**Administrativia**

- Standard notification: 140/200h total compared with 20/28h with lecturer(s) + 16/6 with TAs.
- Language-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 you, then machine (you get tests) and then teaching assistants (Sven Gestegård Robertz and Noric Couderc, possibly more).
- Any problems (deadlines?) – please discuss **IN ADVANCE** with me!
- Slides based a lot on Lennart Andersson and Lennart Ohlsson's material. Thank you.

**Important!**

Please read:  
http://cs.lth.se/utbildning/samarbete-eller-fusk/
Textbooks


Software

- Glasgow Haskell Compiler, or ghc
- Interpreter is called ghci
- Currently in its version 7.10.3. (@ 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 fp@cs.lth.se if you want to contact a human;
- Plan your time!
- Use our time (JM Mo 15.30-16.30, NC ..., SGR ..., AR during the labs)!

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

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

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

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

Our domain and range here: natural numbers

$$f_0 = 1$$
$$f_n = n \times f_{(n-1)}$$

mathematical induction vs. computational recursion vs. mathematical recursion

If

$$f_0 = 1$$
$$f_n = n \times f_{(n-1)}$$

then what is $f_3$?
**Equals for equals**

If

\[ f_0 = 1 \]
\[ f_n = n \cdot f(n-1) \]

then what is \( f_3 \)?

\[ f_3 = 3 \cdot f_2 = 3 \cdot 2 \cdot f_1 = 6 \cdot 1 \cdot f_0 = 6 \cdot 1 = 6 \]

called also *rewrite semantics*

**Imperative programming**

Think like a computer:

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

Then

\[ f(3) = y = y \cdot i = ???? \]

**The basic principle**

**NO ASSIGNMENTS!**

not exactly, but the meaning is:

**NO SIDE EFFECTS!**
The problem with side effects

Example:

```java
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:

```java
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?
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)

Note 1: recursion is intimately connected to computability.

Note 2: (x:xs) - a very important idiom in FP/Haskell.

Higher order functions

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

ackumulate f i [] = i
ackumulate f i (x:xs) = f x (ackumulate f i xs)

sum₂ = ackumulate (+) 0
Higher order functions

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

ackumulate f i [] = i
ackumulate f i (x:xs) = f x (ackumulate f i xs)

sum2 = ackumulate (+) 0
product2 = ackumulate (*) 1
anyTrue2 = ackumulate (||) False
allTrue2 = ackumulate (&&) 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:

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

Running sums
Data flow programming

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:

\[
eExp x = \text{runningSums } [ (x^i)/(\text{fac } i) | i <- [0..] ]
\]