Functional programming

Higher order functions

But first… tuples

```
λ: [1, 2, 3]
=> [1,2,3]
```

a list

same type

```
λ: ("One", 1)
=> ("One",1)
```

a tuple

different types

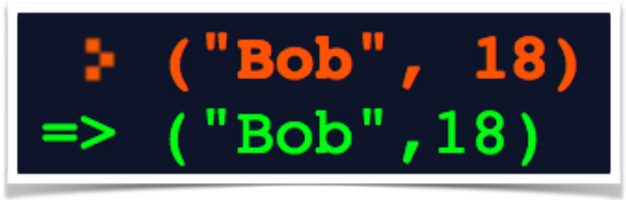
```
λ: [("One", 1), ("Two", 2)]
=> [("One",1),("Two",2)]
```

list of tuples

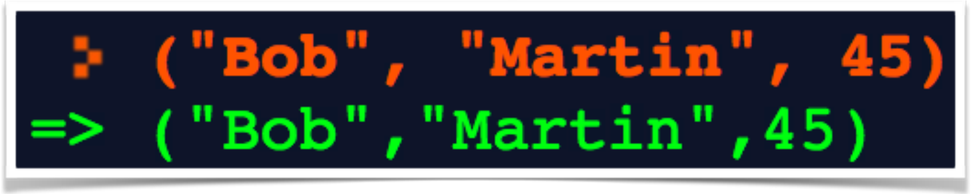
```
λ: ("ABC", [1,2,3])
=> ("ABC",[1,2,3])
```

tuple of lists

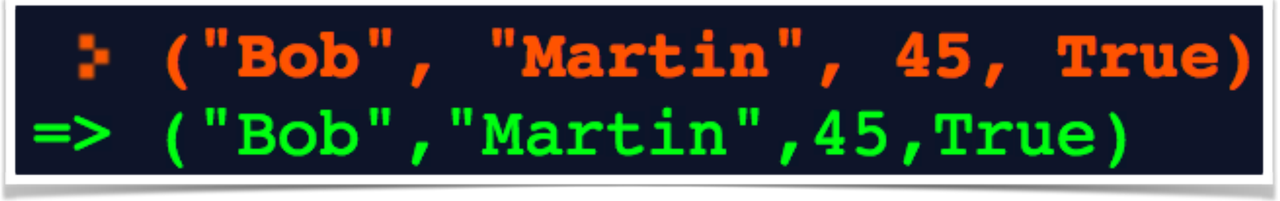
Tuples



a pair



a triple



a 4-tuple

Tuples. Elements.

```
λ> ("Bob", 18)
=> ("Bob",18)
```

a pair

```
λ> fst ("Bob", 18)
=> "Bob"
```

```
λ> snd ("Bob", 18)
=> 18
```

```
fst ("Bob", "Martin", 45)
```

?

```
first :: (a, b, c) -> a
first (a, _, _) = a
```

Zip into tuples

- Zip is function that takes two lists and produces list of pairs

```
λ: zip [1..10] ["Ann", "Bob", "Igor"]
=> [(1,"Ann"),(2,"Bob"),(3,"Igor")]
```

```
zip' [] _ = []
zip' _ [] = []
zip' (a:as) (b:bs) = [(a, b)] ++ zip' as bs
```

Function types

- Let’s make a simple function
- Check our function type
- What does it mean?
- Or…?

```
sum' a b = a + b
```

```
> :t sum'
sum' :: Num a => a -> a -> a
```

a is Num arguments: **a**, **a** returns: **a**

Returning a function

```
sum' a b = a + b
```

```
λ> :t sum'
sum' :: Num a => a -> a -> a
```

- Function gets **a** as an argument
- and returns a function, that takes **a** as an argument
- and returns **a**

Function currying

```
f :: a -> b -> c
```

curried

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

not curried


```
f = curry g
g = uncurry f
```

```
λ> :t sum
sum :: Num a => a -> a -> a
λ> :t (uncurry sum)
(uncurry sum) :: Num c => (c, c) -> c
```

Returning a function. Examples.


```
mult' a b = a * b
double a = mult' 2 a
```

```
λ> mult' 2 3
=> 6
λ> double 3
=> 6
```

```
half = (/2)
```

```
λ> half 5
=> 2.5
```

```
isDigit = (`elem` ['0'..'9'])
```

```
λ> isDigit '3'
=> True
λ> isDigit 'A'
=> False
```

Function as an argument


```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
λ: applyTwice (+2) 1
=> 5
λ: applyTwice (++ "I") "V"
=> "VII"
```

Some really really useful functions

function	arguments	description	example
zipWith	function and two lists	joins lists by applying function between elements	`> zipWith (+) [1,2,3] [2,2,1]` `=> [3,4,4]`
flip	function and two arguments	flips arguments for the function	`> flip (++) "A" "B"` `=> "BA"`
map	function and a list	applies function to each element and returns a new list	`> map (*2) [1..5]` `=> [2,4,6,8,10]`
filter	predicate and a list	returns a list where elements satisfy the predicate	`> filter odd [1..10]` `=> [1,3,5,7,9]`

Some really really useful functions

function	arguments	description	example
zipWith	function and two lists	joins lists by applying function between elements	`> zipWith (+) [1,2,3] [2,2,1]` `=> [3,4,4]`
flip	function and two arguments	flips arguments for the function	`> flip (++) "A" "B"` `=> "BA"`
map	function and a list	applies function to each element and returns a new list	`> map (*2) [1..5]` `=> [2,4,6,8,10]`
filter	predicate and a list	returns a list where elements satisfy the predicate	`> filter odd [1..10]` `=> [1,3,5,7,9]`

Lambdas

- A lambda abstraction is another name for an anonymous function. It gets its name from the usual notation for writing it: for example, $\lambda x \rightarrow x^2$
- In Haskell lambdas are written as:
    - \ x -> x * x
- Lambdas are usually used when function is required once

Lambdas


```
λ> filter (\x -> x > 2 && even x) [1..10]
=> [4,6,8,10]
```

```
λ> map (\e -> e * e + e) [1..10]
=> [2,6,12,20,30,42,56,72,90,110]
```

```
λ> zipWith (\a b -> take b (repeat a))
"HELLO" [1..10]
=> ["H","EE","LLL","LLLL","OOOOO"]
```

Challenges

- Implement own versions of:
  - zipWith
  - flip
  - map
  - filter