Getting Funcy about Programming

Giles Reger

November 5, 2012

Funcy Programming

1. Lists
2. Higher Order Functions

Lists of lists of lists

- From history : Lisp = LISt Processing
- Prefix notation

```
(+ 1 2 3 4)
```

```
(+ (* 10 10) (- 400 70) 7)
```

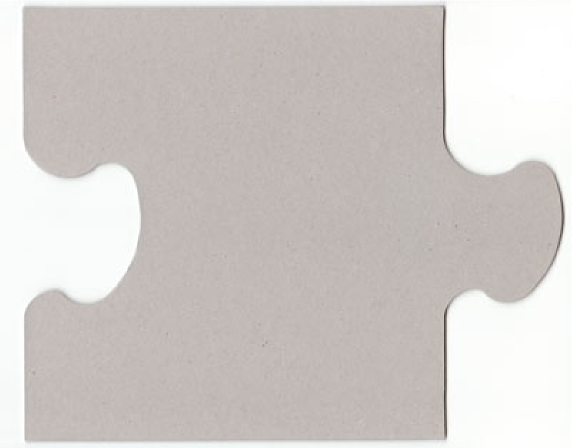
- Commands and data are lists
- Elegant? perhaps
- Easy to write... perhaps not
- But this idea of manipulating lists is important

Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy

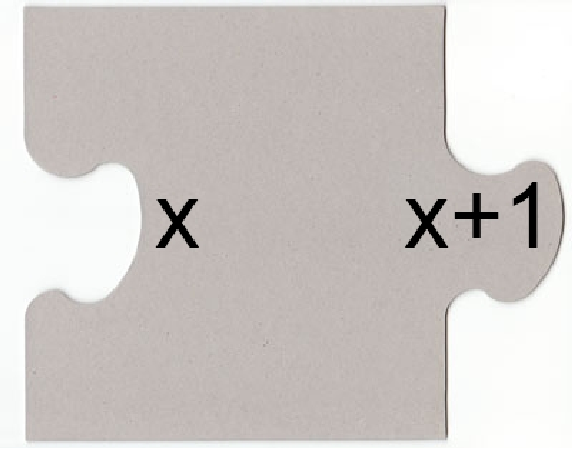
Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy



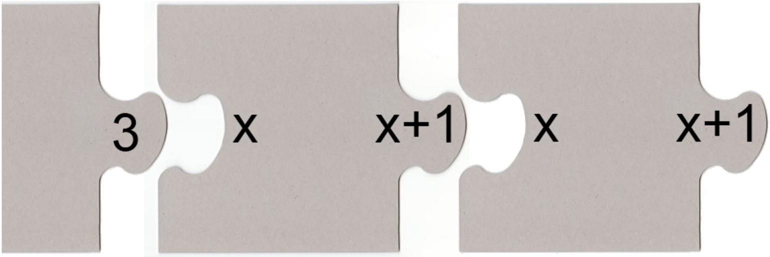
Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy



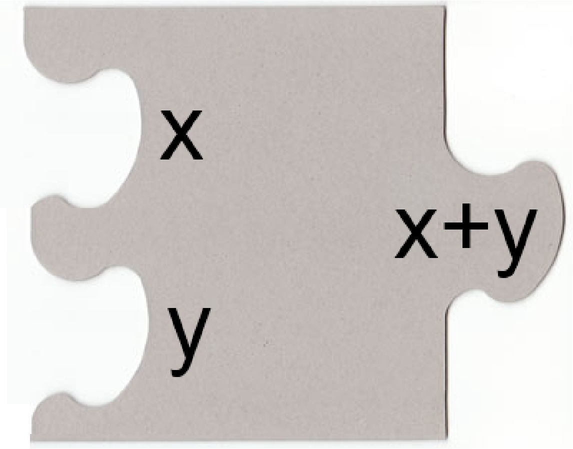
Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy



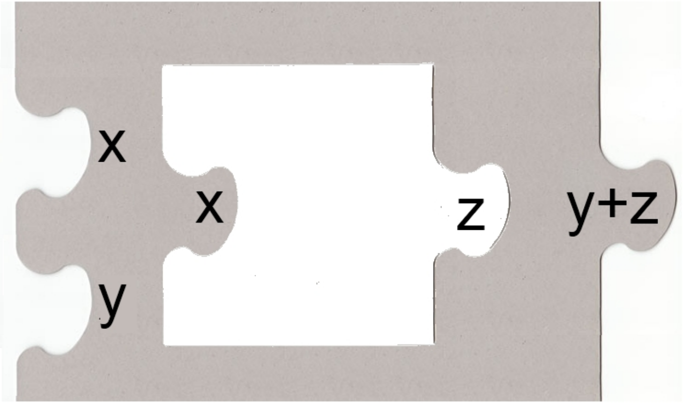
Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy



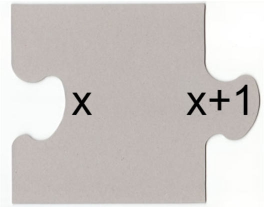
Functions

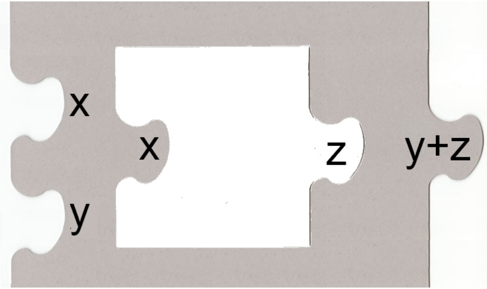
- Not necessarily, but often *side-effect* free
- Jisgaw analogy
- Higher Order Functions



Functions

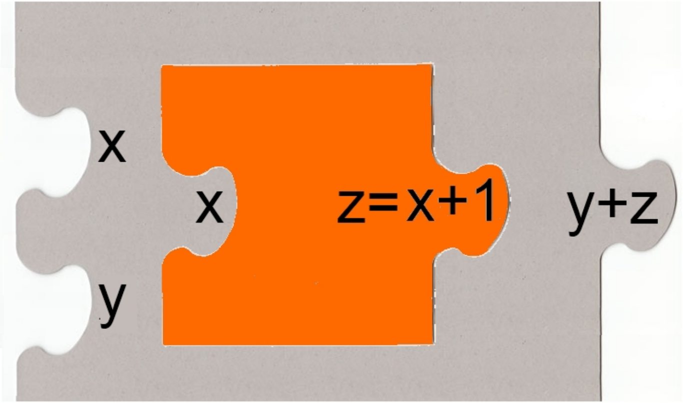
- Not necessarily, but often *side-effect* free
- Jisgaw analogy
- Higher Order Functions





Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy
- Higher Order Functions



Functions

- Not necessarily, but often *side-effect* free
- Jisgaw analogy
- Higher Order Functions

- We treat functions as values
- The jigsaw analogy quickly falls apart.... recursion

Lists + Functions

- A big concept in Functional Programming is applying functions to lists
- The most famous of these is `map`
- This takes a function and applies it to every element of a list

```
 map f over [1,2,3,4]

[ f(1), f(2), f(3), f(4)]
```

Writing Map

Scala

```
def map[S,T](list:List[S],f:S=>T) : List[T] =
list match {
      case List() => List()
      case x::xs  => f(x)::map(xs,f)
}
```

OCaml

```
let rec map f l =
  match l with
    [] -> []
  | hd :: tl => f hd :: map f tl
```

Using Map

Python	`map(lambda x:  x*x, [1,2,3,4])`
Scala	`List(1,2,3,4) map (x => x*x)`
Java 8	
Haskell	`map (\x -> x*x) [1,2,3,4]`
OCaml	`List.map (fun x-> x*x) [1;2;3;4]`
Ruby	
Erlang	
C#	

Reduce

- `map` allows us to transform a list
- `reduce` allows us to collapse a list into a single value

```
reduce f 0 [1,2,3,4]
```

```
f( 0, f( 1, f( 2, f( 3, 4 ) ) ) )
```

Reduce

Python	`reduce(operator.add,[1,2,3,4],0)`
Scala	`List(1,2,3,4).foldLeft(0)(_ + _)`
Java 8	
Haskell	`fold (+) [1,2,3,4]`
OCaml	`List.fold_left (fun x y -> x+y) [1;2;3;4]`
Ruby	
Erlang	
C#	

Project Euler in a Funcy way (Scala)

- Add all the natural numbers below one thousand that are multiples of 3 or 5

```
(1 to 999).filter(x => (x%3 ==0) || (x%5==0))
          .foldLeft(0)(_+_)}
```

- By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

```
def expand_fibs(list:List[Long]):List[Long] = {
  if(list.head > 4000000) list
  else
  expand_fibs((list.head+list.tail.head)::list)
}
expand_fibs(List(1,0)).filter(_%2==0)
                      .foldLeft(0L)(_+_)
```

MapReduce

- Approach to concurrency
- Originated in Google
- Idea:
    - Split list up into lots of small lists
    - carry out lots of maps in parallel to produce a new list
    - reduce that list
- A bit more complicated than that - look it up
- Hadoop

Lambda Calculus

- If you're mathematically minded then look at this
- Loads of cool stuff
  - Type theory
  - Constructive logic
  - Programming language design
  - Computation models
  - Concurrency models