## Assessment

- Exam (75%)
  - Section A, three questions, do two (70% of exam)
  - Section B, multiple choice, 15Q (30% of exam)
- Coursework (25%)
  - Formative—to help you learn.
  - A series of exercises, done as 'homework'
  - Lab (or Labs?) simply to help get started with Haskell and all its bits. Only if you need help.

## Lab 0 : Lab plan & procedure

- Details of exercise released before lab session
  - resources
  - task
  - submission procedure & deadline
- Lab Class
  - Help with exercises, any anything else Haskell-related.
  - Attendance at lab class is *NOT* required.
  - ICTLab I has 40 seats, class-size is 110 approx.
  - Attend *only* if you need help!
- Don't forget to submit the final version of your work!
- Deadlines are important:
  - exercises may be linked: solution to lab $n$ could be input to lab $n + 1$.
- First lab: Thursday 20th September, 2pm, ICTLAB1
- Other labs will only occur if there is a need.

## 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).

## Atomic Types

We have some Atomic types builtin to Haskell:

- `()` the unit type which has only one value, also written as `()`.
- `Bool` boolean values, of which there are just two: `True` and `False` .
- `Ordering` comparison outcomes, with three values: `LT`, `EQ`, and `GT`.
- `Char` character values, representing Unicode characters.
- `Int` fixed-precision integer type with at least the range $[-2^{2}9 \ldots 2^{2}9 - 1]$
- `Integer` infinite-precision integer type
- `Float` floating point number of precision at least that of IEEE single-precision
- `Double` floating point number of precision at least that of IEEE double-precision

## 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:

$$\frac{\texttt{x :: a} \qquad \texttt{e :: b} \qquad \texttt{f x = e}}{\texttt{f :: a -> b}}\ [\text{FunDef}]$$

- 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:

$$\frac{\texttt{f :: a -> b} \qquad \texttt{v :: a}}{\texttt{f v :: b}}\ [\text{FunUse}]$$

## Symbolically ???

- The notation introduced on the previous slide is a standard way of defining typing rules.
  (also a common way to present rules of logical reasoning)
- The rules have the form: given some assumptions ($A_1$,...,$A_n$), we can draw some conclusion $C$
  Symbolically:

$$\frac{A_1 \qquad \ldots \qquad A_n}{C}\ [\text{RuleName}]$$

- These rules can be used bidirectionally:
  - If we know $A_1 \ldots A_n$ then we can claim $C$ is true (top-down).
  - If we want to show $C$ is true then we need to find a way to show the $A_i$ are true.

## Type Checking and 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 (Live Demo)

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

$$\frac{\texttt{x :: a} \quad \texttt{e :: b} \quad \texttt{f x = e}}{\texttt{f :: a -> b}}\ \text{[FunDef]}$$

$$\frac{\texttt{f :: a -> b} \quad \texttt{v :: a}}{\texttt{f v :: b}}\ \text{[FunUse]}$$

$$\frac{\texttt{f :: a -> b -> c} \quad \texttt{u :: a} \quad \texttt{v :: b}}{\texttt{f u v :: c}}\ \text{[Fun2Use]}$$

```
length  ::  [Char] -> Int
(>)     ::  Int -> Int -> Bool
```

## Type Checking with Inference

Having worked out the correct type, 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.