Polymorphism,
Higher order functions

Polymorphism (again)

- Types like `Integer, String, [Int], (String, Int)` are simple to interpret
  - we know exactly what they are
- We have also seen *polymorphic* types
  - `[] :: [a]`
    - empty list with unknown content
  - `length :: [a] -> Int`
    - compute length of list without caring about the actual contents
  - `map :: (a->b) -> [a] -> [b]`
    - apply a function to each element in the list
    - the *type variables* connect the first and second argument; the elements of the list argument must be of the same type that the function takes
- A polymorphic type can be said to be *partially* known
  - type variables work the same as arguments to a function
  - replace (instantiate) type variables consistently with another type
- polymorph - many shapes

Polymorphism (more)

- We have also seen types like
  - `index :: (Num t, Eq a) => a -> [a] -> t`
  - These can *not* be arbitrary types, but must fulfil additional *properties*
  - `t` must be a “number”
  - `a` must be testable for equality
- This is called *type classes* (or just classes) in Haskell
  - This is what makes *overloading* possible
  - Same operator used on different types, different implementations
- `(<) :: Ord a => a -> a -> Bool`
  - conceptually simple “less than”
  - different object compare in different ways
  - numbers
  - characters
  - strings
  - tuples
  - lists

Type Classes

- A type class is defined by functions that need to be implemented by the types that belong to the type
  - `Eq a` consists of only `(==) :: a -> a -> Bool`
- We can make a type *belong* to a type class by defining the required function(s)
- Not really relevant for the types we define now, but will be later.

Type classes are a really beautiful concept!

Abstraction (part 1)

- Polymorphic types allow us to write functions without caring about the full details of the types involved.
- Overloading with type classes allow us to use the same operator (express the same concept) with different implementations.
- These are two instances of hiding details and leveraging concepts.
- Abstraction allows us to write programs that are
  - easy to understand
  - easy to maintain (change)
  - easy to test
- More to come..

Abstraction

Higher Order Functions

- We have seen functions that can take another function as argument
  - `map :: (a->b) -> [a] -> [b]`
  - First argument is a function applied to each element in the list
- We have seen examples where we give the *name* of function
  - `map length [“lisp”, “haskell”, “erlang”, “sml”]`
- ..the *value* of `length` is a *function*..
- ..but we don’t have to name every value we use..
- ..can we write a function directly..?

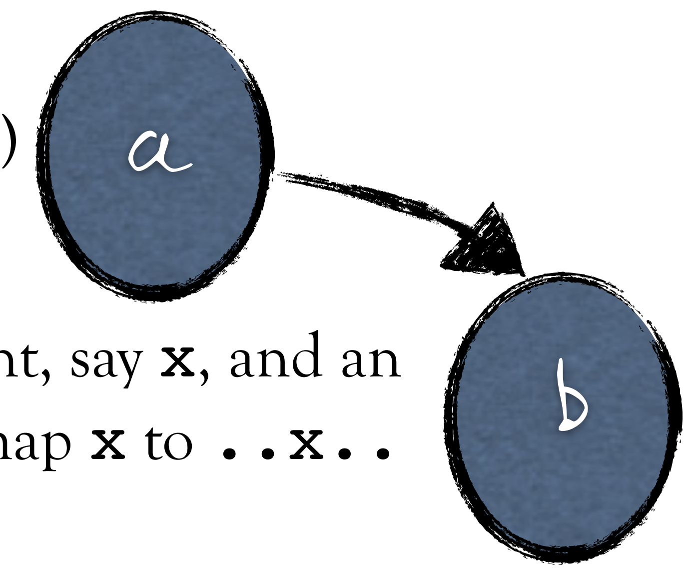
Yes!

First Class Values

- In Haskell functions are said to be *first class* values
  - equivalent in treatment to *normal* values
- A name can be bound to a function
  - `add1 x = x + 1`
- A function can be passed as an argument to a function
  - As in `map` and `filter`
- A function can be returned as the result of a computation
  - We can *construct* functions at runtime
- We can write functions without having to give them a name
  - ..ok..
- Why *first class*?
  - Unusual treatment of functions compared to other language
  - passing a function name as argument quite common
  - returning a function unusual
- Functional Programming Languages have it, and always have
  - They are first class!

Constructing Functions (pt.1)

- When using `map` it is somewhat roundabout to define a named function and then passing the name, especially when the function is “small”
- What, then, is a function?
  - A mapping from one set (of values) to another set (of values)
  - Reflected in a function type `a -> b`



- For *computing* a function (of one argument) we have an argument, say `x`, and an expression `..x..`, called a *body*, containing `x`, so we want to map `x` to `..x..`
  - ~~`x -> ..x..`~~
  - Add something to distinguish from a function type
  - `\x -> ..x..`
    - `\x -> x+1` is a function that adds 1
  - Original notation: λx.x+1
    - lambda calculus - the basis of functional programming (1931)
    - That’s all you really need!

$\boxed{\text{Anonymous function}}$

Using Anonymous Functions

- To add 1 to each element in a list
  - `map (\x->x+1) [19,31,43,59]`
- Filter numbers divisible by 7
  - `filter (\n->mod n 7==0) [1,19,47,117,35]`
- Write a function that filters numbers divisible by a given argument
  - `filterDiv :: Int -> [Int] -> [Int]`
  - Obvious to use `filter`

```
filterDiv :: Int -> [Int] -> [Int]
filterDiv n list =
  filter (\e->mod e n == 0) list
```

Use of Function Arguments

- Nothing special when using a function argument
  - define and use as normal

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

List Function Patterns

- In the example for evaluating simple expressions, we had
  - one function for *adding* all elements in a list
  - one function for *multiplying* all elements in a list
- Very similar pattern
- Differences
  - Value returned for empty list
  - Operation in inductive case
- Can we write a function that can do both?
  - Use common pattern
  - *Inject* operation and value with arguments

Value

```
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs

multlist [] = 1
multlist (x:xs) = x * multlist xs
```

Operation

```
fold :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
fold f a [] = a
fold f a (x:xs) = f x (fold f a xs)
```

Usage

```
fold :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
fold f a [] = a
fold f a (x:xs) = f x (fold f a xs)
```

```
sumlist list = fold (+) 0 list

multlist list = fold (*) 1 list

— factorial in another way
fac n = multlist [1..n]
```

- Note: `+` and `*` *commutative*
  - `a+b=b+a` and `a*b=b*a`
- `fold (+) 0 [1,2,3]` is
  - `1 + (2 + (3 + 0)) = 6`
- `-` is *not* commutative
- `fold (-) 0 [1,2,3]` is
  - `1 - (2 - (3 - 0)) = 2`
- Alternatives:
  - `((1 - 2) - 3) - 0) = -5`
  - `((0 - 1) - 2) - 3) = -6`
- Second is a reasonable symmetric alternative
- Two definitions of `fold`

Variants of fold

```
– fold from the right
foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)

– fold from the left
foldl :: (t1 -> t2 -> t1) -> t1 -> [t2] -> t1
foldl f a [] = a
foldl f a (x:xs) = (foldl f (f a x) xs)
```

- Both exist in Haskell
- Note different types of function argument
  - `foldr`: first argument is the element in the list, second is the value
  - `foldl`: first argument is the value, second is the element in the list
- .. and note that the function argument takes *different* types
  - we can do more than just numeric operations
- `foldl` is an abstraction for list recursion with an accumulator

Abstraction (part 2)

- With higher order list operations like `map`, `filter`, `foldl` and `foldr` we
  - hide the details of the list recursion
  - focus on the important part, i.e., the function being passed as argument
- A program using these will be
  - more compact
  - easier to read
  - easier to maintain
  - less cluttered will common patterns
- Side note:
  - `map` and `filter` can be defined in terms of a fold