# Functional Programming in LISP

Niko Laaksonen

January 21, 2015

# What is Functional Programming?

- Functional programming is a more *mathematical* approach to programming

# What is Functional Programming?

- Functional programming is a more *mathematical* approach to programming
- Functions are first-class objects (treated as any other variable)

# What is Functional Programming?

- Functional programming is a more *mathematical* approach to programming
- Functions are first-class objects (treated as any other variable)
- In practice, our programs should be stateless

# What is Functional Programming?

- Functional programming is a more *mathematical* approach to programming
- Functions are first-class objects (treated as any other variable)
- In practice, our programs should be stateless
- Data should be immutable

# Scheme

- Scheme is a dialect of LISP, which is one of the oldest programming languages still in use

# Scheme

- Scheme is a dialect of LISP, which is one of the oldest programming languages still in use

**Greenspun's Tenth Rule**

Any sufficiently complicated C or Fortran program contains a bug-ridden implementation of half of Common Lisp.

## Scheme

- Scheme is a dialect of LISP, which is one of the oldest programming languages still in use

**Greenspun's Tenth Rule**

Any sufficiently complicated C or Fortran program contains a bug-ridden implementation of half of Common Lisp.

- It has a very simple syntax which is based entirely on lists

```
> (+ 1 2)
3
> (* (+ 2 3) 5)
25
```

## Variables and Functions

- We assign variables with the **define** keyword:

```
> (define x 5)
> (* x x)
25
```

## Variables and Functions

- We assign variables with the **define** keyword:

```
> (define x 5)
> (* x x)
25
```

- Similarly, we can define functions:

```
> (define (square y) (* y y))
> (square 3)
9
```

## Variables and Functions

- We assign variables with the **define** keyword:

```
> (define x 5)
> (* x x)
25
```

- Similarly, we can define functions:

```
> (define (square y) (* y y))
> (square 3)
9
```

- The substitution model for evaluation

# Logic

- Finally, we need conditionals (if, then, else)

```
(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (else (- x))))
```

# Logic

- Finally, we need conditionals (if, then, else)

```
(define (abs x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (else (- x))))
```

- Logical operands work as you'd expect:

```
> (and (> 3 0) (not (= 1 2)))
#t
> (or (< 1 0) (> 2 2))
#f
```

## Recursion and Loops

- LISP doesn't have **for**, **while** etc.

## Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself

## Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself
- Many types of recursive processes: linear recursion, iterative recursion

## Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself
- Many types of recursive processes: linear recursion, iterative recursion

```
(define (factorial n)
    (if (= n 0) 1
    (* n (factorial (- n 1)))))

>(factorial 3)
6
```

# Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself
- Many types of recursive processes: linear recursion, iterative recursion

```
(define (factorial n)
    (if (= n 0) 1
    (* n (factorial (- n 1)))))

>(factorial 3)
6

(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

# Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself
- Many types of recursive processes: linear recursion, iterative recursion

```
(define (factorial n)
    (if (= n 0) 1
    (* n (factorial (- n 1)))))

>(factorial 3)
6

(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

```
(define (factorial n)
    (fact-iter 1 1 n))
(define (fact-iter product step max-count
    (if (> step max-count)
        product
        (fact-iter (* step product)
                   (+ step 1)
                   max-count)))
```

## Recursion and Loops

- LISP doesn't have **for**, **while** etc.
- Recursive function is one that calls itself
- Many types of recursive processes: linear recursion, iterative recursion

```
(define (factorial n)
    (if (= n 0) 1
    (* n (factorial (- n 1)))))

>(factorial 3)
6

(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

```
(define (factorial n)
    (fact-iter 1 1 n))
(define (fact-iter product step max-count
    (if (> step max-count)
        product
        (fact-iter (* step product)
                   (+ step 1)
                   max-count)))
(factorial 3)
(fact-iter 1 1 3)
(fact-iter 1 2 3)
(fact-iter 2 3 3)
(fact-iter 6 4 3)
6
```

# Tree Recursion

- How to compute Fibonacci numbers?

## Tree Recursion

- How to compute Fibonacci numbers?

```
(define (fib n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (else (+ (fib (- n 1)
                   (fib (- n 2)))))))

>(fib 5)
8
```

- The process generated by this function looks like a tree!

# Tree Recursion

- How to compute Fibonacci numbers?

```
(define (fib n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (else (+ (fib (- n 1)
                   (fib (- n 2)))))))


>(fib 5)
8
```

- The process generated by this function looks like a tree!
- Write an iterative version of this!

## Counting Change

- Consider the following problem: In how many ways can you make change of £$x$ amount of money by using standard denominations (50p, 20p, 10p, 5p, 2p, 1p)

## Counting Change

- Consider the following problem: In how many ways can you make change of £$x$ amount of money by using standard denominations (50p, 20p, 10p, 5p, 2p, 1p)
- Key observation: The number of ways to change £$x$ with $n$ coins is equal to

$$\text{(number of ways to change £}x\text{ using all but the first coin)}$$

$$+\text{(number of ways to change £}x - d\text{ using all }n\text{ coins)}$$

## Counting Change

- Consider the following problem: In how many ways can you make change of £$x$ amount of money by using standard denominations (50p, 20p, 10p, 5p, 2p, 1p)
- Key observation: The number of ways to change £$x$ with $n$ coins is equal to

$$\text{(number of ways to change £}x\text{ using all but the first coin)}$$

$$+\text{(number of ways to change £}x - d\text{ using all }n\text{ coins)}$$

- $d$ is the denomination of the removed coin

## Counting Change

- Consider the following problem: In how many ways can you make change of £$x$ amount of money by using standard denominations (50p, 20p, 10p, 5p, 2p, 1p)
- Key observation: The number of ways to change £$x$ with $n$ coins is equal to

$$\text{(number of ways to change £}x\text{ using all but the first coin)}$$

$$+\text{(number of ways to change £}x - d\text{ using all }n\text{ coins)}$$

- $d$ is the denomination of the removed coin
- Trivial cases: if $x = 0$ then there is 1 way, if $x < 0$ or $n < 0$ then there is 0 ways to make change.

## Counting Change

- Consider the following problem: In how many ways can you make change of £$x$ amount of money by using standard denominations (50p, 20p, 10p, 5p, 2p, 1p)
- Key observation: The number of ways to change £$x$ with $n$ coins is equal to

$$(\text{number of ways to change £}x\text{ using all but the first coin})$$

$$+(\text{number of ways to change £}x - d\text{ using all }n\text{ coins})$$

- $d$ is the denomination of the removed coin
- Trivial cases: if $x = 0$ then there is 1 way, if $x < 0$ or $n < 0$ then there is 0 ways to make change.
- It is difficult to turn this into an iterative process!

## Counting Change (cont.)

```
(define (count-change x) (cc x 6))
(define (cc x kinds-of-coins)
    (cond ((= x 0) 1)
          ((or (< x 0) (= kinds-of-coins 0)) 0)
          (else (+ (cc x
                       (- kinds-of-coins 1))
                   (cc (- x (first-coin kinds-of-coins))
                       kinds-of-coins)))))
(define (first-coin n)
    (cond ((= n 1) 1)
          ((= n 2) 2)
          ((= n 3) 5)
          ((= n 4) 10)
          ((= n 5) 20)
          ((= n 6) 50)))
```

## Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.

# Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.
- Write a program to compute the sum of integers from $a$ to $b$.

# Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.
- Write a program to compute the sum of integers from $a$ to $b$.
- Extend the counting change program to include one and two pound coins.

## Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.
- Write a program to compute the sum of integers from $a$ to $b$.
- Extend the counting change program to include one and two pound coins.
- Can you make it to an iterative process (difficult!)

## Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.
- Write a program to compute the sum of integers from $a$ to $b$.
- Extend the counting change program to include one and two pound coins.
- Can you make it to an iterative process (difficult!)
- Write a recursive process to compute the binomial coefficient $\binom{n}{k}$ (Hint: think of Pascal's triangle)

# Exercises

- Write a program to compute the sum of squares of the largest two of given three integers.
- Write a program to compute the sum of integers from $a$ to $b$.
- Extend the counting change program to include one and two pound coins.
- Can you make it to an iterative process (difficult!)
- Write a recursive process to compute the binomial coefficient $\binom{n}{k}$ (Hint: think of Pascal's triangle)
- **References:** Abelson and Sussman, *Structure and Interpretation of Computer Programs* (SICP)