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/`.

0.1 Class 10

0.2 Lexical and Syntactical Matters (III)

Value Declarations [H2010 Sec 4.4.3]

- A value declaration can be either a function or a pattern.
- We have already seen function declarations, which involve patterns

```
funname patn_1 patn_2 ... patn_n = expr
```

- A declaration can also have a lhs that is a single pattern:

```
patn = expr
```

  The simplest example is defining a variable to denote a (fixed) value:

```
molue = 42
```

- But we can also match complicated patterns against equally complicated expressions:

```
(a,b,c)   =  (1,2,3)
[d,e,f]   =  take 3 [1..10]
(before,after) = splitAt 42 someBigList
```

Haskell Layout Rule [H2010 Sec 2.7]

- Some Haskell syntax specifies lists of declarations or actions as follows: $\{item_1; item_2; item_3; ...; item_n\}$
- In some cases (after keywords `where`, `let`, `do`, `of`), we can drop $\{$, $\}$ and `;`.
- The layout (or "off-side") rule takes effect whenever the open brace is omitted.
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
  - For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
  - if it is indented the same amount, then a new item begins (a semicolon is inserted);
  - and if it is indented less, then the layout list ends (a close brace is inserted).

Layout Example

Offside rule (silly) example: consider $\textbf{let } x = y + 3 \wedge z = 10 \wedge f(a) = a + 2z \textbf{ in } f(x)$

- Full syntax:

```
let {  x = y + 3 ; z = 10; f a = a + 2 * z } in f x
```

- Using Layout:

```
let x = y + 3
    z = 10
    f a = a + 2 * z
in f x
```

- Using Layout (alternative):

```
let
  x =  y + 3
  z = 10
  f a
   = a + 2 * z
in f x
```

Local Declarations [H2010 Sec 3.12]

- A let-expression has the form:

$$\texttt{let } \{d_1\texttt{;} \ldots \texttt{;}d_n\}\texttt{in } e$$

  d_i are declarations, e is an expression. *The offside-rule applies.*

- Scope of each d_i is e and righthand side of all the d_is (mutual recursion)

- Example: $ax^2 + bx + c = 0$ means $x = \frac{-b \pm (\sqrt{b^2 - 4ac})}{2a}$

- ```
  solve a b c
   = let twoa = 2 * a
         discr = b*b - 2 * twoa * c
         droot = sqrt discr
     in ((droot-b)/twoa , negate ((droot+b)/twoa))
  ```

Local Declarations [H2010 Sec 3.12]

- A where-expression has the form:

$$\texttt{where } \{d_1\texttt{;} \ldots \texttt{;}d_n\}$$

  d_i are declarations. *The offside-rule applies.*

- Scope of each d_i is the expression that *precedes* `where` and righthand side of all the d_is (mutual recursion)

- ```
  solve a b c
   = ((droot-b)/twoa , negate ((droot+b)/twoa))
   where
     twoa = 2 * a
     discr = b*b - 2 * twoa * c
     droot = sqrt discr
  ```

`let` ([H2010 Sec 3.12]) vs. `where` [H2010 Sec 4.?]

- What is the difference between `let` and `where` ?
- The `let ...in ...` is a full expression and can occur anywhere an expression is expected.
- The `where` keyword occurs at certain places in *declarations*

$$\ldots\texttt{where}\{d_1\texttt{;}\ldots\texttt{;}d_n\}$$

  of

  - case-expressions [H2010 Sec 3.13]
  - `module`s [H2010 Sec 4]
  - `class`es [H2010 Sec 4.3.1]
  - `instance`s [H2010 Sec 4.3.2]
  - function and pattern righthand sides (rhs) [H2010 Sec 4.4.3]
- Both allow mutual recursion among the declarations.

Conditionals [H2010 Sec 3.6]

- For expressions, we can write a conditional using `if` ...`then`...`else`

$$exp \quad \rightarrow \quad \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp$$

- The else-part is compulsory, and cannot be left out (why not?)
- The (boolean-valued) expression after `if` is evaluated: If true, the value is of the expression after `then` If false, the value is of the expression after `else`

Case Expression [H98 3.13]

- A case-expression has the form:

$$\texttt{case}\ e\ \texttt{of}\ \{p_1\ \texttt{->}\ e_1\texttt{;}\ldots\texttt{;}p_n\ \texttt{->}\ e_n\}$$

  p_i are patterns, e_i are expressions. *The offside rule applies.*

```
odd x =
  case (x `mod` 2) of
    0 -> False
    1 -> True
```

```
empty x =
  case x of
    [] -> True
    _  -> False
```

```
vowel x =
  case x of
    'a' -> True
    'e' -> True
    'i' -> True
    'o' -> True
    'u' -> True
    _   -> False
```

Lambda abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch.

There are times when it is handy to just write a function "inline". The notation is:

```
\ x -> e
```

where `x` is a variable, and `e` is an expression that (usually) mentions `x`.

This notation reads as "the function taking `x` as input and returning `e` as a result". We can have nested abstractions

```
\ x -> \ y -> e
```

Read as "the function taking `x` as input and returning a function that takes `y` as input and returns `e` as a result".

There is syntactic sugar for nested abstractions:

```
\ x  y -> e
```

It's just notation!

The following definition groups are equivalent:

```
sqr     =  \ n -> n * n
sqr n   =  n * n
```

```
add     =  \ x y -> x+y
add x   =    \ y -> x+y
add x y =          x+y
```

Lambda application

In general, an application of a lambda abstraction to an argument looks like:

```
(\ x -> x + x) a
       ^--e--^
-- Applied:
(a + a)
```

The result is a copy of `e` where any free occurrence of `x` has been replaced by `a`. This is just the β-reduction rule of the lambda calculus.

Factorial: a comparison

A simple definition of factorial, ignoring negative numbers, is the following:

```
fac 0 = 1
fac n = n * fac (n-1)
```

But what *is* `fac`?

```
fac = ..... ?
```

```
fac = \n -> case n of
             0 -> 1
             m -> m * fac (m-1)
```

0.3 Class 11

0.4 Polymorphism and HOFs

Polymorphism brings great power!

What is the type of `length`?

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

It's `length :: [a] -> Int`

One piece of code can handle all lists, no matter what their contents!

"Polymorphism sets us free" , . . . or does it?

```
f11 :: a -> a
```

```
f11 x = ?
```

We are totally constrained here, and all we can do is reproduce the input:

```
f11 x = x
```

`f11` is in the Prelude, where it is called `id`.

```
f12 :: a -> b
```

```
f12 x = ?
```

Here all we can do is induce a runtime failure

```
f12 x = undefined
```

This will happen for values/functions with types: `a`, `a->b`, `a->b->c`, `a->b->c->d` and so on . . .

```
f121 ::  a -> b -> a
```

```
f121 x y = ?
```

We are totally constrained here, and all we can do is reproduce the first input:

```
f121 x y = x
```

`f121` is in the Prelude, where it is called `const`.

```
f122 ::  a -> b -> b
```

```
f122 x y = ?
```

We are totally constrained here, and all we can do is reproduce the second input:

```
f122 x y = y
```

`f122` is in the Prelude, where it is called `seq`, but it's strange!

```
f111 ::  a -> a -> a
```

```
f111 x y = ?
```

We are less constrained here, and have two choices:

```
f111 x y = x
OR
f111 x y = y
```

`f111`, first version, is in the Prelude, where it is called `asTypeOf`.

```
f321 ::  (a -> b -> c) -> (a,b) -> c
```

```
f321 f (x,y) = ?
```

We are totally constrained here, and all we can do apply `f` to the components of the second pair input

```
f321 f (x,y) = f x y
```

`f321` is in the Prelude, where it is called `uncurry`.

```
f213 ::  ((a,b) -> c) -> a -> b -> c
```

```
f213 g x y = ?
```

We are totally constrained here, and all we can do apply `g` to a pair built from the other inputs

```
f213 g x y = g (x,y)
```

`f213` is in the Prelude, where it is called `curry`.

```
f1221 ::  (a -> b -> c) -> b -> a -> c
```

```
f1221 f x y = ?
```

We are totally constrained here, and all we can do is apply `f` to `x` and `y` in reversed order.

```
f1221 f x y = f y x
```

`f1221` is in the Prelude, where it is called `flip`.

Prelude “PolyHOF” Summary

```
id :: a -> a              ;  id x = x
f12 :: a -> b             ;  f12 _ = undefined
const :: a -> b -> a      ;  const x y = x
seq :: a -> b -> b        ;  seq x y = y
hasTypeOf :: a -> a -> a  ;  hasTypeOf x y = x

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y

curry :: ((a,b) -> c) -> a -> b -> c
curry g x y = g (x,y)

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Polymorphism is a constraint

A polymorphic type in fact drastically reduces the options for coding a function because such code cannot use functions that require specific types (or type classes).

Defining new types (3 possibilities)

- *Type Synonyms*

  ```
  type Name = String
  ```

  Haskell considers both `String` and `Name` to be exactly the same type.

- *“Wrapped” Types*

  ```
  newtype Name = N String
  ```

  If `s` is a value of type `String`, then `N s` is a value of type `Name`. Haskell considers `String` and `Name` to be different types.

- *Algebraic Data Types*

  ```
  data Name = Official String String | NickName String
  ```

  If `f`, `s` and `n` are values of type `String`, then `Official f s` and `NickName n` are different values of type `Name`

Type Synonyms

```
type MyType = ExistingType
```

Haskell considers both `MyType` and `ExistingType` to be exactly the same type.

- Advantages Clearer code documentation Can use all existing functions defined for `ExistingType`
- Disadvantages Typechecker does not distinguish `ExistingType` from any type like `MyType` defined like this

  ```
  type Name = String ; name :: Name ; name = "Andrew"
  type Addr = String ; addr :: Addr ; addr = "TCD"
  name ++ addr  -- is well-typed
  ```

"Wrapping" Existing Types

```
newtype NewType = NewCons ExistingType
```

If `v` is a value of type `ExistingType`, then `NewCons v` is a value of type `NewType`.

- Advantages Typechecker treats `NewType` and `ExistingType` as different and incompatible. Can use type-class system to specify special handling for `NewType`. No runtime penalties in time or space !
- Disadvantages Needs to have explicit `NewCons` in front of values Need to pattern-match on `NewCons v` to define functions None of the functions defined for `ExistingType` can be used directly

Algebraic Data Types (ADTs)

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
```

```
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

- If `vi1`, ... `vik` are values of types `Typei1` ... `Typeik`, then `Dconi vi1 ... vik` is a value of type `ADTName`, and values built with different `Dconi` are always different
- Note that a `Dconi` can have no `Typeij`, in which case `Dconi` itself is a value of type `ADTName`.

Algebraic Data Types (ADTs)

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

- Advantages The only way to add genuinely *new* types to your program
- Disadvantages As per `newtype` — the need to use the `Dconi` data-constructors, and to pattern match Unlike `newtype`, these `data` types do have runtime overheads in space and time.

0.5 Class 12

Defining Functions with ADT Patterns

Consider a generic example of a `data`-declaration:

```
data ADTName
   = Dcon1 Type11 Type12 ... Type1a
   | Dcon2 Type21 Type22 ... Type2b
   ...
   | DconN TypeN1 TypeN2 ... TypeNz
```

We can define a function `myfun :: ADTName -> a` as follows:

```
myfun (Dcon1 pat11 pat12 ... pat1a) = exp1
myfun (Dcon2 pat21 pat22 ... pat2b) = exp2
...
myfun (DconN patN1 patN2 ... patNz) = expN
```

Here `patIJ` has type `TypeIJ` and all `expK` have type `a`.

User-defined Datatypes (`data`): enums

With the `data` keyword we can easily define new *enumerated* types.

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

We can define operations on values of this type by *pattern matching*:

```
weekend :: Day -> Bool
weekend Saturday = True
weekend Sunday   = True
weekend _        = False
```

The identifiers `Monday` thru `Sunday` are Data Constructors, and like the types themselves, must begin with *uppercase* letters (functions and parameters in Haskell begin with lowercase letters).

User-defined Datatypes (`data`): Recursive structures

Haskell also allows data types to be defined *recursively*.

If lists were not built-in, we could define them with `data`:

```
data List = Empty
          | Node Int List
```

compare:

```
typedef struct {
  int value;
  node *next;
} node;
```

User-defined Datatypes (`data`): Recursive structures

```
data List = Empty
          | Node Int List
```

Using this definition the list $\langle 1, 2, 3 \rangle$would be written

```
Node 1 (Node 2 (Node 3 Empty))
```

Recursive types usually mean recursive functions:

```
length :: List -> Integer
length Empty = 0
length (Node _ rest) = 1 + (length rest)
```

Parameterised data types

Of course, those lists are not as flexible as the built-in lists, because they are not *polymorphic*. We can fix that by introducing a *type-variable*:

```
data List t = Empty
            | Node t (List t)
```

compare:

```
class Node<T> {
  T value;
  Node<T> *next;
}
```

No change to the length function, but the type becomes:

```
length :: (List a) -> Integer
```

Type Parameters

The types defined using `type`, `newtype` and `data` can have type parameters themselves:

- `type TwoList t = ([t],[t])`
- `newtype BiList t = BiList ([t],[t])`
- `data ListPair t = LPair [t] [t]`
- The type "list-of-`a`", (`[a]`) can be considered a parameterised type: `[] a`.
- The names `TwoList`, `BiList`, `ListPair`, and `[]` (in the type-language of Haskell) are considered to be *Type Constructors*. They take a type as argument and build a new type using that argument.

What's in a Name?

Consider the following `data` declaration:

```
data MyType = AToken | ANum Int | AList [Int]
```

- the name `MyType` after the `data` keyword is the type name.
- the names `AToken`, `ANum` and `AList` on the rhs are data-constructor names.
- type names and data-constructor names are in different namespaces so they can overlap, e.g.:

  ```
  data Thing = Thing String | Thang Int
  ```

- The same principle applies to newtypes:

  ```
  newtype Nat = Nat Int
  ```

- We call these **Algebraic Datatypes** (ADTs)
- For a nice explanation of the name (if interested) see: [1]

Multiply-parameterised data types

Here is a useful data type:

```
data Pair a b = Pair a b

divmod :: Integer -> Integer -> (Pair Integer Integer)
divmod x y = Pair (x / y) (x `mod` y)
```

Actually, like lists, "tuples" (of various sizes) are built in to Haskell and have a convenient syntax:

```
divmod :: Integer -> Integer -> (Integer,Integer)
divmod x y = (x / y, x `mod` y)
```

As you would expect, we can use pattern matching to open up the tuple:

```
f (x,y,z) = x + y + z
```

`data`-types in the Prelude (I)

- `data () = () -- Not legal syntax; builtin`
- `data Bool = False | True`
- `data Char = ...  'a' | 'b' ... -- Unicode values`
- `data Maybe a = Nothing | Just a`
- `data Either a b = Left a | Right b`
- `data Ordering = LT | EQ | GT`
- `data [a] = [] | a :  [a] -- Not legal syntax; builtin`

[1] https://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/

`data`-types in the Prelude (II)

- `data IO a = ... -- abstract`
- `data (a,b) = (a,b)` `data (a,b,c) = (a,b,c)` `-- Not legal syntax; builtin`
- `data IOError -- internals system dependent`

`data`-types in the Prelude (III)

Standard numeric types.

The data declarations for these types cannot be expressed directly in Haskell since the constructor lists would be far too large.

- `data Int = minBound ...  -1 | 0 | 1 ...  maxBound`
- `data Integer = ...  -1 | 0 | 1 ...`
- `data Float`
- `data Double`

Another example: failure

We can write functions such as `head` so that they fail outright:

```
head (x:xs) = x -- no [] pattern, so runtime fail for head []
```

Define type `Maybe a` to represent a optional value of type `a`:

```
data Maybe a = Nothing
             | Just a
```

Now we can handle failure for `head` in a more manageable way:

```
mhead :: [a] -> Maybe a
mhead []     = Nothing
mhead (x:xs) = Just x
```

This technique is so common that `Maybe` and some useful functions are included in the standard Prelude.

An running example: Expressions

We are going to write functions that manipulate expressions in a variety of ways

```
data Expr
 = Val Double
 | Add Expr Expr
 | Mul Expr Expr
 | Sub Expr Expr
 | Dvd Expr Expr
 deriving Show -- makes it possible to see values (DEMO!)
```

$(10 + 5) * 90$ becomes `Mul (Add (Val 10) (Val 5)) (Val 90)` $10 + (5 * 90)$ becomes `Add (Val 10) (Mul (Val 5) (Val 90))`

An evaluator

We can write a function to calculate the result of these expressions:

```
eval :: Expr -> Double

eval (Val x) = x

eval (Add x y) = eval x + eval y

eval (Mul x y) = ... -- similar to above
-- similarly for Sub and Dvd
```

```
> eval (Add (Val 10) (Mul (Val 5) (Val 90)))
460.0
```

A simplifier

We can write a function to simplify expressions:

```
simp :: Expr -> Expr

simp (Val x) = (Val x)

-- we can use pattern matching in let-expressions!
simp (Add e1 e2) = let (Val x) = simp e1
                       (Val y) = simp e2
                   in Val (x+y)

simp (Dvd x y) = ... -- similar to above
-- similar for Mul and Sub
```

```
> simp (Add (Val 10) (Mul (Val 5) (Val 90)))
Val 460.0
```