Student Online Teaching Advice Notice

The materials and content presented within this session are intended solely for use in a context of teaching and learning at Trinity.

Any session recorded for subsequent review is made available solely for the purpose of enhancing student learning.

Students should not edit or modify the recording in any way, nor disseminate it for use outside of a context of teaching and learning at Trinity.

Please be mindful of your physical environment and conscious of what may be captured by the device camera and microphone during videoconferencing calls.

Recorded materials will be handled in compliance with Trinitys statutory duties under the Universities Act, 1997 and in accordance with the Universitys policies and procedures.

Further information on data protection and best practice when using videoconferencing software is available at `https://www.tcd.ie/info_compliance/data-protection/`.

1 Week 2

1.1 Class 4

1.1.1 Exercise Zero

Exercise Zero

- Getting `stack` installed and working
- Modifying given code (`src/Ex00.hs`) so all tests pass.
- Test driven:
  - Code compiles, all tests pass: `Full Marks`
  - Code compiles, some tests pass: `Marks reduce pro-rata.`
  - Code compiles, no tests pass: `Zero Marks`
  - Code does not compile: `Zero Marks`

1.1.2 Guards

Definition by cases

Often we want to have different equations for different cases.

Mathematically we sometimes write something like this:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Signum as Cases

Math:

$$signum(x) = \begin{cases} 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Haskell:

```
signum x | x < 0  = -1
         | x == 0 =  0
         | x > 0  =  1
```

If all else fails...

Each guard is tested in turn, and the first one to match selects an alternative. This means that it is OK to have a guard that would always be true, as long as it is the *last* alternative.

So the previous definition could have been written like this:

```
signum x | x < 0   =  -1
         | x == 0  =   0
         | True    =   1
```

For readability the name `otherwise` is allowed as a synonym for `True`:

```
signum x | x < 0      =  -1
         | x == 0     =   0
         | otherwise  =   1
```

Cases by guarded alternatives

We can use guards to select special cases in functions. This function is `True` when the year number is a leap year:

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  =  True   -- 2000 was
           | mod y 100 == 0  =  False  -- 1900 wasn't
           | mod y 4 == 0    =  True   -- 2020 is
           | otherwise       =  False  -- 2021 won't be
```

Cases by guarded alternatives

Guards and patterns can be combined:

```
startswith _ [] = False
startswith c (x:xs) | x == c    = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual way.

If no guard matches then we return to the pattern matching stage and try to find another equation.

Behaviour of `startwith`

The second argument of `startswith` is a list of things, while the first argument must have the same type as the things in the list. The first line matches the case when the second list argument is empty, and ignores the first argument, returning `False`

```
> startswith 1 []
False
```

If the second list argument is not empty, then the second pattern match succeeds, and we proceed to compare the first list element (here called `x`), with the first argument (here called `c`).

```
> startswith 42 [42,41,40]
True
```

We can also avoid using the guards: the following replacement for lines two and three of the function definition have the same effect:

```
startswith c (x:xs)  =  c == x
```

Factorial! as Patterns (again!)

Math:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!, \qquad n > 0 \end{aligned}$$

Haskell:

```
factorial 0          =  1
factorial n | n > 0  =  n * factorial (n-1)
```

1.1.3 Function Notation (I)

Function Notation (I)

Consider the definition and application/use of a function in mathematics:

$$f(x) \;\widehat{=}\; x+1 \qquad\qquad f(42)$$

In a C-like language we might write:

```
int f (int x) { return (x+1) }          f(42)
```

In Haskell we could write:

```
f1(x) = x+1                             f1(42)
```

Usually, however, in Haskell, we write:

```
f2 x = x+1                              f2 42
```

Function Notation (II)

Lets add a few more arguments:

$$g(x, y, z) \;\widehat{=}\; x + y + z \qquad\qquad g(42, 57, 99)$$

In a C-like language we might write:

```
int g (int x,y,z) { return (x+y+z) }    g(42,57,99)
```

In Haskell we could write:

```
g1(x,y,z) = x+y+z                       g1(42,57,99)
```

Usually, however, in Haskell, we write:

```
g2 x y z = x+y+z                        g2 42 57 99
```

Function Notation (III)

Why does Haskell have this strange function notation?

Reason 1 Because, defining and using functions is so common that the notation should be as lightweight as possible.

Reason 2 With more than one argument, the Haskell notation proves to be surprisingly flexible (and powerful!)

We'll learn about this flexibility and power later.

Function Notation (IV)

As far as Haskell is concerned, `f1 x` and `f2(x)` are the same.

However, `g1(x,y,z)` and `g2 x y z` are not:

- Their types are different:

```
g1 :: Num a => (a,a,a) -> a
g2 :: Num a => a -> a -> a -> a
```

  Function `g1` takes a triple of numbers and returns a number, whereas, `g2` takes a number, another number and another number, and returns a number. (`Num a =>` says that `a` must be a numeric type).

- The implementation of `g2` is faster and uses less memory than that of `g1`.

1.1.4 Patterns

Pattern Matching

- Inputs: a pattern, and a Haskell expression
- Output: either a "fail", or a "success", with a binding from each variable to the part of the expression it matched.

Example:

pattern `('c':zs)`,

matches expression `'c':('a':('t':[]))`,

with `zs` bound to `'a':('t':[])`

Patterns in Haskell

- We can build patterns from atomic values, variables, and certain kinds of "constructions".
- An atomic value, such as `3`, `'a'` or `"abc"` can only match itself
- A variable, or the wildcard `_`, will match anything.
- A "construction" is either:
  1. a tuple such as `(a,b)` or `(a,b,c)`, etc.,
  2. a list built using `[]` or `:`,
  3. or a user-defined datatype.

  A construction pattern matches if all its sub-components match.

Pattern Sequences

- Function definition equations may have a sequence of patterns (e.g., the `and` function example.)
- Each pattern is matched against the corresponding expression, and all such matches must succeed. *One* binding is returned for all of the matches
- Any given variable may only occur once in any pattern sequence (it can be reused in a different equation.).

Pattern Examples

- Expect three arbitrary arguments (of the appropriate type!)

  ```
  myfun x y z
  ```

- Illegal — if we want first two arguments to be the same then we need to use a conditional (somehow).

  ```
  myfun x x z
  ```

- First argument must be zero, second is arbitrary, and third is a non-empty list.

```
myfun 0 y (z:zs)
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose first element is character 'c'

```
myfun 0 y ('c':zs)
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose tail is a singleton.

```
myfun 0 y (z:[z'])
```

COMMENTARY: Instead of

```
myfun x x z = whatever
```

we should write:

```
myfun x y z | x == y = whatever
```

.

Pattern Matching (summary)

- Pattern-matching can *succeed* or *fail*.

- If successful, a pattern match returns a (possibly empty) *binding*.
- A binding is a mapping from (pattern) variables to values.
- Examples:

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
x (3:ys) 3	99 [3,2,1] 3	Ok, $x \mapsto 99, ys \mapsto [2,1]$

- Binding $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$ can also be written as a (simultaneous) *substitution* $[99, 1, [2,3]/x, y, ys]$ where 99, 1 and $[2,3]$ replace x, y and ys respectively.

So Bindings are Substitutions!

In the λ-calculus we say that $((\lambda x \bullet M)\ N)$ can become $M[N/x]$.

We note that $f\ x = fexpr$ can be considered the same as $f = \lambda x \bullet fexpr$, s $f\ arg$ becomes $fexpr[arg/x]$.

From this we can say the way Haskell pattern matching works is that when a match succeeds, we use the relevant binding as a substitution for the righthand-side of that pattern.

E.g., given

```
myfun x (y:ys) 3 = x:y:y:y:ys
```

then we would observe the following:

```
myfun 99 [1,2,3] 3
 = (x:y:y:y:ys)[99,1,[2,3]/x,y,ys]
 = (99:1:1:1:[2,3])
```

1.2 Class 5

1.2.1 Lexical and Syntactical Matters (I)

Haskell: Syntactical Details

- Now time for a proper introduction to the *language* of Haskell.
- Official Reference: "Haskell 2010 Language Report"
  - Online: `http://www.haskell.org/onlinereport/haskell2010/`
- In this course we refer to sections of that report thus:
  - [*H2010* 3.4]
  - Haskell 2010 Language Report, Section 3.4

Haskell is Case-Sensitive [H2010 2.4]

For example, the following names are all *different*:

```
ab aB Ab AB
```

Program Structure [H2010 1.1]

A Haskell script can be viewed as having four levels:

1. A Haskell program is a set of modules, that control namespaces and software re-use in large programs.
2. A module consists of a collection of declarations, defining ordinary values, datatypes, type classes, and fixity information.
3. Next are expressions, that denote values and have static types.
4. At the bottom level is the lexical structure, capturing the concrete representation of programs in text files.

(We focus on the bottom three for now).

Notational Conventions [H2010 2.1]

- The report uses the following notation for syntax:

$[syn]$	optional occurence of syn
$\{syn\}$	zero or more repetitions of syn
(syn)	grouping
$syn_1 \vert syn_2$	choice between alternatives
$syn_{\langle syn' \rangle}$	difference—elements generated by syn, except those generated by syn'
`fibonacci`	terminal syntax in `typewriter font`

- It uses BNF-like syntax, with productions of the form:

$$nonterm \quad \rightarrow \quad alt_1 | alt_2 | \ldots | alt_n$$

  "$nonterm$ is either an alt_1 or alt_2 or ..."

- The trick is distinguishing | (alternative separator) from | , the vertical bar character (and similarly for characters {}[]()).

Comments [H2010 2.3]

A Haskell script has two kinds of comments:

1. End-of-line comments, starting with --.
2. Nested Comments, started with {- and ending with -}

Example, where comments are in red.

```
myfun x -- end-of-line, but -} won't end it
 = let
     y = 2 {- nested, but -- ignored here -} ; z = 3
     {-
     a = 4 {- was 42 but I changed my mind -}
     b = 5
     -}
   in y + z * x
```

Namespaces [H2010 1.4]

- Six kinds of names in Haskell:
  1. Variables, denoting values;
  2. (Data-)Constructors, denoting values;

3. Type-variables, denoting types;
4. Type-constructors, denoting 'type-builders';
5. Type-classes, denoting groups of 'similar' types;
6. Module-names, denoting program modules.

- Two constraints (only) on naming:
  - Variables (1) and Type-variables (3) begin with lowercase letters or underscore, Other names (2,4,5,6) begin with uppercase letters.
  - An identifier cannot denote both a Type-constructor (4) and Type-class (5) in the same scope.
- So the name `Thing` (*e.g.*) can denote a module, data-constructor, and either a class or type-constructor in a single scope.

Character Types (I) [H2010 2.2]

The characters can be grouped as follows:

- *special* : `( ) , ; [ ] ` { }`
- *whitechar* → *newline*|*vertab*|*space*|*tab*
- *small* → `a`|`b`|...|`z`|`_`
- *large* → `A`|`B`|...|`Z`
- *digit* → `0`|`1`|...|`9`
- *symbol* : `! # % & * + . / < = > ? @ \ ^ | - ~`
- the following characters are not explicitly grouped- `: " '`

(There is also stuff regarding Unicode characters (beyond ASCII) that we shall ignore—so the above is not exactly as shown in [*H2010* 2.2]).

Lexemes (I) [H2010 2.4]

The term "lexeme" refers to a single basic "word" in the language.

- Variable Identifiers (*varid*) start with lowercase and continue with letters, numbers, underscore and single-quote. `x x' a123 myGUI _HASH very_long_Ident_indeed''`

- Constructor Identifiers (*conid*) start with uppercase letters and continue with letters, numbers, underscore and single-quote. `T  Tree  Tree'  My_New_Datatype  Variant123`
- Variable Operators (*varsym*) start with any symbol, and continue with symbols and the colon. `<+>  |:|  ++  +  -  ==>  ==  &&  #!#`
- Constructor Operators (*consym*) start with a colon and continue with symbols and the colon. `:+:  :~  :===  :$%&`

Identifiers (*varid*, *conid*) are usually prefix, whilst operators (*varsym*,*consym*) are usually infix.

Lexemes (II) [H2010 2.4]

- Reserved Identifiers (*reservedid*):

```
case class data default deriving do else foreign if
import in infix infixl infixr instance let module
newtype of then type where _
```

- Reserved Operators (*reservedop*): `..  :  ::  =  \  |  <-  ->  @  ~  =>`

Literals [H2010 2.5,2.6]

We give a simplified introduction to literals (actual basic values)

- Integers (*integer*) are sequences of digits Examples: `0  123`
- Floating-Point (*float*) has the same syntax as found in mainstream programming languages. `0.0  1.2e3  1.4e-45`
- Characters (*char*) are enclosed in single quotes and can be escaped using backslash in standard ways. `'a'  '$'  '\''  '"'  '\64'  '\n'`
- Strings (*string*) are enclosed in double quotes and can also be escaped using backslash in standard ways. `"Hello World"  "I 'like' you"  "\" is a dbl-quote"  "line1\nline2"`

1.2.2 Function Notation (again)

Function Notation (V)

We can define and use functions whose names are either Variable Identifiers ($varid$) or Variable Operators ($varsym$)

For $varid$ names, the function definition uses "prefix" notation, where the function name appears before the arguments:

```
myfun x y  =  x+y+y                    myfun 57 42
```

For $varsym$ names, the function definition uses "infix" notation, where the function has exactly two arguments and the name appears inbetween the arguments:

```
x +++ y  =   x+y+y                     57 +++ 42
```

Function Notation (VI)

For $varid$ names, with functions having two[1] arguments, we can define and use them "infix-style" by surrounding them with backticks:

```
x `plus2` y  =  x+y+y                   57 `plus2` 42
```

For $varsym$ names, we can define and use them "prefix-style" by enclosing them in parentheses:

```
(++++) x y  =   x+y+y                   (++++) 57 42
```

We can define one way and use the other—all these are valid:

```
57 `myfun` 42            (+++) 57 42
plus2 57 42               57 ++++ 42
```

[1] or more ?!?

1.3 Class 6

1.3.1 Types

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 $[-2^{29} \ldots 2^{29} - 1]$

`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:

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

- 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:

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

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:

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

- 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
```

COMMENTARY: We want to work out the type of `notNull`. It is defined using the syntax: `notNull xs = (length xs) > 0`.

We want its type, something of the form: `a -> b`. This matches the form on the bottom of type rule [FunDef]. We see that the top of the rule has a pattern `f x = e`. If we apply this rule, we match `f` to `notNull`, `x` to `xs` and `e` to `(length xs) > 0`. Given that binding we can conclude, that if `notNull :: a -> b`, then the following must be true: `xs :: a` and `((length xs) > 0) :: b`.

We want to figure out what type `b` is. We note that we can rewrite `(length xs) > 0` as `(>) (length xs) 0`, using the Haskell trick to turn infix operator `>` into a prefix function `(>)`. This form of the expression matches the lower part of rule [Fun2Use], with `f` (here) bound to `(>)`, `u` bound to `length xs`, and `v` bound to `0`. The upper part of the rules includes `f :: a -> b -> c`. Given that here `f` stands for `(>)`, then we know its type (`Int -> Int -> Bool`), so types `a`,`b`,`c` must be `Int`,`Int`,`Bool`, respectively. So the rule can be used to conclude that `(>) (length xs) 0` has type `Bool`, provided that `length xs` and `0` each have type `Int`.

`0` clearly has type `Int`, and we can use rule [FunUse] to show that `length xs` has type `Int`, and `xs` has type `[Char]`

With all of this information we can go back to rule [FunDef], and deduce that: `xs :: [Char]`, `(length xs) > 0 :: Bool`, and `notNull :: [Char] -> 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.