Introduction*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin for slide material

Course setup

- Course in two parts
  1. Andreas Lochbihler (until April 1st): Functional Programming
  2. Peter Müller (from April 3rd): Formal Methods
- Instruction in English. You may use German on assignments.
- Course times: Tuesday 10–12 and Thursday 10–12
- Tutorials: Tuesday 13–15 and Wednesday 15–17
  - Starts this week

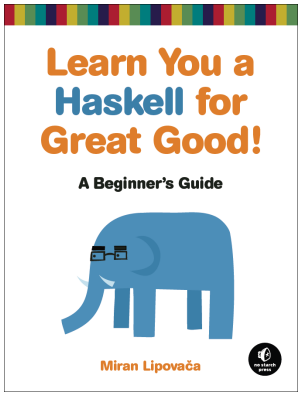
Tutorials, homework, and grading

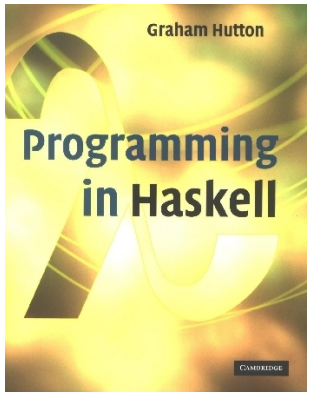
- Tutorial registration
  - Takes place during today's break
  - Those absent should contact `omaric@inf.ethz.ch`
- Weekly homework available **on the web** each Tuesday
  - Deadline is Monday the following week (before 11:00 am)
  - Submit to tutor by email with subject: `[FMFP] Exercise <n>` or use drop box behind the glass door at CNB F
  - Starts this week
- Homework is optional but **very strongly encouraged**
- Grade determined by final exam, during break (Sessionsprüfung)

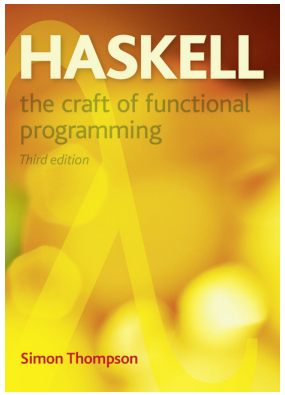
Additional resources

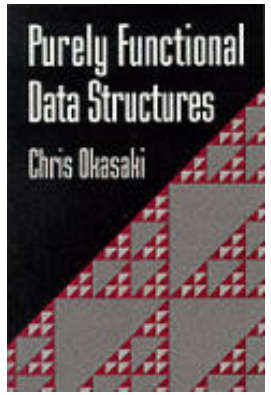
- Course web page contains numerous resources
  - Course announcements
  - Slides and exercises
  - Lots of helpful links, e.g., `www.haskell.org`
- Us!

Recommended books









- "Learn You a Haskell for Great Good" (M. Lipovača)
  Freely available beginner's guide to programming in Haskell
- "Programming in Haskell" (G. Hutton)
  Concise introduction to programming in Haskell
- "Haskell – the craft of functional programming" (S. Thompson)
  Beginners text book with a focus on testing and some induction proofs
- "Purely Functional Data Structures" (C. Okasaki)
  Advanced topics in functional programming

Why formal methods and functional programming?

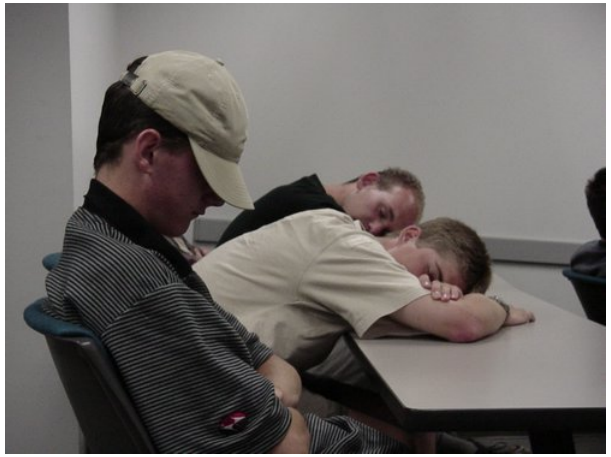
- **Apples** and **Oranges**? Yes but ...

  Studies program groups topics together in modules
  when this makes sense

- Combination FM+FP is somewhat unusual, but sensible
  - Both focus on how to **formalize & reason** about programs
  - I.e., programs as mathematical objects

- Fundamental topic in computer science

What else should you know?



Both parts start out slow, but become increasingly abstract.

Some ideas presented first informally and later formally.

Material will be new to most of you and hopefully exciting.

But it takes work, too. Be prepared for new ways of thinking!

If you have taken FMFP before, beware that the lecturer has changed.
$\Rightarrow$ some topics are new!

Overview to part I

1. Introduction, syntax
2. Logic, proofs, correctness
3. Lists
4. Abstraction, higher-order programming
5. Type classes and polymorphism
6. Algebraic data types
7. Lazy evaluation and efficiency
8. Monads, conclusions

Focus on idiomatic programs and provable correctness

Correctness

Ariane 5: $500+ million development, 1 arithmetic overflow

This week and next

- Introduction: basic concepts and some history
- Syntax and base types
  - **Boring** but necessary. Explained only in parts
  - Read manuals outside of lectures

- Proofs and correctness. **Exciting**, necessary material
  - The advantage of functional languages is not just that you can write clear, concise programs, but that you can easily understand and demonstrate what your programs actually do.
  - Requires some logic. I will provide a brief refresher.

Does language make a difference?

- Algorithm: a precise recipe describing computation steps.
- Can be presented as:
  - a Turing machine
  - a register machine
  - a Java program
  - an executable function description
- Difference in **level of abstraction**

  Which operations are available to model and solve problems?
- Difference in whether reasoning is **stateful**

  Compute values versus update memory

Example: GCD

- **The problem**
  Compute the **g**reatest **c**ommon **d**ivisor of two natural numbers

- **Specification**
  Let $x, y \in \mathcal{N}$ be given. The number z is the **greatest common divisor** of x and y iff $z|x$ and $z|y$ and there is no z', with $z' > z$, such that $z'|x$, and $z'|y$.

  Here $z|x \equiv \exists a \in \mathcal{N}.\ a \cdot z = x$

- The problem specification is not **constructive**
  - It does not describe how GCD should be computed
  - Euclid gave an algorithm to solve this, ca. 300 BC

GCD as imperative program

```
public static int gcd (int x, int y) {
   while (x != y) {
         if (x > y) x = x - y;
         else y = y - x;
   }
   return x;
}
```

- Consists of control flow and assignment

  Assignment changes computer's **state**

- To understand program, one must understand how state changes
  - Poor man's reasoning: simulate, tracking memory content
  - Better: Hoare logic: $\{P\}$ *prog* $\{Q\}$
  - Formal reasoning possible, but not so easy! **See part II**

GCD as functional program

```
gcd x y
  | x == y    = x
  | x > y     = gcd (x - y) y
  | otherwise = gcd x       (y - x)
```

- Formalizes **what** should be computed, rather than **how**
- Is this an algorithm?

  Yes, provided we have also specified how functions are executed
- Examples
  - $3 + (7 - 2) \leadsto 3 + 5 \leadsto 8$
  - $\mathtt{gcd}\ 4\ 6 \leadsto \mathtt{gcd}\ 4\ 2 \leadsto \mathtt{gcd}\ 2\ 2 \leadsto 2$

Basic concepts in functional programming

- Functions and values
  - Functions compute values
  - Functions are values: can compute and return them
- No side effects: $f(x)$ always returns the same value. Compare

```
class test {
  static int y = 0;    /* class variable: shared by all objects */
  static int f(int x) {
    y = y + 1;
    return y;
  }

  public static void main(String[] args) {
    System.out.println(f(0));
    System.out.println(f(0));
  }
}
```

Basic concepts (cont.)

- Since no side effects, can reason as in mathematics

  Example: if $f(0) = 2$ then $f(0) + f(0) = 2 + 2 = 4$

- This property is called **referential transparency**:
  an expression evaluates to the same value in every context.
  - No assignments
  - No global variables
  - . . .

- Easy to parallelize as computations cannot interfere.

More basic concepts

- Recursion instead of iteration

```
gcd x y
  | x == y    = x
  | x > y     = gcd (x-y) y
  | otherwise = gcd x     (y-x)
```

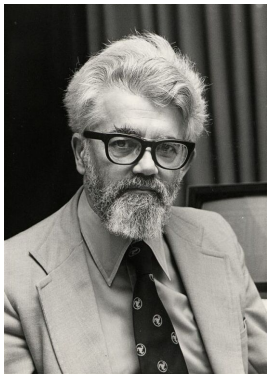
```
public static int gcd (int x, int y) {
   while (x != y) {
         if (x > y) x = x - y;
         else y = y - x;
   }
   return x;
}
```

- Flexible type system
  - Avoids many kinds of programming errors (e.g., no runtime errors: $3 +$ True)
  - Polymorphism supports reusability

```
sort [5,3,4]
sort ["hello", "there", "world"]
```

Functional programming: a short history

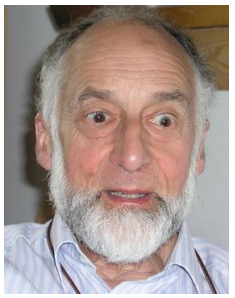
- **Lambda calculus: Church, 1930s**
  - Provides a theoretical framework for describing functions and their evaluation
  - Equivalent to Turing machines in terms of computational power
  - Church-Turing thesis



- **LISP: McCarthy, 1960s**
  - LISP = List processor. Lists as basic data structure
    Example: ("+", 3, ("–", 7, 2))
  - Developed for symbolic computing/AI applications
  - Untyped
  - Although primitive, still actively used by AI programmers
  - Basis for OS, hardware, . . . (e.g. Symbolics Inc.)

Short history (cont.)

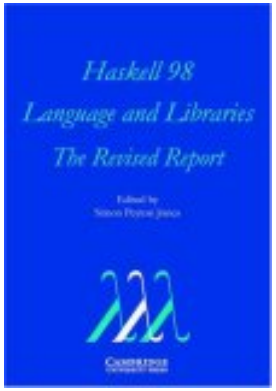
- **FP: Backus 1978 (Turing Award Winning Lecture)**
  - "Can Programming Be Liberated From the von Neumann Style"
  - Programming based on building blocks and combinators
  - Ideas still relevant!
    How does one construct reusable software libraries?



- **ML: Milner 1980s**
  - Powerful (polymorphic, static) type system
  - Sophisticated module system, for structuring "in the large"
  - Serious industrial applications using successor languages SML and OCaml. Competitive with OO-approach for complex software-engineering tasks.

Short history (cont.)

- **Miranda: Turner 1985**
  Similar to ML but features lazy evaluation.
  Can compute with representations of
  - finite data, like [1,2,3,4,5] or [1 .. 5]
  - infinite data [1 ..]



- **Haskell: 1992 – present**
  - State of the art in functional programming languages with lazy evaluation.
  - Many features like an advanced type system, efficient interpreters and compilers, huge library, . . .
  - Influenced other programming languages (e.g., Curry, Java, Python, C#, F#, Scala)

Using Haskell

- We will use **GHC**, the Glasgow Haskell Compiler
- Features
  - Public domain
  - Easy to install on many systems
  - Has an interactive interpreter: **ghci** (fine for assignments)
  - Has a compiler producing optimized code
- Actively supported and used in many real-world projects
- See `www.haskell.org` for binaries, libraries, documentation, ...

Introduction to functional programming

- Idea based on computing with expressions

$$3 + (7 - 2) \leadsto 8$$

  The computer functions like an ordinary calculator

- Functions can be defined

```
? :load gcd.hs
```

- and executed

```
? gcd 4 6
2
```

ghci — demo

```
? 3 + (7 - 2)
8

? 2 + 4 == 1 + 2 + 3
True

? 2 + True
ERROR: ...

? head [1,2,3]
1

? tail [1,2,3]
[2,3]

? :load gcd.hs
? gcd 10 15
5
```

Expression evaluation

- In mathematics, e.g., $f(x, y) = x - y$.

  Compute $f(5, 7)$ by substituting 5 for x and 7 for y and continue evaluation.

$$\begin{aligned} f(5, 7) &= 5 - 7 \\ &= -2 \end{aligned}$$

- Same holds for Haskell:

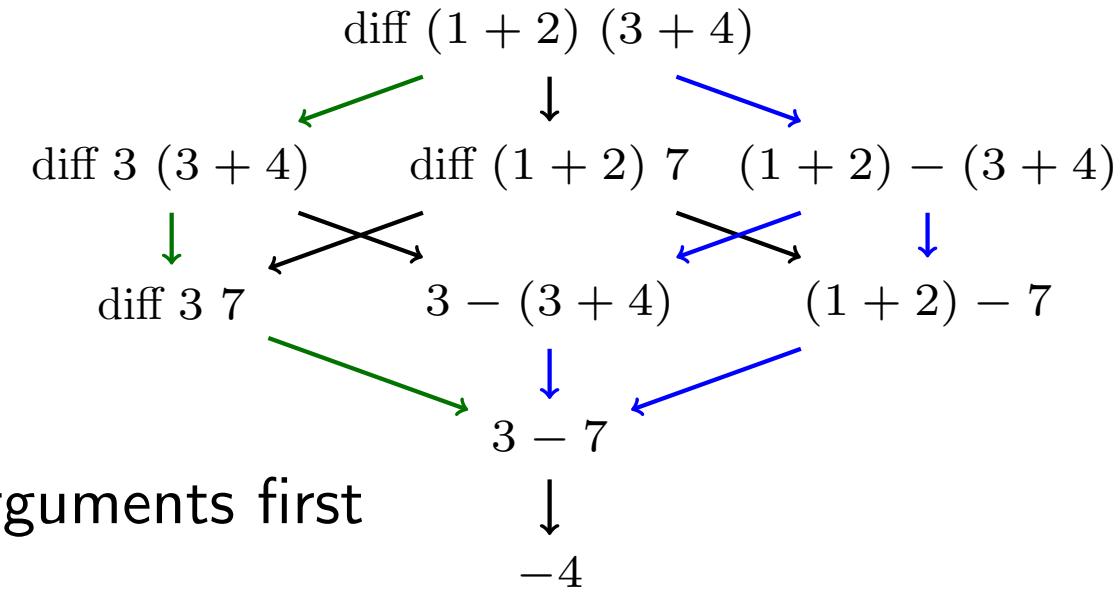
```
gcd x y
  | x == y    = x
  | x > y     = gcd (x-y) y
  | otherwise = gcd x     (y-x)
```

$$\begin{array}{rcll} \text{gcd } 10\ 15 & = & \text{gcd } 10\ (15-10) & \text{case 3 (otherwise)} \\ & = & \text{gcd } 10\ 5 & \text{as } 5 = 15 - 10 \\ & = & \text{gcd } (10-5)\ 5 & \text{case 2 } (10 > 5) \\ & = & \text{gcd } 5\ 5 & \text{as } 5 = 10 - 5 \\ & = & 5 & \text{case 1 } (5 == 5) \end{array}$$

Evaluation: strategies

For the program:

```
diff x y = x - y
```



- Eager evaluation: evaluate arguments first
  - Also called "call-by-value"
  - Corresponds to left path in picture
- Lazy evaluation: used in Haskell
  - Also called "call-by-need" or "left-most/outermost"
  - Certain functions force evaluation, e.g., arithmetic
  - We will study this in detail later

Syntax and Types

Introduction to syntax

```
gcd x y     -- functions and arguments start with lower-case letter
  | x == y    = x
  | x > y     = gcd (x-y) y       -- arguments written in sequence &
  | otherwise = gcd x     (y-x)   -- separated by whitespace
```

- Function consists of different cases:

```
functionName x1 ... xn
  | guard1 = expr1
    :
  | guardm = exprm
```

- Program consists of several definitions:

```
myConstant = 5

aFunction y1 ... ym
  | guard1 = expr1
  | guard2 = expr2

anotherFunction z1 ... zk = ...
```

2D layout

- Indentation determines separation of definitions:
  - All function definitions must start at same indentation level.
  - If a definition runs over n lines, indent lines 2 to n further.
- Recommended layout:

```
f1 x1 x2
   | a long guard which may go over
     a number of lines
       = a long expression that also can go over
         several lines
   | g2  = e2

f2 x1 x2 x3 = ...
```

- Erroneous layout:

```
square x = x * x
 cube x = x * x * x            -- parse error on input `='
```

- Spaces are important. Do not use TABs!

Types

- Haskell is a strongly typed language
- Types avoid runtime errors, like `3 + True`
- Either programmer provides types along with function definition

```
gcd :: Int -> Int -> Int
```

  or system computes types itself

- Function/argument types must "match" (formal account later)

```
? gcd 3 True

<interactive>:1:0:
    No instance for (Integral Bool)
      arising from a use of ‘gcd’
```

Type Int

- Values: $0, 1, 2, \ldots, -1, -2, \ldots$

  `Int` type with at least the range $\{-2^{29}, \ldots, 2^{29} - 1\}$
  Support for arbitrary bit numbers and arithmetic: `Integer`

- Functions: `+`, `*`, `^`, `-`, `div`, `mod`, `abs`

```
? mod 7 2
1
```

- An infix binary function is also called an "operator"

```
? 7 `mod` 2
1
```

- Operators can also be written in prefix notation

```
? + 3 4
<interactive>:1:0: parse error on input `+'
? (+) 3 4
7
```

Type Int (cont.)

- Operators have different binding strength

```
? 2 + 3 ^ 4          -- ^ binds stronger than +
83
? (2 + 3) ^ 4
625
```

- Order and equality return `True` or `False` of type Bool
  - `>` greater than
  - `>=` greater than or equal
  - `==` equal
  - `/=` unequal
  - `<=` less than or equal
  - `<` less than

Type Bool

- Values: `True`, `False`
- Binary operators `&&`, `||`, and unary function `not` as expected

```
? True && False
False

? (10 < 1) || (10 == 1) || (10 > 1)
True

? not (9 >= 7) || (3 /= 3)    -- not binds stronger than && and ||
False

? (3 > 5) == (7 < 6)          -- (==) on Bool is "if and only if"
True
```

Examples of function definition — XOR

- `xor` defined using other operators:

```
xor x y = (x || y) && not (x && y)
```

- `xor` defined using guards:

```
xor x y
  | x         = not y
  | otherwise = y
```

- `xor` defined using cases (new):

```
xor True  True  = False
xor True  False = True
xor False True  = True
xor False False = False
```

- Cases can contain variables ("patterns"):

```
xor True  y = not y
xor False y = y
```

Types Char, String, and Double

Char: `'a'`, `'b'`, ..., `'0'`, `'1'`, ..., `'\t'`, `'\n'`.

```
? ord 'a'   --- requires Char module loaded with :module Data.Char
97
? chr 97
'a'
```

String: `"hello"`, `"123"`, `"a"`.

```
? "Hello " ++ "there"
"Hello there"
```

Double: `0.3456`, `-2.85e03` $= -2.85 * 10^3$, . . .

Functions like `+`, `-`, `*`, `/`, `abs`, `acos`, `asin`, `ceiling`, ...

For documentation see:

www.haskell.org/ghc/docs/latest/html/libraries

Type tuple

- Name reflects: pair, triple, 4-tuple, ...
  - Used to model composite objects ("records")
- **Example**: Student has name, ID number, starting year

  Record type `(String, Int, Int)`
  with element `("Ueli Naef", 1234, 2011)`
- First example of a **type constructor**
  - if $T_1, \ldots, T_n$ are types, then $(T_1, \ldots, T_n)$ is a (tuple) type.
    e.g., `(Int, String, Bool)`
  - if $v_1 :: T_1, \ldots, v_n :: T_n$ then $(v_1, \ldots, v_n) :: (T_1, \ldots, T_n)$
    e.g., `(3, "hi", True) :: (Int, String, Bool)`
  - N.B.: $n \geq 2$, i.e., `("foo")` is not a tuple.
  - We can nest tuples: `(3, ("hi", True)) :: (Int, (String, Bool))`

Tuples

- Functions can take tuples as arguments or return tupled values

```
addPair :: (Int, Int) -> Int
addPair (x, y) = x + y
? addPair (3, 4)
7
```

- Patterns can be nested

```
shift :: ((Int, Int), Int) ->  (Int, (Int, Int))
shift ((x, y), z) = (x, (y, z))
```

- Pattern matching can be used to decompose tuples

```
name (s, id, y) = s
studentNumber (s, id, y) = id
year (s, id, y) = y
```

or (probably not sensible)

```
silly (s, id, y) = id + 2 * y
```

Patterns and function definition

- Function definition built from both patterns `mi` and guards `gi`

```
fun m1 m2 ... mn
  | g1          = e1
     :
  | gm          = em
  | otherwise = e   -- optional!
```

  - Patterns `mi` are variables, constants, or built from data constructors (like tuples)
  - Guards `gi` are Boolean expressions

- Example:

```
silly b (x, y)
  | b           = x + y
  | otherwise = x * y
```

Functions: scope

- Global scope: a function can be called from any other

```
f x y = ...
g x = ... h ...
h z = ... f ... g ...
```

- Local scope with "`let`" and "`where`"

```
let x1 = e1
     :
    xn = en
in e
```

- `let` builds one expression from others:
  - `xi` can bind a variable or a (local) function
  - Local definitions may refer to each other

Functions: scope (cont.)

- Example:

```
f x = let sq y = y * y
      in sq x + sq x
```

We can evaluate `f 10`, but not `sq 10`

```
let y   = a * b
    f x = (x + y) / 2
in f c + f d
```

WHERE: local scope after a function definition

```
f p1 p2 ... pm
  | g1 = e1
  | g2 = e2
    :
  | gk = ek
  where
    v1 a1 ... an = r1
    v2 = r2
    :
```

N.B.

- "`where`" comes directly after a function definition
- Bindings defined over all guards

Program definition with local definitions

- Let's define a function that takes three numbers and returns the largest and how often it occurs

```
maxThreeOccurs :: Int -> Int -> Int -> (Int, Int)
```

- Top-down development

```
maxThreeOccurs n m p = (maxVal, maxCount)
  where
    maxVal   = max3 n m p
    maxCount = count maxVal n m p
```

- Then write the new subroutines

```
max3 a b c = max a (max b c)

count val n m p = isval n + isval m + isval p
  where
    isval x
      | x == val  = 1
      | otherwise = 0
```

Example (cont.)

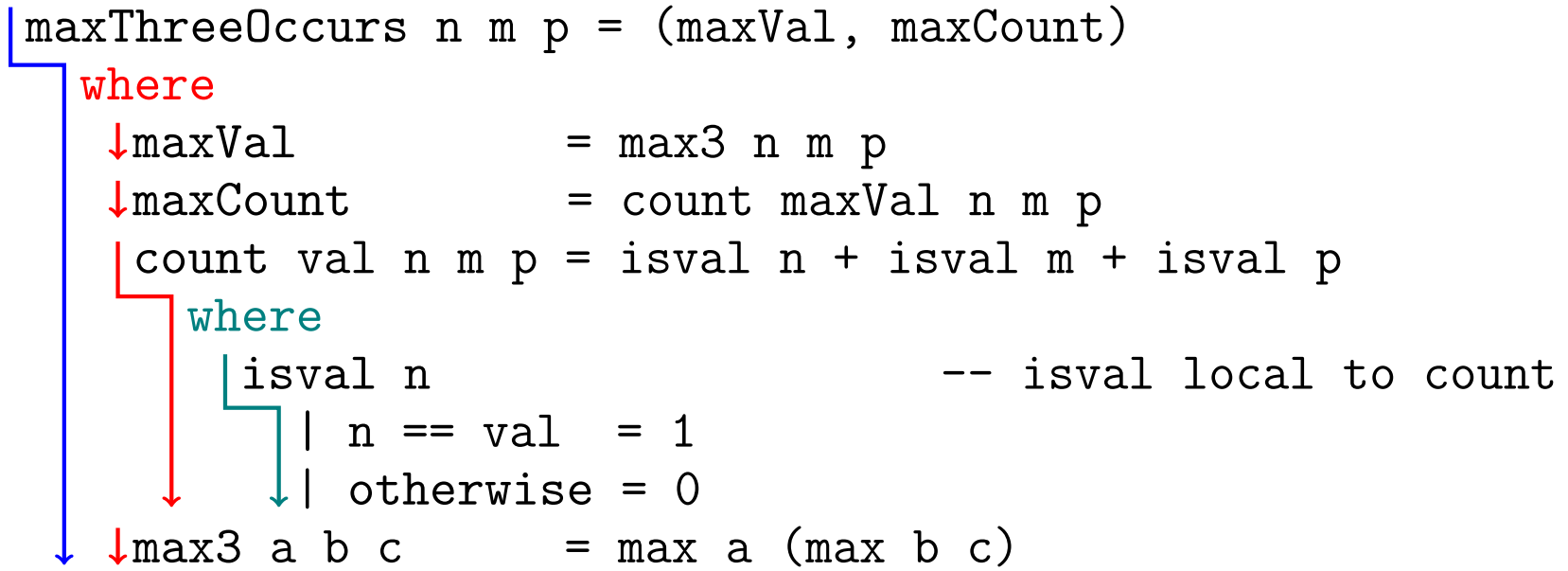
- Alternative structure where all definitions are local

```
maxThreeOccurs n m p = (maxVal, maxCount)
  where
    maxVal           = max3 n m p
    maxCount         = count maxVal n m p
    count val n m p = ...
    max3 a b c       = max a (max b c)
```

- Which program is better?
  - No general answer
  - Depends on how general the defined functions are

2D layout for `let` and `where`

- `let` and `where` open blocks for local definitions.
  Layout rule applies here, too.
  - First definition determines indentation for all definitions in block.
  - Multi-line definitions must indent more.
  - Less indentation closes block.



2D layout for `let` and `where`

- `let` and `where` open blocks for local definitions.
  Layout rule applies here, too.
  - First definition determines indentation for all definitions in block.
  - Multi-line definitions must indent more.
  - Less indentation closes block.

```
maxThreeOccurs n m p = (maxVal, maxCount)
  where
   maxVal           = max3 n m p
   maxCount         = count maxVal n m p
   count val n m p = isval n + isval m + isval p
     where
       isval n                    -- isval local to count
         | n == val  = 1
         | otherwise = 0
   max3 a b c       = max a (max b c)
```

- Spaces are important. **Do not use TABs!**

Converting to/from `String`

- `show` converts values to `String`s

```
? show 23
"23"

? show True
"True"

? show (17, 'a')
"(17,'a')"

? show (17 + 42)
"59"
```

Converting to/from `String`

- `show` converts values to `String`s

```
? show 23
"23"

? show True
"True"

? show (17, 'a')
"(17,'a')"

? show (17 + 42)
"59"
```

- `read` converts `String`s to values

  always specify desired type

```
? read "23" :: Integer
23

? read "23" :: Double
23.0

? read "(17, 'a')" :: (Int, Char)
(17,'a')

? (read "17" :: Int) + (read "42" :: Int)
59

? read "17+42" :: Int
*** Exception: Prelude.read: no parse
```

Input and Output

- How would we write a program like the following in Haskell?

```
void f(String out) {
  String inp1 = Console.readLine();
  String inp2 = Console.readLine();
  if (inp2.equals(inp1))
    System.out.println(out); }
```

Input and Output

- How would we write a program like the following in Haskell?

```
void f(String out) {
  String inp1 = Console.readLine();
  String inp2 = Console.readLine();
  if (inp2.equals(inp1))
    System.out.println(out); }
```

- Assume there would be functions

```
getLine :: String
putStrLn :: String -> ()      -- () is the unit type in Haskell


f :: String -> ()
f out = let
    inp1 = getLine
    inp2 = getLine
  in if inp2 == inp1
     then putStrLn out
     else ()
```

Input and Output

- How would we write a program like the following in Haskell?

```
void f(String out) {
  String inp1 = Console.readLine();
  String inp2 = Console.readLine();
  if (inp2.equals(inp1))
    System.out.println(out); }
```

- Assume there would be functions

```
getLine :: String
putStrLn :: String -> ()      -- () is the unit type in Haskell
```

```
f :: String -> ()
f out = let
    inp1 = getLine
    inp2 = getLine
  in if inp2 == inp1
     then putStrLn out
     else ()
```

- What does `inp2 == inp1` evaluate to?
- In which order are arguments evaluated?
- These functions cause side-effects! Which state changes?

IO: Input and Output

- Cannot use normal functions `putStrLn` and `getLine`
- Tag types with `IO` to capture side effects

```
getLine :: IO String
putStrLn :: String -> IO ()
```

- Syntax for `IO` type:
  - `do` block sequences side effects
  - `<-` extracts values from `IO`
  - `return` tags values with `IO`

```
f :: String -> ()
f out = let
    inp1 = getLine
    inp2 = getLine
  in if inp2 == inp1
     then putStrLn out
     else ()
```

```
f :: String -> IO ()
f out = do
  inp1 <- getLine
  inp2 <- getLine
  if inp2 == inp1
    then putStrLn out
    else return ()
```

main

- `main :: IO ()` is the entry function for Haskell programs

```
main :: IO ()
main = do
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

- compile with GHC

```
> ghc hello.hs
> ./hello
Enter your name:
Andreas
Hello, Andreas!
```

- run within GHCi

```
> ghci
? :load hello.hs
? main
Enter your name:
Andreas
Hello, Andreas!
```

No escape from `IO`

- `IO` tag sticks to values
  Can compute with `IO` values only in `do` blocks
  - Clumsier than with pure expressions
  - Results are tagged, too.
- Stay out of `IO` as long as possible
- Separate computations from user interface

Side effects

```
main :: IO ()
main = do
  n <- getLine
  m <- getLine
  let x = gcd (read n) (read m)
  putStrLn (show x)
```

Pure

```
gcd :: Int -> Int -> Int
gcd x y
  | x == y    = x
  | x > y     = gcd (x-y) y
  | otherwise = gcd x (y-x)
```

No escape from `IO`

- `IO` tag sticks to values
  Can compute with `IO` values only in `do` blocks
  - Clumsier than with pure expressions
  - Results are tagged, too.

NB:
Inside a `do` block,
`let` needs no `in`

- Stay out of `IO` as long as possible
- Separate computations from user interface

Side effects

```
main :: IO ()
main = do
  n <- getLine
  m <- getLine
  let x = gcd (read n) (read m)
  putStrLn (show x)
```

Pure

```
gcd :: Int -> Int -> Int
gcd x y
  | x == y    = x
  | x > y     = gcd (x-y) y
  | otherwise = gcd x (y-x)
```