CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 07: HO Programming and Type Classes

- Curried Functions
- Folding
- Type Classes

Reading: Hutton Ch. 3 & beginning of 7

You should also look at the Standard Prelude in Appendix B!

HO Programming: Curried Functions

Recall that **function slices** are created from infix functions/operators by giving one of the operands, and leaving the other out. The missing operand is a parameter – this turns a function of two arguments into a function of one argument:

```
Main> (3^2)
9

Main> (^2) 3
9

Main> (3^) 2
9
```

```
Main> (\x -> \y -> x^y) 3 2
9

Main> (\x -> x^2) 3
9

Main> (\y -> 3^y) 2
9
```

```
(3 ^ 2) 2            =>    9

(\y -> 3^y) 2 =>β 3^2   => 9
```

HO Programming: Curried Functions

But notice that what we are doing here is partially applying a function to one of its arguments, and then stopping halfway through and calling it a new function:

```
(\x -> (\y -> x^y)) 3 2
```

$=>_\beta$

```
(\y -> 3^y))      2
```

$=>_\beta$

```
3^2
```

=>

```
9
```

HO Programming: Curried Functions

We can do this any time we want, with any lambda expression with more than one argument:

```
Main> f = (\x -> (\y -> x^y)) 3

Main> f 2
9
```

By **referential transparency**, this is the same as:

```
Main> (\x -> (\y -> x^y)) 3 2
9
```

except that we "froze" the computation after applying the first argument.

HO Programming: Curried Functions

This explains why the following are all completely equivalent:

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

f x y = \z -> (x,y,z)

f x = \y -> (\z -> (x,y,z))

f x = \y z -> (x,y,z)

f = \x -> (\y -> (\z -> (x,y,z)))

f = \x y z -> (x,y,z)
```

which is proved by the type: **all these will have the same type**:

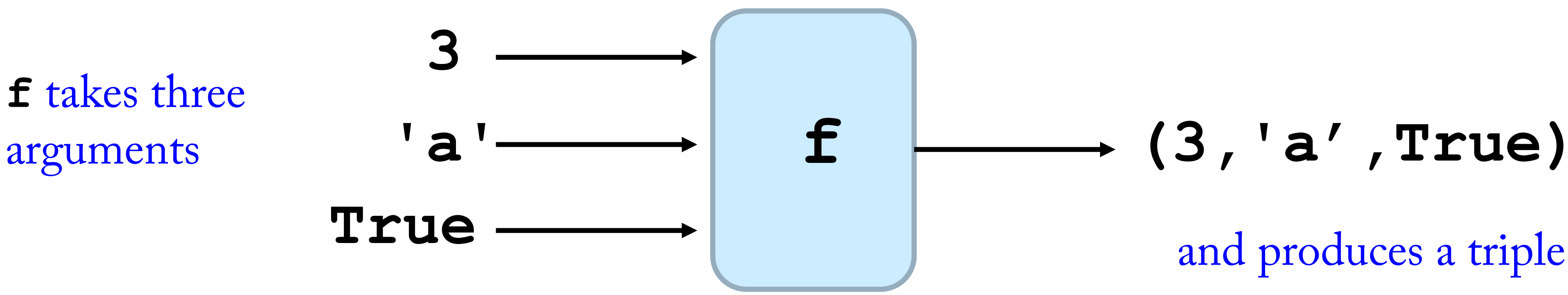
```
f :: a ->  b ->  c -> (a,b,c)

f = \x -> \y -> \z -> (a,b,c)
```

Notice how the type arrows line up with the arrows in the lambda expression!
Not a coincidence!

HO Programming: Curried Functions

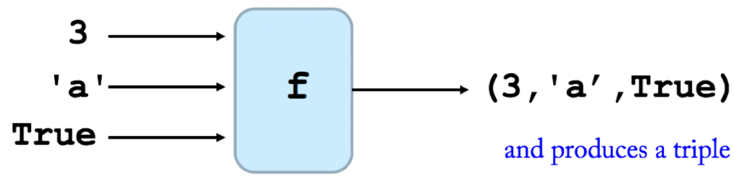
It also explains why **all functions** can be thought of as unary (one-parameter) functions.

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

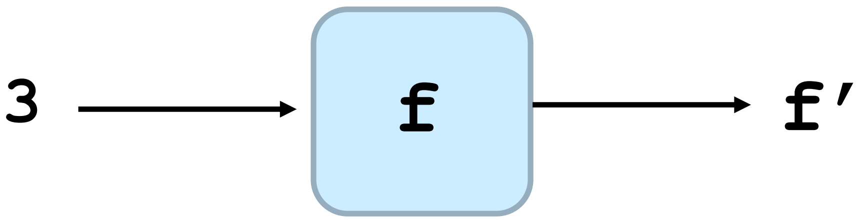


HO Programming: Curried Functions

`f` takes three arguments




```
f =  \x -> \y -> \z -> (x,y,z)
```

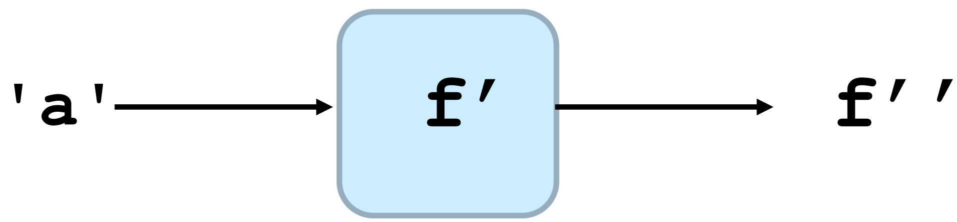


`f` takes one argument and produces a function `f'` of two arguments:

```
f x = \y -> \z -> (x,y,z)
```

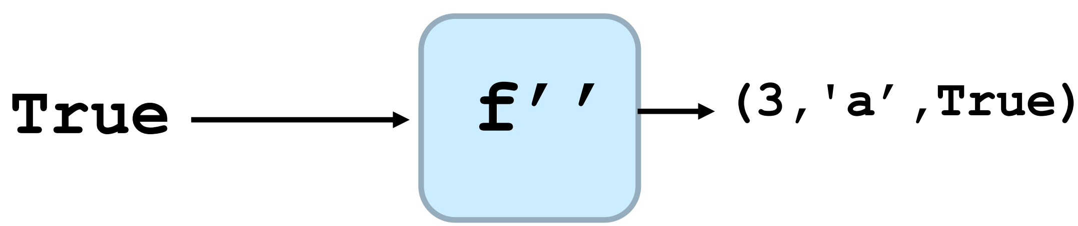
`f'` takes one argument and produces a function `f''` of one argument:

```
f' y = \z -> (3,y,z)
```



`f''` takes one argument and produces a value:

```
f'' z = (3,'a',z)
```



HO Programming: Curried Functions

This also explains why function application is left-associative and the arrow (in lambda expressions OR in type expressions) is right-associative:

```
f  3 'a' True             f :: a ->  b ->  c -> (a,b,c)
                          f = \x -> \y -> \z ->( x,y,z)


 (f 3) 'a' True           f :: a -> ( b  -> c -> (a,b,c))
                          f = \x -> (\y -> \z -> (x,y,z))


 ((f 3) 'a') True         f :: a -> ( b -> ( c -> (a,b,c)))
                          f = \x -> (\y -> (\z -> (x,y,z)))
```

HO Programming: Curried Functions

NOTE carefully that these functions DO have the same type:

```
g ::  a ->  b -> c

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

But these functions do NOT have the same type:

```
g' ::  a -> b  -> c

h' :: (a -> b) -> c
```

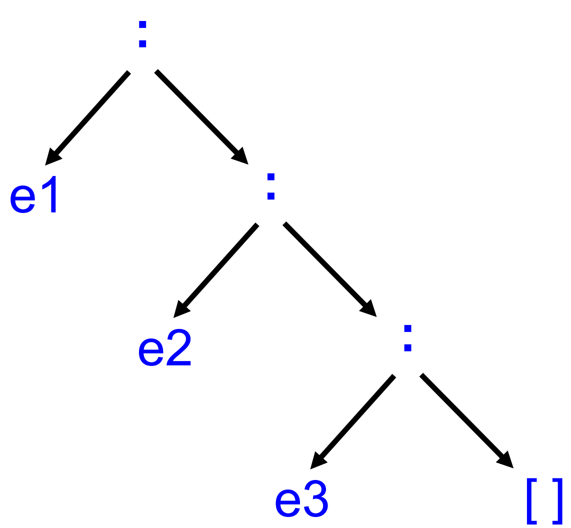
Higher-order Programming Paradigms

Reading: Hutton Ch. 7.3

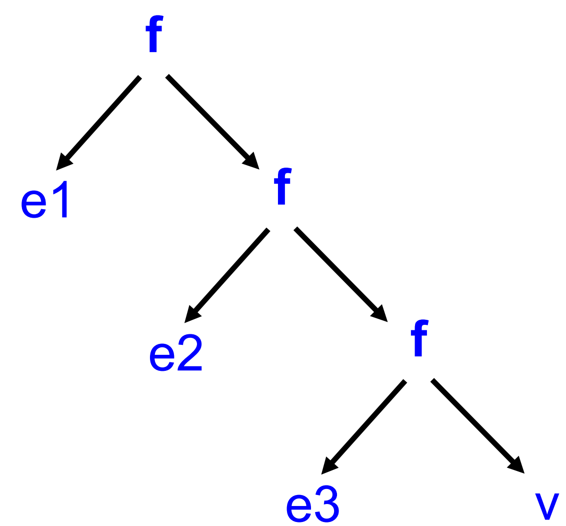
Fold (also called reduce) is another function which uses a function as a parameter. There are two versions foldr (foldr) and foldl (fold left).

Fold right takes a list (constructed with the cons operator :) and effectively replaces the cons with a function of two arguments, and the empty list with an "initial value" to get the recursion started:

```
[ e1, e2, e3 ]

e1 : ( e2 : e3 : [] )
```



`foldr f v [e1,e2,e3]`



Higher-order Programming Paradigms

Reading: Hutton Ch. 7.3

Here is a version of foldr similar to that given in the Prelude:

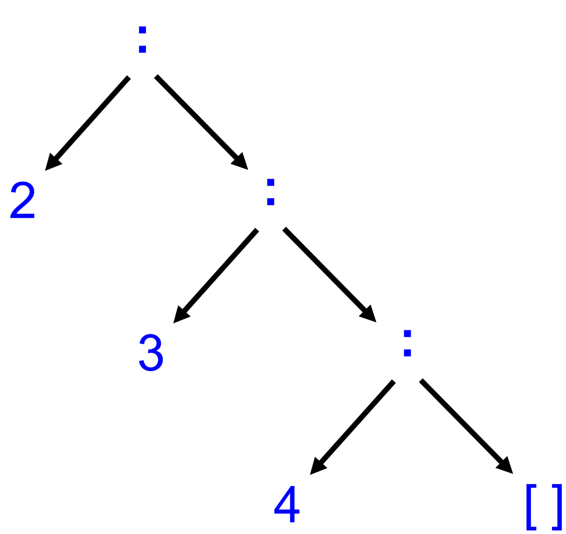
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

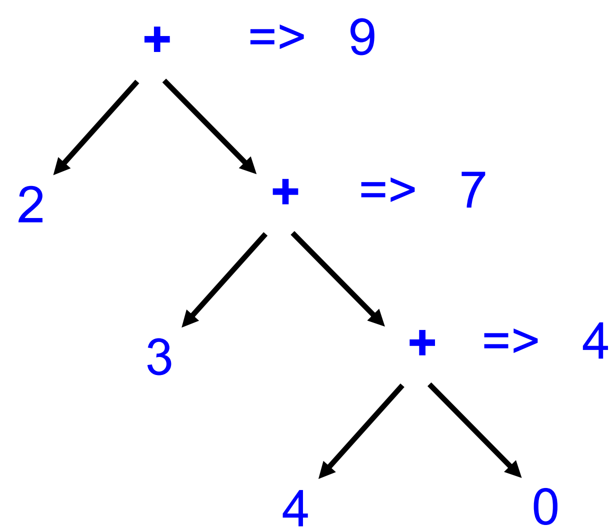
Thus, to sum the elements of the list, we could write:

`foldr (+) 0 [2,3,4]   =>   9`

`2 : ( 3 : 4 : [] )`

`2 + ( 3 + 4 + 0 )`





Higher-order Programming Paradigms

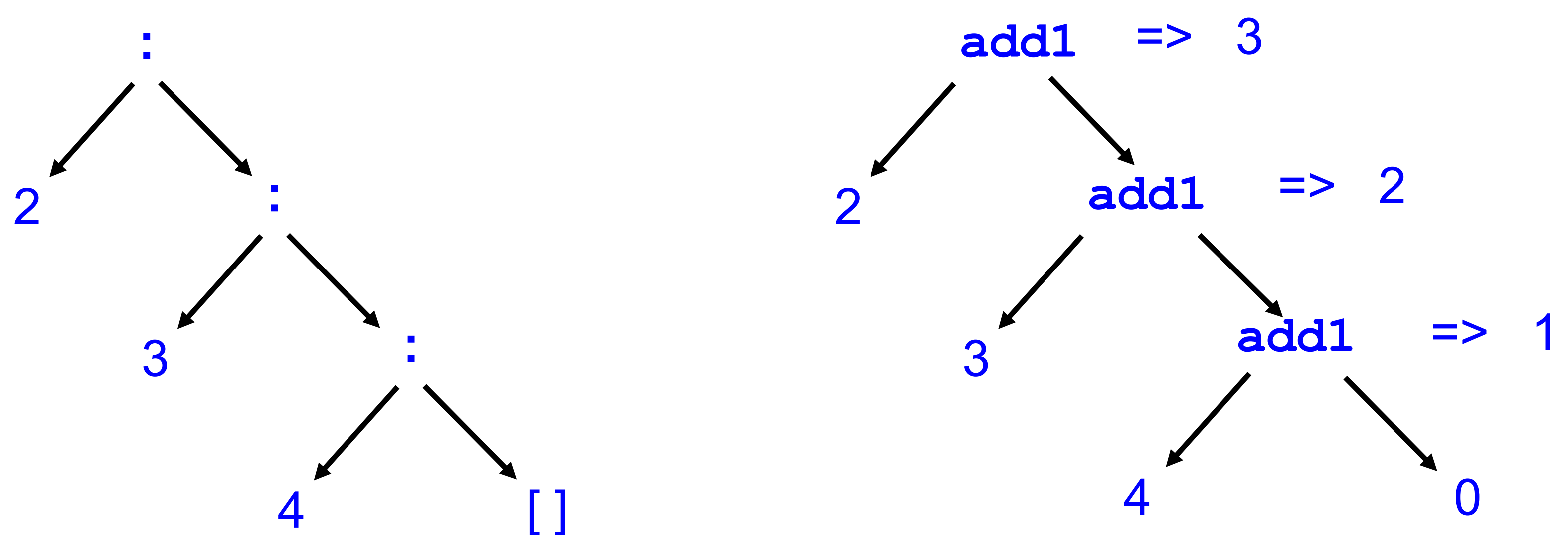
Reading: Hutton Ch. 7.3

Here are some other applications of foldr – it is actually more powerful than you might think at first!

Calculating the length of a list:

```
foldr add1 0 [2,3,4]

add1 x y = y + 1
```

```
2 : ( 3 : 4 : [] )        2 `add1` ( 3 `add1` 4 `add1` 0 )
```



Higher-order Programming Paradigms

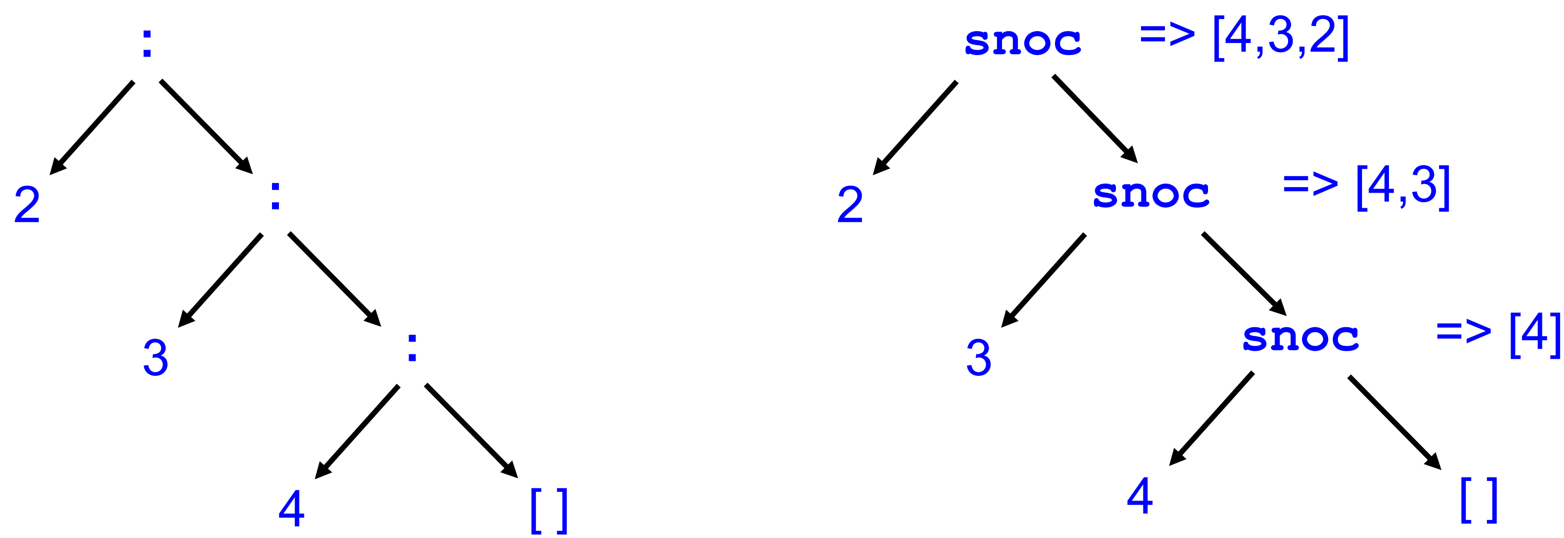
Reading: Hutton Ch. 7.3

Here are some other applications of foldr – it is actually more powerful than you might think at first!

Reversing a list:

```
snoc :: a -> [a] -> [a]          -- snoc is "cons" reversed
snoc x xs = xs ++ [x]            -- because it adds to end instead of front
```

```
foldr snoc [] [2,3,4]
```



Higher-order Programming Paradigms

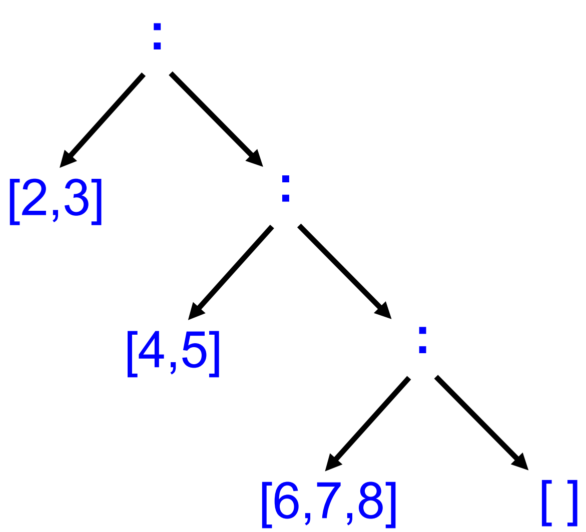
Reading: Hutton Ch. 7.3

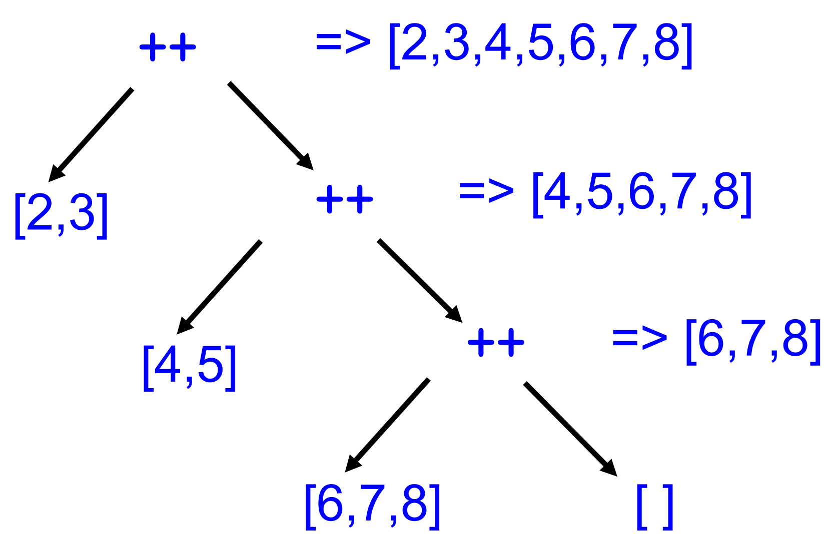
Here is another applications of foldr – it is actually more powerful than you might think at first!

Collapsing a list:

```
foldr (++) [] [ [2,3], [4,5], [6,7,8] ]
```

```
[2,3]:[4,5]:[6,7,8]:[]          [2,3]++[4,5]++[6,7,8]++[]
```






```
foldr (++) [] [ "hi ", "there ", "folks!" ] => "hi there folks!"
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

An overloaded operator is the same symbol or name, but used for more than one type of argument:

Note: there is really no difference between an "operator" and "function" – an operator IS a function, but usually is represented infix.

`2 + 4    3.4 + 5.6`   also   `* - /`

`"hi" + " there"`   (Python)

`True == False    3 /= 5`   (Haskell)

`Note that data or other syntax is sometimes overloaded`

`'hi there!'    "hi there!"`   (Python)

34 can be `Int    Integer    Float    Double`   (Haskell)

Why do we do this? Flexibility and convenience and standard math practice!

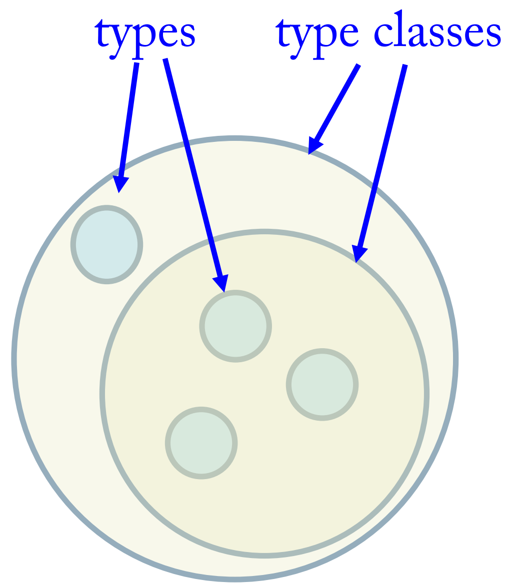
Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5
Hutton Appendix B

Recall: A type is a set of related values and its associated operators/functions.

A type class is a set of types that share some overloaded operations/functions. In specific:

- The type class is defined by a set of data objects and the set of shared operators/functions;
- A type may be a member of multiple type classes;
- A type class may be a subset of another type class



A type class is similar to an interface in Java: it defines what operations you can use with the type.

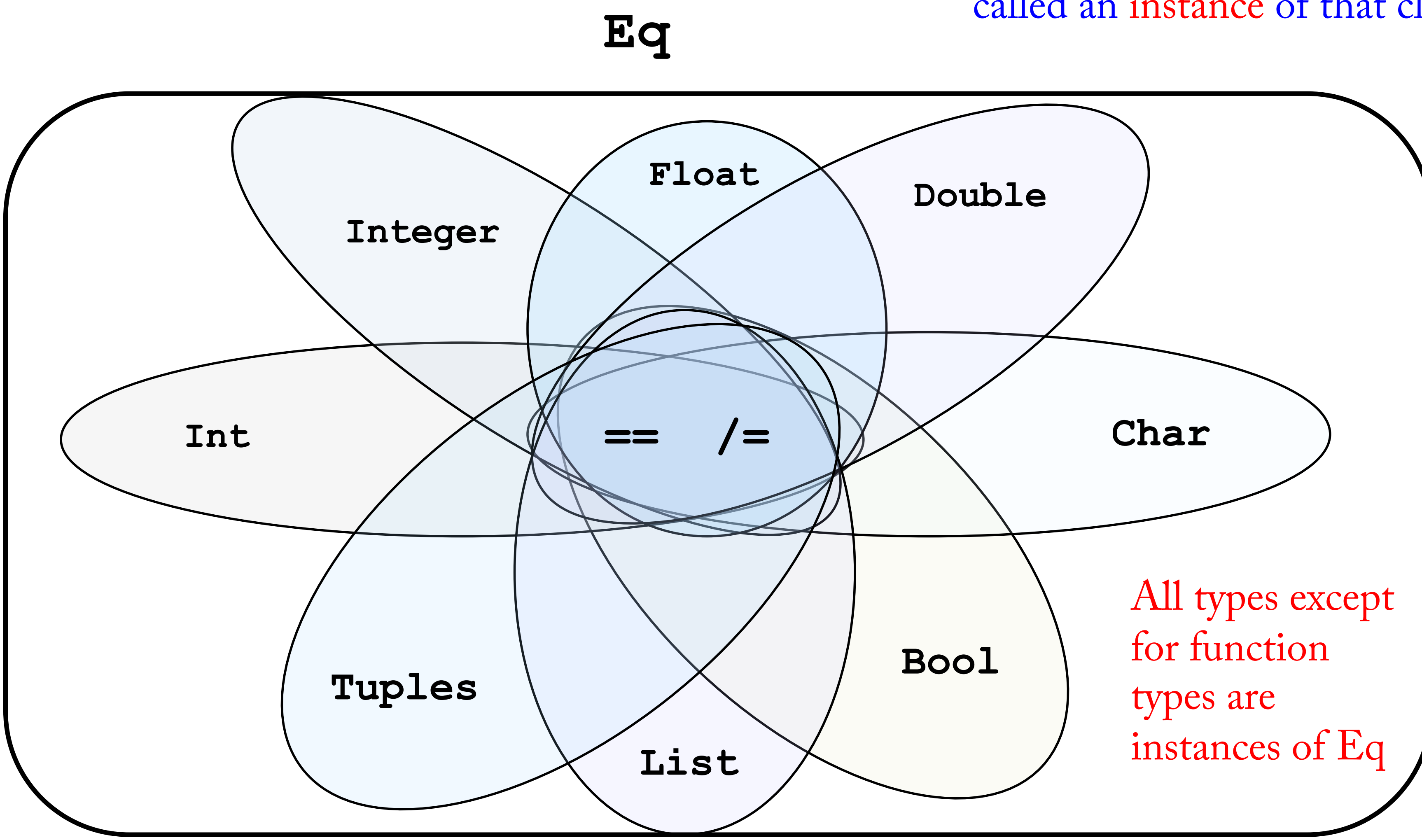
```
// Queueable Interface

public interface Queueable {
      void enqueue(int n);     // insert at the rear of the queue
      int dequeue();           // Remove and return head of queue
      int peek();              // Return head of queue without removing
      boolean isEmpty();
      int size();              // returns number of integers in queue
}
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Example: The type class **`Eq`** contains all the Equality Types, those that implement the equality operators:

A type contained in a type class is called an instance of that class.

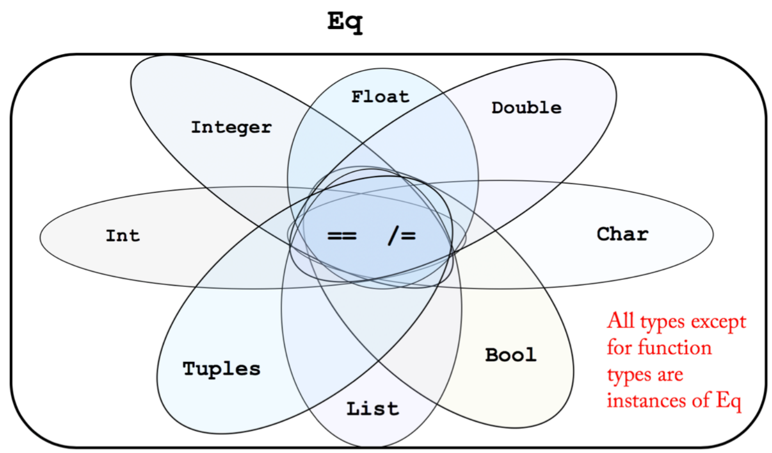


All types except for function types are instances of Eq

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

```
*Main> 5 == 6
False
*Main> 3.4 == 3.3999999999999999
False
*Main> ('a',(0,["hi","there"])) == ('a',(0,["hi","there"]))
True
*Main> [2,3,4,5] /= [3,2,4,5]
True
*Main> a = 5
*Main> b = 5
*Main> a == b
True
*Main> (+) == (+)

<interactive>:176:1: error:
    • No instance for (Eq (Integer -> Integer -> Integer))
        arising from a use of '=='
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: (+) == (+)
      In an equation for 'it': it = (+) == (+)
*Main> incr x = x + 1
*Main> :t incr
incr :: Num a => a -> a
*Main> incr == incr

<interactive>:179:1: error:
    • No instance for (Eq (Integer -> Integer))
        arising from a use of '=='
        (maybe you haven't applied a function to enou
    • In the expression: incr == incr
      In an equation for 'it': it = incr == incr
*Main>
```



Naturally, these operators are polymorphic:

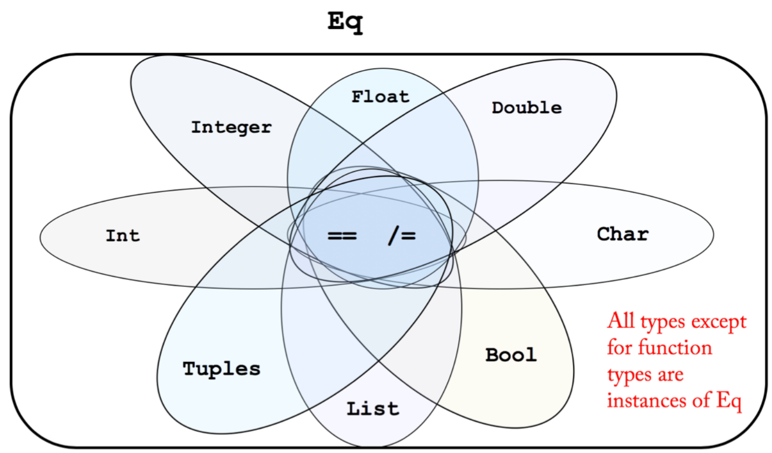
```
*Main> :t (==)
(==) :: Eq a => a -> a -> Bool
*Main> :t (/=)
(/=) :: Eq a => a -> a -> Bool
*Main>
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

Naturally, these operators are polymorphic:

```
*Main> :t (==)
(==) :: Eq a => a -> a -> Bool

*Main> :t (/=)
(/=) :: Eq a => a -> a -> Bool
*Main>
```



However, the polymorphism is restricted to types which are instances of Eq:

```
Eq a => a -> a -> Bool
```

class constraint

This says: "For any type **`a`** which is an instance of **`Eq`**, the function has type **`a -> a -> Bool`** "; any other type is forbidden.

```
<interactive>:176:1: error:
    • No instance for (Eq (Integer -> Integer -> Integer))
        arising from a use of '=='
        (maybe you haven't applied a function to enough arguments?)
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

The type class **`Ord`** is a superset of **`Eq`**, and contains those types that can be totally ordered and compared using the standard relational operators:

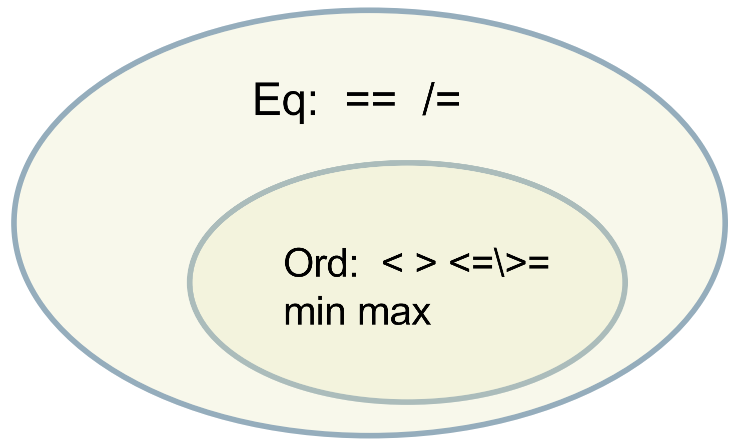
```
(<)  :: Ord a => a -> a -> Bool

(>)  :: Ord a => a -> a -> Bool

(<=)  :: Ord a => a -> a -> Bool

(<=)  :: Ord a => a -> a -> Bool

min  :: Ord a => a -> a -> a

max  :: Ord a => a -> a -> a
```

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

The type class **`Eq`** is a superset of **`Ord`**, which contains those types that can be totally ordered and compared using the standard relational operators:

Ord ⊆ Eq



Relational tests on tuples and lists is lexicographic:

```
*Main> "abc" < "abd"
True
*Main> "abc" < "abcd"
True
*Main> [2,3,4] <= [2,3,6]
True
*Main> [2,3,4] > [2,3]
True
*Main> [2,3] < [2,4,5]
True
*Main> ('a',5) < ('a',7)
True
*Main> (2,3) < (2,3,4)

<interactive>:202:9: error:
    • Couldn't match expected type '(Integer, Integer)'
                  with actual type '(Integer, Integer, Integer)'
    • In the second argument of '(<)', namely '(2, 3, 4)'
      In the expression: (2, 3) < (2, 3, 4)
      In an equation for 'it': it = (2, 3) < (2, 3, 4)
```

The ordering on lists and tuples is also recursive:

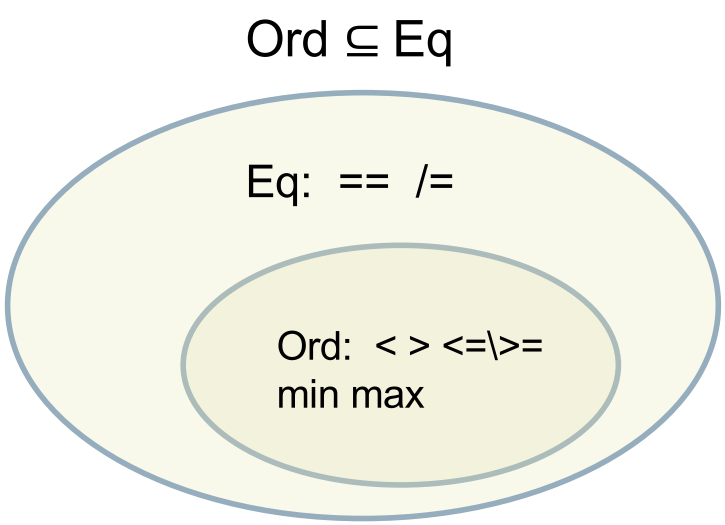
```
*Main> [ [2,3], [2,4] ] < [ [2,3], [2,5] ]
True
```

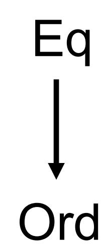
Type Classes and Overloading

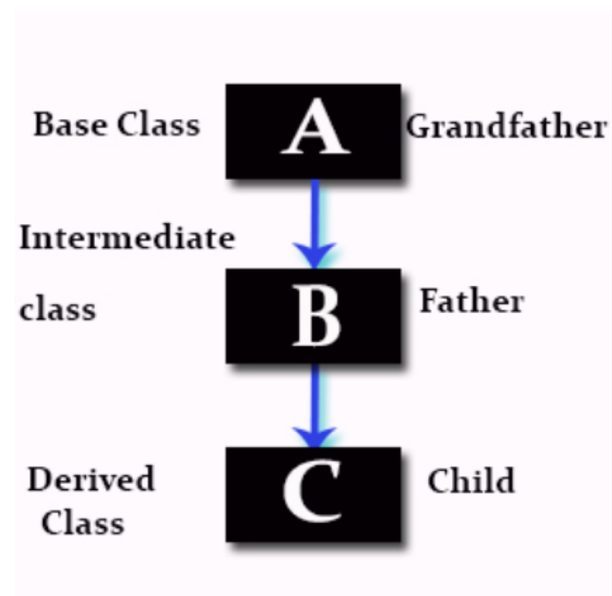
Reading: Hutton Ch. 3.8, 3.9, 8.5

The type class **`Eq`** is a superset of **`Ord`**, which contains those types that can be totally ordered and compared using the standard relational operators.

Every instance of Ord is an instance of Eq, i.e., **`Ord`** ⊆ **`Eq`**, which is similar to inheritance in Java and object-oriented languages:







Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Num` – numeric types

The **`Num`** class contains numeric values, and consists of the following overloaded operators:

```
(+)  :: Num a => a -> a -> a

(*)  :: Num a => a -> a -> a

(-)  :: Num a => a -> a -> a

negate :: Num a => a -> a

abs :: Num a => a -> a

signum :: Num a => a -> a
```

Hm...   where is division?

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Integral` – integer types

These are the instances of Num whose values are integers, and support integer division and modulus:

```
div :: Integral a => a -> a -> a

mod :: Integral a => a -> a -> a
```

```
*Main> div 5 3
1
*Main> 5 `div` 3
1
*Main> mod 10 4
2
*Main> 10 `mod` 4
2
*Main>
```

Note that mod and div are prefix functions, to turn any function into infix, use back-quotes.

Type Classes and Overloading

Reading: Hutton Ch. 3.8, 3.9, 8.5

`Fractional` – floating-point types

These are the instances of Num whose values are floating point, and support floating-point division and reciprocation:

```
(/) :: Fractional a => a -> a -> a

recip :: Fractional a => a -> a
```

```
*Main> 4.0 / 2.2
1.8181818181818181
*Main> recip 5
0.2
*Main>  4 / 2
2.0
*Main> 5 / 2
2.5
*Main> 5 / 2.2
2.2727272727272725
```

The symbols for integers are overloaded, so there is no "type-coercion" from integer to float here. The values are already fractional!