



## EDAF40/EDAN40: Functional Programming Introduction

Jacek Malec

Dept. of Computer Science, Lund University, Sweden

March 20th, 2017



## Administrativa

- Standard notification: 140/200h total compared with 20/28h with me + 6 with TAs.
- Self-learning period in the beginning (syntax, basics).
- Two/Three not too tough programming assignments. (15, 10, 6 hrs)
- Kursombud (course representative) must be chosen. Today!
- Programming assignments verified by machine and then teaching assistants (Christian Söderberg and Sten Gestegård Robertz).
- Any problems (deadlines?) – please discuss **IN ADVANCE** with me!
- Slides based a lot on Lennart Andersson and Lennart Ohlsson's material. Thank you.



## Textbooks

- 1 Graham Hutton, *Programming in Haskell*, Cambridge University Press, 2007, ISBN 978-0-521-69269-4
- 2 Bryan O'Sullivan, Don Stewart, and John Goerzen, *Real World Haskell*, O'Reilly Media, 2008, ISBN 0-596-51498-0
- 3 Miran Lipovača, *Learn You a Haskell for Great Good!*, No Starch Press, 2011, ISBN 1-59327-283-9
- 4 Simon Thompson, *Haskell - The Craft of Functional Programming*, 3rd edition, Addison-Wesley 2011, ISBN 0-201-88295-7



## Software

- Glasgow Haskell Compiler, or `ghc`
- Interpreter is called `ghci`
- Currently in its version 7.10.3. (@ [login.student.lth.se](http://login.student.lth.se)), or higher
- \*.student.lth.se all run this version (please report issues)
- consider installing haskell-stack environment on your machine (<http://haskellstack.org>)



## Suggestions

- Read the assignment completely before you begin coding;
- Read the assignment text **after** the official announcement date;
- Complain to me or to a course student representative, if something does not work or is unclear;
- Check the course web;
- Do not mail `fp@cs.lth.se` unless you are filing in a **working** solution to an assignment;
- Do not mail `edan40@cs.lth.se` if you want to contact a human;
- Plan your time!
- Use our time (JM Mo 15.30-16.30, CS ..., SGR ...)!



## What is functional programming?

*“Functional programming is so called because a program consists entirely of functions. [...] These functions are much like ordinary mathematical functions [...] defined by ordinary equations.”*

(John Hughes)



## A function

Let  $A$  and  $B$  be arbitrary sets.

Any subset of  $A \times B$  will be called a *relation* from  $A$  to  $B$ .

A relation  $R \subset A \times B$  is a *function* if and only if

$$\forall x \in A \forall y_1, y_2 \in B ((x, y_1) \in R \wedge (x, y_2) \in R) \rightarrow (y_1 = y_2)$$



## A function

Our domain and range here: natural numbers

$$\begin{aligned} f\ 0 &= 1 \\ f\ n &= n * f\ (n-1) \end{aligned}$$



## A function

Our domain and range here: natural numbers

$$f\ 0 = 1$$

$$f\ n = n * f\ (n-1)$$

mathematical induction vs. computational recursion vs. mathematical recursion



## Equals for equals

If

$$f\ 0 = 1$$

$$f\ n = n * f\ (n-1)$$

then what is  $f\ 3$ ?



## Equals for equals

If

$$f\ 0 = 1$$

$$f\ n = n * f\ (n-1)$$

then what is  $f\ 3$ ?

$$f\ 3 = 3 * f\ 2$$

$$= 3 * 2 * f\ 1$$

$$= 6 * 1 * f\ 0$$

$$= 6 * 1$$

$$= 6$$

called also *rewrite semantics*



## Imperative programming

Think like a computer:

```
public int f(int x) {
    int y = 1;
    for (int i=1; i<=x; i++) {
        y = y*i;
    }
    return y;
}
```

Then

$$f(3) = y = y*i = \text{????}$$



## The basic principle

**NO ASSIGNMENTS!**



## The basic principle

**NO ASSIGNMENTS!**

not exactly, but the meaning is:

**NO SIDE EFFECTS!**



## The problem with side effects

Example:

```
public int f(int x) {
    int t1 = g(x) + g(x);
    int t2 = 2*g(x);
    return t1-t2;
}
```



## The problem with side effects

Example:

```
public int f(int x) {
    int t1 = g(x) + g(x);
    int t2 = 2*g(x);
    return t1-t2;
}
```

Then **of course**

$$f(x) = t1-t2 = g(x) + g(x) - 2*g(x) = 0$$



## The problem with side effects

Example:

```
public int f(int x) {
    int t1 = g(x) + g(x);
    int t2 = 2*g(x);
    return t1-t2;
}
```

Then **of course**

$$f(x) = t1-t2 = g(x) + g(x) - 2*g(x) = 0$$

But suppose:

```
public int g(int x) {
    int y = input.nextInt();
    return y; }

```



## The concept of a variable

Is a variable the name of a

**memory cell**

or the name of an

**expression?**



## The core of functional programming

Functional programming  
=  
ordinary programming – assignments / side effects

It provides good support for

- higher order functions
- infinite data structures
- lazy evaluation



## Recursion: The sum of a list

```
sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)
```

Note1: *recursion* is intimately connected to *computability*.

Note2:  $(x:xs)$  - a very important idiom in FP/Haskell.



## Higher order functions

```
sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)
```

```
accumulate f i [] = i
accumulate f i (x:xs) = f x (accumulate f i xs)
```



## Higher order functions

```
sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)
```

```
accumulate f i [] = i
accumulate f i (x:xs) = f x (accumulate f i xs)
```

```
sum2 = accumulate (+) 0
```



## Higher order functions

```
sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)
```

```
accumulate f i [] = i
accumulate f i (x:xs) = f x (accumulate f i xs)
```

```
sum2 = accumulate (+) 0
```

```
product2 = accumulate (*) 1
anyTrue2 = accumulate (||) False
allTrue2 = accumulate (&&) True
```



## Infinite lists

Primes computed with Eratosthenes sieve:

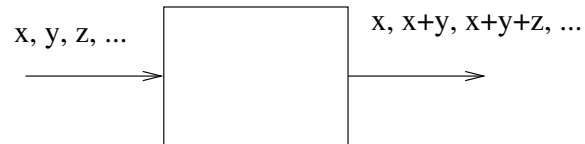
```
primes = sieve [2..]
  where
    sieve (n:ns) =
      n : sieve [ x | x <- ns, x `mod` n > 0 ]
```

Is this programming? Or just math?



## Data flow programming

The running sums of a list of numbers:

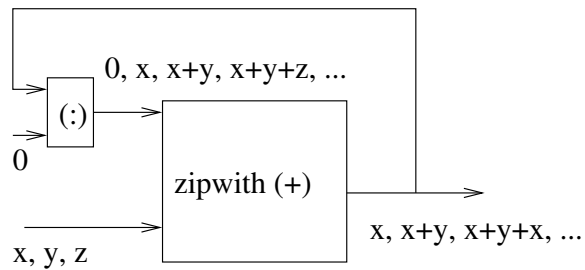


## Running sums

```
runningSums xs = theSolution
  where
    theSolution = zipWith (+) xs (0:theSolution)
```



## Data flow programming



## Exact approximations

The Taylor series of the exponential function:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$



## Exact approximations

The Taylor series of the exponential function:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

can be implemented exactly!



## Exact approximations

The Taylor series of the exponential function:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

can be implemented exactly!

for example like a list of approximations:

```
eExp x = runningSums [ (x^i)/(fac i) | i <- [0..] ]
```