

Functional Programming TDA 451

- Visit the course homepage. Find up-to-date information relevant for the course
 - Schedule
 - Lab assignments
 - Exercises
 - Last-minute changes
 - Sign up for the google group!



shortcut:
tr.im/tda451

www.cs.chalmers.se/Cs/Grundutb/Kurser/e4fun

Functional Programming

David Sands*

*Adapted slides from Koen Lindström Claessen & John Hughes

Why learn FP?

- Functional programming will make you think differently about programming
 - mainstream programming is all about *state* and how to *transform* state
 - Functional programming is all about *values* and how to construct values using *functions*
- Whether you use it later or not, it will make you a better programmer

Why Haskell?

- Haskell is a very *high-level language* (many details taken care of automatically).
- Haskell is expressive and concise (can achieve a lot with a little effort).
- Haskell is good at handling complex data and combining components.
- Haskell is a high-productivity language (prioritise programmer-time over computer-time).

Naming a Value

- We give a name to a value by making a *definition*.
- Definitions are put in a file, using a text editor such as emacs.

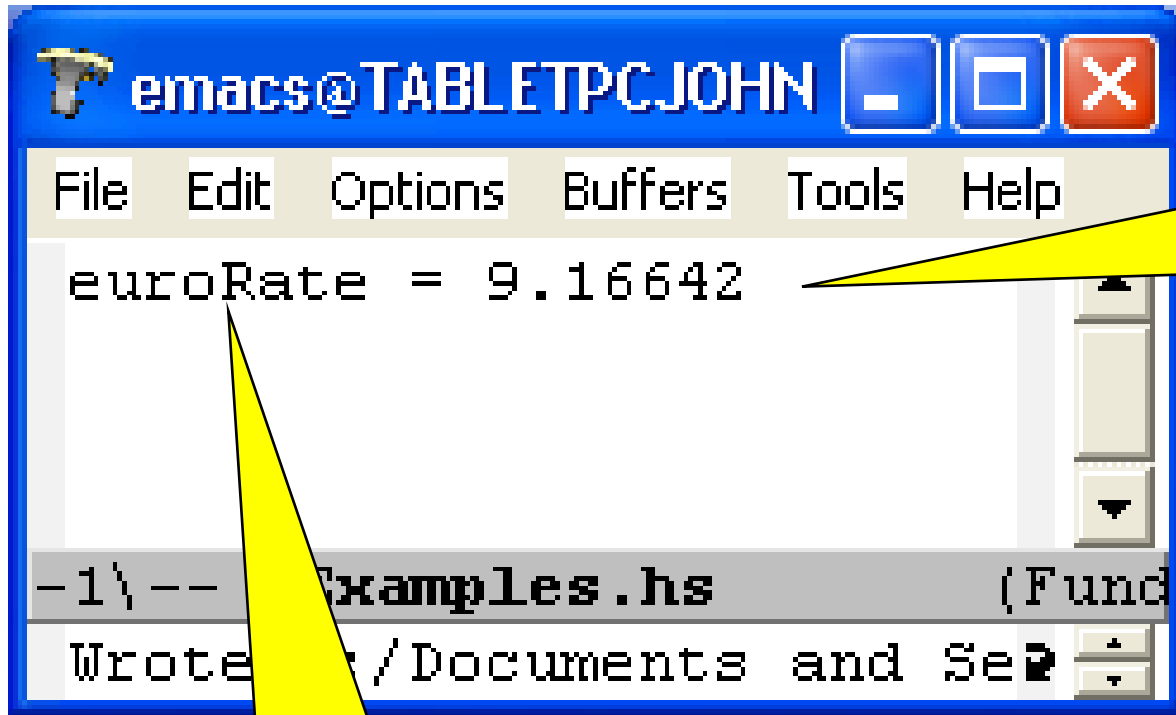
```
> emacs Examples.hs
```

The UNIX prompt,
not the ghci one!

Haskell files
end in `.hs`

Do this in a separate window, so you can edit and run hugs simultaneously.

Creating the Definition



The image shows a screenshot of an Emacs window. The title bar reads "emacs@TABLETPCJOHN". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", and "Help". The main text area contains the code "euroRate = 9.16642". The status bar at the bottom shows "-1\ -- Examples.hs (Fund" and "Wrote /Documents and Se".

```
emacs@TABLETPCJOHN
File Edit Options Buffers Tools Help
euroRate = 9.16642
-1\ -- Examples.hs (Fund
Wrote /Documents and Se
```

Give the name *euroRate* to the value 9.16642

variable

Using the Definition

The prompt changes – we have now loaded a program.

```
Prelude> :l Examples
Main> euroRate
9.16642
Main> 53*euroRate
485.82026
Main>
```

Load the file Examples.hs into ghci – make the definition available.

We are free to make use of the definition.

A Function to convert Euros to SEK

A definition –
placed in the
file
Examples.hs

A *comment* – to help us
understand the program

```
-- convert euros to SEK  
sek x = x*euroRate
```

Function name
– the name we
are defining.

Name for the
argument

Expression to
compute the
result

Using the Function

Reload the file to make the new definition available.

```
Main> :r  
Main> sek 53  
485.82026  
Main> euro (sek 49) == 49  
True
```

The operator == tests whether two values are equal

Converting Back Again

```
-- convert SEK to euros  
euro x = x/euroRate
```

```
Main> :r  
Main> euro 485.82026  
53.0  
Main> euro (sek 49)  
49.0  
Main> sek (euro 217)  
217.0
```



A Test

Automated Testing

- Define a function to perform the test for us

```
prop_EuroSek x = euro (sek x) == x
```

```
Main> prop_EuroSek 53  
True  
Main> prop_EuroSek 49  
True
```

Performs the same tests as before – but now we need only remember the function name!

Writing Properties in Files

- Convention: functions names beginning "prop_" are *properties we expect to be True*
- Writing properties in files
 - Tells us *how* functions should behave
 - Tells us *what* has been tested
 - Lets us *repeat* tests after changing a definition

Automatic Testing

- Testing account for *more than half* the cost of a software project
- We will use a widely used Haskell library for *automatic random* testing

```
import Test.QuickCheck
```

Add *first* in the file of definitions – makes QuickCheck available.

Names are case sensitive.

Running Tests

```
Main> quickCheck prop_EuroSek  
Falsifiable, after 10 tests:  
3.75
```

The value for which
the test fails.

It's *not* true!

Runs 100
randomly
chosen
tests

```
Main> sek 3.75  
34.374075  
Main> euro 34.374075  
3.75
```

Looks OK

The Problem

- There is a very tiny difference between the initial and final values

```
Main> euro (sek 3.75) - 3.75  
4.44089209850063e-016
```

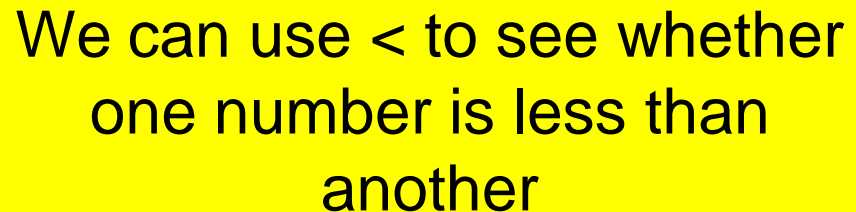
e-016 means
.10⁻¹⁶

- Calculations are only performed to about 15 significant figures
- The property is wrong!

Fixing the Problem

- The result should be *nearly* the same
- The difference should be small – say < 0.000001

```
Main> 2<3  
True  
Main> 3<2  
False
```



We can use `<` to see whether one number is less than another

Defining "Nearly Equal"

- We can define new *operators* with names made up of symbols

```
x ~== y = x-y < 0.000001
```

Define a new operator ~==

With arguments x and y

Which returns True if the *difference* between x and y is less than 0.000001

Testing \approx

Main> 3 \approx 3.0000001

True

OK

Main> 3 \approx 4

True

Huh? What's
wrong?

$x \approx y = x - y < 0.000001$

Fixing the Definition

- A useful function

```
Main> abs 3
```

```
3
```

```
Main> abs (-3)
```

```
3
```



Absolute value

```
x ~== y = abs (x-y) < 0.000001
```

```
Main> 3 ~== 4
```

```
False
```

Fixing the Property

```
prop_EuroSek x = euro (sek x) == x
```

```
Main> prop_EuroSek 3
```

```
True
```

```
Main> prop_EuroSek 56
```

```
True
```

```
Main> prop_EuroSek 2
```

```
True
```

Name the Price

- Let's define a name for the price of the game we want

```
price = 53
```

```
Main> sek price
```

```
ERROR - Type error in application
```

```
*** Expression      : sek price
```

```
*** Term            : price
```

```
*** Type            : Integer
```

```
*** Does not match : Double
```

Every Value has a Type

- The `:i` command prints *information* about a name

```
Main> :i price  
price :: Integer
```

Integer (whole number) is the *inferred type* of price

```
Main> :i euroRate  
euroRate :: Double
```

Double is the type of *real* numbers
Funny name, but refers to *double the precision* that computers originally used

More Types

```
Main> :i True
```

```
True :: Bool -- data constructor
```

```
Main> :i False
```

```
False :: Bool -- data constructor
```

```
Main> :i sek
```

```
sek :: Double -> Double
```

The type of a *function*

Type of the result

```
Main> :i prop_EuroSek
```

```
prop_EuroSek :: Double -> Bool
```

Type of the argument

Types Matter

- Types determine *how* computations are performed

Specify which type to use

```
Main> 123456789*123456789 :: Double  
1.52415787501905e+016
```

Correct to 15 figures

```
Main> 123456789*123456789 :: Integer  
15241578750190521
```

The *exact* result – 17 figures
(but must be an integer)

GHCi *must know* the type
of each expression
before computing it.

Type Checking

- Infers (works out) the type of every expression
- Checks that all types match – *before* running the program

Our Example

```
Main> :i price  
price :: Integer
```

```
Main> :i sek  
sek :: Double -> Double
```

```
Main> sek price  
ERROR - Type error in application  
*** Expression      : sek price  
*** Term            : price  
*** Type            : Integer  
*** Does not match : Double
```

Why did it work before?

```
Main> sek 53  
485.82026
```

Certainly works to say 53
What is the type of 53?

```
Main> 53 :: Integer  
53
```

53 can be used with *several*
types – it is *overloaded*

```
Main> 53 :: Double  
53.0
```

```
Main> price :: Integer  
53
```

Giving it a name *fixes* the
type

```
Main> price :: Double  
ERROR - Type error in type annotation  
*** Term           : price  
*** Type           : Integer  
*** Does not match : Double
```

Fixing the Problem

- Definitions can be given a *type signature* which *specifies* their type

```
price :: Double  
price = 53
```

```
Main> :i price  
price :: Double
```

```
Main> sek price  
485.82026
```

Always Specify Type Signatures!

- They help the reader (and *you*) understand the program
- The type checker can find your errors more easily, by checking that definitions have the types you say
- Type error messages will be easier to understand
- Sometimes they are necessary (as for price)

Example with Type Signatures

```
euroRate :: Double  
euroRate = 9.16642
```

```
sek, euro :: Double -> Double  
sek x     = x*euroRate  
euro x    = x/euroRate
```

```
prop_EuroSek :: Double -> Bool  
prop_EuroSek x = euro (sek x) ~== x
```

Function Definition by Cases and Recursion

Example: Absolute Value

- Find the absolute value of a number
 - If x is positive, result is x
 - If x is negative, result is $-x$

```
-- returns the absolute value of x
absolute :: Integer -> Integer
absolute x | x >= 0 = x
absolute x | x < 0 = -x
```

Notation

- We can abbreviate repeated left hand sides

```
absolute x | x >= 0 = x  
absolute x | x < 0  = -x
```

```
absolute x | x >= 0 = x  
           | x < 0  = -x
```

- Haskell also has **if then else**

```
absolute x = if x >= 0 then x else -x
```

Recursion

- First example of a *recursive* function
 - Compute x^n (without using built-in x^n)

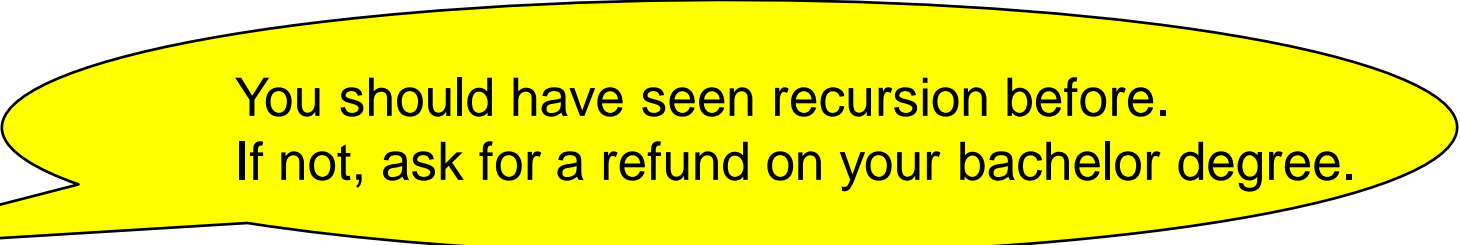
```
power x 0 = 1
power x n | n > 0 = x * power x (n-1)
```

- Calculate to find the answer:

$$\begin{aligned} \text{power } 2 \ 2 &= 2 * \text{power } 2 \ (2-1) \\ &= 2 * \text{power } 2 \ 1 = 2 * 2 * \text{power } 2 \ (1-1) \\ &= 2 * 2 * \text{power } 2 \ 0 = 2 * 2 * 1 = 4 \end{aligned}$$

Recursion

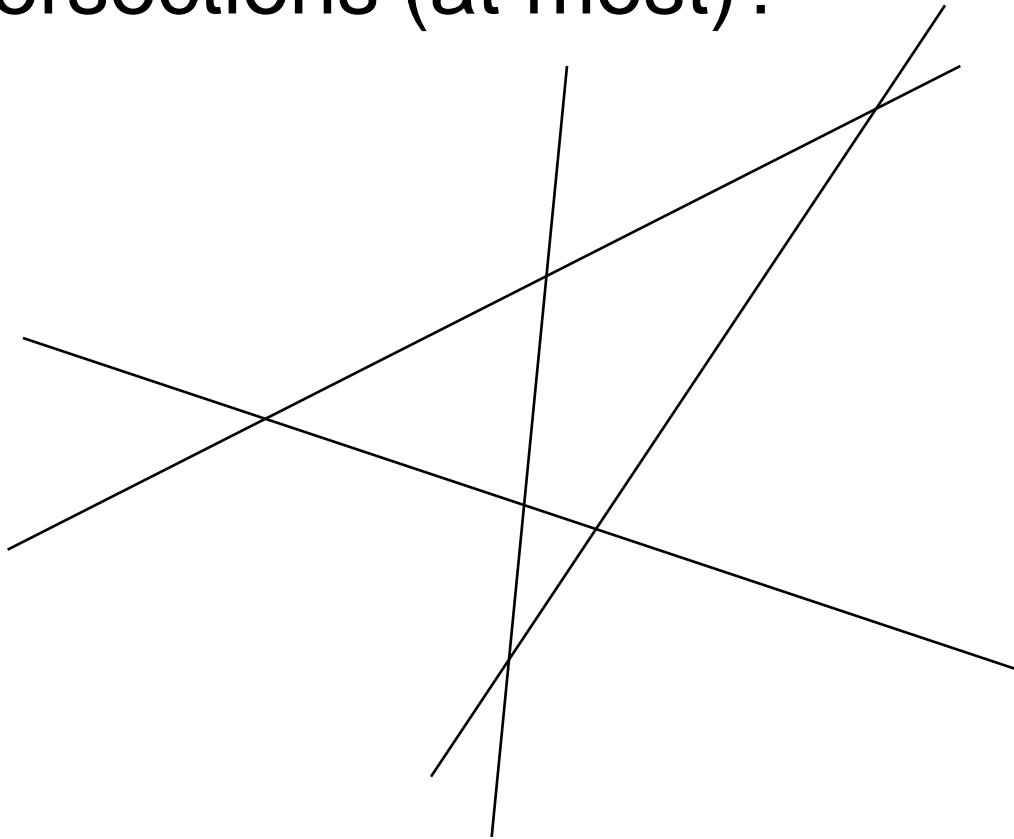
- Reduce a problem (e.g. power x n) to a *smaller* problem of the same kind
- So that we eventually reach a "smallest" *base case*
- Solve base case separately
- Build up solutions from smaller solutions



You should have seen recursion before.
If not, ask for a refund on your bachelor degree.

Example: Counting intersections

- n non-parallel lines. How many intersections (at most)?



The Solution

- Always pick the base case as simple as possible!

```
intersect :: Integer -> Integer
intersect 0 = 0
intersect n
  | n > 0 = intersect (n - 1) + n - 1
```