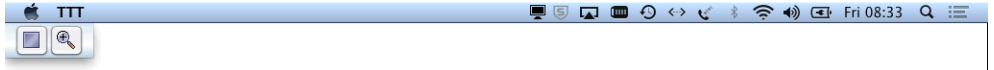
Script **generated by TTT**

Title: Nipkow: Info2 (10.10.2014)

Date: Fri Oct 10 06:33:23 GMT 2014

Duration: 82:08 min

Pages: 143



Informatik 2: Functional Programming

Tobias Nipkow

Fakultät für Informatik
TU München

http://fp.in.tum.de

Wintersemester 2014/15

October 9, 2014

Literatur

- Vorlesung orientiert sich stark an
  Thompson: *Haskell, the Craft of Functional Programming*

Literatur

- Vorlesung orientiert sich stark an
  Thompson: *Haskell, the Craft of Functional Programming*
- Für Freunde der kompakten Darstellung:
  Hutton: *Programming in Haskell*

Klausur und Hausaufgaben

- Klausur am Ende der Vorlesung
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.

Klausur und Hausaufgaben

- Klausur am Ende der Vorlesung
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.
- Hausaufgabenstatistik:
  Wahrscheinlichkeit, die Klausur (oder W-Klausur) zu bestehen:

Klausur und Hausaufgaben

- Klausur am Ende der Vorlesung
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.
- Hausaufgabenstatistik:
  Wahrscheinlichkeit, die Klausur (oder W-Klausur) zu bestehen:
  - $\geq 40\%$ der Hausaufgabenpunkte $\Longrightarrow 100\%$
  - $< 40\%$ der Hausaufgabenpunkte $\Longrightarrow < 50\%$

Klausur und Hausaufgaben

- Klausur am Ende der Vorlesung
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt, hat seinen Notenbonus sofort verwirkt.
- Hausaufgabenstatistik:
  Wahrscheinlichkeit, die Klausur (oder W-Klausur) zu bestehen:
  - $\geq 40\%$ der Hausaufgabenpunkte $\Longrightarrow 100\%$
  - $< 40\%$ der Hausaufgabenpunkte $\Longrightarrow < 50\%$
- Aktueller persönlicher Punktestand im WWW über Statusseite

Programmierwettbewerb — Der Weg zum Ruhm

- Jede Woche eine Wettbewerbsaufgabe
- Punktetabellen im Internet:
  - Die Top 20 jeder Woche

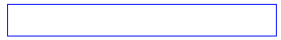
Programmierwettbewerb — Der Weg zum Ruhm

- Jede Woche eine Wettbewerbsaufgabe
- Punktetabellen im Internet:
  - Die Top 20 jeder Woche
  - Die kumulative Top 20

Programmierwettbewerb — Der Weg zum Ruhm

- Jede Woche eine Wettbewerbsaufgabe
- Punktetabellen im Internet:
  - Die Top 20 jeder Woche
  - Die kumulative Top 20
- Ende des Semesters: Trophäen fuer die Top k Studenten

Piazza: Frage-und-Antwort Forum



Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonymn)
  Natürlich keine Lösungen posten!

Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonymn)
  Natürlich keine Lösungen posten!

- Fragen werden an alle Tutoren weitergeleitet

Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonymn)
  Natürlich keine Lösungen posten!
- Fragen werden an alle Tutoren weitergeleitet

- Mehr über Piazza: Video auf `http://piazza.com`

Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonymn)
  Natürlich keine Lösungen posten!
- Fragen werden an alle Tutoren weitergeleitet
- Mehr über Piazza: Video auf `http://piazza.com`

- Zugang zu Piazza für Info 2 über Vorlesungsseite
- Funktioniert erst nach Anmeldung zur Übung

Haskell Installation

Haskell Installation

- Bei Problemen mit der Installation des GHC:
  Zwei Beratungstermine, siehe Vorlesungsseite
  (10.10. 10:00-12:00, 13.10. 10:00-13:00)

Haskell Installation

- Bei Problemen mit der Installation des GHC:
  Zwei Beratungstermine, siehe Vorlesungsseite
  (10.10. 10:00-12:00, 13.10. 10:00-13:00)
- Tutoren leisten in der Übung keine Hilfestellung mehr!

2. Functional Programming: The Idea

Functions are pure/mathematical functions:
Always same output for same input

Functions are pure/mathematical functions:
Always same output for same input

Computation = Application of functions to arguments

Example 1

In Haskell:

```
sum [1..10]
```

Example 1

In Haskell:

```
sum [1..10]
```

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
  total = total + i;
```

Example 2

In Haskell:

```
wellknown [] = []
wellknown (x:xs) = wellknown ys ++ [x] ++ wellknown zs
   where ys = [y | y <- xs, y <= x]
         zs = [z | z <- xs, x < z]
```

In Java:

```
void sort(int[] values) {
   if (values ==null || values.length==0){ return; }
   this.numbers = values;
   number = values.length;
   quicksort(0, number - 1);
}

void quicksort(int low, int high) {
   int i = low, j = high;
   int pivot = numbers[low + (high-low)/2];
   while (i <= j) {
      while (numbers[i] < pivot) { i++; }
      while (numbers[j] > pivot) { j--; }
      if (i <= j) {exchange(i, j); i++; j--; }
   }
   if (low < j) quicksort(low, j);
   if (i < high) quicksort(i, high);
}

void exchange(int i, int j) {
   int temp = numbers[i];
   numbers[i] = numbers[j];
   numbers[j] = temp;
}
```

There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies.

The other way is to make it so complicated that there are no obvious deficiencies.

From the Turing Award lecture by Tony Hoare (1985)

There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies.

The other way is to make it so complicated that there are no obvious deficiencies.

From the Turing Award lecture by Tony Hoare (1985)

Characteristics of functional programs

elegant
expressive

Characteristics of functional programs

elegant
expressive
concise
readable

Characteristics of functional programs

elegant
expressive
concise
readable
predictable pure functions, no side effects

Characteristics of functional programs

elegant
expressive
concise
readable
predictable pure functions, no side effects
provable it's just (very basic) mathematics!

Aims of functional programming

- Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures

Aims of functional programming

- Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures
- Minimize time to read and write programs:
  $\Rightarrow$ reduced development and maintenance time and costs

Aims of functional programming

- Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures
- Minimize time to read and write programs:
  $\Rightarrow$ reduced development and maintenance time and costs
- Increased confidence in correctness of programs:
  clean and simple syntax and semantics
  $\Rightarrow$ programs are easier to
  - understand
  - test (Quickcheck!)
  - prove correct

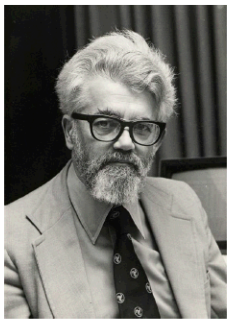
Historic Milestones

1930s

Alonzo Church develops the lambda calculus,
the core of all functional programming languages.

Historic Milestones

1950s



John McCarthy (Turing Award 1971) develops Lisp, the first functional programming language.

Historic Milestones

1970s

Robin Milner (FRS, Turing Award 1991) & Co. develop ML, the first modern functional programming language with *polymorphic types* and *type inference*.

Historic Milestones

1987

Haskell

A Purely Functional Language

An international committee of researchers initiates the development of Haskell, a standard lazy functional language.

Popular languages based on FP

F# (Microsoft) = ML for the masses

Popular languages based on FP

F# (Microsoft) = ML for the masses

Erlang (Ericsson) = distributed functional programming

Popular languages based on FP

F# (Microsoft) = ML for the masses

Erlang (Ericsson) = distributed functional programming

Scala (EPFL) = Java + FP

FP concepts in other languages

Garbage collection: Java, C#, Python, Perl, Ruby, Javascript

FP concepts in other languages

Garbage collection: Java, C#, Python, Perl, Ruby, Javascript

Higher-order functions: Java, C#, Python, Perl, Ruby, Javascript

FP concepts in other languages

Garbage collection:	Java, C#, Python, Perl, Ruby, Javascript
Higher-order functions:	Java, C#, Python, Perl, Ruby, Javascript
Generics:	Java, C#
List comprehensions:	C#, Python, Perl 6, Javascript

FP concepts in other languages

Garbage collection:	Java, C#, Python, Perl, Ruby, Javascript
Higher-order functions:	Java, C#, Python, Perl, Ruby, Javascript
Generics:	Java, C#
List comprehensions:	C#, Python, Perl 6, Javascript
Type classes:	C++ "concepts"

Why we teach FP

- FP is a fundamental programming style (like OO!)

Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# ...

Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# ...
- It gives you the edge over Millions of Java/C/C++ programmers out there

Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# ...
- It gives you the edge over Millions of Java/C/C++ programmers out there
- FP concepts make you a better programmer, no matter which language you use

Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# ...
- It gives you the edge over Millions of Java/C/C++ programmers out there
- FP concepts make you a better programmer, no matter which language you use
- To show you that programming need not be a black art with magic incantations like `public static void` but can be a science

3.1 Notational conventions

$e :: T$ means that expression e has type T

	Mathematics	Haskell
Function types:	$f : A \times B \to C$	`f :: A -> B -> C`

3.1 Notational conventions

$e :: T$ means that expression e has type T

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

Function application:

Mathematics	Haskell
$f(a)$	`f a`

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

Function application:

Mathematics	Haskell
$f(a)$	`f a`
$f(a, b)$	`f a b`

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

Function application:

Mathematics	Haskell
$f(a)$	`f a`
$f(a, b)$	`f a b`
$f(g(b))$	`f (g b)`

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

Function application:

Mathematics	Haskell
$f(a)$	`f a`
$f(a, b)$	`f a b`
$f(g(b))$	`f (g b)`
$f(a, g(b))$	`f a (g b)`

3.1 Notational conventions

$e :: T$ means that expression e has type T

Function types:

Mathematics	Haskell
$f : A \times B \to C$	`f :: A -> B -> C`

Function application:

Mathematics	Haskell
$f(a)$	`f a`
$f(a, b)$	`f a b`
$f(g(b))$	`f (g b)`
$f(a, g(b))$	`f a (g b)`

Prefix binds stronger than infix:

`f a + b` means `(f a) + b`
not `f (a + b)`

3.2 Type `Bool`

Predefined: `True False not && || ==`

3.2 Type `Bool`

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
```

3.2 Type `Bool`

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
xor x y  =  (x || y) && not(x && y)
```

3.2 Type `Bool`

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
xor x y  =  (x || y) && not(x && y)

xor2 :: Bool -> Bool -> Bool
```

3.2 Type `Bool`

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
xor x y  =  (x || y) && not(x && y)

xor2 :: Bool -> Bool -> Bool
xor2 True  True   =  False
xor2 True  False  =  True
xor2 False True   =  True
xor2 False False  =  False
```

3.2 Type `Bool`

Predefined: `True False not && || ==`

Defining new functions:

```
xor :: Bool -> Bool -> Bool
xor x y  =  (x || y) && not(x && y)

xor2 :: Bool -> Bool -> Bool
xor2 True  True   =  False
xor2 True  False  =  True
xor2 False True   =  True
xor2 False False  =  False
```

This is an example of pattern matching.
The equations are tried in order. More later.

Testing with QuickCheck

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Check property with GHCi:

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Check property with GHCi:


```
> quickCheck prop_xor2
```

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Check property with GHCi:

```
> quickCheck prop_xor2
```


GHCi answers

```
+++ OK, passed 100 tests.
```

QuickCheck

- Essential tool for Haskell programmers

QuickCheck

- Essential tool for Haskell programmers
- Invaluable for regression tests
- Important part of exercises & homework
- Helps you to avoid bugs
- Helps us to discover them

QuickCheck

- Essential tool for Haskell programmers
- Invaluable for regression tests
- Important part of exercises & homework
- Helps you to avoid bugs
- Helps us to discover them

Every nontrivial Haskell function
should come with one or more QuickCheck properties/tests

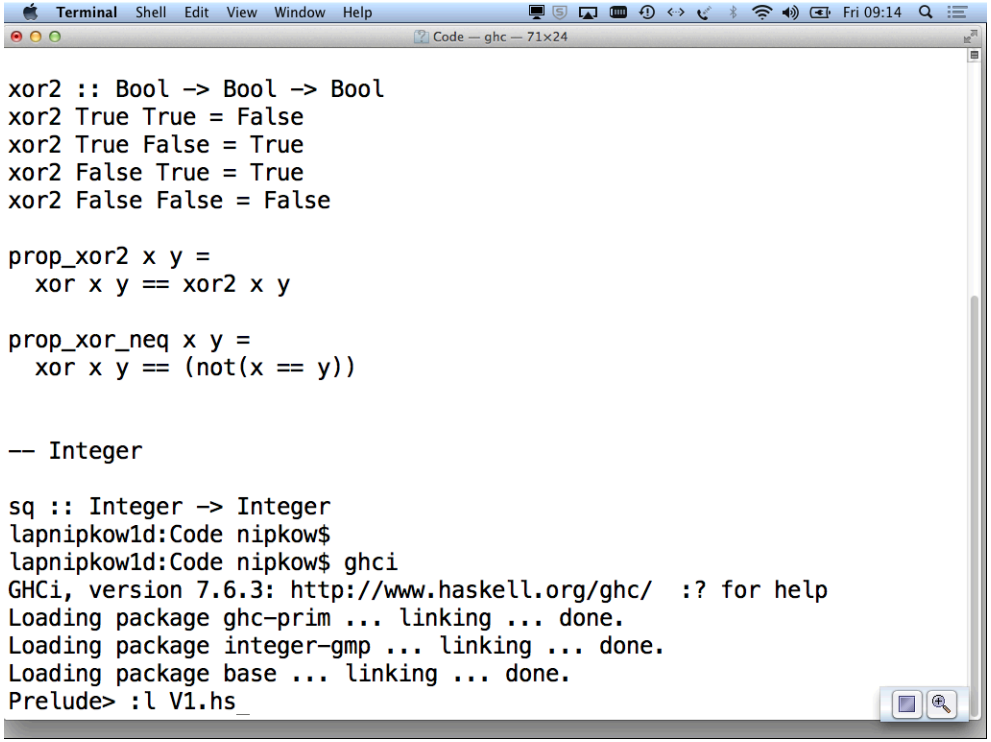
QuickCheck

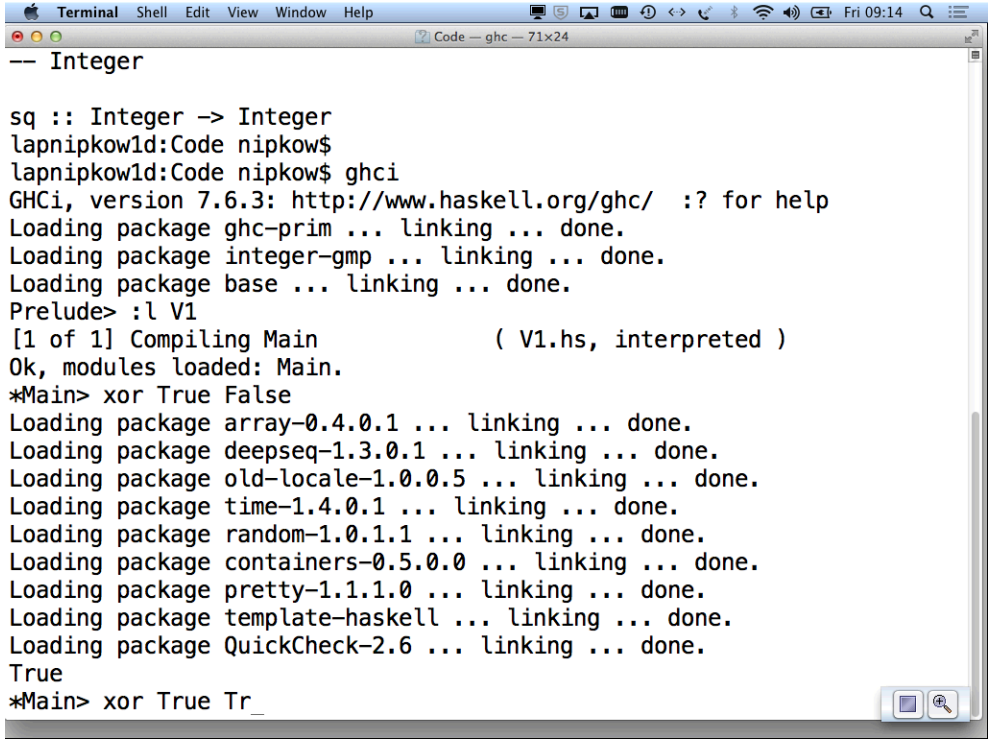
- Essential tool for Haskell programmers
- Invaluable for regression tests
- Important part of exercises & homework
- Helps you to avoid bugs
- Helps us to discover them

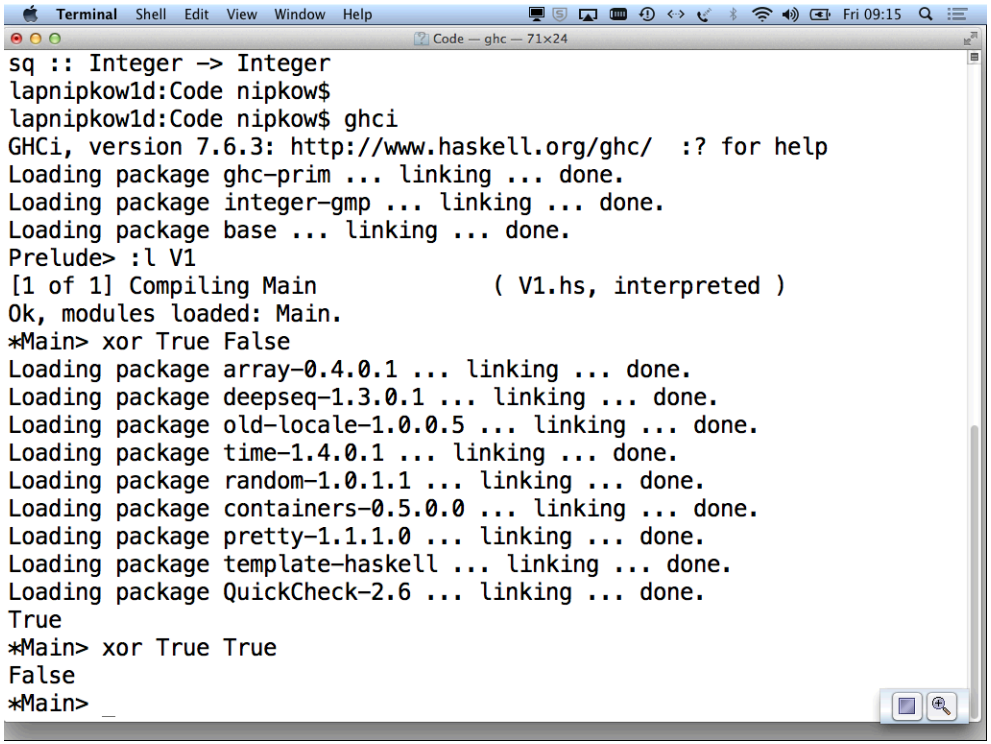
Every nontrivial Haskell function
should come with one or more QuickCheck properties/tests

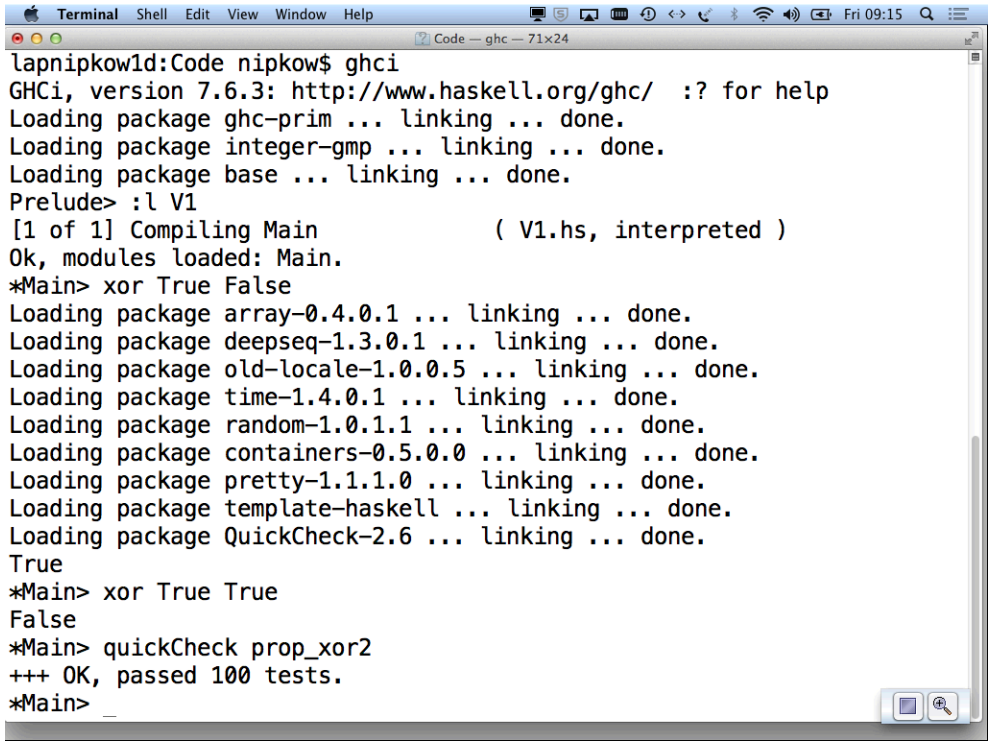
Typical test:

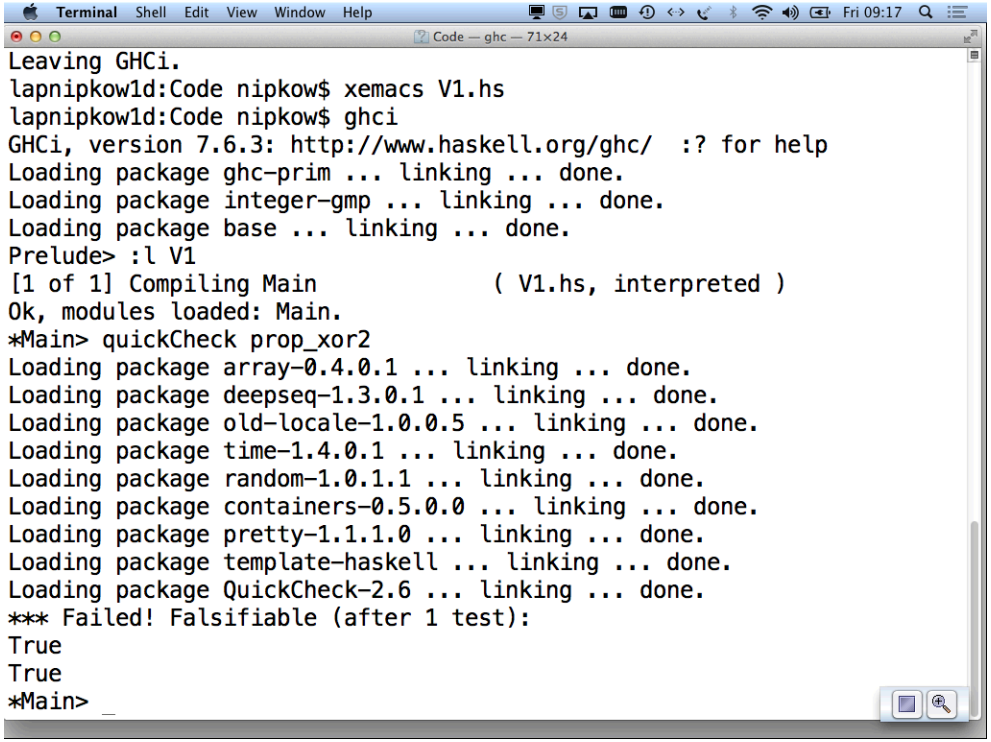
```
prop_f x y =
  f_efficient x y == f_naive x y
```

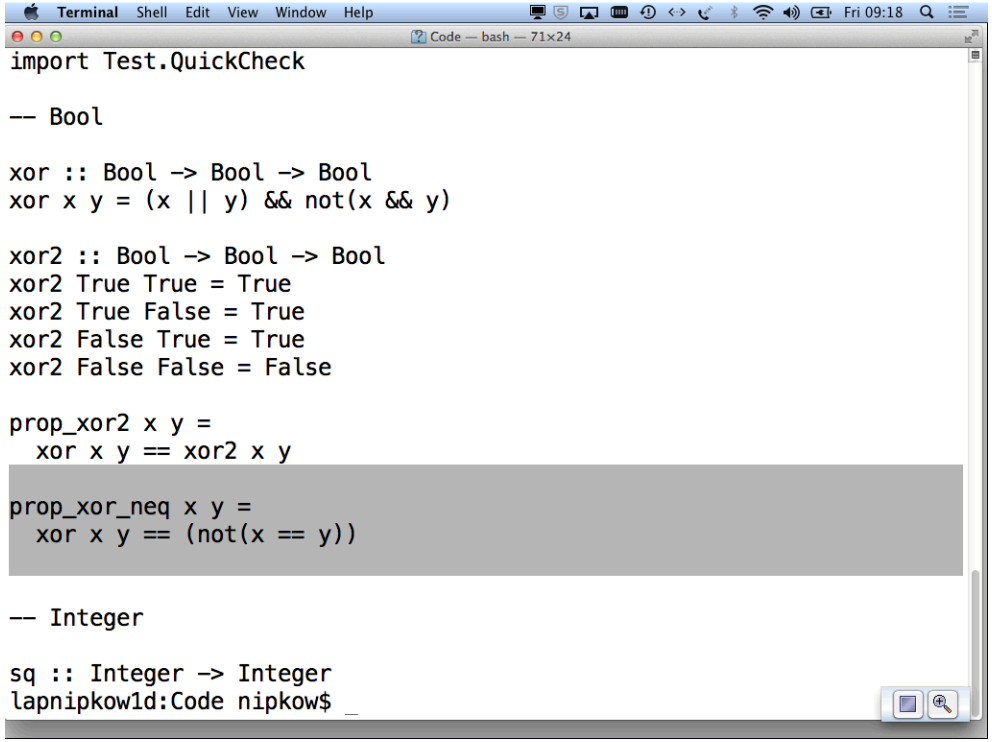

Terminal Shell Edit View Window Help Fri 09:14
Code — ghc — 71×24
xor2 :: Bool -> Bool -> Bool
xor2 True True = False
xor2 True False = True
xor2 False True = True
xor2 False False = False
prop_xor2 x y =
xor x y == xor2 x y
prop_xor_neq x y =
xor x y == (not(x == y))
-- Integer
sq :: Integer -> Integer
lapnipkow1d:Code nipkow$
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l V1.hs


Terminal Shell Edit View Window Help Fri 09:14
Code — ghc — 71×24
-- Integer
sq :: Integer -> Integer
lapnipkow1d:Code nipkow$
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l V1
[1 of 1] Compiling Main (V1.hs, interpreted)
Ok, modules loaded: Main.
*Main> xor True False
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.5.0.0 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.6 ... linking ... done.
True
*Main> xor True Tr


Terminal Shell Edit View Window Help Fri 09:15
Code — ghc — 71×24
sq :: Integer -> Integer
lapnipkow1d:Code nipkow$
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l V1
[1 of 1] Compiling Main (V1.hs, interpreted)
Ok, modules loaded: Main.
*Main> xor True False
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.5.0.0 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.6 ... linking ... done.
True
*Main> xor True True
False
*Main>


Terminal Shell Edit View Window Help Fri 09:15
Code — ghc — 71×24
lapnipkow1d:Code nipkow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l V1
[1 of 1] Compiling Main (V1.hs, interpreted)
Ok, modules loaded: Main.
*Main> xor True False
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.5.0.0 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.6 ... linking ... done.
True
*Main> xor True True
False
*Main> quickCheck prop_xor2
+++ OK, passed 100 tests.
*Main>





3.3 Type `Integer`

Unlimited precision mathematical integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

There is also the type `Int` of 32-bit integers.

3.3 Type `Integer`

Unlimited precision mathematical integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

There is also the type `Int` of 32-bit integers.
Warning: `Integer: 2 ^ 32 = 4294967296`

3.3 Type `Integer`

Unlimited precision mathematical integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

There is also the type `Int` of 32-bit integers.
Warning:
```
Integer:  2 ^ 32 = 4294967296
Int:      2 ^ 32 = 0
```

3.3 Type `Integer`

Unlimited precision mathematical integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

There is also the type `Int` of 32-bit integers.
Warning:
```
Integer:  2 ^ 32 = 4294967296
Int:      2 ^ 32 = 0
```

==, <= etc are overloaded and work on many types!

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3)
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
          = (3 * 3) * (3 * 3)
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
          = (3 * 3) * (3 * 3)
          = 81
```

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
          = (3 * 3) * (3 * 3)
          = 81
```

Evaluation of Haskell expressions
means
Using the defining equations from left to right.

3.4 Guarded equations

3.4 Guarded equations

Example: maximum of 2 integers.

```
max :: Integer -> Integer -> Integer
max x y
  | x >= y     = x
  | otherwise  = y
```

Haskell also has `if-then-else`:

```
max x y  =  if x >= y then x else y
```

3.4 Guarded equations

Example: maximum of 2 integers.

```
max :: Integer -> Integer -> Integer
max x y
  | x >= y     = x
  | otherwise  = y
```

3.4 Guarded equations

Example: maximum of 2 integers.

```
max :: Integer -> Integer -> Integer
max x y
  | x >= y     = x
  | otherwise  = y
```

Haskell also has `if-then-else`:

```
max x y  =  if x >= y then x else y
```

3.4 Guarded equations

Example: maximum of 2 integers.

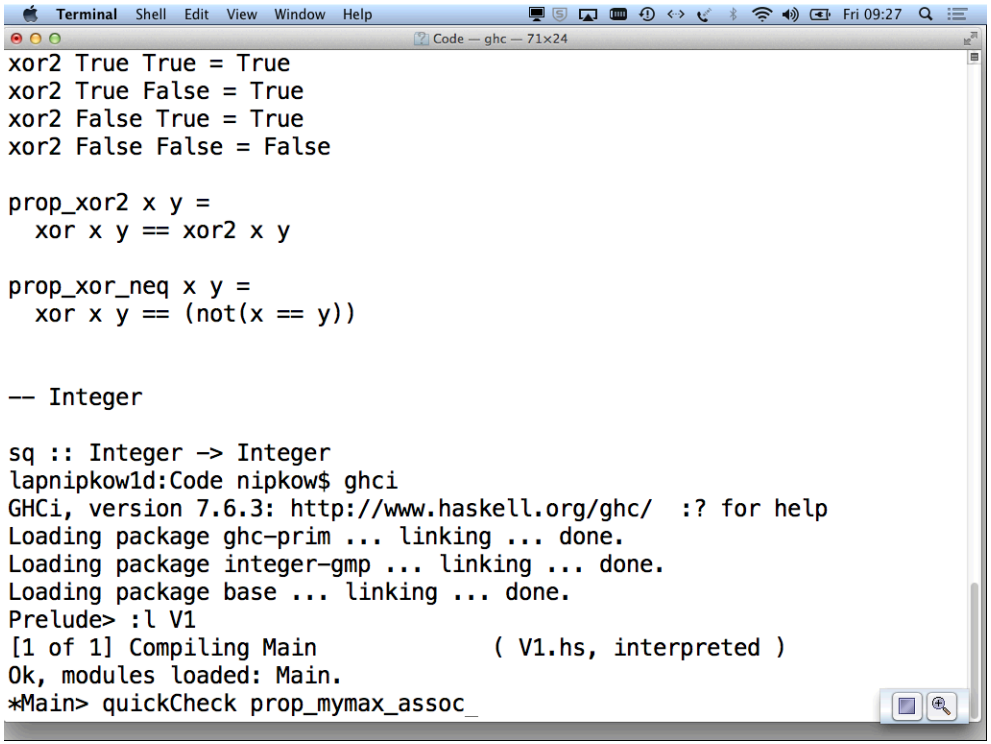
```
max :: Integer -> Integer -> Integer
max x y
  | x >= y     = x
  | otherwise  = y
```

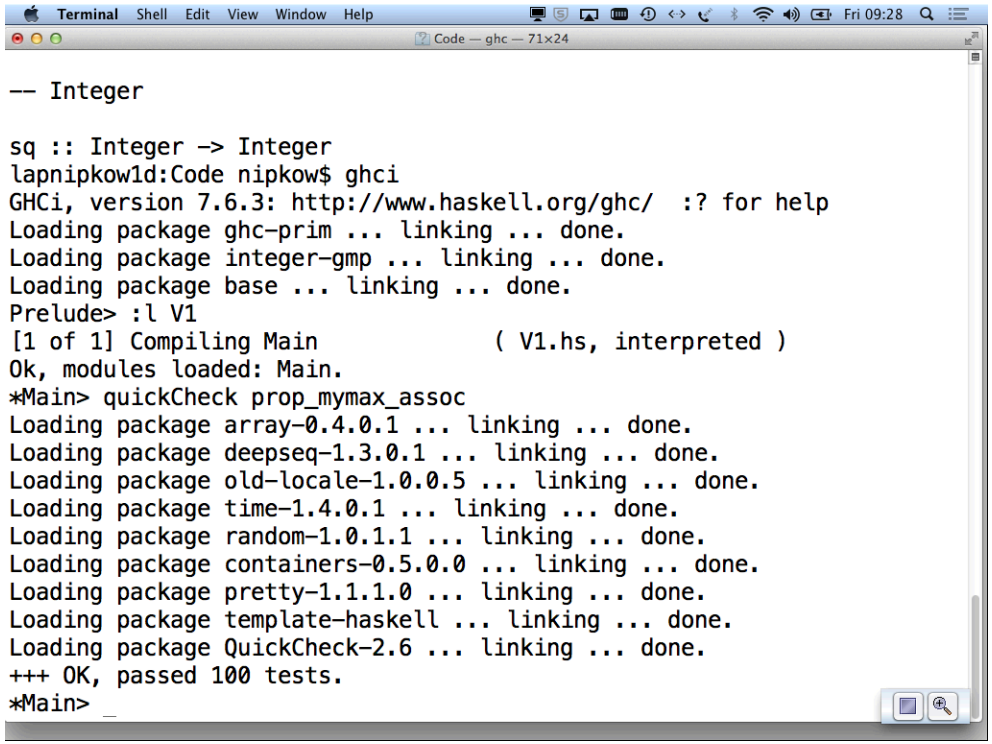
Haskell also has `if-then-else`:

```
max x y  =  if x >= y then x else y
```

True?

```
prop_max_assoc x y z =
  max x (max y z) == max (max x y) z
```





3.5 Recursion

Example: x^n (using only $*$, not $\hat{}$)

3.5 Recursion

Example: x^n (using only $*$, not $\hat{}$)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

3.5 Recursion

Example: x^n (using only $*$, not $\hat{}$)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
```

3.5 Recursion

Example: x^n (using only $*$, not $\hat{}$)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0  = 1                  -- the base case
```

3.5 Recursion

Example: x^n (using only $*$, not ^)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0  = 1                -- the base case
  | n > 0   = x * pow x (n-1)
```

3.5 Recursion

Example: x^n (using only $*$, not ^)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0  = 1                -- the base case
  | n > 0   = x * pow x (n-1)  -- the recursive case
```

3.5 Recursion

Example: x^n (using only $*$, not ^)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0  = 1                -- the base case
  | n > 0   = x * pow x (n-1)  -- the recursive case
```

More compactly:

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

Evaluating pow

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

```
pow 2 3
```

Evaluating pow

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

pow 2 3

Evaluating pow

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

pow 2 3 = 2 * pow 2 2

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

```
> pow 2 (-1)
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

```
> pow 2 (-1)
```

GHCi answers

```
*** Exception: PowDemo.hs:(1,1)-(2,33):
    Non-exhaustive patterns in function pow
```

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

```
> pow 2 (-1)
```

GHCi answers

```
*** Exception: PowDemo.hs:(1,1)-(2,33):
    Non-exhaustive patterns in function pow
```

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

Evaluating pow

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

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

```
> pow 2 (-1)
```

GHCi answers

```
*** Exception: PowDemo.hs:(1,1)-(2,33):
    Non-exhaustive patterns in function pow
```

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

In either case:

State your preconditions clearly!

As a guard, a comment

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

In either case:

State your preconditions clearly!

As a guard, a comment or using QuickCheck:

```
P x ==> isDefined(f x)
```

Partially defined functions

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

versus

```
pow x n  =  x * pow x (n-1)
```

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

In either case:

State your preconditions clearly!

As a guard, a comment or using QuickCheck:

```
P x ==> isDefined(f x)
```

where `isDefined y  =  y == y`.

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

```
sumTo :: Integer -> Integer
sumTo 0  =  0
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

```
sumTo :: Integer -> Integer
sumTo 0  =  0
sumTo n | n > 0  =
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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


prop_sumTo n =
             sumTo n ==
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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


prop_sumTo n =
             sumTo n ==
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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


prop_sumTo n =
             sumTo n == n*(n+1) `div` 2
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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


prop_sumTo n =
  n >= 0 ==> sumTo n == n*(n+1) `div` 2
```

Properties can be *conditional*

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                              -- base case
```

Example sumTo

The sum from 0 to n = n + (n-1) + (n-2) + ... + 0

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

```
prop_sumTo n =
  n >= 0 ==> sumTo n == n*(n+1) `div` 2
```

Properties can be *conditional*

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                              -- base case
```

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                              -- base case
f n | n > 0  =  ... f(n - 1) ...       -- recursive call(s)
```

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                              -- base case
f n | n > 0  =  ... f(n - 1) ...       -- recursive call(s)
```

Always make the base case as simple as possible,
typically 0, not 1

Many variations:

- more parameters

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                          -- base case
f n | n > 0  =  ... f(n - 1) ...   -- recursive call(s)
```

Always make the base case as simple as possible,
typically 0, not 1

Many variations:

- more parameters
- other base cases, e.g. `f 1`

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                          -- base case
f n | n > 0  =  ... f(n - 1) ...   -- recursive call(s)
```

Always make the base case as simple as possible,
typically 0, not 1

Many variations:

- more parameters
- other base cases, e.g. `f 1`
- other recursive calls, e.g. `f(n - 2)`

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                          -- base case
f n | n > 0  =  ... f(n - 1) ...   -- recursive call(s)
```

Always make the base case as simple as possible,
typically 0, not 1

Many variations:

- more parameters
- other base cases, e.g. `f 1`
- other recursive calls, e.g. `f(n - 2)`
- also for negative numbers

Recursion in general

- Reduce a problem to a *smaller* problem,
  e.g. `pow x n` to `pow x (n-1)`
- Must eventually reach a *base case*
- Build up solutions from smaller solutions

Recursion in general

- Reduce a problem to a *smaller* problem,
  e.g. `pow x n` to `pow x (n-1)`
- Must eventually reach a *base case*
- Build up solutions from smaller solutions

General problem solving strategy
in *any* programing language

3.6 Syntax matters

Functions are defined by one or more equations.
In the simplest case, each function is defined
by one (possibly conditional) equation:

$$
\begin{array}{l}
f\ x_1\ \ldots\ x_n \\
\quad |\ test_1\ =\ e_1 \\
\quad \vdots \\
\quad |\ test_n\ =\ e_n
\end{array}
$$

Each right-hand side e_i is an expression.

3.6 Syntax matters

Functions are defined by one or more equations.
In the simplest case, each function is defined
by one (possibly conditional) equation:

$$
\begin{array}{l}
f\ x_1\ \ldots\ x_n \\
\quad |\ test_1\ =\ e_1 \\
\quad \vdots \\
\quad |\ test_n\ =\ e_n
\end{array}
$$

Each right-hand side e_i is an expression.

Note: `otherwise = True`

Function and parameter names must begin with a lower-case letter

3.6 Syntax matters

Functions are defined by one or more equations.
In the simplest case, each function is defined
by one (possibly conditional) equation:

$$
\begin{array}{l}
f\ x_1\ \ldots\ x_n \\
\quad |\ test_1\ =\ e_1 \\
\quad \vdots \\
\quad |\ test_n\ =\ e_n
\end{array}
$$

Each right-hand side e_i is an expression.
Note: `otherwise = True`

Function and parameter names must begin with a lower-case letter