Getting GHC

- Can't wait for the 1st exercise in order to get going?
- Strongly recommended:
  install `stack` (see
  https://docs.haskellstack.org/en/stable/README/)
- Follow the Quickstart guide
  for Unix/OS X the default behaviour is usually fine
  for WIndows read the Windows stuff carefully
  (use the installers, rather than manual download).

Types

- Haskell is strongly typed
  — every expression/value has a well-defined type:
  `myExpr ::  MyType`
  Read: "Value `myExpr` has type `MyType`"
- Haskell supports *type-inference:* we don't have to declare types of functions in advance. The compiler can figure them out automatically.
- Haskell's type system is *polymorphic*, which allows the use of arbitrary types in places where knowing the precise type is not necessary.
- This is just like *generics* in Java or C++ − think of `List<T>`, `Vector<T>`, etc.

More about Types

- Some Literals have simple pre-determined types.
  `'a' :: Char`
  `"ab" :: String"`
- Numeric literals are more complicated
  `1 :: ?`
  Depending on context, `1` could be an integer, or floating point number.
- Live demo regarding numerical types!
- This is common with many other languages where notation for numbers (and arithmetic operations) are often "overloaded".
- Haskell has a standard powerful way of handling overloading (the `class` mechanism).

Function Types

- A function type consists of the input type, followed by a right-arrow and then the output type
  `myFun ::  MyInputType -> MyOutputType`
- Given a function declaration like `f x = e`, if `e` has type `b`, and we know that the usage of `x` in `e` has type `a`, then `f` must have type `a -> b`. Symbolically:

$$\frac{x :: a \quad e :: b \quad f\ x = e}{f :: a \rightarrow b}$$

- Given a function application `f v`, if `f` has type `a -> b`, then `v` must have type `a`, and `f v` will have type `b`. Symbolically:

$$\frac{f :: a \rightarrow b \quad v :: a}{f\ v :: b}$$

Type Checking with Inference

When a type is provided the compiler checks to see that it is consistent with the equations for the function.

```
notNull :: [Char] -> Int
notNull xs = (length xs) > 0
```

The function `notNull` is valid, but the compiler rejects it. Why?
The compiler knows the types of `(>)` and `length`:

```
length :: [Char] -> Int      -- not quite, see later
(>) :: Int -> Int -> Bool    -- not quite either
```

Type Inference Demo !
We can correct things:

```
notNull :: [Char] -> Bool
notNull xs = (length xs) > 0
```

Now the compiler accepts the code, because the written and inferred types match.

More about types

What is the type of this function?

```
length [] = 0
length (x:xs) = 1 + length xs
```

Could it be: `length ::  [Integer] -> Integer` ?
What about ?

```
> length "abcde"
5
```

This would imply a type: `length ::  [Char] -> Integer` !
We *could* make an arbitrary decision...

Parametric polymorphism (I)

In Haskell we are allowed to give that function a general type:

```
length :: [a] -> Integer
```

This type states that the function `length` takes a list of values and returns an integer. There is no constraint on the kind of values that must be contained in the list, except that they must all have the same type `a`.
What about this: `head (x:xs) = x` ?
This takes a list of values, and returns one of them. There is no constraint on the types of things that can be in the list, but the kind of thing that is returned must be that same type:

```
head :: [a] -> a
```

Revisting `notNull`

Reminder

```
notNull xs = (length xs) > 0
```

The compiler knows the types of `(>)` and `length`:

```
length :: [a] -> Int
(>) :: Int -> Int -> Bool   -- still not quite right
```

Type inference will deduce

```
notNull :: [a] -> Bool
notNull xs = (length xs) > 0
```

This is the most general type possible for `notNull`

Parametric polymorphism (II)

What is the type of `sameLength`?

```
sameLength [] [] = True
sameLength (x:xs) [] = False
sameLength [] (y:ys) = False
sameLength (x:xs) (y:ys) = sameLength xs ys
```

Could it be:

```
sameLength :: [a] -> [a] -> Bool
```

This type states that `sameLength` takes a list of values of type `a` and another list of values of *that same type* `a` and returns a `Bool`. It's overconstrained - why?

Parametric polymorphism (III)

A type signature can use more than one type variable (it can vary in more than one type). Again, we consider:

```
sameLength [] [] = True
sameLength (x:xs) [] = False
sameLength [] (y:ys) = False
sameLength (x:xs) (y:ys) = sameLength xs ys
```

What would the most general type that could work be?

```
sameLength :: [a] -> [b] -> Bool
```

The two lists do not have to contain the same type of elements for `length` to work. `sameLength` has two *type parameters*. When doing type inference, Haskell will *always* infer the **most general type** for expressions.

The Haskell Prelude [*H2010* 9]

- The "`Standard Prelude`" is a library of functions loaded automatically (by default) into any Haskell program.
- Contains most commonly used datatypes and functions
- [*H2010* 9] is a *specification* of the Prelude the actual code is compiler dependent

Prelude extracts (I)

- Infix declarations

```
infixr 9  .
infixr 8  ^, ^^, ..
infixl 7  *, /, 'quot', 'rem', 'div', 'mod'
infixl 6  +, -
infixr 5  :      —builtin, and cannot be altered.
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, 'seq'
```

Higher precedence numbers bind tighter.
Function application binds tightest of all

Prelude extracts (II)

- Numeric Functions

```
subtract  :: (Num a) => a -> a -> a
even, odd :: (Integral a) => a -> Bool
gcd       :: (Integral a) => a -> a -> a
lcm       :: (Integral a) => a -> a -> a
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)      :: (Fractional a, Integral b) => a -> b -> a
```

The `Num`, `Integral` and `Fractional` annotations have to do with *type-classes* — see later.

Prelude extracts (III)

- Function Functions

```
id         :: a -> a
const      :: a -> b -> a
(.)        :: (b -> c) -> (a -> b) -> a -> c
flip       :: (a -> b -> c) -> b -> a -> c
seq        :: a -> b -> b
($), ($!)  :: (a -> b) -> a -> b
```

Prelude extracts (IV)

- Boolean Type & Functions

```
data Bool  =  False | True
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool
```

Prelude extracts (V)

- List Functions

```
map    :: (a -> b) -> [a] -> [b]
(++)   :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
head   :: [a] -> a
tail   :: [a] -> [a]
null   :: [a] -> Bool
length :: [a] -> Int
(!!)   :: [a] -> Int -> a
repeat :: a -> [a]
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
elem   :: Eq a => a -> [a] -> Bool
```