Functional programming

Svarny Petr

Katedra logiky FF UK

6. dubna 2021

Overview

- Functional programming
- Lambda calculus
- Python
- Haskell
- Clojure - Read-Eval-Print-Loop
- Racket

Functional programming - languages

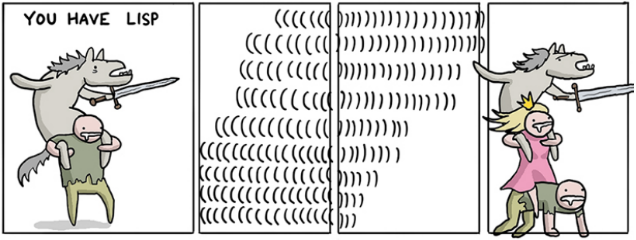
- =Lisp=,
- =Clojure=,
- =Scala=,
- =Haskell=...

Functional programming ur-father

Functional programming ur-father

Language **I**ntended for **S**mart **P**eople
LISt **P**rocessing
Lost **I**n **S**imple **P**arentheses
Lots of **I**nsidious **S**illy **P**arenthesis

Functional programming ur-father



=Princess comics=

Functional programming ur-father

Functional programming - benefits

- Higher-order functions (i.e., functions on functions),
- Lazy evaluation (i.e., evaluating data on need),
- **No state**, immutable data:
  - no side-effects, less errors,
  - no mutable states - efficient parallel computing.
- Opposite to OOP - don't bundle data and functions.

Functional programming - brain hurts

- No state,
- No usual flow control (e.g., if) - control only by function calls,
- Iteration with recursion (not loops).

Lambda sources

=LearnXinY=
=Online Calculus=
=UC Berkeley Online=
=lambdacalc.io=

Getting to abstraction

```
replace x and y
    in _ y x _
    by hello and world

>>> _ world hello _
```

Getting to abstraction

```
replace x and y
    in _ y x _
    by hello and world
```

```
((replace x & y)
    in _ y x _)
    by hello & world
```

Getting to abstraction

`((replace x & y) in _ y x _) by hello & world`

Getting to abstraction

```
((replace x & y) in _ y x _) by hello & world

((replace x & y) _ y x _) hello & world
```

Getting to abstraction

```
((replace x & y) in _ y x _) by hello & world

((replace x & y) _ y x _) hello & world

((replace x, y) _ y x _) hello , world
```

Getting to abstraction

```
((replace x & y) in _ y x _) by hello & world

((replace x & y) _ y x _) hello & world

((replace x, y) _ y x _) hello, world

((lambda x, y) _ y x _) hello, world
```

Getting to abstraction

```
((replace x & y) in _ y x _) by hello & world

((replace x & y) _ y x _) hello & world

((replace x, y) _ y x _) hello, world

((lambda x, y) _ y x _) hello, world

(lambda xy._ y x _) hello, world

(lambda x.x)y
```

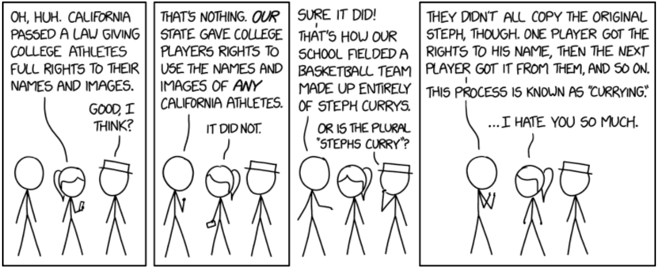
Lambda Currying

$$(\lambda xyz.xyz) = (\lambda x.\lambda y.\lambda z.xyz)$$

Functional programming - Currying

- i.e., multi-arity function turned into sequence of unary functions
- $f(x, y, z) \rightarrow g(x), h(y), i(z)$
- =Haskell currying=

Functional programming - Currying



=XKCD WTF=

Functional Python

=Python Docs=
=RealPython=

Python lambda

```
lambda s: s[::-1]
```

Python lambda

```
lambda s: s[::-1]

(lambda x1, x2: (x1 + x2)/ 3)(9, 6)
```

Python lambda

```
lambda s: s[::-1]

(lambda x1, x2: (x1 + x2)/3)(9, 6)

my_function = lambda x: x + 2
my_function(2)
```

Lazy evaluation in Python

```
>>> import sys
>>> sys.getsizeof([0, 1, 2, 3, 4])
104
>>> sys.getsizeof(range(5))
??
```

Lazy evaluation in Python

```
>>> import sys
>>> sys.getsizeof([0, 1, 2, 3, 4])
104
>>> sys.getsizeof(range(5))
48
```

Lazy evaluation in Python

```
>>> import sys
>>> sys.getsizeof([0, 1, 2, 3, 4])
104
>>> sys.getsizeof(range(5))
48
>>> sys.getsizeof(range(500))
??
```

Lazy evaluation in Python

```
>>> import sys
>>> sys.getsizeof([0, 1, 2, 3, 4])
104
>>> sys.getsizeof(range(5))
48
>>> sys.getsizeof(range(500))
48
```

Lazy evaluation in Python

- generators $\subset$ iterators
- __next__, __iter__ methods for iterators, see =Programiz=
- yield instead of return, see =RealPython=

Haskell reasons

=10 reasons for Haskell=

Haskell

- functional language,
- strong static typed,
- lazy evaluation,
- lambda expressions,
- list comprehension,
- type polymorphism,
- ...

Py vs Haskell

```
PY:
xs = [1, 2, 3, 4, 5]
xs_ = map(lambda x : x + 1, xs)

Haskell:
xs = [1, 2, 3, 4, 5]
xs' = map (+1) xs
```

Haskell tutorial

=Learn Haskell!=
=Slant: Python vs Haskell=
=Haskell review=

```
'lambda' x : x + x
doubleMe x = x + x
doubleMe 8
> 16
```

Haskell - Function definition

```
doubleUs x y = x*2 + y*2
```

Haskell - Type definition

```haskell
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Haskell - Recursive function

```haskell
-- Integral = Int and not bounded Integer
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Haskell - Guards

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b      = GT
    | a == b     = EQ
    | otherwise  = LT

3 `myCompare` 2
> GT
```

Haskell - Bindings

```
4 * (let a = 9 in a + 1) + 2

> ?
```

Haskell - Bindings

```
4 * (let a = 9 in a + 1) + 2
> 42
```

Haskell - Conditional example

```
doubleSmallNumber x = if x > 100
                        then x
                        else x*2

doubleSmallNumber' x = (if x > 100 then x else x*2)
```

Haskell - list definition

```
let listNumbers = [1,2,3,4]

sum listNumbers
> 10

4 `elem` [3,4,5,6]
> True
```

Haskell - list comprehension

```
[ x | x <- [50..100], x `mod` 7 == 3]
> [52,59,66,73,80,87,94]
```

Haskell - list comprehension

```haskell
mapConcurrently :: (a -> IO b) -> [a] -> IO [b]
  -- specialised to lists

main =
  mapConcurrently putStrLn
    [ "hello",
      "this",
      "is",
      "concurrent" ]

-- async processing each in own thread
```

Haskell - Polymorphism

```
data List a = Nil | Cons a (List a)

length :: List a -> Integer
length Nil          = 0
length (Cons x xs) = 1 + length xs

map :: (a -> b) -> List a -> List b
map f Nil          = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Clojure sources

=Clojure io=
=LearnXinY=
=Clojure for the brave!=
=Clojure koans=

Clojure and REPL

=REPL=
Dynamic compilation of each line into JVM bytecode
Option for ahead-of-time compilation

Clojure - basic elements

```
(1 2 3 4) ; list
[1 2 3 4] ; vector
{:name "Tom", :age 5} ; map
#{1 2 3} ; set
```

Clojure - function

```
(+ 1 2) ; basically a list... LISP legacy
```

Clojure - function

```
(defn do-something
    "This function does something."
    [x]
    "Something")
```

Clojure - anonymous functions

```
(fn [coll] (filter even? coll))
```

Clojure - Multiple arity

```
(defn do-something
  ([] "nothing")
  ([one] "easy!")
  ([one two] "hm, ok, will do")
  ([one two & more] "oh, no, so many!"))
```

Clojure - Higher order function

```
(defn concat-some
  [f vec1 vec2]
  ((fn [x] (filter f x))
   (concat vec1 vec2)))
>(concat-some even? [1 2 3] [4 5 6])
```

Clojure - For loop

```
(for [x (range 10 15)]
  (str "|" x "|"))
```

Clojure - For options

```
(for [i (range 1 10)
      :when (even? i)
      :let [inverse (/ 1 i)]]
  [i inverse])
```

Clojure - Polymorphism

```
(defmulti encounter (fn [x y] [(:Species x)
                              (:Species y)]))
(defmethod encounter [:Bunny :Lion] [b l] :run)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)
(def b1 {:Species :Bunny :other :stuff})
(def b2 {:Species :Bunny :other :stuff})
(def l1 {:Species :Lion :other :stuff})
(def l2 {:Species :Lion :other :stuff})
(encounter b1 b2)
```

Racket

Based on Scheme dialect of LISP, thus functional origins.
=Racket lang=
=Teach Yourself Racket=

Racket - basic elements

```
#b111 ; binary
1/2 ; rationals
(exact->inexact 1/3) ; => 0.3333333333333333
#t ; true
#f ; false
"Hello, world!"
```

Racket - lists

```
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3 4)
(quote (1 2 3))
'(1 2 3)

; be careful with the quotes
`(1 ,(+ 1 1) 3)
```

Racket - others

```
#(1 2 3) ; vector a.k.a. "tuple"
(set 1 2 3) ; set
(define m (hash 'a 1 'b 2 'c 3)) ; hash-table

(list->set '(1 2 3 1 2 3 3 2 1 3 2 1))
```

Racket - lambda function

```
(lambda () "Hello World")
(l () "Hello World") ; imagine unicode lambda

(define hello-world (lambda () "Hello World"))

(define (hello-world) "Hello World") ; shortening o
```

Racket - input for functions

```
(define hello
  (lambda (name)
    (string-append "Hello " name)))
(hello "World")

(define (hello2 name)
  (string-append "Hello " name))
```

Racket - multi-variadic functions

```
(define hello
  (case-lambda
    [() "Hello World"]
    [(name) (string-append "Hello " name)]))

; default arguments
(define (hello2 [name "World"])
  (string-append "Hello " name))
```

Racket - Conditions

```
(if #t                  ; test expression
    "this is true"      ; then expression
    "this is false")    ; else expression

(cond [(> 2 2) (error "wrong!")]
      [(< 2 2) (error "wrong again!")]
      [else 'ok])
```

Racket - For loop

```
(define (loop i)
  (when (< i 10)
    (printf "i=~a\n" i)
    (loop (add1 i)))) ; tail recursion

; in-built version
(for ([i 10])
  (printf "i=~a\n" i))
```

Racket - For options

```
(for [i (range 1 10)
        #:when (even? i)
        #:let [inverse (/ 1 i)]]
  [i inverse])
```

Racket - While macro

```
(define-syntax-rule (while condition body ...)
  (let loop ()
    (when condition
      body ...
      (loop))))

(let ([i 0])
  (while (< i 10)
    (displayln i)
    (set! i (add1 i))))
```

Racket -Contracts

```
(module bank-account racket
  (provide (contract-out
            [deposit (-> positive? any)] ; amounts
            [balance (-> positive?)]))

  (define amount 0)
  (define (deposit a) (set! amount (+ amount a)))
  (define (balance) amount)
  )

(require 'bank-account)
(deposit 5)

(balance) ; => 5

;; (deposit -5) ; => deposit: contract violation
```

Bonus

=Fermats library - Dijkstra recursive programming=