xs == xs
+++ OK, passed 100 tests.
```

7.1.3 Run more tests

Sometimes it is not enough to run 100 random tests:

```
Test.QuickCheck Data.List>  quickCheck $ \a -> a < 0.9 || a > 1
+++ OK, passed 100 tests.
```

One can specifiy the number of random tests with `withMaxSuccess`:

```
Test.QuickCheck Data.List>  quickCheck $ withMaxSuccess 10000 $ \a -> a < 0.9 || a > 1
*** Failed! Falsifiable (after 307 tests):
0.9784714901633009
```

7.1.4 Testing higher order functions

Testing higher order functions is interesting because random functions should be generated for the inputs. Without further explanation, here is an example how to do this with `QuickCheck`:

```
>  quickCheck $ \(Fn (f :: Int -> Int)) (Fn (g :: Int -> Int)) x -> (f . g) x == (g . f) x
*** Failed! Falsifiable (after 3 tests and 20 shrinks):
{_->0}
{0->1, _->0}
0
```

7.1.5 Custom data types

Any data type is supported if it has an `Arbitrary` instance.

The `Arbitrary` class provides a random generator for each instance:

```
class Arbitrary a where
  arbitrary :: Gen a
```

`Gen` is explained at Random value generation.

7.1.6 Sized random value generation

Size of randomly generated values matters:

- `quickCheck` tries smaller values first and larger values later
- For recursive or deeply nested data structures, smaller and smaller values should be generated with increasing recursion levels, otherwise the generated value could blow up.

The meaning of size depends on the actual data structure:

- length for lists
- maximum value for `Integer`
- …

Size control can be achieved by the following two functions:

```
scale :: (Int -> Int) -> Gen a -> Gen a

sized :: (Int -> Gen a) -> Gen a
```

`sized` is used in `Arbitrary` instances.
For example, the `Arbitrary` instance for lists is implemented with `listOf` which is implemented with `sized`:

```
listOf :: Gen a -> Gen [a]
listOf gen = sized $ \n ->
  do k <- choose (0,n)
     vectorOf k gen
```

We can observe the increasing list sizes with sample too:

```
Test.QuickCheck>  sample (arbitrary :: Gen [Int])
[]
[]
[-1,0]
[0,-6]
[-6]
[6,-6,-9,-5,-2,-6,-9,4,-3]
[1,-8,11,7,5,-11,-9,6]
[1,-8,-5,12,-8,0,-8,-4,-9,-4,-4,3,-9]
[0,0,-1,6,-11,-9,-9,-14,0,-16,-11,-15,15,-4,14]
[-4,15]
[20,-12,-4,-11,-18,17,-13,-10,2,-12]
```

`scale` can be used directy during debugging random generators:

```
Test.QuickCheck>  sample $ scale (*2) (arbitrary :: Gen [Int])
[]
[3,0,-1]
[5,2,4,-8,7,-2,4]
[1,12,-8,-11,12,-12,2,-10,0]
[2,-4,15,14,2,-10]
[8,-10]
[20,14,-10,-24,3,-17,21,-14,-4,-14,-22,8,-7,-20,-3,-18]
[16,-18,13,-5]
[17,-14,0,-11,16,15,3,27,-17,-2,5,27,15,9,23,27,10,18,13,21,-25,4,-16,27,8,-8,-30,13,3]
[15,-8,-23]
[3,-26,-30,-23,-14,19,-25,18,-19,34,-12,-5,26]
```

`scale` is used implicitly by `mapSize`:

```
Test.QuickCheck>  quickCheck $ \a -> a == take 100 a
+++ OK, passed 100 tests.                                   -- wrong!!!
```

```
Test.QuickCheck>  quickCheck $ mapSize (*10) $ \a -> a == take 100 a
*** Failed! Falsifiable (after 13 tests and 2 shrinks):
[(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),()
,(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),()
,(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),(),()
,(),(),(),(),(),(),(),()]
```

Chapter 8

Pandoc's source code

8.1 Web locations

The most recent *released* source code and API of Pandoc packages can be found at HackageDB, the Haskell community's central package archive of open source software. The source code of the development versions is on GitHub.

The two main Pandoc related packages and their main contents are the following:

- **pandoc-type** on HackageDB and on GitHub
  - inner document data structure definition (1 `.hs` module, ~24KB)
  - basic operations on the inner representation (5 `.hs` module, ~49KB)
  - basic tests on the inner representation
- **pandoc** on HackageDB and on GitHub
  - pandoc API implementation
    * infrastructure (45 `.hs` modules, ~495KB)
    * source code of readers in the `Readers` directory (45 `.hs` modules, ~918KB)
    * source code of writers in the `Writers` directory (36 `.hs` modules, ~885KB)
  - pandoc executable source code (1 `.hs` module, ~1.3KB)
  - static data files
    * templates for each output format (39 files, ~69Kb)
  - test framework
    * source code (46 `.hs` files, ~336KB)
    * tests (576 files, ~4762KB)

8.2 Overview of Pandoc's software architecture

Pandoc has a modular design: it has a reader for each input format, which parses the input document and produces an inner representation of the document (like an abstract syntax tree or AST), and a writer for each output format, which converts this inner representation into the target.

The inner representation can be observed via the `native` output format:

```
$ echo 'Hello, *World*!' | pandoc --to native
[Para [Str "Hello,",Space,Emph [Str "World"],Str "!"]]
```

The inner representation can be serialized in `JSON` format:

```
$ echo 'Hello, *World*!' | pandoc --to json
{"blocks":[{"t":"Para","c":[{"t":"Str","c":"Hello,"},{"t":"Space"}
,{"t":"Emph","c":[{"t":"Str","c":"World"}]},{"t":"Str","c":"!"}]}]
,"pandoc-api-version":[1,17,3,1],"meta":{}}
```

The `json` format has a fast de-serializer. An example of de-serialization:

```
$ echo 'Hello, *World*!' | pandoc --to json | pandoc --from json
<p>Hello, <em>World</em>!</p>
```

One can run filters on the JSON format. The filter can be implemented in Haskell or in any other languages.

Readers and writers have *extensions* which can be turned on and off individually.

Each output format has a default *template* file which is used to produce standalone documents in the given format. The default template is customizable with variables like `title`, `date`, `lang`, … or one can replace the default template by a custom template file.

```
________                  _____________                 _____________                  ______
input(s)  ---reader--->  native format  ---filter--->  native format  ---writer--->  output
          customized by                 freely given                  customized by
          - language                                                  - language
            - extensions                                                - extensions
                                                                        - template file
                                                                          - variables
```

The detailed data flow of pandoc and related main data structures are shown on ref{fig:flow}.

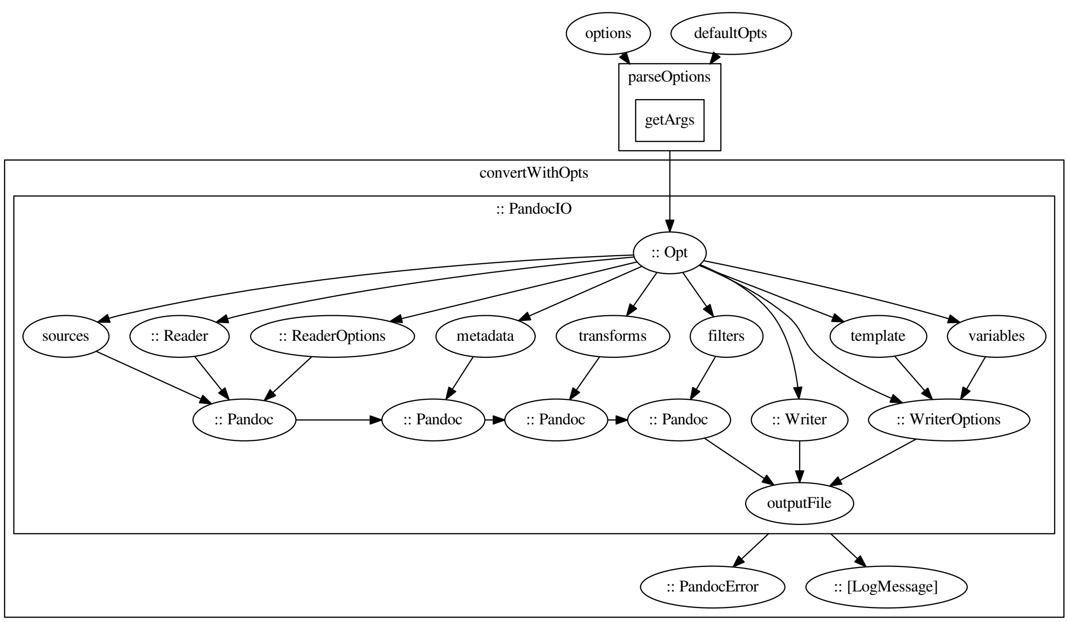

Figure 8.1: Main data flow

Main components of Pandoc and related main data structures are shown on 8.2. The next sections discuss the components one-by-one.

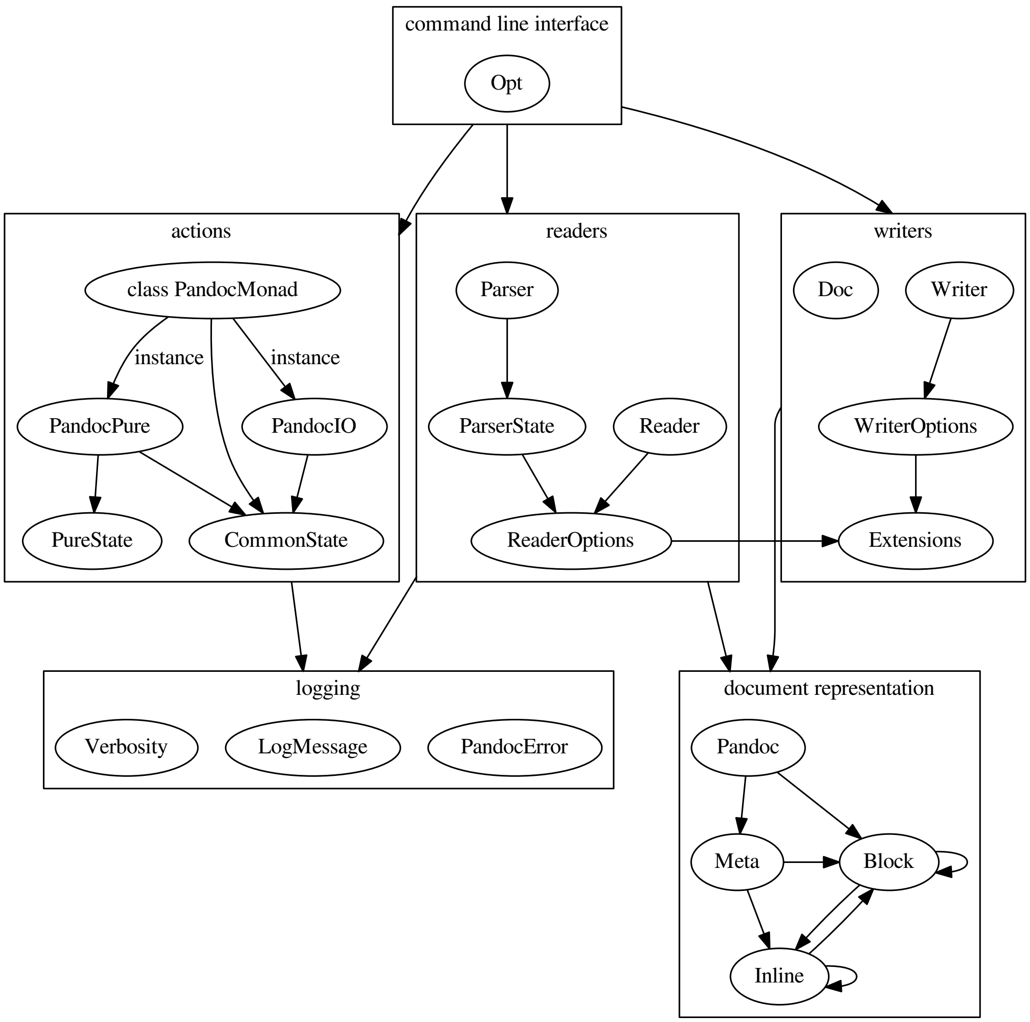

Figure 8.2: Dependencies between main components and their data structures

8.3 Document representation

A Pandoc document has two main parts: metavalues and contents (which is a list of `Blocks`):

```
data Pandoc = Pandoc Meta [Block]
```

The content is built up with `Inline` and `Block` elements:

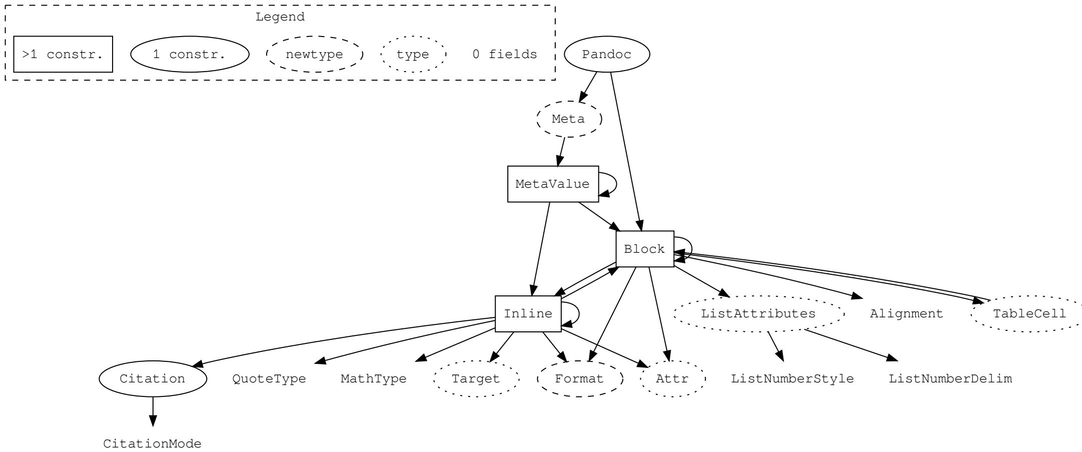

Figure 8.3: Dependencies between data types for document representation

```
data Inline      -- content placed inline, i.e. without line breaks before and after it
data Block       -- content placed with line breaks around it
```

8.3.1 Inline elements

8.3.1.1 Basic inline elements

```
data Inline
    = Str String            -- ^ Text (string)
    | Space                 -- ^ Inter-word space
    | ...
```

The most basic inline element is a string of characters:

```
Str "Hello" :: Inline
```

Inter-word spaces are represented by `Space`:

```
Space :: Inline
```

For example, "Hello world!" is represented as list of inline elements:

```
[Str "Hello", Space, Str "world!"] :: [Inline]
```

8.3.1.2 Non-canonical representations

There are several representations of the same text, but only one of them is *canonical*.

The general rules for canonicity (in order of importance):

1. **Use the most specific constructor**

For example, `Space` should be used instead of `Str " "`.

2. **Use less constructors**

   For example, `[Str "world!"]` should be used instead of `[Str "world", Str "!"]`.

Examples:

```
[Str "Hello", Space, Str "world!"] :: [Inline]          -- canonical
```

```
[Str "Hello world!"] :: [Inline]                        -- non-canonical (space inside Str)
```

```
[Str "Hello", Space, Str "world", Str "!"] :: [Inline]  -- non-canonical (Str next to Str)
```

8.3.1.3 Spaces

```
data Inline
    = ...
    | Space                 -- ^ Inter-word space
    | SoftBreak             -- ^ Soft line break
    | LineBreak             -- ^ Hard line break
    | ...
```

A *soft line break* is rendered as a space or as a newline depending on the page width.

Spaces cannot be next to each other:

```
[Space, SoftBreak]      -- non-canonical (space next to space)
```

Non-breaking space `'\160'` is stored inside `Str` elements.

8.3.1.4 Basic markups

```
data Inline
    = ...
    | Emph [Inline]             -- ^ Emphasized text
    | Strong [Inline]           -- ^ Strongly emphasized text
    | Strikeout [Inline]        -- ^ Strikeout text
    | Superscript [Inline]      -- ^ Superscripted text
    | Subscript [Inline]        -- ^ Subscripted text
    | SmallCaps [Inline]        -- ^ Small caps text
    | Quoted QuoteType [Inline] -- ^ Quoted text
    | ...
```

The type of quotations:

```
data QuoteType = SingleQuote | DoubleQuote
```

The general rules of canonicity apply:

```
[Emph [Str "a"], Emph [Str "b"]]             -- non canonical (Emph next to Emph)
```

```
[Emph [Str "a", Emph [Str "b"]]]             -- non canonical (Emph inside Emph)
```

```
[Emph [Str "a", Strikeout [Emph [Str "b"]]]] -- non canonical (Emph deeply inside Emph)
```

```
[Strong [Str "a", Emph [Str "b"]]]           -- non canonical (Strong shadows Emph)
```

8.3.1.5 Links and images

Only links to actual images are stored in the document, so the representation of images is similar to the representation of links:

```
data Inline
    = ...
    | Link  Attr [Inline] Target -- ^ Hyperlink: alt text (list of inlines), target
    | Image Attr [Inline] Target -- ^ Image:  alt text (list of inlines), target
    | ...
```

`Target` is a synonym for (URL, title) pairs:

```
type Target = (String, String)  --  (URL, title)
```

Example image:

```
Image nullAttr [Str "dog_cannot_be_shown"] ("http://...", "pitbull")
```

Attributes are discussed later.

8.3.1.6 Notes

Footnotes are stored inline at the place where they are referred in the text. They are moved to the end of the text when the text is rendered.

```
data Inline
    = ...
    | Note [Block]          -- ^ Footnote or endnote
    | ...
```

8.3.2 Block elements

8.3.2.1 Paragraphs

```
data Block
    = Plain [Inline]        -- ^ Plain text, not a paragraph
    | Para [Inline]         -- ^ Paragraph
    | LineBlock [[Inline]]  -- ^ Multiple non-breaking lines
    | ...
```

`Plain xs` is rendered as a plain block of text.

`Para xs` is rendered as a paragraph (with `<p>` tags in HTML).

`LineBlock xss` is rendered as non-breaking lines with line breaks between them (like a verse).

8.3.2.2 Block quotes

Block quotes are usually rendered with more indentation.

```
data Block
    = ...
    | BlockQuote [Block]    -- ^ Block quote (list of blocks)
    | ...
```

8.3.2.3 Headers

Headers with level 1, 2, 3, 4, 5, 6 are supported in HTML output.

```
data Block
    = ...
    | Header Int Attr [Inline] -- ^ Header - level (integer) and text (inlines)
    | ...
```

Attributes are discussed in Attributes.

8.3.2.4 Horizontal rule

Rendered as a horizontal rule.

```
data Block
    = ...
    | HorizontalRule        -- ^ Horizontal rule
    | ...
```

8.3.3 Attributes

There are 3 different kind of attributes:

- identifiers

  Identifiers are rendered as HTML identifiers in HTML output.

  Identifiers are used for labels and link anchors in a few other writers.

- classes

  The typical use case of classes is to style HTML output with CSS.

  Other use cases:

  - Headers with the class `unnumbered` will not be numbered.
  - Code blocks supports the `numberLines` and `lineAnchors` classes.
  - Code blocks and code fragments use classes like `haskell` to set the language (for syntax highlighting).
  - `smallcaps` class is used to write text in small caps
  - Native divs use the `column` class to set multi-column layout.

- key-value pairs

  Use cases:

  - `lang` and `dir` keys are used to set the language and direction (`rtl` or `ltr`) of text.
  - `startFrom` key is used to set the start number if code block lines are numbered.
  - Native divs with `column` class use the `with` key for specifying the number of columns.

  - The `width` and `height` keys on images are treated specially.

`Attr` is a triple of an identifier, classes and key-value pairs. The `""` string is used if there is no identifier attribute.

```
-- | Attributes: identifier, classes, key-value pairs
type Attr = (String, [String], [(String, String)])
```

`nullAttr` is the default:

```
nullAttr :: Attr
nullAttr = ("",[],[])
```

8.3.3.1 Native spans and divs

Native spans and divs are used to set attributes of any inlines or list of blocks.

```
data Inline
    = ...
    | Span Attr [Inline]    -- ^ Generic inline container with attributes
    | ...
```

```
data Block
    = ...
    | Div Attr [Block]      -- ^ Generic block container with attributes
    | ...
```

8.3.4 Source code in documents

Inline code fragments:

```
data Inline
    = ...
    | Code Attr String      -- ^ Inline code (literal)
    | ...
```

Code blocks:

```
data Block
    = ...
    | CodeBlock Attr String -- ^ Code block (literal) with attributes
    | ...
```

The following attributes are treated specially in writers:

- Classes like `haskell` are used to set the language (for syntax highlighting).
- The `numberLines` (or `number-lines`) class will cause the lines of the code block to be numbered, starting with 1 or the value of the `startFrom` key.
  - `startFrom` key is used to set the start number
- The `lineAnchors` (or `line-anchors`) class will cause the lines of code blocks to be clickable anchors in HTML output.

8.3.5 Raw elements

Raw inlines and raw blocks is treated as raw content with the designated format.

```
data Inline
    = ...
    | RawInline Format String -- ^ Raw inline
    | ...
```

```
data Block
    = ...
    | RawBlock Format String -- ^ Raw block
    | ...
```

```
-- | Formats for raw blocks
newtype Format = Format String
```

8.3.6 Walking documents

`Text.Pandoc.Walk` defines

```
class Walkable a b where
    walk  ::            (a ->   a) -> b ->   b    -- pure walk
    walkM :: Monad m => (a -> m a) -> b -> m b    -- effectful walk
    query :: Monoid c => (a -> c) -> b -> c
```

There are lots of `Walkable` instances, notable ones are:

```
instance Walkable Block  Pandoc
instance Walkable Inline Pandoc
```

Usage examples:

```
allCaps :: Pandoc -> Pandoc     -- capitalize all string
allCaps = walk f
  where
    f :: Inline -> Inline
    f (Str xs) = Str $ map toUpper xs
    f x = x
```

```
every2ndCaps :: Pandoc -> Pandoc     -- capitalize every 2nd inline element
every2ndCaps p = evalState (walkM f p) False
  where
    f :: Inline -> State Bool Inline
    f (Str xs) = do
        b <- state $ \b -> (b, not b)
        pure $ Str $ if b then map toUpper xs else xs
    f x = pure x
```

```
getLinks :: Pandoc -> [Target]
getLinks = query f
  where
    f :: Inline -> [Target]
    f (Link _ _ t) = [t]
    f _ = []
```

8.3.7 Pandoc filters

Program which normalizes a pandoc file:

```
import qualified Data.Text.IO as Text
import Text.Pandoc

main :: IO ()
main = do
    text <- Text.readFile "example.md"
    p <- runIOorExplode $ readMarkdown (def {readerExtensions = pandocExtensions}) text
    text' <- runIOorExplode $ writeMarkdown (def {writerExtensions = pandocExtensions}) p
    Text.writeFile "exampleOut.md" text'
```

It is easy to modify this program to transform the document, for example capitalize headers in it.

Another option is to use *pandoc filters*.

`Text.Pandoc.JSON` defines

```
class ToJSONFilter a where
    toJSONFilter :: a -> IO ()
```

Notable instances are

```
instance Walkable a Pandoc => ToJSONFilter (a -> a)
instance Walkable a Pandoc => ToJSONFilter (a -> [a])
```

so `toJSONFilter` can be specialized as, for example

```
toJSONFilter :: (Inline -> Inline) -> IO ()
-- produces a program which walks a pandoc JSON representation with the given function
```

Example usage:

```
------------------------- contents of capitalize.hs
#!/usr/bin/env runghc
import Data.Char
import Text.Pandoc
import Text.Pandoc.JSON

main :: IO ()
main = toJSONFilter f
  where
    f :: Inline -> Inline
    f (Str xs) = Str $ map toUpper xs
    f x = x
```

This "filter" can be used with pandoc like this:

```
> pandoc --filter ./capitalize.hs
```

8.3.8 Additional features

The document representation also supports features like lists, tables, math formulas, citations slide shows, internationalization and filters.

8.3.8.1 Lists

The constructors related to lists in the `Block` data type:

```
data Block
    = ...
    | OrderedList ListAttributes [[Block]] -- ^ Ordered list (attributes
                            -- and a list of items, each a list of blocks)
    | BulletList [[Block]]  -- ^ Bullet list (list of items, each
                            -- a list of blocks)
    | DefinitionList [([Inline],[[Block]])]  -- ^ Definition list
                            -- Each list item is a pair consisting of a
                            -- term (a list of inlines) and one or more
                            -- definitions (each a list of blocks)
    | ...
```

List attributes:

```
type ListAttributes = (Int, ListNumberStyle, ListNumberDelim)
```

Style of list numbers:

```
data ListNumberStyle = DefaultStyle
                     | Example
                     | Decimal
                     | LowerRoman
                     | UpperRoman
                     | LowerAlpha
                     | UpperAlpha deriving (Eq, Ord, Show, Read, Typeable, Data, Generic)
```

Delimiter of list numbers:

```
data ListNumberDelim = DefaultDelim
                     | Period
                     | OneParen
                     | TwoParens deriving (Eq, Ord, Show, Read, Typeable, Data, Generic)
```

8.3.8.2 Tables

The constructors related to tables in the `Block` data type:

```
data Block
    = ...
    | Table [Inline] [Alignment] [Double] [TableCell] [[TableCell]]  -- ^ Table,
                            -- with caption, column alignments (required),
                            -- relative column widths (0 = default),
                            -- column headers (each a list of blocks), and
                            -- rows (each a list of lists of blocks)
    | ...
```

8.3.8.3 Math formulas

The constructor related to math formulas in the `Inline` data type:

```haskell
data Inline
    = ...
    | Math MathType String  -- ^ TeX math (literal)
    | ...
```

8.3.8.4 Citations

The constructor related to citations in the `Inline` data type:

```haskell
data Inline
    = ...
    | Cite [Citation]  [Inline] -- ^ Citation (list of inlines)
    | ...
```

8.4 Custom classes

Given the following types:

```haskell
-- | Metadata for the document:  title, authors, date.
newtype Meta = Meta { unMeta :: M.Map String MetaValue }
```

```haskell
data MetaValue = MetaMap (M.Map String MetaValue)
               | MetaList [MetaValue]
               | MetaBool Bool
               | MetaString String
               | MetaInlines [Inline]
               | MetaBlocks [Block]
```

We would like to have a polymorphic `setMeta`:

```haskell
setMeta "title" (text "The title") meta   --  :: String -> Inlines -> Meta -> Meta
setMeta "title" (text "The title") doc    --  :: String -> Inlines -> Pandoc -> Pandoc
setMeta "author" [text "author #1", text "author #2"] doc
                                          --  :: String -> [Inlines] -> Pandoc -> Pandoc
```

so `setMeta` should have type something like

```haskell
setMeta :: (MetaSettable b a) => String -> b -> a -> a
```

or better if we can factor the constraint `(MetaSettable b a)` into `(ToMetaValue b, HasMeta a)` because we have to define less instances later:

```haskell
setMeta :: (ToMetaValue b, HasMeta a) => String -> b -> a -> a
```

What should be `ToMetaValue`?

```haskell
class ToMetaValue a where
  toMetaValue :: a -> MetaValue
```

```haskell
instance ToMetaValue Bool where
  toMetaValue = MetaBool
```

```haskell
instance ToMetaValue Inlines where
  toMetaValue = MetaInlines . toList
```

```haskell
instance ToMetaValue a => ToMetaValue [a] where
  toMetaValue = MetaList . map toMetaValue
```

```haskell
instance ToMetaValue MetaValue where
  toMetaValue = id
```

What should be `HasMeta`? The idiomatic solution would be

```haskell
class HasMeta a where
  setMetaPre :: String -> MetaValue -> a -> a
  deleteMeta :: String -> a -> a

setMeta :: (ToMetaValue b, HasMeta a) => String -> b -> a -> a
setMeta key val = setMetaPre key (toMetaValue val)
```

The actual solution does the same thing with a bit more work.

To specialize the too polymorphic `setMeta` some wrappers are defined with restricted types:

```haskell
setTitle :: Inlines -> Pandoc -> Pandoc
setTitle = setMeta "title"

setAuthors :: [Inlines] -> Pandoc -> Pandoc
setAuthors = setMeta "author"

setDate :: Inlines -> Pandoc -> Pandoc
setDate = setMeta "date"
```

Usage examples: in the haddock documentation of `Text.Pandoc.Builder`.

8.5 JSON conversion

```
$ echo 'Hello *World*!' | pandoc --to json
{"blocks":[{"t":"Para","c":[{"t":"Str","c":"Hello"},{"t":"Space"},{"t":"Emph","c":[{"t":"Str"
,"c":"World"}]},{"t":"Str","c":"!"}]}],"pandoc-api-version":[1,17,3,1],"meta":{}}
```

```
$ echo 'Hello *World*!' | pandoc --to json | pandoc --from json
<p>Hello <em>World</em>!</p>
```

JSON conversion scheme:

```
                    decode
        <-----------------------------
        parseJSON
Pandoc  <--------     Value    <-------    ByteString
        -------->              ------->
         toJSON
        ----------------------------->
                    encode
```

JSON conversion API:

```haskell
module Data.Aeson where
```

```haskell
...

encode :: ToJSON a => a -> ByteString
decode :: FromJSON a => ByteString -> Maybe a

class ToJSON a where
    toJSON :: a -> Value
    ...

class FromJSON a where
    parseJSON :: Value -> Parser a
    ...

-- JSON "syntax tree" in Haskell
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null

type Object = HashMap Text Value

type Array = Vector Value

data Parser a

instance Monad Parser
```

`ToJSON` instances in `Text.Pandoc.Definition`:

```haskell
instance ToJSON Inline where
  toJSON (Str s) = tagged "Str" s
  toJSON (Emph ils) = tagged "Emph" ils
  toJSON (Strong ils) = tagged "Strong" ils
  toJSON (Strikeout ils) = tagged "Strikeout" ils
  ...
  toJSON Space = taggedNoContent "Space"
  ...

tagged :: ToJSON a => [Char] -> a -> Value
tagged x y = object [ "t" .= x, "c" .= y ]

taggedNoContent :: [Char] -> Value
taggedNoContent x = object [ "t" .= x ]

object :: [Pair] -> Value  -- imported from Data.Aeson

type Pair = (Text, Value)  -- imported from Data.Aeson

(.=) :: ToJSON v => Text -> v -> Pair   -- imported from Data.Aeson, specialized type
```

`FromJSON` instances in `Text.Pandoc.Definition`:

```haskell
instance FromJSON Inline where
  parseJSON (Object v) = do
    t <- v .: "t"
    case t of
      "Str"         -> Str <$> v .: "c"
      "Emph"        -> Emph <$> v .: "c"
      "Strong"      -> Strong <$> v .: "c"
      "Strikeout"   -> Strikeout <$> v .: "c"
      ...
      "Space"       -> return Space
      ...

(.:) :: FromJSON a => Object -> Text -> Parser a   -- imported from Data.Aeson
```

8.6 Logging framework

The `PandocError` data type describes fatal errors which may happen during the execution of `pandoc`:

```haskell
data PandocError = PandocIOError String IOError
                 | PandocHttpError String HttpException
                 | PandocShouldNeverHappenError String
                 | PandocSomeError String
                 | PandocParseError String
                 | ...
```

`handleError` handles the possible fatal error, usually by exiting with a message and an exit code in case of errors:

```haskell
handleError :: Either PandocError a -> IO a
```

The `LogMessage` data type describes errors, warnings and infos produced during the execution of `pandoc`:

```haskell
data LogMessage = SkippedContent String SourcePos
                | CouldNotParseYamlMetadata String SourcePos
                | DuplicateLinkReference String SourcePos
                | DuplicateNoteReference String SourcePos
                | NoteDefinedButNotUsed String SourcePos
                | ...
                | NoTitleElement String
                | NoLangSpecified
                | ...
```

Each `LogMessage` has a verbosity level:

```haskell
data Verbosity = ERROR | WARNING | INFO
```

```haskell
messageVerbosity :: LogMessage -> Verbosity
```

8.7 Actions

The `PandocMonad` type class contains all the potentially IO-related functions used in pandoc's readers and writers. There are two instances of `PandocMonad`:

- `PandocIO`, which implements `PandocMonad`'s functions in `IO`
- `PandocPure`, which implements `PandocMonad`'s functions in an internal state that represents a file system, time, etc.
  `PandocPure` is used to run test cases.

Main data structured used by `PandocMonad`'s functions:

- The `CommonState` data type represents state that is used by all modes.
- The `MediaBag` data type holds binary resources, and an interface for interacting with it.
- `MimeType` is a synonym for `String`; the related functions detect the input and output format of document if it is not give explicitly by the user.

8.8 Readers

Readers are described with the `Parsec` parser combinator library. There are common combinators defined in the `Text.Pandoc.Parsing` module which are useful in several input formats.

The `ParserState` data type contain all information necessary during parsing for any input format. `ParserState` contains the command line options related to readers, described by the `ReaderOptions` data type. `ReaderOptions` refer to the `Extensions` data type, which describes the turned on extensions in a compact way.

The common interface of readers for the input formats is described by the `Reader` data type. The available readers are enumerated in the `readers` list.

8.9 Writers

Writers which produce indented output use the `Doc` data type, which supports formatted text output for a given screen width.

The common interface of writers for the output formats is described by the `Writer` data type. The available writers are enumerated in the `writers` list. Writers have a `WriterOptions` argument which describes the command line arguments related to writers.