# Functional Programming with Haskell

CSC 372, Spring 2016
The University of Arizona
William H. Mitchell
**whm@cs**

# Programming Paradigms

# Paradigms

Thomas Kuhn's *The Structure of Scientific Revolutions* (1962) describes a *paradigm* as a scientific achievement that is...

- "...sufficiently unprecedented to attract an enduring group of adherents away from competing modes of scientific activity."

- "...sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to resolve."

Kuhn cites works such as Newton's *Principia*, Lavoisier's *Chemistry*, and Lyell's *Geology* as serving to document paradigms.

# Paradigms, continued

A paradigm provides a conceptual framework for understanding and solving problems.

A paradigm has a world view, a vocabulary, and a set of techniques that can be applied to solve a problem.

(Another theme for us.)

A question to keep in mind:

What are the problems that programming paradigms attempt to solve?

# The procedural programming paradigm

From the early days of programming into the 1980s the dominant paradigm was *procedural programming*:

> Programs are composed of bodies of code (procedures) that manipulate individual data elements or structures.

Much study was focused on how best to decompose a large computation into a set of procedures and a sequence of calls.

Languages like FORTRAN, COBOL, Pascal, and C facilitate procedural programming.

Java programs with a single class are typically examples of procedural programming.

# The object-oriented programming paradigm

In the 1990s, object-oriented programming became the dominant paradigm. Problems are solved by creating systems of objects that interact.

> "*Instead of a bit-grinding processor plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.*"—Dan Ingalls

Study shifted from how to decompose computations into procedures to how to model systems as interacting objects.

Languages like C++ and Java facilitate use of an object-oriented paradigm.

# The influence of paradigms

The programming paradigm(s) we know affect how we approach problems.

If we use the procedural paradigm, we'll first think about breaking down a computation into a series of steps.

If we use the object-oriented paradigm, we'll first think about modeling the problem with a set of objects and then consider their interactions.

# Language support for programming paradigms

If a language makes it easy and efficient to use a particular paradigm, we say that the language supports the paradigm.

What language features are required to support procedural programming?

- The ability to break programs into procedures.

What language features does OO programming require, for OO programming as you know it?

- Ability to define classes that comprise data and methods
- Ability to specify inheritance between classes

# Multiple paradigms

Paradigms in a field of science are often incompatible.

Example: geocentric vs. heliocentric model of the universe

Can a programming language support multiple paradigms?

Yes! We can do procedural programming with Java.

The programming language Leda fully supports the procedural, imperative, object-oriented, functional, and logic programming paradigms.

Wikipedia's **Programming_paradigm** cites 60+ paradigms!

But, are "programming paradigms" really paradigms by Kuhn's definition or are they just characteristics?

# The imperative programming paradigm

The imperative paradigm has its roots in programming at the machine level.

Machine-level programming:

- Instructions change memory locations or registers
- Instructions alter the flow of control

Programming with an imperative language:

- Expressions compute values based on memory contents
- Assignments alter memory contents
- Control structures guide the flow of control, perhaps iterating to accumulate a result.

# The imperative programming paradigm

Solutions using the procedural or object-oriented paradigms typically make use of the imperative programming paradigm, too.

Two fundamental characteristics of languages that support the imperative paradigm:

- "Variables"—data objects whose values typically change as execution proceeds.
- Support for iteration—a “while” control structure, for example.

# Imperative programming, continued

Here's an imperative solution in Java to sum the integers in an array:

```
int sum(int a[])
{
   int sum = 0;
   for (int i = 0; i < a.length; i++)
      sum += a[i];

   return sum;
}
```

The **for** loop causes **i** to vary over the indices of the array, as the variable **sum** accumulates the result.

How can the above solution be improved?

# Imperative programming, continued

With Java's "enhanced **for**", also known as a for-each loop, we can avoid array indexing.

```
int sum(int a[])
{
    int sum = 0;
    for (int val: a)
        sum += val;

    return sum;
}
```

Is this an improvement? If so, why?

Can we write **sum** in a non-imperative way?

# Imperative programming, continued

We can use recursion to get rid of loops and assignments, but...ouch!

```
int sum(int a[]) { return sum(a, 0); }

int sum(int a[], int i)
{
   if (i == a.length)
      return 0;
   else
      return a[i] + sum(a, i+1);
}
```

Wrt. correctness, which of the three versions would you bet your job on?

# The level of a paradigm

Programming paradigms can apply at different levels:

- Making a choice between procedural and object-oriented programming fundamentally determines the high-level structure of a program.

- The imperative paradigm is focused more on the small aspects of programming—how code looks at the line-by-line level.

Java combines the object-oriented and imperative paradigms.

The procedural and object-oriented paradigms apply to *programming in the large*.

The imperative paradigm applies to *programming in the small*.

# Background:
# Value, type, side effect

# Value, type, and side effect

An *expression* is a sequence of symbols that can be evaluated to produce a value.

Here are some Java expressions:

```
'x'
i + j * k
f(args.length * 2) + n
```

There are three questions that are commonly considered when looking at an expression in conventional languages like Java and C:

- What value does the expression produce?
- What's the type of that value?
- Does the expression have any side effects?

Mnemonic aid: Imagine you're wearing a vest that's reversed.
"vest" reversed is "t-se-v": type/side-effect/value.

# Value, type, and side effect, continued

What is the value of the following Java expressions?

```
3 + 4
```
7

```
1 < 2
```
true

```
"abc".charAt(1)
```
'b'

```
s = "3" + 4
```
"34"

```
"a,bb,c3".split(",")
```
An array with three elements: "a", "bb" and "c3"

```
"a,bb,c3".split(",")[2]
```
"c3"

```
"a,bb,c3".split(",")[2].charAt(0) == 'X'
```
false

# Value, type, and side effect, continued

What is the type of each of the following Java expressions?

```
3 + 4
```
int

```
1 < 2
```
boolean

```
"abc".charAt(1)
```
char

```
s = "3" + 4
```
String

```
"a,bb,c3".split(",")
```
String []

```
"a,bb,c3".split(",")[2]
```
String

```
"a,bb,c3".split(",")[2].charAt(0) == 'X'
```
boolean

When we ask,

"What's the type of this expression?"

we're actually asking this:

"What's the type of the value produced by this expression?"

# Value, type, and side effect, continued

A "side effect" is a change to the program's observable data or to the state of the environment in which the program runs.

Which of these Java expressions have a side effect?

`x + 3 * y`

*No side effect. A computation was done but no evidence of it remains.*

`x += 3 * y`

*Side effect:* `3 * y` *is added to* `x`.

`s.length() > 2 || s.charAt(1) == '#'`

*No side effect. A computation was done but no evidence of it remains.*

# Value, type, and side effect, continued

More expressions to consider wrt. side effects:

**"testing".toUpperCase()**

*A string "TESTING" was created somewhere but we can't get to it. No side effect.*

**L.add("x")**, where **L** is an **ArrayList**

*An element was added to **L**. Definitely a side-effect!*

**System.out.println("Hello!")**

*Side effect: "Hello!" went somewhere.*

**window.checkSize()**

*We can't tell without looking at **window.checkSize()**!*

# The hallmark of imperative programming

Side effects are the hallmark of imperative programing.

Programs written in an imperative style are essentially an orchestration of side effects.

Recall:

```
int sum = 0;
for (int i = 0; i < a.length; i++)
   sum += a[i];
```

Can we program without side effects?

# The Functional Paradigm

# The functional programming paradigm

A key characteristic of the functional paradigm is writing functions that are like pure mathematical functions.

Pure mathematical functions:

- Always produce the same value for given input(s)
- Have no side effects
- Can be easily combined to produce more powerful functions

Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions.

# Functional programming, continued

Other characteristics of the functional paradigm:

- Values are never changed but lots of new values are created.
- Recursion is used in place of iteration.
- Functions are values. Functions are put into data structures, passed to functions, and returned from functions. Lots of temporary functions are created.

Based on the above, how well would Java support functional programming? How about C?

# Haskell basics

# What is Haskell?

Haskell is a pure functional programming language; it has no imperative features.

Was designed by a committee with the goal of creating a standard language for research into functional programming.

First version appeared in 1990. Latest version is known as Haskell 2010.

Is said to be *non-strict*—it supports *lazy evaluation*.

It is not object-oriented in any way.

# Haskell resources

Website: **haskell.org**
All sorts of resources!

Books: (on Safari, too)
*Learn You a Haskell for Great Good!*, by Miran Lipovača
**http://learnyouahaskell.com** (Known as LYAH.)

*Programming in Haskell*, by Hutton
Note: See appendix B for mapping of non-ASCII chars!

*Real World Haskell*, by O'Sullivan, Stewart, and Goerzen
**http://realworldhaskell.org** (I'll call it RWH.)

Haskell 2010 Report (I'll call it H10.)
**http://haskell.org/definition/haskell2010.pdf**

# Interacting with Haskell

On lectura we can interact with Haskell by running **ghci**:

```
% ghci
GHCi, version 7.4.1: ...more...  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
>
```

With no arguments, **ghci** starts a read-eval-print loop (REPL)—expressions that we type at the prompt (>) are evaluated and the result is printed.

**Note**: the standard prompt is **Prelude>** but I've got

```
:set prompt "> "
```

in my **~/.ghci** file.

# Interacting with Haskell, continued

Let's try some expressions with **ghci**:

Go *live!*

```
> 3+4
7

> 3 * 4.5
13.5

> (3 > 4) || (5 < 7)
True

> 2 ^ 200
1606938044258990275541962092341162602522202993782792835301376

> "abc" ++ "xyz"
"abcxyz"
```

# Interacting with Haskell, continued

We can use **:help** to see available commands:

```
> :help
 Commands available from the prompt:
   <statement>                 evaluate/run <statement>
   :                           repeat last command
   :{\n ..lines.. \n:}\n       multiline command
   ...lots more...
```

# Interacting with Haskell, continued

The command **:set +t** causes types to be shown:

```
> :set +t
> 3+4
7
it :: Integer

> 3 == 4
False
it :: Bool
```

"**::**" is read as "has type". The value of the expression is "bound" to the name **it**.

Note that **:set +t** is not a Haskell expression—it's a command recognized by **ghci**.

# Interacting with Haskell, continued

We can use **it** in subsequent computations:

```
> 3+4
7
it :: Integer

> it + it * it
56
it :: Integer

> it /= it
False
it :: Bool
```

# Extra Credit Assignment 1

For two assignment points of extra credit:

1. Run **ghci** (or WinGHCi) somewhere and try ten Haskell expressions with some degree of variety. (Not just ten additions, for example!) Do a **:set +t** at the start.

2. Capture the output and put it in a plain text file, eca1.txt. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in eca1.txt with the following command:

```
% turnin 372-eca1 eca1.txt
```

Due: At the start of the next lecture after we hit this slide.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.

# Haskell version issues

lectura has version 2012.1.0.0 of the Haskell Platform, which has version 7.4.1 of **`ghci`** but the latest version is 7.10.3.

In 7.10.x a number of functions that operate on lists were switched to operating on "Foldable"s instead. IMO, this extra level of abstraction makes the language harder to learn, so I plan to avoid 7.10.x this semester.

If you want to install Haskell on your own machine, I recommend that you get the Haskell Platform 2014.2.0.0, which has version 7.8.3 of **`ghci`**. (URLs on next slide.)

As far as I know, there are no significant compatibility issues between 7.8.3 and lectura's 7.4.1 that will impact our usage.

# Haskell downloads

**https://www.haskell.org/platform/prior.html** has prior versions of the Haskell Platform.

Under **2014**, on the line **2014.2.0.0, August 2014 ⟹ ...**

For OS X, get "**Mac OS X, 64bit**".

For Windows, "**Windows, 32bit**" should be fine, but if you have trouble, (1) let us know and (2) go ahead and try the 64-bit version.

# The ~/.ghci file

When **ghci** starts up on Linux or OS X it looks for the file **~/.ghci** – a **.ghci** file in the user's home directory.

I have these two lines in my **~/.ghci** file on both my Mac and on lectura:

```
:set prompt "> "
:m +Text.Show.Functions
```

The first line simply sets the prompt to something I like.

*The second line is very important:*

It loads a module that allows functions to be printed as values, although just showing **<function>** for function values. Without it, lots of examples in these slides won't work!

# ~/.ghci, continued

Goofy fact: **~/.ghci** must not be group- or world-writable!

If you see something like this,

```
*** WARNING: /pl/hw/whm/.ghci is writable by
someone else, IGNORING!
```

Fix it at the shell prompt with this:

```
% chmod og-w ~/.ghci
```

Details on **.ghci** and lots more can be found in
downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

# ~/.ghci, continued

On Windows, **ghci** and WinGHCi use a different initialization file:

**%APPDATA%\ghc\ghci.conf**

(Note: the file is named **ghci.conf**, not **.ghci**!)

**%APPDATA%** represents the location of your **Application Data** directory. You can find that path by typing **set appdata** in a command window, like this:

```
C:\>set appdata
APPDATA=C:\Users\whm\Application Data
```

Combing the two, the full path to the file for me would be

**C:\Users\whm\Application Data\ghc\ghci.conf**

# Functions and function types

# Calling functions

In Haskell, *juxtaposition* indicates a function call:

```
> negate 3
-3
it :: Integer

> even 5
False
it :: Bool

> pred 'C'
'B'
it :: Char

> signum 2
1
it :: Integer
```

Note: These functions and many more are defined in the Haskell "Prelude", which is loaded by default when **ghci** starts up.

# Calling functions, continued

Function call with juxtaposition is left-associative.

**signum negate 2** means **(signum negate) 2**

```
> signum negate 2
<interactive>:40:1:    -- It's an error!
   No instance for (Num (a0 -> a0)) arising from a
use of `signum'
...
```

We add parentheses to call **negate 2** first:

```
> signum (negate 2)
-1
it :: Integer
```

# Calling functions, continued

Function call with juxtaposition has higher precedence than any operator.

```
> negate 3+4
1
it :: Integer
```

negate 3 + 4 means (negate 3) + 4. Use parens to force + first:

```
> negate (3 + 4)
-7
it :: Integer

> signum (negate (3 + 4))
-1
it :: Integer
```

# Function types

Haskell's **Data.Char** module has a number of functions for working with characters. We'll use it to start learning about function types.

```
> :m +Data.Char
```

*(:m(odule) loads a module)*

```
> isLower 'b'
True
it :: Bool

> toUpper 'a'
'A'
it :: Char

> ord 'A'
65
it :: Int

> chr 66
'B'
it :: Char
```

We can also reference a function in a module with a *qualified name*:

```
% ghci
...
> Data.Char.ord 'G'
71
```

# Function types, continued

We can use **ghci**'s **:type** command to see what the type of a function is:

```
> :type isLower
isLower :: Char -> Bool   (read -> as "to")
```

The type **Char -> Bool** means that **isLower** is a function that takes an argument of type **Char** and produces a result of type **Bool.**

Using **ghci**, what are the types of **toUpper**, **ord**, and **chr**?

We can use **:browse Data.Char** to see everything in the module.

# Type consistency

Like most languages, Haskell requires that expressions be *type-consistent* (or *well-typed*).

Here is an example of an inconsistency:

```
> chr 'x'
<interactive>:32:5:
  Couldn't match expected type Int with actual type Char
  In the first argument of `chr', namely 'x'

> :type chr
chr :: Int -> Char

> :type 'x'
'x' :: Char
```

**chr** requires its argument to be an **Int** but we gave it a **Char**. We can say that **chr 'x'** is *ill-typed*.

# Type consistency, continued

State whether each expression is well-typed and if so, its type.

```
'a'
isUpper
isUpper 'a'
not (isUpper 'a')
not not (isUpper 'a')
toUpper (ord 97)
isUpper (toUpper (chr 'a'))
isUpper (intToDigit 100)
```

```
'a' :: Char
chr :: Int -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
isUpper :: Char -> Bool
not :: Bool -> Bool
ord :: Char -> Int
toUpper :: Char -> Char
```

# Sidebar: Key bindings in `ghci`

`ghci` uses the `haskeline` package to provide line-editing.

A few handy bindings:

| | |
|---|---|
| `TAB` | completes identifiers |
| `^A` | Start of line |
| `^E` | End of line |
| `^R` | Incremental search backwards |

More:

`http://trac.haskell.org/haskeline/wiki/KeyBindings`

# Sidebar: Using a REPL to help learn a language

As we've seen, **ghci** provides a REPL (read-eval-print loop) for Haskell.

What are some other languages that have a REPL available?

How does a REPL help us learn a language?

Is there a REPL for Java?

**javarepl.com**

What characteristics does a language need to support a REPL?

If there's no REPL for a language, how hard is it to write one?

# Type classes

# What's the type of **negate**?

Recall the **negate** function:

```
> negate 5
-5
it :: Integer

> negate 5.0
-5.0
it :: Double
```

What's the type of **negate**? (Is it both **Integer -> Integer** and **Double -> Double**??)

# Type classes

**Bool**, **Char**, and **Integer** are examples of Haskell types.

Haskell also has *type classes*. A type class specifies the operations must be supported on a type in order for that type to be a member of that type class.

**Num** is one of the many type classes defined in the Prelude.

**:info Num** shows that for a type to be a **Num,** it must support addition, subtraction, multiplication and four functions: **negate**, **abs**, **signNum**, and **fromInteger**. (The **Num** club!)

The Prelude defines four *instances* of the **Num** type class: **Int** (word-size), **Integer** (unlimited size), **Float** and **Double**.

# Type classes, continued

Here's the type of **negate:**

```
> :type negate
negate :: Num a => a -> a
```

The type of **negate** is specified using a *type variable*, **a**.

The portion **a -> a** specifies that **negate** returns a value having the same type as its argument.

*"If you give me an **Int**, I'll give you back an **Int**."*

The portion **Num a =>** is a *class constraint*. It specifies that the type **a** must be an instance of the type class **Num**.

How can we state the type of **negate** in English?

***negate** accepts any value whose type is an instance of **Num**. It returns a value of the same type.*

# Type classes, continued

What type do integer literals have?

```
> :type 3
3 :: Num a => a

> :type (-27)              -- Note: Parens needed!
(-27) :: Num a => a
```

Literals are typed with a class constraint of **Num**, so they can be used by any function that accepts **Num a => a**.

# Type classes, continued

Let's check the type of a decimal fraction:

```
> :type 3.4
3.4 :: Fractional a => a
```

Will **negate 3.4** work?

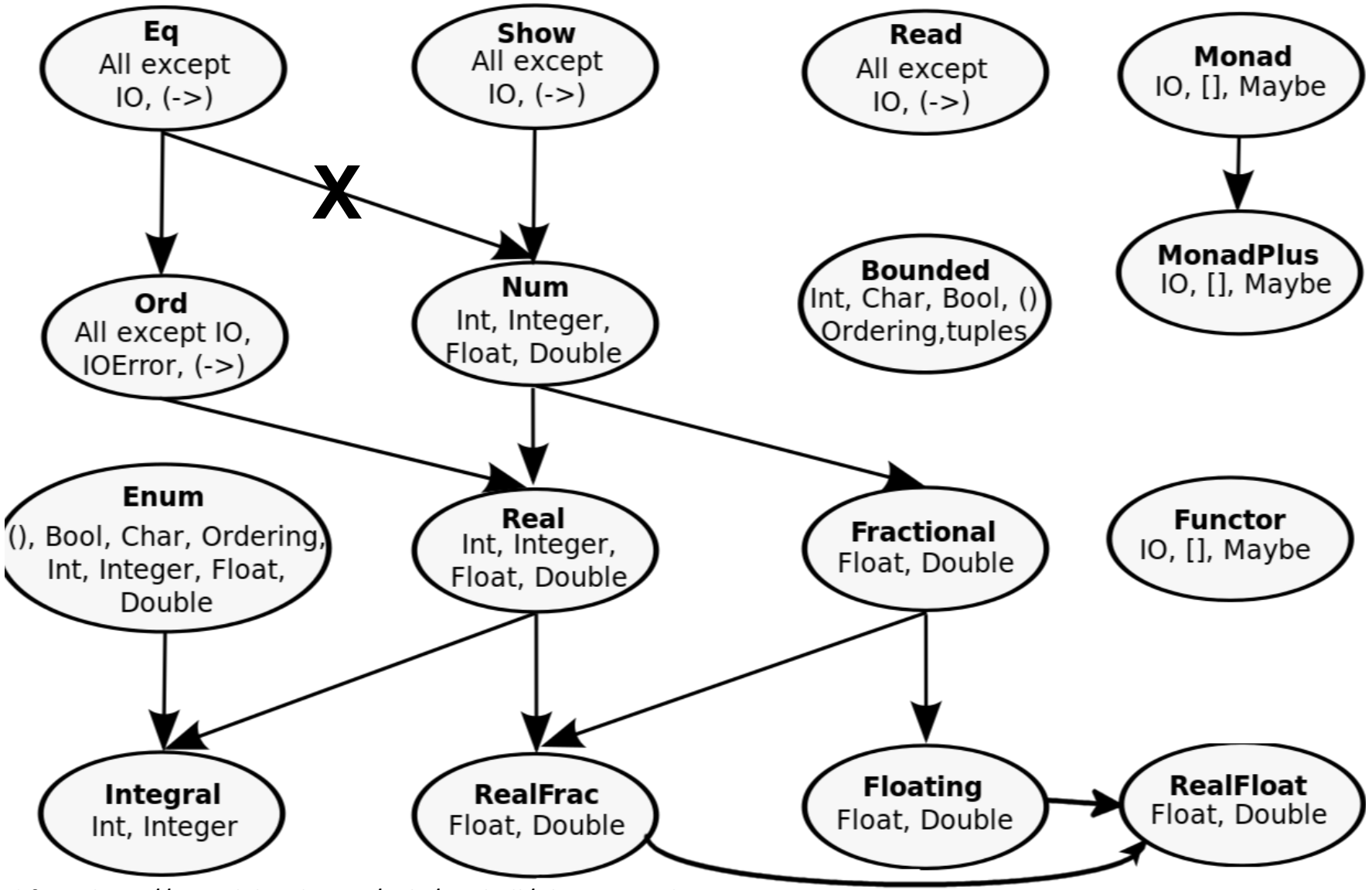
```
> :type negate
negate :: Num a => a -> a

> negate 3.4
-3.4
```

Speculate: Why does it work?

# Type classes, continued

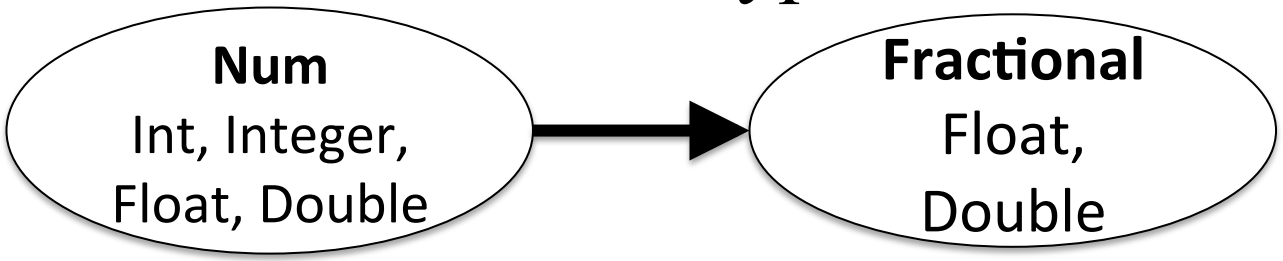
Haskell type classes form a hierarchy. The Prelude has these:




Adapted from http://en.wikibooks.org/wiki/Haskell/Classes_and_types

# Type classes, continued

Excerpt:




The arrow from **Num** to **Fractional** means that a **Fractional** can be used as a **Num**. (What does that remind you of?)

Given

```
negate :: Num a => a -> a
```

and

```
5.0 :: Fractional a => a
```

then

`negate 5.0` is valid.

Note that the bubbles also show the types that are instances of the type class. (Do **:info Num** again, and **:info Fractional**, too.)

# Type classes, continued

What's meant by the type of **truncate**?

```
truncate :: (Integral b, RealFrac a) => a -> b
```

**truncate** accepts a type whose type class is an instance of **RealFrac** but produces a type whose type class is an instance of **Integral**.

LYAH pp. 27-33 has a good description of the Prelude's type classes. ("Type Classes 101")

Note that type classes are not required for functional programming but because Haskell makes extensive use of them, we must learn about them.

## `negate` is *polymorphic*

In essence, `negate :: Num a => a -> a` describes many functions:

```
negate :: Integer -> Integer
negate :: Int -> Int
negate :: Float -> Float
negate :: Double -> Double
...and more...
```

`negate` is a *polymorphic function*. It handles values of many forms.

If a function's type has any type variables, it's a polymorphic function.

How does Java handle this problem? How about C? C++?

# Sidebar: LHtLaL—introspective tools

**:set +t**, **:type** and **:info** are three introspective tools that we can use to help learn Haskell.

When learning a language, look for such tools early on.

Some type-related tools in other languages:

Python: **type(*expr*)** and **repr(*expr*)**

JavaScript: **typeof(expr)**

PHP: **var_dump(*expr1*, *expr2*, ...)**

C: **sizeof(*expr*)**

Java: **getClass()**

What's a difference between **ghci**'s **:type** and Java's **getClass()**?

# Sidebar, continued

Here's a Java program that makes use of the "boxing" mechanism to show the type of values, albeit with wrapper types for primitives.

```
public class exprtype {
   public static void main(String args[]) {
      int n = 1;
      showtype(n++, 3 + 'a');
      showtype(n++, 3 + 4.0);
      showtype(n++, "a,b,c".split(","));
      showtype(n++, new HashMap<String,Integer>());
   }
   private static void showtype(int num, Object o) {
      System.out.format("%d: %s\n", num, o.getClass());
   }
}
```

Output:

```
1: class java.lang.Integer
2: class java.lang.Double
3: class [Ljava.lang.String;
4: class java.util.HashMap
```

*(Note: no **String** or **Integer**—type erasure!)*

# More on functions

# Writing simple functions

A function can be defined in the REPL by using **let**. Example:

```
> let double x = x * 2
double :: Num a => a -> a

> double 5
10
it :: Integer

> double 2.7
5.4
it :: Double

> double (double (double 111111111111))
888888888888
it :: Integer
```

# Simple functions, continued

More examples:

```
> let neg x = -x
neg :: Num a => a -> a

> let isPositive x = x > 0
isPositive :: (Num a, Ord a) => a -> Bool

> let toCelsius temp = (temp - 32) * 5/9
toCelsius :: Fractional a => a -> a
```

The determination of types based on the operations performed is known as *type inferencing*. (More on it later!)

Note: function and parameter names must begin with a lowercase letter or _. (If capitalized they're assumed to be *data constructors*.)

# Simple functions, continued

We can use :: *type* to constrain a function's type:

```
> let neg x = -x :: Integer
neg :: Integer -> Integer

> let toCelsius temp = (temp - 32) * 5/9 :: Double
toCelsius :: Double -> Double
```

:: *type* has low precedence; parentheses are required for this:

```
> let isPositive x = x > (0::Integer)
isPositive :: Integer -> Bool
```

Note that :: *type* applies to an expression, not a function.

We'll use :: *type* to simplify some following examples.

# Sidebar: loading functions from a file

We can put function definitions in a file. When we do, **we leave off the let**!

I've got four function definitions in the file **simple.hs**, as shown with the UNIX **cat** command:

```
% cat simple.hs
double x = x * 2 :: Integer   -- Note: no "let"!
neg x = -x :: Integer
isPositive x = x > (0::Integer)
toCelsius temp = (temp - 32) * 5/(9::Double)
```

The **.hs** suffix is required.

# Sidebar, continued

Assuming **simple.hs** is in the current directory, we can load it with **:load** and see what we got with **:browse**.

```
% ghci
> :load simple
[1 of 1] Compiling Main             ( simple.hs, interpreted )
Ok, modules loaded: Main.

> :browse
double :: Integer -> Integer
neg :: Integer -> Integer
isPositive :: Integer -> Bool
toCelsius :: Double -> Double
```

Note the colon in **:load**, and that the suffix **.hs** is assumed.

We can use a path, like **:load ~/372/hs/simple**, too.

# Sidebar: My usual edit-run cycle

**ghci** is clumsy to type! I've got an **hs** alias in my **~/.bashrc**:

```
alias hs=ghci
```

I specify the file I'm working with as an argument to **hs**.

```
% hs simple
GHCi, version 7.8.3 ...
[1 of 1] Compiling Main             ( simple.hs, interpreted )
Ok, modules loaded: Main.
> ... experiment ...
```

After editing in a different window, I use **:r** to reload the file.

```
> :r
[1 of 1] Compiling Main             ( simple.hs, interpreted )
Ok, modules loaded: Main.
> ...experiment some more...
```

Lather, rinse, repeat.

# Functions with multiple arguments

Here's a function that produces the sum of its two arguments:

```
> let add x y = x + y :: Integer
```

Here's how we call it: (no commas or parentheses!)

```
> add 3 5
8
```

Here is its type:

```
> :type add
add :: Integer -> Integer -> Integer
```

The operator -> is right-associative, so the above means this:

```
add :: Integer -> (Integer -> Integer)
```

But what does that mean?

# Multiple arguments, continued

Recall our negate function:

```
> let neg x = -x :: Integer
neg :: Integer -> Integer
```

Here's **add** again, with parentheses added to show precedence:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)
```

**add** is a function that takes an integer as an argument and produces a function as its result!

**add 3 5** means **(add 3) 5**

Call **add** with the value 3, producing a nameless function.
Call that nameless function with the value 5.

# Partial application

When we give a function fewer arguments than it requires, the resulting value is called a *partial application*. It is a function.

We can bind a name to a partial application like this:

```
> let plusThree = add 3
plusThree :: Integer -> Integer
```

The name **plusThree** now references a function that takes an **Integer** and returns an **Integer**.

What will **plusThree 5** produce?

```
> plusThree 5
8
it :: Integer
```

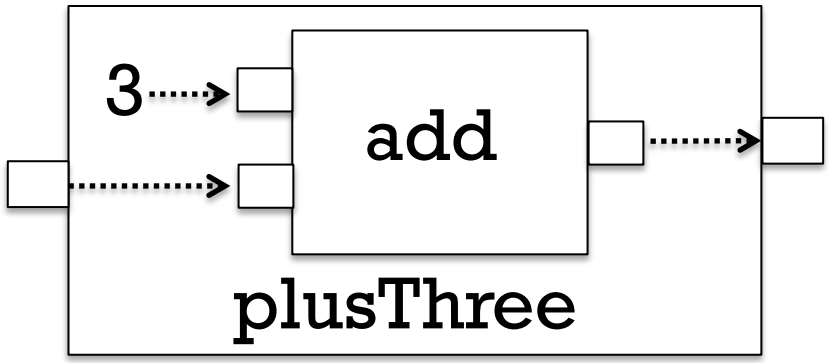
# Partial application, continued

At hand:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)  -- parens added

> let plusThree = add 3
plusThree :: Integer -> Integer
```

Let's picture **add** and **plusThree** as boxes with inputs and outputs:




An analogy: **plusThree** is like a calculator where you've clicked 3, then +, and handed it to somebody.

# Partial application, continued

At hand:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)  -- parens added
```

Another: *(with parentheses added to type to aid understanding)*

```
> let add3 x y z = x + y + z :: Integer
add3 :: Integer -> (Integer -> (Integer -> Integer))
```

These functions are said to be defined in *curried* form, which allows partial application of arguments.

The idea of a partially applicable function was first described by Moses Schönfinkel. It was further developed by Haskell B. Curry. Both worked wtih David Hilbert in the 1920s.

$log_2 n$

What prior use have you made of partially applied functions?

**Note: Next set of slides!**

# REPLACEMENTS

Put a big "X" on slides 74-76 in the 1-76 set and continue with this set.

# Some key points

- The *general form* of a function definition (for now):
  ***name param1 param2 ... paramN = expression***
  —At the **ghci** prompt, use **let ...**

- A function with a type like **Integer -> Char -> Char** takes two arguments, an **Integer** and a **Char**. It produces a **Char**.

- Remember that -> is a right-associative type operator.
  **Integer -> Char -> Char** means **Integer -> (Char -> Char)**

- A function call like
  **f x y z**
  means
  **((f x) y) z**
  and (conceptually) causes two temporary, unnamed functions to be created.

# Some key points, continued

- Calling a function with fewer arguments than it requires creates a *partial application*, a function value.

- There's really nothing special about a partial application—it's just another function.

# Another view of partial application

Consider this function:

```
let f x y z = x + y + y * z

let f1 = f 3
```

is equivalent to

```
let f1 y z = 3 + y + y * z

let f2 = f1 5
```

is equivalent to

```
let f2 z = 3 + 5 + 5 * z

let val = f2 7
```

is equivalent to

```
let val = f 3 5 7
```

and

```
let val = f1 5 7
```

One way to think of partial application is that as each argument is provided, a parameter is dropped and the argument's value is "wired" into the expression for the function, producing a new function with one less parameter.

# Exercise

Add parentheses to show the order of operations for the following expression:

```
f g 3 4 + x f 3 g(5*x)
```

Note that the expression is function calls and an addition.

Recall that function call is the highest precedence operation and is left-associative.

Let's first note that the addition is lowest precedence, and last:

```
(f g 3 4) + (x f 3 g (5*x))
```

Let's now reflect the left-associativity of function call:

```
(((f g) 3) 4) + ((((x f) 3) g) (5*x))
```

# Exercise

Problem: Define a function **min3** that computes the minimum of three values. The Prelude has a **min** function.

```
> min3 5 2 10
2
```

Solution:

```
> let min3 a b c = min a (min b c)
min3 :: Ord a => a -> a -> a -> a
```

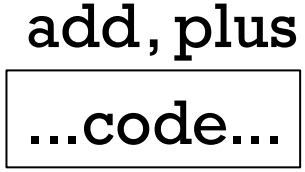
What are some types that **min3** can be used with?

# Functions are values

A fundamental characteristic of a functional language: functions are values that can be used as flexibly as values of other types.

This **let** creates a function value and binds the name **add** to it.

```
> let add x y = x + y
```




This **let** binds the name **plus** to the value of **add**, whatever it is.

```
> let plus = add
```

(Diagram here merged w/ above)

Either name can be used to reference the function value:

```
> add 3 4
7
> plus 5 6
11
```

# Functions as values, continued

What does the following suggest to you?

```
> :info add
add :: Num a => a -> a -> a

> :info +
class Num a where
  (+) :: a -> a -> a
  ...
infixl 6 +
```

Operators in Haskell are simply functions that have a symbolic name bound to them.

**infixl 6 +** shows that the symbol + can be used as a infix operator that is left associative and has precedence level 6.

Use **:info** to explore these operators: ==, >, +, *, | |, ^, ^^ and **.

# Function/operator equivalence

To use an operator like a function, enclose it in parentheses:

```
> (+) 3 4
7
```

Conversely, we can use a function like an operator by enclosing it in backquotes:

```
> 3 `add` 4
7

> 11 `rem` 3
2
```

Speculate: do `` `add` `` and `` `rem` `` have precedence and associativity?

# Sidebar: Custom operators

Haskell lets us define custom operators.

Example: (loaded from a file)

```
(+%) x percentage = x + x * percentage / 100
infixl 6 +%
```

Usage:

```
> 100 +% 1
101.0
> 12 +% 25
15.0
```

The characters ! # $ % & * + . / < = > ? @ \ ^ | - ~ : and non-ASCII Unicode symbols can be used in custom operators.

Modules often define custom operators.

# Reference: Operators from the Prelude

| Precedence | Left associative operators | Non associative operators | Right associative operators |
|---|---|---|---|
| 9 | !! | | . |
| 8 | | | ^, ^^, ** |
| 7 | *, /, `div`, `mod`, `rem`, `quot` | | |
| 6 | +, - | | |
| 5 | | | :, ++ |
| 4 | | ==, /=, <, <=, >, >=, `elem`, `notElem` | |
| 3 | | | && |
| 2 | | | \|\| |
| 1 | >>, >>= | | |
| 0 | | | $, $!, `seq` |

Note: From page 51 in Haskell 2010 report

# Type Inferencing

# Type inferencing

It was briefly mentioned that Haskell performs type infererencing: the types of values are inferred based on the operations performed.

Example:

```
> let isCapital c = c >= 'A' && c <= 'Z'
isCapital :: Char -> Bool
```

Because `c` is being compared to `'A'` and `'Z'`, both of which are type `Char`, `c` is inferred to be a `Char`.

# Type inferencing, continued

Recall **ord** in the **Data.Char** module:

```
> :t ord
ord :: Char -> Int
```

What type will be inferred for the following function?

```
f x y = ord x == y
```

1. The argument of **ord** is a **Char**, so **x** must be a **Char**.

2. The result of **ord**, an **Int**, is compared to **y**, so **y** must be an **Int**.

Let's try it:

```
> let f x y = ord x == y
f :: Char -> Int -> Bool
```

# Type inferencing, continued

Recall this example:

```
> let isPositive x = x > 0
isPositive :: (Num a, Ord a) => a -> Bool
```

**:info** shows that **>** operates on types that are instances of **Ord**:

```
> :info >
class Eq a => Ord a where
  (>) :: a -> a -> Bool
  ...
```

Because **x** is an operand of **>**, Haskell infers that the type of **x** must be a member of the **Ord** type class.

Because **x** is being compared to 0, Haskell also infers that the type of **x** must be a member of the **Num** type class.

# Type inferencing, continued

If a contradiction is reached during type inferencing, it's an error.

The function below uses **x** as both a **Num** and a **Char**.

```
> let g x y = x > 0 && x > '0'

<interactive>:20:17:
    No instance for (Num Char) arising from the literal `0'
    Possible fix: add an instance declaration for (Num Char)
    In the second argument of `(>)', namely `0'
    In the first argument of `(&&)', namely `x > 0'
    In the expression: x > 0 && x > '0'
```

Note that Haskell's suggested fix, making **Char** be an instance of the **Num** type class, isn't very good.

# Type Specifications

# Type specifications for functions

It's a good practice to specify the type of a function along with its definition in a file.

Examples, using **cat** to make it clear that they're in a file:

```
% cat typespecs.hs
min3::Ord a => a -> a -> a
min3 x y z = min x (min y z)

isCapital :: Char -> Bool
isCapital c = c >= 'A' && c <= 'Z'

isPositive :: (Num a, Ord a) => a -> Bool
isPositive x = x > 0
```

# Type specifications, continued

Sometimes type specifications can backfire. What's the ramification of the difference in these two type specifications?

```
add1::Num a => a -> a -> a
add1 x y = x + y

add2::Integer -> Integer -> Integer
add2 x y = x + y
```

`add1` can operate on **Num**s but `a2` requires **Integer**s.

Challenge: Without using *::type*, show an expression that works with `add1` but fails with `add2`.

# Type specification for functions, continued

There are two pitfalls for Haskell novices related to type specifications for functions:

1. Specifying a type, such as **`Integer`**, rather than a type class, such as **`Num`**, may make a function's type needlessly specific, like **`add2`** on the previous slide.

2. In some cases the type can be plain wrong without the mistake being obvious, leading to a baffling problem. (An "Ishihara".)

Recommendation:

Try writing functions without a type specification and see what type gets inferred. If the type looks reasonable, and the function works as expected, add a specification for that type.

Type specifications can prevent Haskell's type inferencing mechanism from making a series of bad inferences that lead one far away from the actual source of an error.

# Continuation with indentation

A Haskell source file is a series of *declarations*. Here's a file with two declarations:

```
% cat indent1.hs
add::Integer -> Integer -> Integer
add x y = x + y
```

A declaration can be continued across multiple lines by indenting subsequent lines more than the first line of the declaration. These weaving declarations are poor style but are valid:

```
add
      ::
  Integer-> Integer-> Integer
add x y
    =
  x
     + y
```

# Indentation, continued

A line that starts in the same column as the previous declaration ends that previous declaration and starts a new one.

```
% cat indent2.hs
add::Integer -> Integer -> Integer
add x y =
x + y

% ghci indent2
...
indent2.hs:3:1:
    parse error (possibly incorrect indentation or
mismatched brackets)
Failed, modules loaded: none.
```

Note that 3:1 indicates line 3, column 1.

# Guards

# Guards

Recall this characteristic of functional programming:

> "Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions."

This function definition uses *guards* to specify three cases:

```
sign x | x < 0  = -1
       | x == 0 = 0
       | otherwise = 1
```

Notes:

- No **let**—this definition is loaded from a file with **:load**
- **sign x** appears just once. First guard might be on next line.
- The *guard* appears between **|** and **=**, and produces a **Bool**
- What is **otherwise**?

# Guards, continued

Problem: Using guards, define a function **smaller**, like **min**:

```
> smaller 7 10
7

> smaller 'z' 'a'
'a'
```

Solution:

```
smaller x y
    | x <= y = x
    | otherwise = y
```

# Guards, continued

Problem: Write a function **weather** that classifies a given temperature as hot if 80+, else nice if 70+, and cold otherwise.

```
> weather 95
"Hot!"
> weather 32
"Cold!"
> weather 75
"Nice"
```

A solution that takes advantage of the fact that guards are tried in turn:

```
weather temp | temp >= 80 = "Hot!"
             | temp >= 70 = "Nice"
             | otherwise = "Cold!"
```

# if-else

# Haskell's **if-else**

Here's an example of Haskell's **if-else**:

```
> if 1 < 2 then 3 else 4
3
```

How does this compare to the **if-else** in Java?

# Sidebar: Java's **if-else**

Java's **if-else** is a statement. It cannot be used where a value is required.

Java's conditional operator is the analog to Haskell's **if-else**.

```
1 < 2 ? 3 : 4
```
(Java conditional, a.k.a ternary operator)

It's an expression that can be used when a value is required.

Java's if-else statement has an else-less form but Haskell's **if-else** does not. Why doesn't Haskell allow it?

Java's **if-else** vs. Java's conditional operator provides a good example of a *statement* vs. an *expression*.

Pythoners: Is there an **if-else** expression in Python?

```
3 if 1 < 2 else 4
```

# Haskell's **if-else**, continued

What's the type of these expressions?

```
> :type if 1 < 2 then 3 else 4
if 1 < 2 then 3 else 4 :: Num a => a

> :type if 1 < 2 then '3' else '4'
if 1 < 2 then '3' else '4' :: Char

> if 1 < 2 then 3 else '4'
  <interactive>:12:15:
   No instance for (Num Char) arising from the literal `3'

> if 1 < 2 then 3
   <interactive>:13:16:
   parse error (possibly incorrect indentation or
   mismatched brackets)
```

# Guards vs. **if-else**

Which of the versions of **sign** below is better?

```
sign x
    | x < 0 = -1
    | x == 0 = 0
    | otherwise = 1
```

```
sign x = if x < 0 then -1
                  else if x == 0 then 0
                                 else 1
```

We'll later see that *patterns* add a third possibility for expressing cases.

# A Little Recursion

# Recursion

A recursive function is a function that calls itself either directly or indirectly.

Computing the factorial of a integer (N!) is a classic example of recursion. Write it in Haskell (and don't peek below!) What is its type?

```
factorial n
    | n == 0 = 1          -- Base case, 0! is 1
    | otherwise = n * factorial (n - 1)

> :type factorial
factorial :: (Eq a, Num a) => a -> a

> factorial 40
815915283247897734345611269596115894272000000000
```

# Recursion, continued

One way to manually trace through a recursive computation is to underline a call, then rewrite the call with a textual expansion.

```
factorial n
    | n == 0 = 1
    | otherwise = n * factorial (n – 1)
```

factorial 4

4 * factorial 3

4 * 3 * factorial 2

4 * 3 * 2 * factorial 1

4 * 3 * 2 * 1 * factorial 0

4 * 3 * 2 * 1 * 1

# Recursion, continued

Consider repeatedly dividing a number until the quotient is 1:

```
> 28 `quot` 3      (Note backquotes to use quot as infix op.)
9
> it `quot` 3      (Remember that it is previous result.)
3
> it `quot` 3
1
```

Problem: Write a recursive function **numDivs divisor x** that computes the number of times **x** must be divided by **divisor** to reach a quotient of 1.

```
> numDivs 3 28
3
> numDivs 2 7
2
```

# Recursion, continued

A solution:

```
numDivs divisor x
    | (x `quot` divisor) < 1 = 0
    | otherwise =
                1 + numDivs divisor (x `quot` divisor)
```

Example:

```
> numDivs 3 28
3
```

What is its type?

```
numDivs :: (Integral a, Num a1) => a -> a -> a1
```

Will `numDivs 2 3.4` work?

```
> numDivs 2 3.4
<interactive>:93:1:
   No instance for (Integral a0) arising from a use of
`numDivs'
```

# Sidebar: Fun with partial applications

Let's compute two partial applications of **numDivs**, using **let** to bind them to identifiers:

```
> let f = numDivs 2
> let g = numDivs 10
> f 9
3
> g 1001
3
```

What are more descriptive names than **f** and **g**?

```
> let floor_log2 = numDivs 2
> floor_log2 1000
9

> let floor_log10 = numDivs 10
> floor_log10 1000
3
```

# Lists

# List basics

In Haskell, a list is a sequence of values of the same type.

Here's one way to make a list. Note the type of **it** for each.

```
> [7, 3, 8]
[7,3,8]
it :: [Integer]

> [1.3, 10, 4, 9.7]  -- note mix of literals
[1.3,10.0,4.0,9.7]
it :: [Double]

> ['x', 10]
<interactive>:20:7:
   No instance for (Num Char) arising from the literal `10'
```

It is said that lists in Haskell are homogeneous.

# List basics, continued

The function **length** returns the number of elements in a list:

```
> length [3,4,5]
3

> length []
0
```

What's the type of **length**?

```
> :type length
length :: [a] -> Int
```

With no class constraint specified, **[a]** indicates that **length** operates on lists containing elements of any type.

# List basics, continued

The **head** function returns the first element of a list.

```
> head [3,4,5]
3
```

What's the type of **head**?

```
head :: [a] -> a
```

Here's what **tail** does. How would you describe it?

```
> tail [3,4,5]
[4,5]
```

What's the type of **tail**?

```
tail :: [a] -> [a]
```

Important: **head** and **tail** are good for learning about lists but we'll almost always use patterns to access list elements!

# List basics, continued

The ++ operator concatenates two lists, producing a new list.

```
> [3,4] ++ [10,20,30]
[3,4,10,20,30]

> it ++ it
[3,4,10,20,30,3,4,10,20,30]

> let f = (++) [1,2,3]
> f [4,5]
[1,2,3,4,5]

> f [4,5] ++ reverse (f [4,5])
[1,2,3,4,5,5,4,3,2,1]
```

What are the types of ++ and **reverse**?

```
> :type (++)
(++) :: [a] -> [a] -> [a]

> :type reverse
reverse :: [a] -> [a]
```

# List basics, continued

A range of values can be specified with a dot-dot notation:

```
> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
it :: [Integer]

> [-5,-3..20]
[-5,-3,-1,1,3,5,7,9,11,13,15,17,19]

> length [-1000..1000]
2001

> [10..5]
[]
it :: [Integer]
```

This is known as the *arithmetic sequence notation*, described in H10 3.10.

# List basics, continued

The !! operator produces a list's Nth element, zero-based:

```
> :type (!!)
(!!) :: [a] -> Int -> a

> [10,20..100] !! 3
40
```

Sadly, we can't use a negative value to index from the right.

```
> [10,20..100] !! (-2)
*** Exception: Prelude.(!!): negative index
```

Should that be allowed?

Important: Extensive use of !! might indicate you're writing a Java program in Haskell!

# Comparing lists

Haskell lists are values and can be compared as values:

```
> [3,4] == [1+2, 2*2]
True

> [3] ++ [] ++ [4] == [3,4]
True

> tail (tail [3,4,5,6]) == [last [4,5]] ++ [6]
True
```

Conceptually, how many lists are created by each of the above?

A programmer using a functional language writes complex expressions using lists (and more!) as freely as a Java programmer might write `f(x) * a == g(a,b) + c`.

# Comparing lists, continued

Lists are compared *lexicographically*: Corresponding elements are compared until an inequality is found. The inequality determines the result of the comparison.

Example:

```
> [1,2,3] < [1,2,4]
True
```

Why: The first two elements are equal, and `3 < 4`.

More examples:

```
> [1,2,3] < [1,1,1,1]
False
> [1,2,3] > [1,2]
True
```

# Lists of Lists

We can make lists of lists.

```
> let x = [[1], [2,3,4], [5,6]]
x :: [[Integer]]
```

Note the type: **x** is a list of **Integer** lists.

**length** counts elements at the top level.

```
> length x
3
```

Recall that **length :: [a] -> Int** Given that, what's the type of **a** for **length x**?

What's the value of **length (x ++ x ++ [3])**?

# Lists of lists, continued

More examples:

```
> let x = [[1], [2,3,4], [5,6]]

> head x
[1]

 > tail x
[[2,3,4],[5,6]]

> x !! 1 !! 2
4

> head (head (tail (tail x)))
5
```

# Strings are [Char]

Strings in Haskell are simply lists of characters.

```
> "testing"
"testing"
it :: [Char]

> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
it :: [Char]

> ["just", "a", "test"]
["just","a","test"]
it :: [[Char]]
```

What's the beauty of this?

# Strings, continued

All list functions work on strings, too!

```
> let asciiLets = ['A'..'Z'] ++ ['a'..'z']
asciiLets :: [Char]

> length asciiLets
52

> reverse (drop 26 asciiLets)
"zyxwvutsrqponmlkjihgfedcba"

> :type elem
elem :: Eq a => a -> [a] -> Bool

> let isAsciiLet c = c `elem` asciiLets
isAsciiLet :: Char -> Bool
```

# Strings, continued

The Prelude defines **String** as **[Char]** (a *type synonym*).

```
> :info String
type String = [Char]
```

A number of functions operate on **String**s. Here are two:

```
> :type words
words :: String -> [String]

> :type unwords
unwords :: [String] -> String
```

What's the following doing?

```
> unwords (tail (words "Just some words!"))
"some words!"
```

# "cons" lists

Like most functional languages, Haskell's lists are "cons" lists.

A "cons" list has two parts:

head: a value

tail: a list of values (possibly empty)

The : ("cons") operator creates a list from a value and a list of values of that same type (or an empty list).

```
> 5 : [10, 20,30]
[5,10,20,30]
```

What's the type of the cons operator?

```
> :type (:)
(:) :: a -> [a] -> [a]
```

# "cons" lists, continued

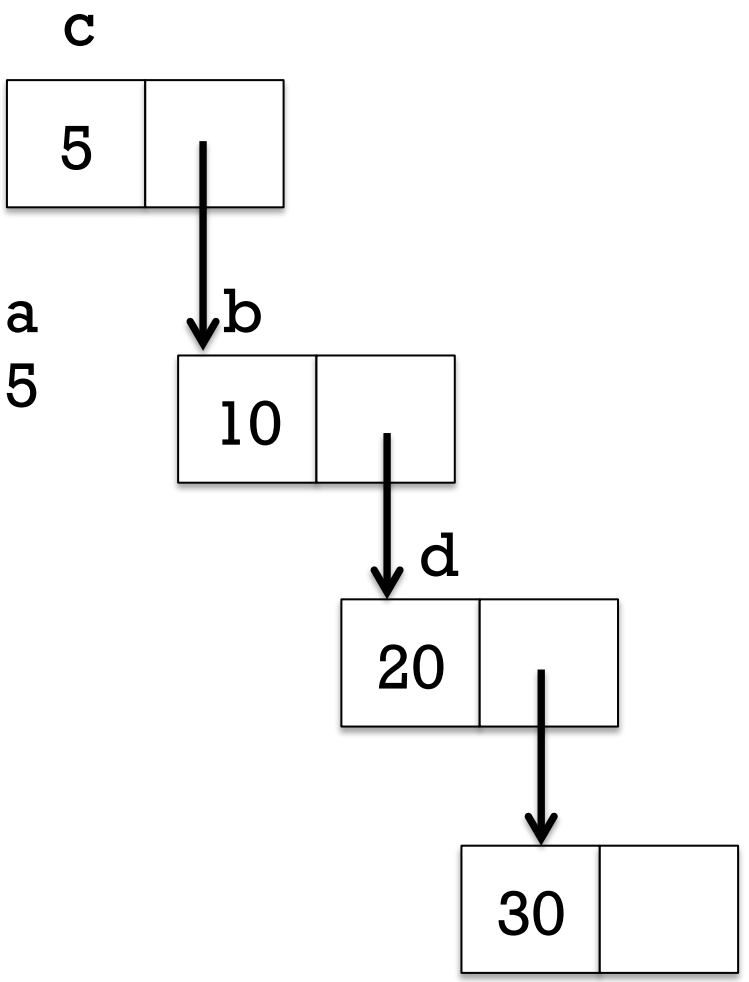
The cons (:) operation forms a new list from a value and a list.

```
> let a = 5
> let b = [10,20,30]
> let c = a:b
[5,10,20,30]

> head c
5

> tail c
[10,20,30]

> let d = tail (tail c)
> d
[20,30]
```



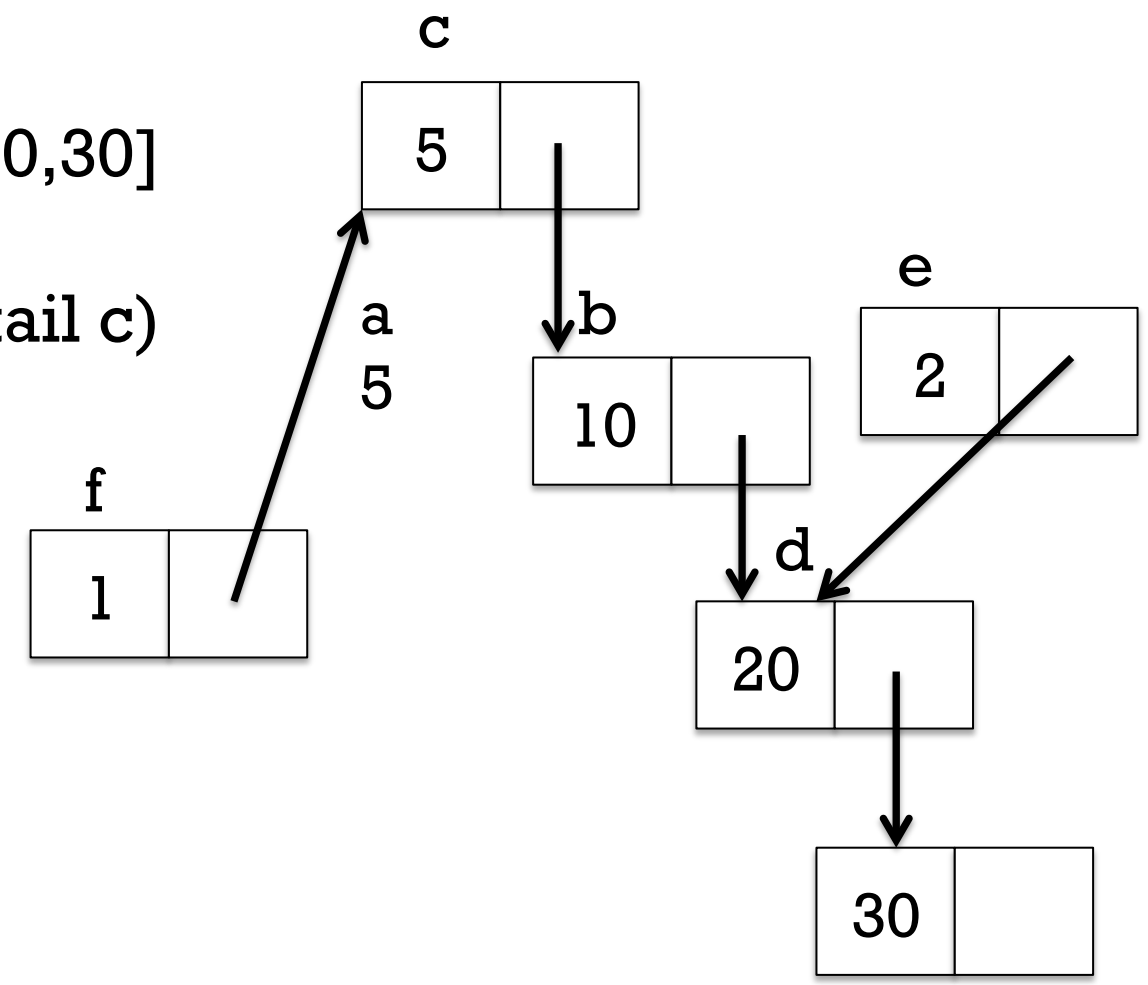
# "cons" lists, continued

A cons node can be referenced by multiple cons nodes.

```
> let a = 5
> let b = [10,20,30]
> let c = a:b
> let d = tail (tail c)
[20,30]

> let e=2:d
[2,20,30]

> let f=1:c
[1,5,10,20,30]
```



# "cons" lists, continued

What are the values of the following expressions?

```
> 1:[2,3]
[1,2,3]

> 1:2
...error...

> chr 97:chr 98:chr 99:[]
"abc"

> []:[]
[[]]

> [1,2]:[]
[[1,2]]

> []:[1]
...error...
```

cons is right associative

```
chr 97:(chr 98:(chr 99:[]))
```

# **head** and **tail** visually

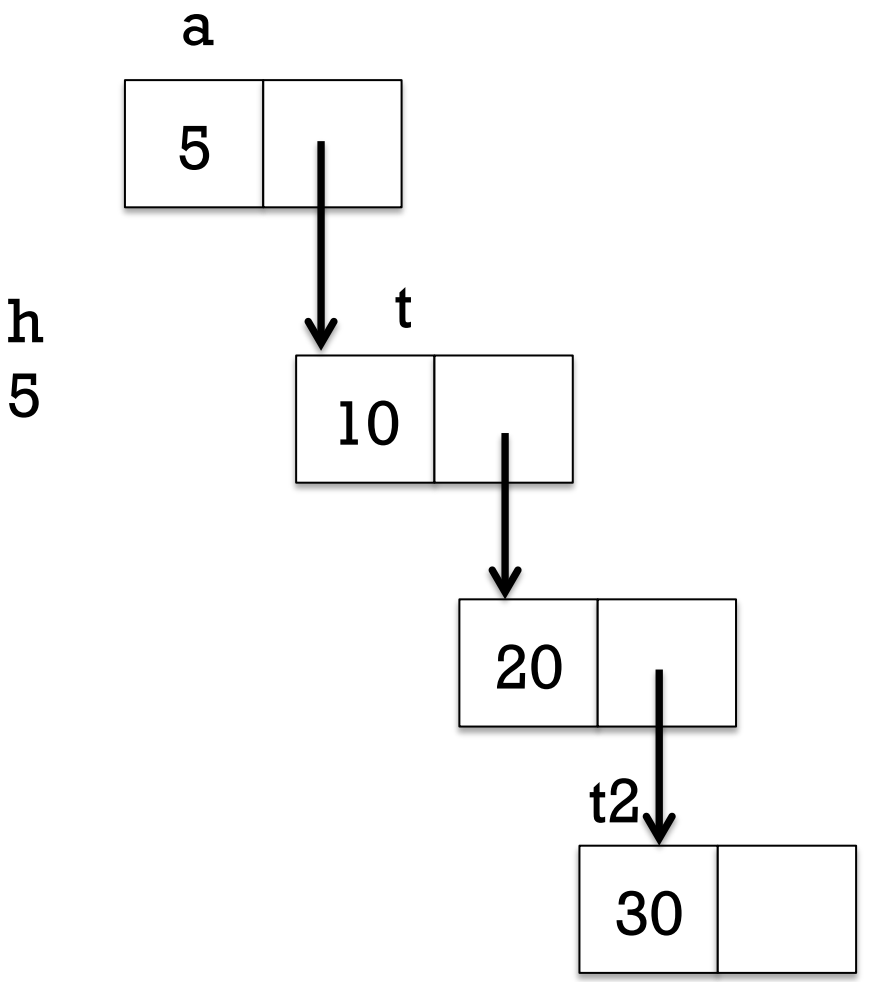
It's important to understand that **tail** does not create a new list. Instead it simply returns an existing cons node.

```
> let a = [5,10,20,30]

> let h = head a
> h
5

> let t = tail a
> t
[10,20,30]

> let t2 = tail (tail t)
> t2
[30]
```



# A little on performance

What operations are likely fast with cons lists?

- Get the head of a list
- Get the tail of a list
- Make a new list from a head and tail ("cons up a list")

What operations are likely slower?

- Get the Nth element of a list
- Get the length of a list

With cons lists, what does list concatenation involve?

```
> let m=[1..10000000]
> length (m++[0])
10000001
```

# True or false?

The head of a list is a one-element list.

False, unless...

...it's the head of a list of lists that starts with a one-element list

The tail of a list is a list.

True

The tail of an empty list is an empty list.

It's an error!

`length (tail (tail x)) == (length x) – 2`

True (assuming what?)

A cons list is essentially a singly-linked list.

True

A doubly-linked list might help performance in some cases.

Hmm...what's the backlink for a multiply-referenced node?

Changing an element in a list might affect the value of many lists.

Trick question! We can't change a list element. We can only "cons up" new lists and reference existing lists.

# fromTo

Here's a function that produces a list with a range of integers:

```
> let fromTo first last = [first..last]

> fromTo 10 15
[10,11,12,13,14,15]
```

Problem: Write a recursive version of **fromTo** that uses the cons operator to build up its result.

# fromTo, continued

One solution:

```
fromTo first last
    | first > last = []
    | otherwise = first : fromTo (first+1) last
```

Evaluation of `fromTo 1 3` via substitution and rewriting:

```
fromTo 1 3
1 : fromTo (1+1) 3
1 : fromTo 2 3
1 : 2 : fromTo (2+1) 3
1 : 2 : fromTo 3  3
1 : 2 : 3 : fromTo (3+1) 3
1 : 2 : 3 : fromTo 4 3
1 : 2 : 3 : []
```

The **Enum** type class has **enumFromTo** and more.

# **fromTo**, continued

Do **:set +s** to get timing and memory information, and make some lists. Try these:

```
fromTo 1 10
let f = fromTo        -- So we can type f instead of fromTo
f 1 1000
let f = fromTo 1      -- Note partial application
f 1000
let x = f 1000000
length x
take 5 (f 1000000)
```

# List comprehensions

Here's a simple example of a *list comprehension*:

```
> [x^2 | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

This describes a list of the squares of **`x`** where **`x`** takes on each of the values from 1 through 10.

List comprehensions are very powerful but in the interest of time and staying focused on the core concepts of functional programming, we're not going to cover them.

Chapter 5 in Hutton has some very interesting examples of practical computations with list comprehensions.

# A little output

# A little output

The **putStr** function outputs a string:

```
> putStr "just\ntesting\n"
just
testing
```

Here's the type of **putStr**:

```
> :t putStr
putStr :: String -> IO ()
```

The return type of **putStr**, **IO ()**, is known as an *action*. It represents an interaction with the outside world, which is a side effect.

The construction **()** is read as "unit". The unit type has a single value, unit. Both the type and the value are written as **()**.

# A little output, continued

For the time being, we'll use this approach for functions that produce output:

- A helper function will produce a ready-to-print string that contains newline characters as needed.

- The top-level function will call the helper function and then call `putStr` with the helper function's result.

# A little output, continued

We can use **show** to produce a string representation of any value whose type is a member of the **Show** type class.

```
> :t show
show :: Show a => a -> String

> show 10
"10"

> show [10,20]
"[10,20]"

> show show
"<function>"
```

# printN

Let's write a function to print the integers from 1 to N:

```
> printN 3
1
2
3
```

First, let's write a helper, **printN'**:

```
> printN' 3
"1\n2\n3\n"
```

Solution:

```
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

# printN, continued

At hand:

```
printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

Usage:

```
> printN' 10
"1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n"
```

Let's write the top-level function:

```
printN::Integer -> IO ()
printN n = putStr (printN' n)
```

# printN, continued

All together, as a file:

```
% cat printN.hs
printN::Integer -> IO ()
printN n = putStr (printN' n)

printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"

% ghci printN
...
> printN 3
1
2
3
```

# printNunary

At hand:

```
printN::Integer -> IO ()
printN n = putStr (printN' n)

printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

Let's modify **printN** to print lines of characters:

```
> printN 3 '|'
|
||
|||
```

We can view it as printing *unary numbers* with a specified "digit", so we'll call the new version **printNunary**.

# printNunary, continued

Useful: The Prelude has **replicate :: Int -> a -> [a]**

```
> replicate 3 7
[7,7,7]
> replicate 3 'a'
"aaa"
```

Let's add a parameter for the character to print and call **replicate** instead of **show**:

```
printNunary::Int -> Char -> IO ()
printNunary n c = putStr (printNunary' n c)

printNunary'::Int -> Char -> String
printNunary' n c
   | n == 0 = ""
   | otherwise = printNunary' (n-1) c ++
                          (replicate n c) ++ "\n"
```

# charbox

Let's write **charbox**:

```
> charbox 5 3 '*'
*****
*****
*****

> :t charbox
charbox :: Int -> Int -> Char -> IO ()
```

How can we approach it?

# **charbox**, continued

Let's work out a sequence of computations with **ghci**:

```
> replicate 5 '*'
"*****"

> it ++ "\n"
"*****\n"

> replicate 2 it
["*****\n","*****\n"]    -- the type of it is [[Char]]

> :t concat
concat :: [[a]] -> [a]

> concat it
"*****\n*****\n"

> putStr it
*****
*****
```

# **charbox**, continued

Let's write **charbox'**:

```
charbox'::Int -> Int -> Char -> String
charbox' w h c = concat (replicate h (replicate w c ++ "\n"))
```

Test:

```
> charbox' 3 2 '*'
"***\n***\n"
```

Now we're ready for the top-level function:

```
charbox::Int -> Int -> Char -> IO ()
charbox w h c = putStr (charbox' w h c)
```

How does this approach contrast with how we'd write it in Java?

# Sidebar: Where's the code?

On the CS machines, selected Haskell code is in this directory:

**/cs/www/classes/cs372/spring16/haskell**

In these slides I'll refer to that directory as **spring16**.

**spring16/slides.hs** has the smaller functions, in rough chronological order. For functions that evolve, there may be multiple versions, with all but one version commented out.

Note: **{- ... -}** is a multi-line comment in Haskell.

Larger examples are in their own files, like **spring16/printN.hs** and **spring16/charbox.hs**.

**spring16** is also accessible on the web:

**http://cs.arizona.edu/classes/cs372/spring16**

# Patterns

# Motivation: Summing list elements

Imagine a function that computes the sum of a list's elements.

```
> sumElems  [1..10]
55

> :type sumElems
sumElems :: Num a => [a] -> a
```

Implementation:

```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

It works but it's not idiomatic Haskell. We should use *patterns* instead!

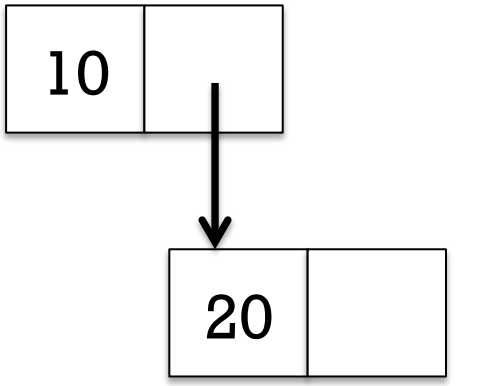
# Patterns

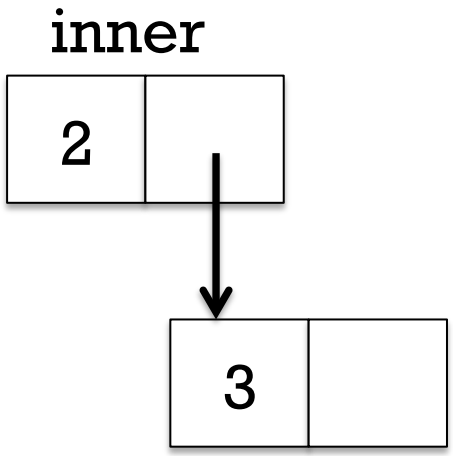
In Haskell we can use *patterns* to bind names to elements of data structures.

```
> let [x,y] = [10,20]
> x
10
> y
20

> let [inner] = [[2,3]]
> inner
[2,3]
```

```
x   y
10  20
```




Speculate: Given a list like [10,20,30] how could we use a pattern to bind names to the head and tail of the list?

# Patterns, continued

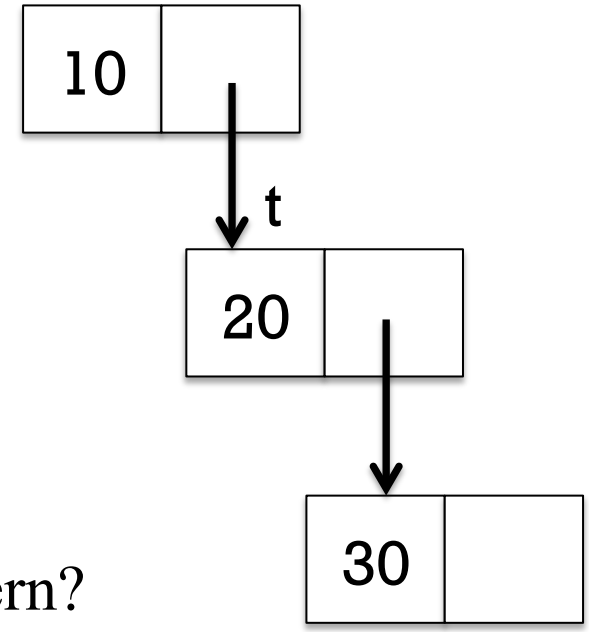
We can use the cons operator in a pattern.

```
> let h:t = [10,20,30]

> h
10

> t
[20,30]
```

```
h
10
```




What values get bound by the following pattern?

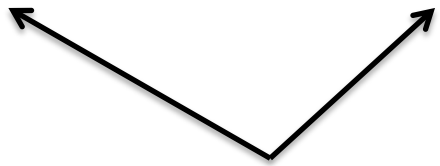
```
> let a:b:c:d = [10,20,30]
> [c,b,a]          -- in a list so I could show them w/ a one-liner
[30,20,10]

> d
[]                 -- Why didn't I do [d,c,b,a] above?
```

# Patterns, continued

If some part of a structure is not of interest, we indicate that with an underscore, known as the *wildcard pattern*.

```
> let _:(a:[b]):c = [[1],[2,3],[4]]
> a
2
> b
3
> c
[[4]]
```



No binding is done for the wildcard pattern.

The pattern mechanism is completely general—patterns can be arbitrarily complex.

# Patterns, continued

A name can only appear once in a pattern. This is invalid:

```
> let a:a:[] = [3,3]
<interactive>:25:5:
    Conflicting definitions for `a'
```

When using **let** as we are here, a failed pattern isn't manifested until we try to see what's bound to a name.

```
> let a:b:[] = [1]
> a
*** Exception: <interactive>:26:5-16: Irrefutable
pattern failed for pattern a : b : []
```

# Practice

Describe in English what must be on the right hand side for a successful match.

`let (a:b:c) = ...`

A list containing at least two elements.

Does `[[1,2]]` match?

`[2,3]` ?

`"abc"` ?

`let [x:xs] = ...`

A list whose only element is a non-empty list.

Does `words "a test"` match?

`[words "a test"]` ?

`[[]]` ?

`[[[]]]` ?

# Patterns in function definitions

Recall our non-idiomatic **sumElems**:

```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

How could we redo it using patterns?

```
sumElems [] = 0
sumElems (h:t) = h + sumElems t
```

Note that **sumElems** appears on both lines and that there are no guards. **sumElems** has two *clauses*. (H10 4.4.3.1)

**The parentheses in (h:t) are required!!**

Do the types of the two versions differ?

```
(Eq a, Num a) => [a] -> a
        Num a => [a] -> a
```

# Patterns in functions, continued

Here's a buggy version of **sumElems**:

```
buggySum [x] = x
buggySum (h:t) = h + buggySum t
```

What's the bug?

```
> buggySum [1..100]
5050
> buggySum []
*** Exception: slides.hs:(62,1)-(63,31): Non-
exhaustive patterns in function buggySum
```

# Patterns in functions, continued

At hand:

```
buggySum [x] = x
buggySum (h:t) = h + buggySum t
```

If we use the **-fwarn-incomplete-patterns** option of **ghci**, we'll get a warning when loading:

```
% ghci -fwarn-incomplete-patterns buggySum.hs
buggySum.hs:1:1:Warning:
    Pattern match(es) are non-exhaustive
    In an equation for 'buggySum': Patterns not matched: []
    >
```

Suggestion: add a bash alias! (See us if you don't know how to.)

```
alias ghci="ghci -fwarn-incomplete-patterns"
```

# Patterns in functions, continued

What's a little silly about the following list-summing function?

```
sillySum [] = 0
sillySum [x] = x
sillySum (h:t) = h + sillySum t
```

The second clause isn't needed.

# An "as pattern"

Consider a function that duplicates the head of a list:

```
> duphead [10,20,30]
[10,10,20,30]
```

Here's one way to write it, but it's repetitious:

```
duphead (x:xs) = x:x:xs
```

We can use an "as pattern" to bind a name to the list as a whole:

```
duphead all@(x:xs) = x:all
```

Can it be improved?

```
duphead all@(x:_) = x:all
```

The term "as pattern" perhaps comes from Standard ML, which uses an "as" keyword for the same purpose.

# Patterns, then guards, then **if-else**

Good coding style in Haskell:

- Prefer patterns over guards
- Prefer guards over **if-else**

Patterns—first choice!

```
sumElems [] = 0
sumElems (h:t) = h + sumElems t
```

Guards—second choice...

```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

**if-else**—third choice...

```
sumElems list =
    if list == [] then 0
    else head list + sumElems (tail list)
```

And, these comparisons imply that **list**'s type must in **Eq**!

# Patterns, then guards, then if-else

Recall this example of guards:

```
weather temp | temp >= 80 = "Hot!"
             | temp >= 70 = "Nice"
             | otherwise = "Cold!"
```

Can we rewrite **weather** to have three clauses with patterns?

No.

The pattern mechanism doesn't provide a way to test ranges.

Design question: should patterns and guards be unified?

# Revision: the general form of a function

We first saw this *general form* of a function definition:

*name param1 param2 ... paramN = expression*

Revision: A function may have one or more clauses, of this form:

*function-name pattern1 pattern2 ... patternN*

{ | *guard-expression1* } = *result-expression1*

...

{ | *guard-expressionN* } = *result-expressionN*

The set of clauses for a given name is the *binding* for that name. (See 4.4.3 in H10.)

If values in a call match the pattern(s) for a clause and a guard is true, the corresponding expression is evaluated.

# Revision, continued

At hand, a more general form for functions:

*function-name pattern1 pattern2 ... patternN*

{ | *guard-expression1* } = *result-expression1*

...

{ | *guard-expressionN* } = *result-expressionN*

How does

**`add x y = x + y`**

conform to the above specification?

- **`x`** and **`y`** are trivial patterns
- **`add`** has one clause, which has no guard

# Pattern/guard interaction

If the patterns of a clause match but all guards fail, the next clause is tried. Here's a contrived example:

```
f (h:_) | h < 0 = "negative head"
f list | length list > 3 = "too long"
f (_:_) = "ok"
f [] = "empty"
```

Usage:

```
> f [-1,2,3]
"negative head"

> f []
"empty"

> f [1..10]
"too long"
```

How many clauses does **f** have?

4

What if 2nd and 3rd clauses swapped?

3rd clause would never be matched!

What if 4th clause is removed?

Warning with **-fwarn-incomplete-patterns**; "non-exhaustive patterns" exception on **f []**.

# Recursive functions on lists

# Simple recursive list processing functions

Problem: Write **`len x`**, which returns the length of list **`x`**.

```
> len []
0

> len "testing"
7
```

Solution:

```
len [] = 0
len (_:t) = 1 + len t  -- since head isn't needed, use _
```

# Simple list functions, continued

Problem: Write **odds x**, which returns a list having only the odd numbers from the list **x**.

```
> odds [1..10]
[1,3,5,7,9]

> take 10 (odds [1,4..100])
[1,7,13,19,25,31,37,43,49,55]
```

Handy: **odd :: Integral a => a -> Bool**

Solution:

```
odds [] = []
odds (h:t)
    | odd h = h:odds t
    | otherwise = odds t
```

# Simple list functions, continued

Problem: write **isElem x vals**, like **elem** in the Prelude.

```
> isElem 5 [4,3,7]
False

> isElem 'n' "Bingo!"
True

> "quiz" `isElem` words "No quiz today!"
True
```

Solution:

```
isElem _ [] = False     -- Why a wildcard?
isElem x (h:t)
    | x == h = True
    | otherwise = x `isElem` t
```

# Simple list functions, continued

Problem: write a function that returns a list's maximum value.

```
> maxVal "maximum"
'x'

> maxVal [3,7,2]
7

> maxVal (words "i luv this stuff")
"this"
```

Note that the Prelude has `max :: Ord a => a -> a -> a`

One solution:

```
maxVal [x] = x
maxVal (x:xs) = max x (maxVal xs)
maxVal [] = error "empty list"
```

# Sidebar: C and Python challenges

C programmers: Write **`strlen`** in C in a functional style. Do **`strcmp`** and **`strchr`**, too!

Python programmers: In a functional style write **`size(x)`**, which returns the number of elements in the string or list **`x`**. Restriction: You may not use **`type()`**.

# Tuples

# Tuples

A Haskell *tuple* is an ordered aggregation of two or more values of possibly differing types.

```
> (1, "two", 3.0)
(1,"two",3.0)
it :: (Integer, [Char], Double)

> (3 < 4, it)
(True,(1,"two",3.0))
it :: (Bool, (Integer, [Char], Double))
```

What's something we can represent with a tuple that we can't represent with a list?

We can't create analogous lists for the above tuples, due to the mix of types. Lists must be homogeneous.

# Tuples, continued

A function can return a tuple:

```
> let pair x y = (x,y)
```

What's the type of `pair`?

```
pair :: t -> t1 -> (t, t1)
-- why not a -> b -> (a,b)?
```

Let's play...

```
> pair 3 4
(3,4)

> pair (3,4)
<function>

> it 5
((3,4),5)
```

# Tuples, continued

The Prelude has two functions that operate on 2-tuples.

```
> let p = pair 30 "forty"
p :: (Integer, [Char])

> p
(30,"forty")

> fst p
30

> snd p
"forty"
```

# Tuples, continued

Recall: patterns used to bind names to list elements have the same syntax as expressions to create lists.

Patterns for tuples are like that, too.

Problem: Write **middle**, to extract a 3-tuple's second element.

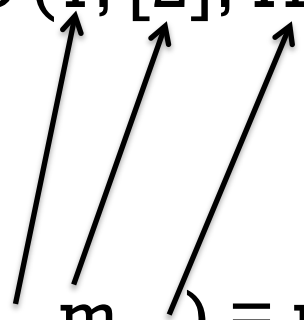
```
> middle ("372", "BIOW 208", "Mitchell")
"BIOW 208"

> middle (1, [2], True)
[2]
```

# Tuples, continued

At hand:

```
> middle (1, [2], True)
[2]
```



Solution:

```
middle (_, m, _) = m
```

What's the type of **middle**?

```
middle :: (t, t1, t2) -> t1
```

Does the following call work?

```
> middle(1,[(2,3)],4)
[(2,3)]
```

# Tuples, continued

Here's the type of **zip** from the Prelude:

```
zip :: [a] -> [b] -> [(a, b)]
```

Speculate: What does **zip** do?

```
> zip ["one","two","three"] [10,20,30]
[("one",10),("two",20),("three",30)]

> zip ['a'..'z'] [1..]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6),('g',7),('h',8),('i',
9),('j',10), ...more..., ('x',24),('y',25),('z',26)]
```

What's especially interesting about the second example?

`[1..]` is an infinite list! **zip** stops when either list runs out.

# Tuples, continued

Problem: Write **elemPos**, which returns the zero-based position of a value in a list, or -1 if not found.

```
> elemPos 'm' ['a'..'z']
12
```

Hint: Have a helper function do most of the work.

Solution:

```
elemPos x vals = elemPos' x (zip vals [0..])

elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps
```

# Sidebar: To curry or not to curry?

Consider these two functions:

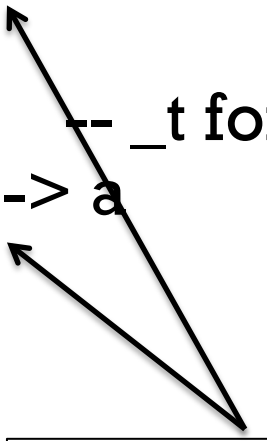
```
> let add_c x y = x + y    -- _c for curried arguments
add_c :: Num a => a -> a -> a

> let add_t (x,y) = x + y    -- _t for tuple argument
add_t :: Num a => (a, a) -> a
```

Usage:

```
> add_c 3 4
7

> add_t (3,4)
7
```



**Important:** Note the difference in types!

Which is better, **add_c** or **add_t**?

# The **Eq** type class and tuples

**:info Eq** shows many lines like this:

```
...
instance (Eq a, Eq b, Eq c, Eq d, Eq e) => Eq (a, b, c, d, e)
instance (Eq a, Eq b, Eq c, Eq d) => Eq (a, b, c, d)
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
instance (Eq a, Eq b) => Eq (a, b)
```

We haven't talked about **instance** declarations but let's speculate: What's being specified by the above?

**instance (Eq a, Eq b, Eq c) => Eq (a, b, c)**

If values of each of the three types **a**, **b**, and **c** can be tested for equality then 3-tuples of type **(a, b, c)** can be tested for equality.

The **Ord** and **Bounded** type classes have similar instance declarations.

# Lists vs. tuples

Type-wise, lists are homogeneous; tuples are heterogeneous.

We can write a function that handles a list of any length but a function that operates on a tuple specifies the arity of that tuple.

> Example: we can't write an analog for **`head`**, to return the first element of an arbitrary tuple.

Even if values are homogeneous, using a tuple lets static type-checking ensure that an exact number of values is being aggregated.

> Example: A 3D point could be represented with a 3-element list but using a 3-tuple guarantees points have three coordinates.

If there were *Head First Haskell* it would no doubt have an interview with List and Tuple, each arguing their own merit.

# More on patterns and functions

# Literals in patterns

Literal values can be part or all of a pattern. Here's a 3-clause binding for f:

```
f 1 = 10
f 2 = 20
f n = n
```

Usage:

```
> f 1
10

> f 3
3
```

For contrast, with guards:

```
f n
  | n == 1 = 10
  | n == 2 = 20
  | otherwise = n
```

Remember: Patterns are tried in the order specified.

# Literals in patterns, continued

Here's a function that classifies characters as parentheses (or not):

```
parens c
    | c == '(' = "left"
    | c == ')' = "right"
    | otherwise = "neither"
```

Could we improve it by using patterns instead of guards?

```
parens '(' = "left"
parens ')' = "right"
parens _  = "neither"
```

Which is better?

Remember: Patterns, then guards, then **if-else**.

# Literals in patterns, continued

**not** is a function:

```
> :type not
not :: Bool -> Bool

> not True
False
```

Problem: Using literals in patterns, define **not**.

Solution:

```
not True = False
not _ = True          -- Using wildcard avoids comparison
```

# Pattern construction

A pattern can be:

- A literal value such as 1, '`x`', or `True`
- An identifier (bound to a value if there's a match)
- An underscore (the wildcard pattern)
- A tuple composed of patterns
- A list of patterns in square brackets (fixed size list)
- A list of patterns constructed with `:` operators
- Other things we haven't seen yet

Note the recursion.

Patterns can be arbitrarily complex.

3.17.1 in H10 shows the full syntax for patterns.

# The **where** clause for functions

Intermediate values and/or helper functions can be defined using an optional **where** clause for a function.

Here's an example to show the syntax; the computation is not meaningful.

```
f x
   | g x < 0 = g a + g b
   | a > b = g b
   | otherwise = g a * g b
  where {
      a = x * 5;
      b = a * 2 + x;
      g t = log t + a
      }
```

The names **a** and **b** are bound to expressions; **g** is a function binding.

The bindings in the **where** clause are done first (!), then the guards are evaluated in turn.

Like variables defined in a method or block in Java, **a**, **b**, and **g** are not visible outside the the function **f**.

# The *layout rule* for **where** (and more)

This is a valid declaration with a **where** clause:

```
f x = a + b + g a where { a = 1; b = 2; g x = -x }
```

The **where** clause has three declarations enclosed in braces and separated by semicolons.

We can take advantage of the *layout rule* and write it like this instead:

```
f x = a + b + g a
    where
        a = 1
        b = 2
        g x =
            -x
```

Besides whitespace what's different about the second version?

# The layout rule, continued

At hand:

```
f x = a + b + g a
    where
        a = 1
        b = 2
        g x =
            -x
```

Another example:

```
f x = a + b + g a where a = 1
                        b = 2
                        g x =
                          -x
```

The absence of a brace after **where** activates the layout rule.

The column position of the first token after **where** establishes the column in which declarations of the **where** must start.

Note that the declaration of **g** is continued onto a second line; if the minus sign were at or left of the line, it would be an error.

# The layout rule, continued

Don't confuse the layout rule with indentation-based continuation of declarations! (See slides 94-95.)

The layout rule allows omission of braces and semicolons in **where**, **do**, **let**, and **of** blocks. (We'll see **do** and **let** later.)

Indentation-based continuation applies

1. outside of **where/do/let/of** blocks
2. inside **where/do/let/of** blocks when the layout rule is triggered by the absence of an opening brace.

The layout rule is also called the "off-side rule".

TAB characters are assumed to have a width of 8.

What other languages have rules of a similar nature?

# countEO

Imagine a function that counts occurrences of even and odd numbers in a list.

```
> countEO [3,4,5]
(1,2)                          -- one even, two odds
```

Code:

```
countEO [] = (0,0)        -- no odds or evens in []
countEO (x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
  where
    (evens, odds) = countEO xs    -- do counts for tail first!
```

Would it be awkward to write it without using **where**?

Try it!

# countEO, continued

At hand:

```
countEO [] = (0,0)
countEO (x:xs)
    | odd x = (evens, odds + 1)
    | otherwise = (1+ evens, odds)
  where (evens, odds) = countEO xs
```

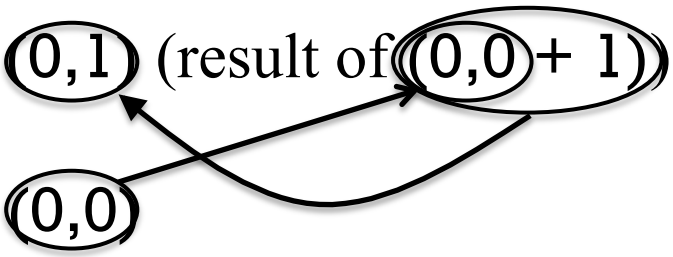
Here's one way to picture this recursion:

```
countEO [10,20,25]    returns (2,1) (result of (1 + 1,1))

   countEO [20,25]    returns (1,1) (result of (1 + 0,1))

      countEO [25]    returns (0,1) (result of (0,0 + 1))

         countEO []   returns (0,0)
```

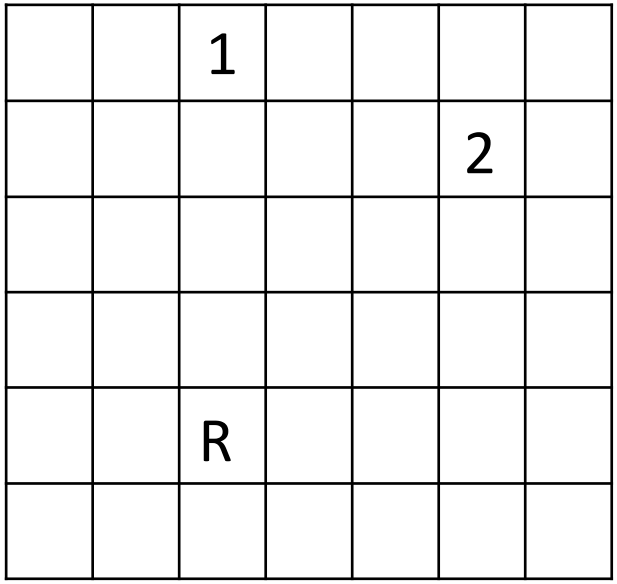


# Larger examples

# travel

Imagine a robot that travels on an infinite grid of cells. Movement is directed by a series of one character commands: **`n`**, **`e`**, **`s`**, and **`w`**.

Let's write a function **`travel`** that moves the robot about the grid and determines if the robot ends up where it started (i.e., it got home) or elsewhere (it got lost).




If the robot starts in square `R` the command string **`nnnn`** leaves the robot in the square marked `1`.

The string **`nenene`** leaves the robot in the square marked `2`.

**`nnessw`** and **`news`** move the robot in a round-trip that returns it to square R.

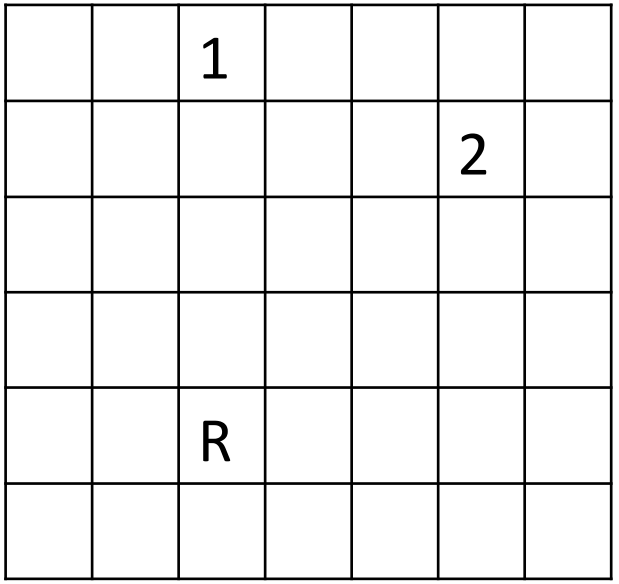
# travel, continued

Usage:

```
> travel "nnnn"        -- ends at 1
"Got lost; 4 from home"

> travel "nenene"      -- ends at 2
"Got lost; 6 from home"

> travel "nnessw"
"Got home"
```




How can we approach this problem?

# **travel**, continued

One approach:

1. Map letters into integer 2-tuples representing X and Y displacements on a Cartesian plane.
2. Sum the X and Y displacements to yield a net displacement.

Example:

Argument value: "**nnee**"
Mapped to tuples: (0,1) (0,1) (1,0) (1,0)
Sum of tuples: (2,2)

Another:

Argument value: "**nnessw**"
Mapped to tuples: (0,1) (0,1) (1,0) (0,-1) (0,-1) (-1,0)
Sum of tuples: (0,0)

# **travel**, continued

First, let's write a helper function to turn a direction into an **(x,y)** displacement:

```
mapMove :: Char -> (Int, Int)
mapMove 'n' = (0,1)
mapMove 's' = (0,-1)
mapMove 'e' = (1,0)
mapMove 'w' = (-1,0)
mapMove c = error ("Unknown direction: " ++ [c])
```

Missing case found with
**ghci -fwarn-incomplete-patterns**



Usage:

```
> mapMove 'n'
(0,1)

> mapMove 'w'
(-1,0)
```

# travel, continued

Next, a function to sum **x** and **y** displacements in a list of tuples:

```
> sumTuples [(0,1),(1,0)]
(1,1)

> sumTuples [mapMove 'n', mapMove 'w']
(-1,1)
```

Implementation:

```
sumTuples :: [(Int,Int)] -> (Int,Int)
sumTuples [] = (0,0)
sumTuples ((x,y):ts) = (x + sumX, y + sumY)
   where
      (sumX, sumY) = sumTuples ts
```

# **travel**, continued

**travel** itself:

```
travel :: [Char] -> [Char]
travel s
    | disp == (0,0) = "Got home"
    | otherwise = "Got lost; " ++ show (abs x + abs y) ++
                  " from home"
   where
      tuples = makeTuples s
      disp@(x,y) = sumTuples tuples -- note "as pattern"

      makeTuples :: [Char] -> [(Int, Int)]
      makeTuples [] = []
      makeTuples (c:cs) = mapMove c : makeTuples cs
```

As is, **mapMove** and **sumTuples** are at the top level but **makeTuples** is hidden inside **travel**. How should they be arranged?

# Sidebar: top-level vs. hidden functions

```
travel s
   | disp == (0,0) = "Got home"
   | otherwise = "Got lost; " ...
   where
      tuples = makeTuples s
      disp = sumTuples tuples

      makeTuples [] = []
      makeTuples (c:cs) =
          mapMove c:makeTuples cs

      mapMove 'n' = (0,1)
      mapMove 's' = (0,-1)
      mapMove 'e' = (1,0)
      mapMove 'w' = (-1,0)
      mapMove c = error ...

      sumTuples [] = (0,0)
      sumTuples ((x,y):ts) = (x + sumX, y + sumY)
         where
            (sumX, sumY) = sumTuples ts
```

Top-level functions can be tested after code is loaded but functions inside a **where** block are not visible.

The functions at left are hidden in the **where** block but they can easily be changed to top-level using a shift or two with an editor.

Note: Types are not shown, to save space.

# tally

Consider a function **tally** that counts character occurrences in a string:

```
> tally "a bean bag"
a 3
b 2
  2
g 1
n 1
e 1
```

Note that the characters are shown in order of decreasing frequency.

How can this problem be approached?

In a nutshell: `[('a',3),('b',2),(' ',2),('g',1),('n',1),('e',1)]`

# **tally**, continued

Let's start by writing **incEntry c tuples**, which takes a list of (*character, count*) tuples and produces a new list of tuples that reflects the addition of the character **c**.

```
incEntry :: Char -> [(Char, Int)] -> [(Char, Int)]
```

Calls to **incEntry** with 't', 'o', 'o':

```
> incEntry 't' []
[('t',1)]

> incEntry 'o' it
[('t',1),('o',1)]

> incEntry 'o' it
[('t',1),('o',2)]
```

```
{-  incEntry c tups

    tups is a list of (Char, Int) tuples that indicate how many
    times a character has been seen.  A possible value for tups:
        [('b',1),('a',2)]

    incEntry produces a copy of tups with the count in the tuple
    containing the character c incremented by one.

    If no tuple with c exists, one is created with a count of 1.
-}

incEntry::Char -> [(Char,Int)] -> [(Char,Int)]
incEntry c [ ] = [(c, 1)]
incEntry c ((char, count):entries)
  | c == char = (char, count+1) :  entries
  | otherwise = (char, count) :  incEntry c entries
```

Next, let's write **mkentries s**. It calls **incEntry** for each character in the string **s** in turn and produces a list of (*char, count*) tuples.

```
mkentries :: [Char] -> [(Char, Int)]
```

Usage:

```
> mkentries "tupple"
[('t',1),('u',1),('p',2),('l',1),('e',1)]

> mkentries "cocoon"
[('c',2),('o',3),('n',1)]
```

Code:

```
mkentries :: [Char] -> [(Char, Int)]
mkentries s = mkentries' s []
   where
      mkentries' [ ] entries = entries
      mkentries' (c:cs) entries =
         mkentries' cs (incEntry c entries)
```

```
{-  insert, isOrdered, and sort  provide an insertion sort -}
insert v [ ] = [v]
insert v (x:xs)
    | isOrdered (v,x) = v:x:xs
    | otherwise =  x:insert v xs

isOrdered ((_, v1), (_, v2)) = v1 > v2

sort [] = []
sort (x:xs) = insert x (sort xs)

> mkentries "cocoon"
[('c',2),('o',3),('n',1)]

> sort it
[('o',3),('c',2),('n',1)]
```

# **tally**, continued

```
{- fmtEntries prints (char,count) tuples one per line -}
fmtEntries [] = ""
fmtEntries ((c, count):es) =
      [c] ++ " " ++ (show count) ++ "\n" ++ fmtEntries es

{- top-level function -}
tally s = putStr (fmtEntries (sort (mkentries s)))

> tally "cocoon"
o 3
c 2
n 1
```

- How does this solution exemplify functional programming? (slide 24)
- How is it like procedural programming (slide 5)

# Running **tally** from the command line

Let's run it on lectura...

```
% code=/cs/www/classes/cs372/spring16/haskell

% cat $code/tally.hs
... everything we've seen before and now a main:
main = do
   bytes <- getContents  -- reads all of standard input
   tally bytes

% echo -n cocoon | runghc $code/tally.hs
o 3
c 2
n 1
```

# **tally** from the command line, continued

**$code/genchars N** generates N random letters:

```
% $code/genchars 20
KVQaVPEmClHRbgdkmMsQ
```

Lets tally a million letters:

```
% $code/genchars 1000000 |
        time runghc $code/tally.hs >out
21.79user 0.24system 0:22.06elapsed
% head -3 out
s  19553
V 19448
J  19437
```

# tally from the command line, continued

Let's try a compiled executable.

```
% cd $code
% ghc --make -rtsopts tally.hs
% ls -l tally
-rwxrwxr-x 1 whm whm 1118828 Jan 26 00:54 tally

% ./genchars 1000000 > 1m
% time ./tally +RTS -K40000000 -RTS < 1m > out
real    0m7.367s
user    0m7.260s
sys     0m0.076s
```

# **tally** performance in other languages

Here are user CPU times for implementations of **tally** in several languages. The same one-million letter file was used for all timings.

| Language | Time (seconds) |
|---|---|
| Haskell | 7.260 |
| Ruby | 0.548 |
| Icon | 0.432 |
| Python 2 | 0.256 |
| C w/ `gcc -O3` | 0.016 |

However, our **tally** implementation is very simplistic. An implementation of **tally** by an expert Haskell programmer, Chris van Horne, ran in 0.008 seconds. (See **spring16/haskell/tally-cwvh[12].hs**.)

Then I revisited the C version (**tally2.c**) and got to 3x faster than Chris' version with a one-billion character file.

# Real world problem: "How many lectures?"

Here's an early question when planning a course for a semester:

"How many lectures will there be?"

How should we answer that question?

Google for a course planning app?

No! Let's write a Haskell program! ☺

# classdays

One approach:

```
> classdays ...arguments...
#1 H 1/15        (for 2015...)
#2 T 1/20
#3 H 1/22
#4 T 1/27
#5 H 1/29
...
```

What information do the arguments need to specify?

- First and last day
- Pattern, like M-W-F or T-H
- How about holidays?

# Arguments for **classdays**

Let's start with something simple:

```
> classdays  (1,15)  (5,6)  [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
>
```

The first and last days are represented with (*month*,*day*) tuples.

The third argument shows the pattern of class days: the first is a Thursday, and it's five days to the next class. The next is a Tuesday, and it's two days to the next class. Repeat!

# Date handling

There's a **Data.Time.Calendar** module but writing two minimal date handling functions provides good practice.

```
> toOrdinal (12,31)
365     -- 12/31 is the last day of the year

> fromOrdinal 32
(2,1)   -- The 32nd day of the year is February 1.
```

What's a minimal data structure that could help us?

```
[(0,0),(1,31),(2,59),(3,90),(4,120),(5,151),(6,181),
(7,212),(8,243),(9,273),(10,304),(11,334),(12,365)]
```

(1,31) *The last day in January is the 31st day of the year*

(7,212) *The last day in July is the 212th day of the year*

# toOrdinal and fromOrdinal

```
offsets = [(0,0),(1,31),(2,59),(3,90),(4,120),(5,151),(6,181),
(7,212),(8,243),(9,273),(10,304),(11,334),(12,365)]

toOrdinal (month, day) = days + day
   where
      (_,days) = offsets!!(month-1)

fromOrdinal ordDay =
      fromOrdinal' (reverse offsets) ordDay
   where
      fromOrdinal' ((month,lastDay):t) ordDay
         | ordDay > lastDay = (month + 1, ordDay - lastDay)
         | otherwise = fromOrdinal' t ordDay
      fromOrdinal' [] _ = error "invalid month?"
```

```
> toOrdinal (12,31)
365
```

```
> fromOrdinal 32
(2,1)
```

Recall:

```
> classdays  (1,15)  (5,6)  [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
...
```

Ordinal dates for (1,15) and (5,6) are 15 and 126, respectively.

With the Thursday-Tuesday pattern we'd see the ordinal dates progressing like this:

15, 20, 22, 27, 29, 34, 36, 41, ...

...

+5 +2 +5 +2 +5 +2 +5 ...

Imagine this series of calls to a helper, **showLecture**:

```
showLecture 1 15  'H'
showLecture 2  20 'T'
showLecture 3  22 'H'
showLecture 4  27  'T'
...
showLecture 32 125 'T'
```

Desired output:

```
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
```

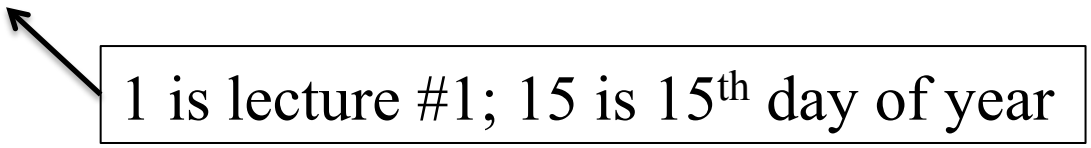
What computations do we need to transform

```
showLecture 1 15  'H'
```

into

```
"#1 H 1/15\n"?
```

We have: showLecture 1 15 'H'
We want: "#1 H 1/15"




Let's write showOrdinal :: Integer -> [Char]

```
> showOrdinal 15
"1/15"

showOrdinal ordDay = show month ++ "/" ++ show day
   where
      (month,day) = fromOrdinal ordDay
```

Now we can write showLecture:

```
showLecture lecNum ordDay dayOfWeek =
      "#" ++ show lecNum ++ " " ++ [dayOfWeek] ++
       " " ++ showOrdinal ordDay ++ "\n"
```

Recall:

```
showLecture 1 15  'H'
showLecture 2  20 'T'
...
showLecture 32 125 'T'
```

Desired output:

```
#1 H 1/15
#2 T 1/20
...
#32 T 5/5
```

Let's "cons up" a list out of the results of those calls...

```
> showLecture 1 15 'H' :
   showLecture 2 20 'T' :
   "...more..." :  -- I literally typed "...more..."
   showLecture 32 125 'T' : []
["#1 H 1/15\n","#2 T 1/20\n", "...more...","#32 T
5/5\n"]
```

How close are the contents of that list to what we need?

Now lets imagine a recursive function **showLectures** that builds up a list of results from **showLecture** calls:

```
showLectures 1 15 126 [('H',5),('T',2)]          "#1 H 1/15\n"
  showLectures 2 20 126 [('T',2),('H',5)]        "#2 T 1/20\n"
      ...
    showLectures 32 125 126 [('T',2),('H',5)]  "#32 T 5/5\n"
      showLectures 33 127 126 [('H',5),('T',2)]
```

Result:

```
["#1 H 1/15\n","#2 T 1/20\n", ... ,"#33 H 5/5\n"]
```

Now let's write **showLectures:**

```
showLectures  lecNum thisDay lastDay
                    (pair@(dayOfWeek, daysToNext):pairs)
  | thisDay > lastDay = []
  | otherwise =  showLecture lecNum thisDay dayOfWeek
    : showLectures (lecNum+1) (thisDay + daysToNext)
                   lastDay (pairs ++ [pair])
```

# **classdays**—top-level

Finally, a top-level function to get the ball rolling:

```
classdays first last pattern = putStr (concat result)
   where
      result =
         showLectures 1 (toOrdinal first) (toOrdinal last) pattern
```

Usage:

```
> classdays (1,15) (5,6) [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
...
#31 H 4/30
#32 T 5/5
```

Full source is in **spring16/haskell/classdays.hs**

**Note: next set of slides!**

# Errors

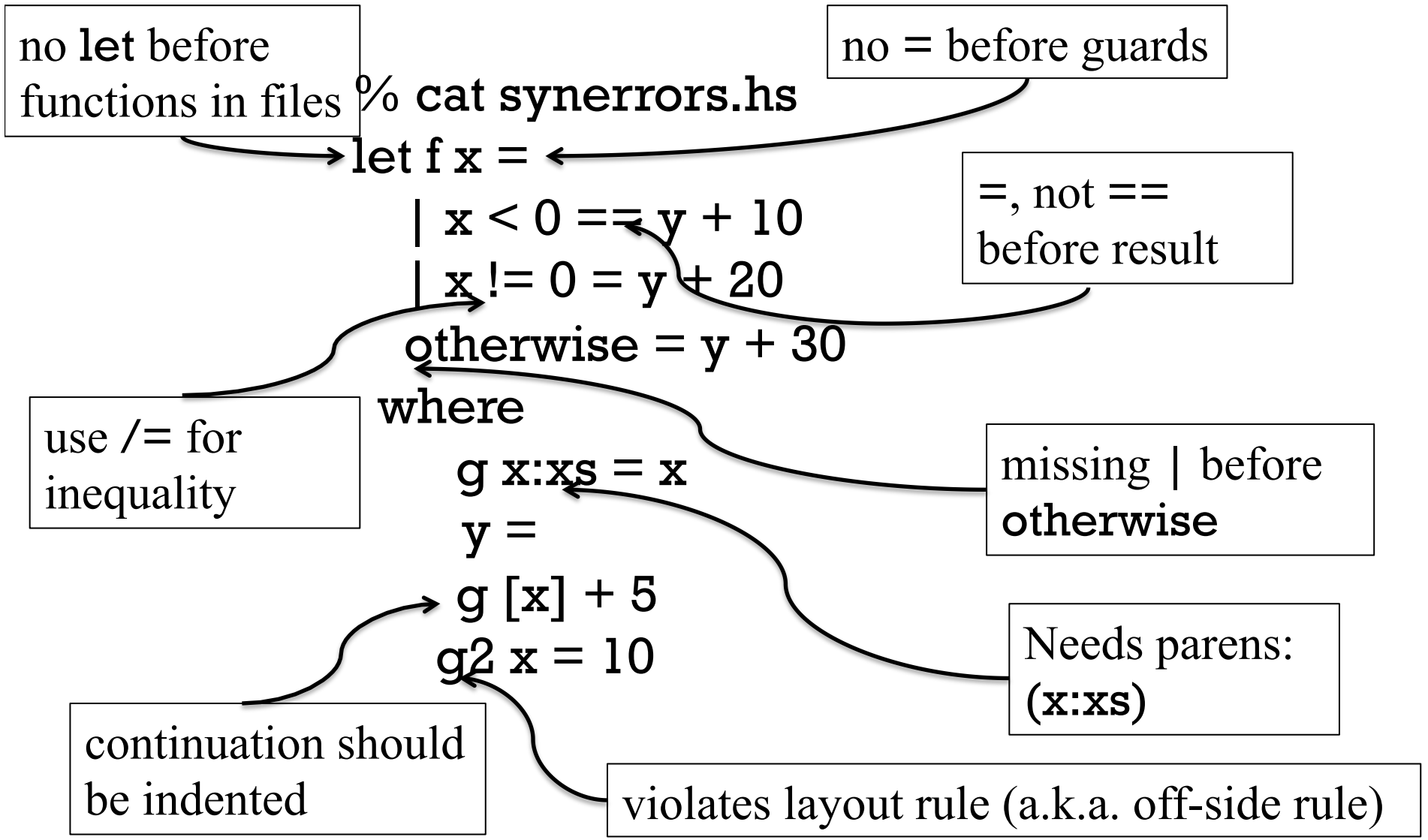
# Syntax errors

What syntax errors do you see in the following file?

```
% cat synerrors.hs
let f x =
    | x < 0 == y + 10
    | x != 0 = y + 20
   otherwise = y + 30
  where
      g x:xs = x
      y =
      g [x] + 5
     g2 x = 10
```

# Syntax errors, continued

What syntax errors do you see in the following file?



# Type errors

In my opinion, producing understandable messages for type errors is what **ghci** is worst at.

If only concrete types are involved, type errors are typically easy to understand.

```
> chr 'x'
  Couldn't match expected type `Int' with actual
    type `Char'
   In the first argument of `chr', namely 'x'
   In the expression: chr 'x'
   In an equation for `it': it = chr 'x'

> :type chr
chr :: Int -> Char
```

# Type errors, continued

Code and error:

```
f x y
    | x == 0 = []
    | otherwise = f x

 Couldn't match expected type `[a1]' with actual type
`t0 -> [a1]'
   In the return type of a call of `f'
   Probable cause: `f' is applied to too few arguments
   In the expression: f x
```

The error message is perfect in this case.

The first clause implies that `f` returns a list but the second clause returns a partial application, of type `t0 -> [a1]`, a contradiction.

# Type errors, continued

Code:

```
countEO (x:xs)
  | odd x = (evens, odds+1)
  | otherwise = (evens+1, odds)
 where (evens,odds) = countEO
```

Error:

```
Couldn't match expected type `(t3, t4)'
            with actual type `[t0] -> (t1, t2)'
  In the expression: countEO
  In a pattern binding: (evens, odds) = countEO
```

What's the problem?

It's expecting a tuple, (t3,t4) but it's getting a function, [t0] -> (t1,t2)

Typically, instead of getting errors about too few (or too many) function arguments, you get function types popping up in unexpected places.

# Type errors, continued

Is there an error in the following?

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

Another way to produce an infinite type:

```
let x = head x
```

```
Occurs check: cannot construct the infinite type:
  a0 = [a0]     ("a0 is a list of a0s"--whm)
  In the first argument of `(:)', namely `x'
  In the expression: x : f xs
  In an equation for `f': f (x : xs) = x : f xs
```

The second and third clauses are fine by themselves but together they create a contradiction.

Technique: Comment out clauses (and/or guards) to find the troublemaker, or incompatibilities between them.

# Type errors, continued

Recall **ord :: Char -> Int.**

Note these two errors:

```
> ord 5
  No instance for (Num Char) arising from the literal `5'
    Possible fix: add an instance declaration for (Num Char)

> length 3
  No instance for (Num [a0]) arising from the literal `3'
    Possible fix: add an instance declaration for (Num [a0])
```

The error "**No instance for (A B)**" means I want a **B** but got an **A**.

The suggested fix, adding an instance declaration, is always wrong in our simple Haskell world.

# Debugging

# Debugging in general

My advice in a nutshell for debugging in Haskell:

Don't need to do any debugging!

My usual development process in Haskell:

1. Work out expressions at the **`ghci`** prompt.
2. Write a function using those expressions and put it in a file.
3. Test that function at the **`ghci`** prompt.
4. Repeat with the next function.

With conventional languages I might write dozens of lines of code before trying them out.

With Haskell I might write a half-dozen lines of code before trying them out.

# Tracing

The **Debug.Trace** module provides a **trace** function that sneakily does output without getting embroiled with the I/O machinery.

Consider a trivial function:

```
f 1 = 10
f n = n * 5 + 7
```

Let's augment it with tracing:

```
import Debug.Trace
f 1 = trace "f: first case" 10
f n = trace "f: default case" n * 5 + 7
```

Execution:

```
> f 1
f: first case
10

> f 3
f: default case
22
```

# Tracing, continued

Here's **countEO** with tracing:

```
import Debug.Trace
countEO [] = (0,0)
countEO list@(x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
  where
   result = countEO xs
   (evens,odds) =
    trace ("countEO " ++ show xs ++ " --> " ++ show result) result
```

Execution:

```
> countEO [3,2,4]
(countEO [] --> (0,0)
countEO [4] --> (1,0)
countEO [2,4] --> (2,0)
2,1)
```

Before tracing the **where** was:

```
(evens,odds) = countEO xs
```

# ghci's debugger

**ghci** does have some debugging support but debugging is *expression-based*. Here's some simple interaction with it on **countEO**:

```
> :step countEO [3,2,4]
Stopped at countEO.hs:(1,1)-(6,29)
_result :: (t, t1) = _
> :step
Stopped at countEO.hs:3:7-11
_result :: Bool = _
x :: Integer = 3
> :step
Stopped at countEO.hs:3:15-29
_result :: (t, t1) = _
evens :: t = _
odds :: t1 = _
> :step
(Stopped at countEO.hs:6:20-29
_result :: (t, t1) = _
xs :: [Integer] = [2,4]
```

```
countEO [] = (0,0)
countEO (x:xs)
   | odd x = (evens, odds+1)
   | otherwise = (evens+1, odds)
  where
    (evens,odds) = countEO xs
```

**_result** shows type of current expression

Arbitrary expressions can be evaluated at the > prompt (as always).

# More on debugging

There's lots more to the debugging support in **gchi**.

**https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci-debugger.html**

**http://www.youtube.com/watch?v=1OYljb_3Cdg**

GHCi's Debugger - Haskell from Scratch #2

In 352, I promote **gdb** heavily but this is the first time in 372 that I've ever mentioned tracing and debugging for Haskell.

Again, my advice in a nutshell for debugging in Haskell:

Don't need to do any debugging!

# Excursion:
# A little bit with infinite lists and lazy evaluation

# Infinite lists

Here's a way we've seen to make an infinite list:

```
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,2
2,23,24,25,26,27,28,29,30,31,32,^C
```

What does the following **let** create?

```
> let f = (!!) [1,3..]
f :: Int -> Integer
```

A function that produces the Nth odd number, zero-based.

Yes, we could say **nthOdd n = (n*2)+1** but that wouldn't be nearly as much fun! (This *is* functional programming!)

I want you to be cognizant of performance but don't let concerns about performance stifle creativity!

# Lazy evaluation

Consider the following **let**. Why does it complete?

```
> let fives=[5,10..]
fives :: [Integer]
```

A simple answer we'll later refine:

Haskell uses *lazy evaluation*. Values aren't computed until needed.

How will the following expression behave?

```
> take (head fives) fives
[5,10,15,20,25]
```

Haskell computes the first element of **fives**, and then four more elements of **fives**.

# Lazy evaluation, continued

Here is an expression that is said to be *non-terminating:*

```
> length fives
...when tired of waiting...^C Interrupted.
```

The value of **length fives** is said to be ⊥, read "bottom".

But, we can bind a name to **length fives**:

```
> let numFives = length fives
numFives :: Int
```

That completes because Haskell hasn't yet needed to compute a value for **length fives**.

We can get another coffee break by asking Haskell to print the value of **numFives:**

```
> numFives
...after a while...^CInterrupted.
```

# Lazy evaluation, continued

We can use **:print** to explore lazy evaluation:

```
> let fives = [5,10..]

> :print fives
fives = (_t2::[Integer])

> take 3 fives
[5,10,15]
```

What do you think **:print fives** will now show?

```
> :print fives
fives = 5 : 10 : 15 : (_t3::[Integer])
```

# Lazy evaluation, continued

Consider this function:

```
f x y z = if x < y then y else z
```

Will the following expressions terminate?

```
> f 2 3 (length [1..])
3

> f 3 2 (length [1..])
^CInterrupted.

> f 3 (length [1..]) 2
^CInterrupted.
```

# Sidebar: Lazy vs. non-strict

In fact, Haskell doesn't fully meet the requirements of lazy evaluation.
The word "lazy" appears only once in the Haskell 2010 Report.

What Haskell does provide is *non-strict evaluation*:
Function arguments are not evaluated until a value is needed.

From the previous slide:

```
f x y z = if x < y then y else z
```

Reconsider the following wrt. non-strict evaluation:

```
> f 2 3 (length [1..]) -- Third argument is not used
3

> f 3 2 (length [1..]) -- Third argument is used
^CInterrupted.
```

See **wiki.haskell.org/Lazy_vs._non-strict** for the fine points of lazy evaluation vs. non-strict evaluation. Google for more, too.

# More with infinite lists

Speculate: Can infinite lists be concatenated?

```
> let values = [1..] ++ [5,10..] ++ [1,2,3]
> :t values
values :: [Integer]
```

What will the following do?

```
> let nums = [1..]
> nums > [1,2,3,5]
False
```

False due to lexicographic comparison—`4 < 5`

How far did evaluation of `nums` progress?

```
> :print nums
nums = 1 : 2 : 3 : 4 : (_t2::[Integer])
```

# Infinite expressions

What does the following expression mean?

```
> let threes = 3 : threes
```

**threes** is a list whose head is **3** and whose tail is **threes**!

```
> take 5 threes
[3,3,3,3,3]
```

How about the following?

```
> let xys = ['x','y'] ++ xys

> take 5 xys
"xyxyx"

> xys !! 100000000
'x'
```

One more:

```
> let x = 1 + x
> x
^CInterrupted.
```

# intsFrom

Problem: write a function **intsFrom** that produces the integers from a starting value.

```
> intsFrom 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...

> intsFrom 1000
[1000,1001,1002,1003,1004,1005,1006,1007,1008,...

> take 5 (intsFrom 1000000)
[1000000,1000001,1000002,1000003,1000004]
```

Solution:

```
intsFrom n = n : intsFrom (n + 1)
```

Does `length (intsFrom (minBound::Int))` terminate?

# repblock

The **cycle** function returns an infinite number of repetitions of its argument, a list:

```
> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

Using **cycle**, write **repblock**:

```
> repblock "abc" 4 3 -- string width height
abca
bcab
cabc

> repblock "+-" 3 2
+-+
-+-
```

# **repblock**, continued

At hand: **repblock s width height**

Approach: Create an infinite repetition of **s** and take **width**-sized lines **height** times.

Solution:

```
repblock :: String -> Int -> Int -> IO ()
repblock s width height =
   putStr (repblock' (cycle s) width height)

repblock' :: String -> Int -> Int -> String
repblock' s width height
   | height == 0 = ""
   | otherwise = take width s ++ "\n" ++
      repblock' (drop width s) width (height - 1)
```

# Higher-order functions

# Remember: Functions are values

Remember:

A fundamental characteristic of a functional language: functions are values that can be used as flexibly as values of other types.

Here are some more examples of that. What do the following do?

```
> let nums = [1..10]

> (if 3 < 4 then head else last) nums
1

> let funcs = (tail, (:) 100)

> fst funcs nums
[2,3,4,5,6,7,8,9,10]

> snd funcs nums
[100,1,2,3,4,5,6,7,8,9,10]
```

# Lists of functions

We can work with lists of functions:

```
> let funcs = [head, last]

> funcs
[<function>,<function>]

> let nums = [1..10]

> head funcs nums
1

> (funcs!!1) nums
10

> last [last]
<function>
```

# Lists of functions, continued

Is the following valid?

```
> [take, tail, init]
Couldn't match type `[a2]' with `Int'
   Expected type: Int -> [a0] -> [a0]
        Actual type: [a2] -> [a2]
   In the expression: init
```

What's the problem?

**take** does not have the same type as **tail** and **init**.

Puzzle: Make **[take, tail, init]** valid by adding two characters.

```
> [take 5, tail, init]
[<function>,<function>,<function>]
```

# Comparing functions

Can functions be compared?

```
> add == plus
No instance for (Eq (Integer -> Integer -> Integer))
    arising from a use of `=='
In the expression: add == plus
```

You might see a proof based on this in 473:

If we could determine if two arbitrary functions perform the same computation, we could solve the halting problem, which is considered to be unsolvable.

Because functions can't be compared, this version of **length** won't work for lists of functions: (Its type: **(Num a, Eq t) => [t] -> a**)

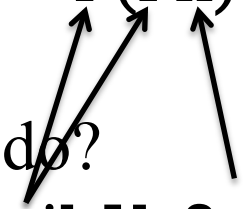
```
len list@(_h:t)
    | list == [] = 0
    | otherwise = 1 + len t
```

# A simple *higher-order function*

Definition: A *higher-order function* is a function that has one or more arguments that are functions.

**twice** is a higher-order function with <u>two</u> arguments: **f** and **x**

```
twice f x = f (f x)
```



What does it do?

```
> twice tail [1,2,3,4,5]
[3,4,5]

> tail (tail [1,2,3,4,5])
[3,4,5]
```

.

# twice, continued

At hand:

```
> let twice f x = f (f x)
> twice tail [1,2,3,4,5]
[3,4,5]
```

Let's make the precedence explicit:

```
> ((twice tail) [1,2,3,4,5])
[3,4,5]
```

Consider a partial application...

```
> let t2 = twice tail    -- like let t2 x = tail (tail x)
> t2
<function>
it :: [a] -> [a]
```

# twice, continued

At hand:

```
> let twice f x = f (f x)
> twice tail [1,2,3,4,5]
[3,4,5]
```

Let's give twice a partial application!

```
> twice (drop 2) [1..5]
[5]
```

Let's make a partial application with a partial application!

```
> twice (drop 5)
<function>
> it ['a'..'z']
"klmnopqrstuvwxyz"
```

Try these!

```
twice (twice (drop 3)) [1..20]
twice (twice (take 3)) [1..20]
```

# **twice**, continued

At hand:

```
twice f x = f (f x)
```

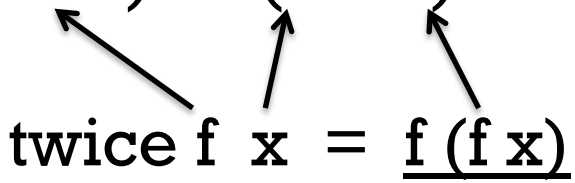
A *higher-order function* is a function that has one or more arguments that are functions.

What's the the type of **twice**?

```
> :t twice
twice :: (t -> t) -> t -> t
```

Parentheses added to show precedence:

```
twice :: (t -> t) -> (t -> t)
```




What's the correspondence between the elements of the clause and the elements of the type?

# The `map` function

# The Prelude's **map** function

Recall **double x = x * 2**

**map** is a Prelude function that applies a function to each element of a list, producing a new list:

```
> map double [1..5]
[2,4,6,8,10]

> map length (words "a few words")
[1,3,5]

> map head (words "a few words")
"afw"
```

Is **map** a higher order function?

Yes! It's first argument is a function.

# `map`, continued

At hand:

```
> map double [1..5]
[2,4,6,8,10]
```

Write it!

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

What is its type?

```
map :: (a -> b) -> [a] -> [b]
```

What's the relationship between the length of the input and output lists?

The lengths are <u>always</u> the same.

# map, continued

Mapping (via **map**) is applying a transformation (a function) to each of the values in a list, always producing a new list of the same length.

```
> map chr [97,32,98,105,103,32,99,97,116]
"a big cat"

> map isLetter it
[True,False,True,True,True,False,True,True,True]

> map not it
[False,True,False,False,False,True,False,False,False]

> map head (map show it) -- Note: show True is "True"
"FTFFFTFFF"
```

# Sidebar: **map** can go parallel

Here's another map:

```
> map weather [85,55,75]
["Hot!","Cold!","Nice"]
```

This is equivalent:

```
> [weather 85, weather 55, weather 75]
["Hot!","Cold!","Nice"]
```

Because functions have no side effects, we can immediately turn a mapping into a parallel computation. We might start each function call on a separate processor and combine the values when all are done.

# `map` and partial applications

What's the result of these?

```
> map (add 5) [1..10]
[6,7,8,9,10,11,12,13,14,15]

> map (drop 1) (words "the knot was cold")
["he","not","as","old"]

> map (replicate 5) "abc"
["aaaaa","bbbbb","ccccc"]
```

# map and partial applications, cont.

What's going on here?

```
> let f = map double
> f [1..5]
[2,4,6,8,10]

> map f [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]
```

Here's the above in one step:

```
> map (map double) [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]
```

Here's one way to think about it:

```
[(map double) [1..3], (map double) [10..15]]
```

# Sections

Instead of using **map (add 5)** to add 5 to the values in a list, we should use a *section* instead: (it's the idiomatic way!)

```
> map (5+) [1,2,3]
[6,7,8]   -- [5+1, 5+2, 5+3]
```

More sections:

```
> map (10*) [1,2,3]
[10,20,30]

> map (++"*") (words "a few words")
["a*","few*","words*"]

> map ("*"++) (words "a few words")
["*a","*few","*words"]
```

# Sections, continued

Sections have one of two forms:

(*infix-operator value*) Examples: (+5), (/10)

(*value infix-operator*) Examples: (5*), ("x"++)

Iff the operator is commutative, the two forms are equivalent.

```
> map (3<=) [1..4]
[False,False,True,True]
```

[3 <= 1, 3 <= 2, 3 <= 3, 3 <= 4]

```
> map (<=3) [1..4]
[True,True,True,False]
```

[1 <= 3, 2 <= 3, 3 <= 3, 4 <= 4]

Sections aren't just for **map**; they're a general mechanism.

```
> twice (+5) 3
13
```

# travel, revisited

# Now that we're good at recursion...

Some of the problems on the next assignment will encourage working with higher-order functions by prohibiting you from writing any recursive functions!

Think of it as isolating muscle groups when weight training.

Here's a simple way to avoid what's prohibited:

*Pretend that you no longer understand recursion!*

*What's a base case? Is it related to baseball?*
*Why would a function call itself? How's it stop?*
*Is a recursive plunge refreshing?*

If you were UNIX machines, I'd do **`chmod 0`** on an appropriate section of your brains.

# travel revisited

Recall our traveling robot: (slide 195)

```
> travel "nnee"
"Got lost"

> travel "nnss"
"Got home"
```

Recall our approach:

Argument value: "nnee"

Mapped to tuples: (0,1) (0,1) (1,0) (1,0)

Sum of tuples: (2,2)

How can we solve it non-recursively?

# travel, continued

Recall:

```
> :t mapMove
mapMove :: Char -> (Int, Int)

> mapMove 'n'
(0,1)
```

Now what?

```
> map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
```

Can we sum them with map?

# travel, continued

We have:

```
> let disps= map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
```

We want: (2,3)

Any ideas?

```
> :t fst
fst :: (a, b) -> a

> map fst disps
[0,0,1,1,0]

> map snd disps
[1,1,0,0,1]
```

# travel, revisited

We have:

```
> let disps= map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
> map fst disps
[0,0,1,1,0]
> map snd disps
[1,1,0,0,1]
```

We want: (2,3)

Ideas?

```
> :t sum
sum :: Num a => [a] -> a

> (sum (map fst disps), sum (map snd disps))
(2,3)
```

# travel—Final answer

```
travel :: [Char] -> [Char]
travel s
    | totalDisp == (0,0) = "Got home"
    | otherwise = "Got lost"
   where
      disps = map mapMove s
      totalDisp = (sum (map fst disps),
                   sum (map snd disps))
```

Did we have to understand recursion to write this?

No.

Did we write any recursive functions?

No.

Did we use any recursive functions?

Maybe so, but using recursive functions doesn't violate the prohibition at hand.

# Filtering

# Filtering

Another higher order function in the Prelude is **filter**:

```
> filter odd [1..10]
[1,3,5,7,9]

> filter isDigit "(800) 555-1212"
"8005551212"
```

What's **filter** doing?

What is the type of **filter**?

```
filter :: (a -> Bool) -> [a] -> [a]
```

Think of **filter** as filtering *in*, not filtering *out*.

# **filter**, continued

More...

```
>  filter (<= 5) (filter odd [1..10])
[1,3,5]

> map (filter isDigit) ["br549", "24/7"]
["549","247"]

> filter (`elem` "aeiou") "some words here"
"oeoee"
```

*Note that (`elem` ...) is a section!*

```
elem :: Eq a => a -> [a] -> Bool
```

# filter, continued

At hand:

```
> filter odd [1..10]
[1,3,5,7,9]

> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Let's write filter!

```
filter _ [] = []
filter f (x:xs)
    | f x = x : filteredTail
    | otherwise = filteredTail
  where
    filteredTail = filter f xs
```

# **filter** uses a *predicate*

**filter**'s first argument (a function) is called a *predicate* because inclusion of each value is predicated on the result of calling that function with that value.

Several Prelude functions use predicates. Here are two:

```
all :: (a -> Bool) -> [a] -> Bool
> all even [2,4,6,8]
True
> all even [2,4,6,7]
False

dropWhile :: (a -> Bool) -> [a] -> [a]
> dropWhile isSpace "  testing  "
"testing  "
> dropWhile isLetter it
"  "
```

# map vs. filter

For reference:

```
> map double [1..10]
[2,4,6,8,10,12,14,16,18,20]

> filter odd [1..10]
[1,3,5,7,9]
```

**map**:

transforms a list of values

length *input* == length *output*

**filter**:

selects values from a list

0 <= length *output* <= length *input*

**map** and **filter** are in Python and JavaScript, to name two of many languages having them. (And, they're trivial to write!)

Put a big "X" on 281-282 and go to slide 305!

~~Anonymous functions~~

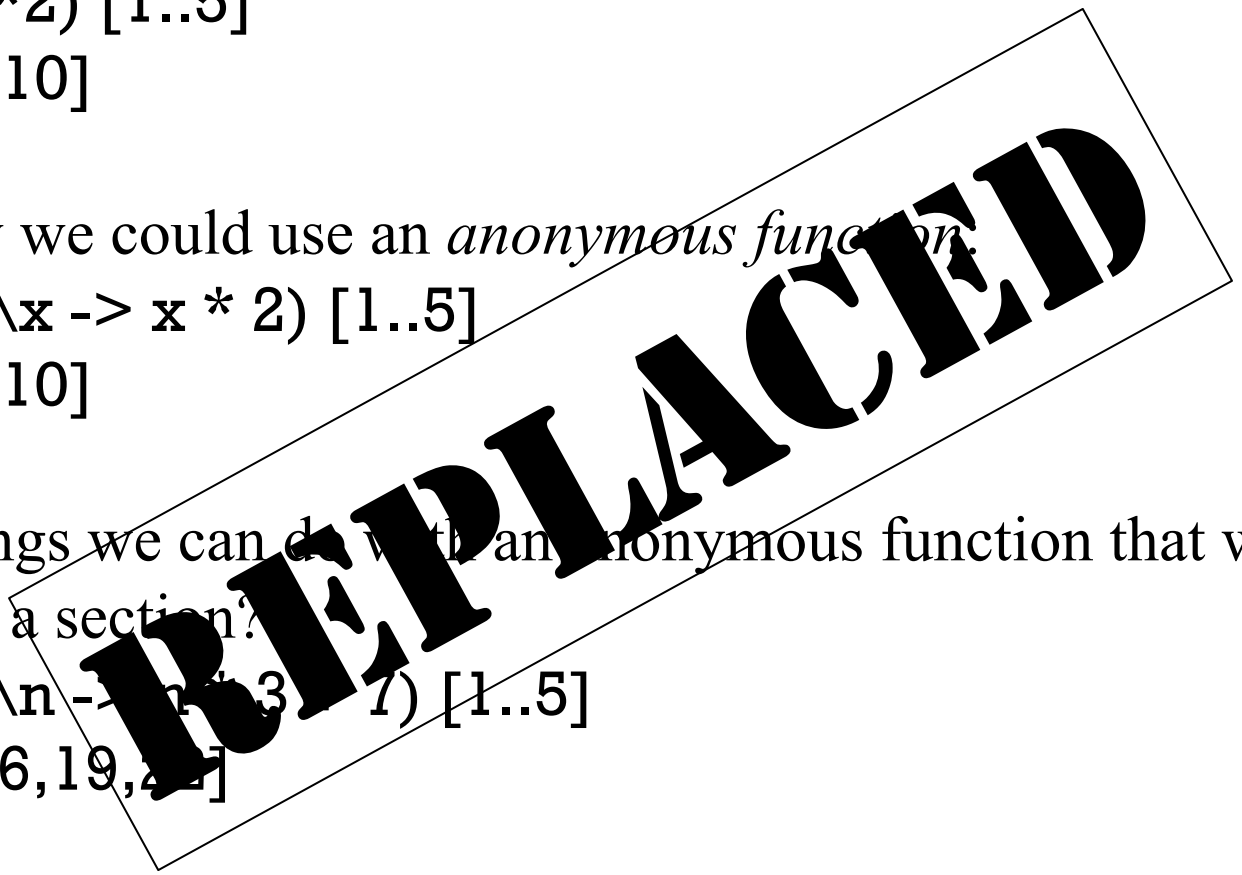
# Anonymous functions

We can map a section to double the numbers in a list:

```
> map (*2) [1..5]
[2,4,6,8,10]
```

Alternatively we could use an *anonymous function*:

```
> map (\x -> x * 2) [1..5]
[2,4,6,8,10]
```




What are things we can do with an anonymous function that we can't do with a section?

```
> map (\n -> n*3 - 7) [1..5]
[10,13,16,19,22]

> filter (\x -> head x == last x) (words "pop top suds")
["pop","suds"]
```

# Anonymous functions, continued

The general form:

\ *pattern1 ... patternN -> expression*

Simple syntax suggestion: enclose the whole works in parentheses.

```
map (\x -> x * 2) [1..5]
```

The typical use case for an anonymous function is a single instance of supplying a higher order function with a computation that can't be expressed with a section or partial application.



Anonymous functions are also called *lambdas*, *lambda expressions*, and *lambda abstractions*.

The \ character was chosen due to its similarity to λ, used in Lambda calculus, another system for expressing computation.

# Larger example: longest

# Example: longest line(s) in a file

Imagine a program to print the longest line(s) in a file, along with their line numbers:

```
% runghc longest.hs /usr/share/dict/web2
72632:formaldehydesulphoxylate
140339:pathologicopsychological
175108:scientificophilosophical
200796:tetraiodophenolphthalein
203042:thyroparathyroidectomize
```

What are some ways in which we could approach it?

# **longest**, continued

Let's work with a shorter file for development testing:

```
% cat longest.1
data
to
test
```

**readFile** in the Prelude lazily returns the full contents of a file as a string:

```
> readFile "longest.1"
"data\nto\ntest\n"
```

To avoid wading into I/O yet, let's focus on a function that operates on a string of characters (the full contents of a file):

```
> longest "data\nto\ntest\n"
"1:data\n3:test\n"
```

# **longest**, continued

Let's work through a series of transformations of the data:

```
> let bytes = "data\nto\ntest\n"

> let lns = lines bytes
["data","to","test"]
```

Note: To save space, values of **let** bindings are being shown immediately after each **let**. E.g., **> lns** is not shown above.

Let's use **zip3** and **map length** to create (length, line-number, line) triples:

```
> let triples = zip3 (map length lns) [1..] lns
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
```

# **longest**, continued

We have (length, line-number, line) triples at hand:

```
> triples
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
```

Let's use **sort :: Ord a => [a] -> [a]** on them:

```
> let sortedTriples = reverse (sort triples)
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
```

Note that by having the line length first, triples are sorted first by line length, with ties resolved by line number.

We use **reverse** to get a descending order.

If line length weren't first, we'd instead use

```
Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

and supply a function that returns an **Ordering**.

# **longest**, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
```

We'll handle ties by using **takeWhile** to get all the triples with lines of the maximum length.

Let's use a helper function to get the first element of a 3-tuple:

```
> let first (len, _, _) = len
> let maxLength = first (head sortedTriples)
4
```

**first** will be used in another place but were it not for that we might have used a pattern:

```
let (maxLength,_,_) = head sortedTriples
```

# **longest**, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]

> maxLength
4
```

Let's use **takeWhile :: (a -> Bool) -> [a] -> [a]** to get the triples having the maximum length:

```
> let maxTriples = takeWhile
    (\triple -> first triple  == maxLength) sortedTriples
[(4,3,"test"),(4,1,"data")]
```

anonymous function for takeWhile

# **longest**, continued

At hand:

```
> maxTriples
[(4,3,"test"),(4,1,"data")]
```

Let's map an anonymous function to turn the triples into lines prefixed with their line number:

```
> let linesWithNums =
      map (\(_,num,line) -> show num ++ ":" ++ line)
            maxTriples
["3:test","1:data"]
```

We can now produce a ready-to-print result:

```
> let result = unlines (reverse linesWithNums)
> result
"1:data\n3:test\n"
```

# longest, continued

Let's package up our work into a function:

```
longest bytes = result
    where
        lns = lines bytes
        triples = zip3 (map length lns) [1..] lns
        sortedTriples = reverse (sort triples)
        maxLength = first (head sortedTriples)
        maxTriples = takeWhile
            (\triple -> first triple  == maxLength) sortedTriples
        linesWithNums =
            map (\(_,num,line) -> show num ++ ":" ++ line)
            maxTriples
        result = unlines (reverse linesWithNums)

        first (x,_,_) = x
```

# longest, continued

At hand:

```
> longest "data\nto\ntest\n"
"1:data\n3:test\n"
```

Let's add a **main** that handles command-line args and does I/O:

```
% cat longest.hs
import System.Environment (getArgs)
import Data.List (sort)

longest bytes = ...from previous slide...

main = do
   args <- getArgs  -- Get command line args as list
   bytes <- readFile (head args)
   putStr (longest bytes)
```

Execution:

```
$ runghc longest.hs /usr/share/dict/words
39886:electroencephalograph's
```

# Composition

# Function composition

Given two functions **f** and **g**, the *composition* of **f** and **g** is a function **c** that for all values of **x**, (**c x**) equals (**f (g x)**)

Here is a primitive **compose** function that applies two functions in turn:

```
> let compose f g x = f (g x)
```

How many arguments does compose have?

Its type:

```
(b -> c) -> (a -> b) -> a -> c
```

```
> compose init tail [1..5]
[2,3,4]
```

```
> compose signum negate 3
-1
```

# Composition, continued

Haskell has a function composition operator. It is a dot (.)

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Its two operands are functions, and its result is a function.

```
> let numwords = length . words

> numwords "just testing this"
3

> map numwords ["a test", "up & down", "done"]
[2,3,1]
```

# Composition, continued

Problem: Using composition create a function that returns the next-to-last element in a list:

```
> ntl [1..5]
4

> ntl "abc"
'b'
```

Two solutions:

```
ntl = head . tail . reverse
ntl = last . init
```

Problem: Recall `twice f x = f (f x)`. Define `twice` as a composition.

```
twice f = f . f
```

# Composition, continued

Problem: Create a function to remove the digits from a string:

```
> rmdigits "Thu Feb  6 19:13:34 MST 2014"
"Thu Feb   :: MST "
```

Solution:

```
> let rmdigits = filter (not  .  isDigit)
```

Given the following, describe f:

```
> let f = (*2) . (+3)

> map f [1..5]
[8,10,12,14,16]
```

Would an anonymous function be a better choice?

# Composition, continued

Given the following, what's the type of **numwords**?

```
> :type words
words :: String -> [String]

> :type length
length :: [a] -> Int

> let numwords = length . words
```

Type:

```
numwords :: String -> Int
```

Assuming a composition is valid, the type is based only on the input of the rightmost function and the output of the leftmost function.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

**Note: Next set of slides!**

# REPLACEMENTS

Put a big "X" on slides 299-300 in the 223-300 set and continue with this set.

# Composition, continued

Consider the following:

```
> let s = "It's on!"
> map head (map show (map not (map isLetter s)))
"FFTFTFFT"
```

Can we use composition to simplify it?

```
> map (head . show . not . isLetter) s
"FFTFTFFT"
```

In general, because there are no side-effects,

```
map f (map g x)
```

is equivalent to

```
map (f . g) x
```

If `f` and `g` did output, how would the output of the two cases differ?

# Point-free style

Recall **rmdigits**:

```
> rmdigits "Thu Feb  6 19:13:34 MST 2014"
"Thu Feb   :: MST "
```

What the difference between these two bindings for **rmdigits**?

```
rmdigits s = filter (not  .  isDigit) s

rmdigits = filter (not  .  isDigit)
```

The latter version is said to be written in *point-free style.*

A point-free binding of a function **f** has NO parameters!

# Point-free style, continued

I think of point-free style as a natural result of fully grasping partial application and operations like composition.

Although it was nameless, we've already seen examples of point-free style, such as these:

```
nthOdd = (!!) [1,3..]
t2 = twice tail
numwords = length . words
ntl = head . tail . reverse
```

There's nothing too special about point-free style but it does save some visual clutter. It is commonly used.

The term "point-free" comes from topology, where a point-free function operates on points that are not specifically cited.

# Point-free style, continued

Problem: Using point-free style, bind **len** to a function that works like the Prelude's **length**.

Handy:

```
> :t const
const :: a -> b -> a

> const 10 20
10

> const [1] "foo"
[1]
```

Solution:

```
len = sum . map (const 1)
```

See also: *Tacit programming* on Wikipedia

Go to slide 312

# Anonymous functions (second attempt at 281-283)

# Anonymous functions

Imagine that for every number in a list we'd like to double it and then subtract 5.

Here's one way to do it:

```
> let f n = n * 2 - 5
> map f [1..5]
[-3,-1,1,3,5]
```

We could instead use an *anonymous function* to do the same thing:

```
> map (\n -> n * 2 - 5) [1..5]
[-3,-1,1,3,5]
```

Which do you like better, and why?

# Anonymous functions, continued

At hand:

```
let f n = n * 2 - 5
map f [1..5]
```

vs.

```
map (\n -> n * 2 - 5) [1..5]
```

If a computation is only used in one place, using an anonymous function lets us specify it on the spot, directly associating its definition with its only use.

We also don't need to think up a name for the function! ☺

# Anonymous functions, continued

The general form of an anonymous function:

```
\ pattern1 ... patternN -> expression
```

Simple syntax suggestion: enclose the whole works in parentheses.

```
map (\n -> n * 2 - 5) [1..5]
```

Anonymous functions are also called *lambda abstractions* (H10*),* *lambda expressions*, and just *lambdas* (LYAH).

The \ character was chosen due to its similarity to λ, used in the *lambda calculus*, another system for expressing computation.

The typical use case for an anonymous function is a single instance of supplying a higher order function with a computation that can't be expressed with a section or partial application.

# Anonymous functions, continued

Speculate: What will **ghci** respond with?

```
> \x y -> x + y * 2
<function>
> it 3 4
11
```

The expression `\x y -> x + y * 2` produces a function value.

Here are three ways to bind the name **double** to a function that doubles a number:

```
double x = x * 2

double = \x -> x * 2

double = (*2)
```

# Anonymous functions, continued

Anonymous functions are commonly used with higher order functions such as **map** and **filter**.

```
> map (\w -> (length w, w)) (words "a test now")
[(1,"a"),(4,"test"),(3,"now")]

> map (\c -> "{" ++ [c] ++ "}") "anon."
["{a}","{n}","{o}","{n}","{.}"]

> filter (\x -> head x == last x) (words "pop top suds")
["pop","suds"]
```

In the above examples, the anonymous functions are somewhat like the bodies of loops in imperative languages.

Go to slide 283

# Hocus pocus with higher-order functions

# Mystery function

What's this function doing?

```
f  a = g
   where
      g  b = a + b
```

Type?

```
f :: Num a => a -> a -> a
```

Interaction:

```
> let f ' = f 10
> f ' 20
30

> f 3 4
7
```

# DIY Currying

Fact:

Curried function definitions are really just *syntactic sugar*—they just save some clutter. The don't provide something we can't do without.

Compare these two completely equivalent declarations for **add**:

```
add x y = x + y
```

```
add x = add'
    where
        add' y = x + y
```

The **x** used in **add'** refers to the **x** parameter of **add**.

The result of the call **add 5** is essentially this function:

```
add' y = 5 + y
```

The combination of the code for **add'** and the binding for **x** is known as a *closure*. It contains what's needed for execution.

# Sidebar: Syntactic sugar

In 1964 Peter Landin coined the term "syntactic sugar".

A language construct that makes something easier to express but doesn't add a new capability is called *syntactic sugar*. It simply makes the language "sweeter" for human use.

Two examples from C:

- "`abc`" is equivalent to a `char` array initialized with {'a', 'b', 'c', '\0'}

- `a[i]` is equivalent to `*(a + i)`

What's an example of syntactic sugar in Java?

The "enhanced for": `for (int i: a) { ... }`

# Syntactic sugar, continued

In Haskell a list like `[5, 2, 7]` can be expressed as `5:2:7:[]`.

Is that square-bracket list literal notation syntactic sugar?

What about `[1..]`, `[1,3..]`, `['a'..'z']`?

The **`Enum`** type class has **`enumFrom`**, **`enumFromTo`**, etc.

Recall these equivalent bindings for **`double`**:

```
double x = x * 2
double = \x -> x * 2
```

Is the first form just syntactic sugar?

What if a function has multiple clauses?

Are anonymous functions syntactic sugar?

# Syntactic sugar, continued

"Syntactic sugar causes cancer of the semicolon."
—Alan J. Perlis.

Another Perlis quote:

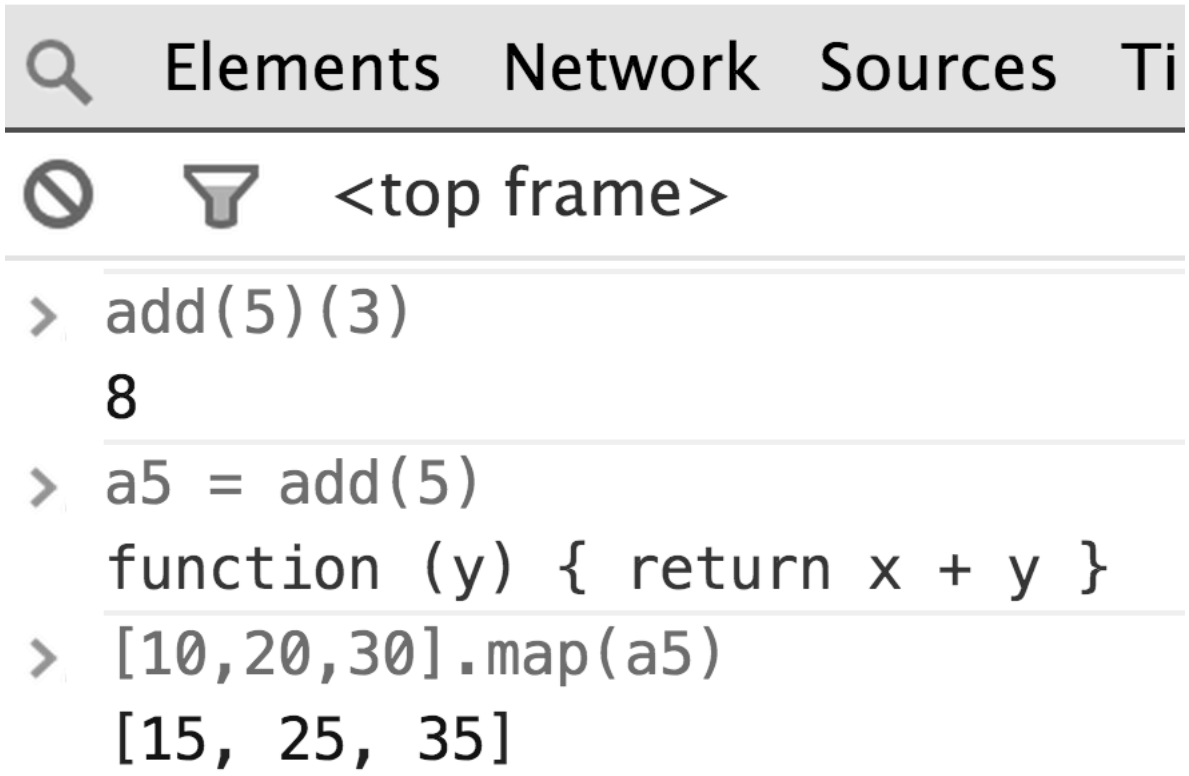
"A language that doesn't affect the way you think about programming is not worth knowing."

Perlis was the first recipient of the ACM's Turing Award.

# DIY currying in JavaScript

JavaScript doesn't provide the syntactic sugar of curried function definitions but we can do this:

```
function add(x) {
    return function (y) { return x + y }
    }
```




Try it in Chrome!

View>Developer> JavaScript Console brings up a console.

Type in the code for **add** on one line.

# DIY currying in Python

```
>>> def add(x):
...         return lambda y: x + y
...

>>> f = add(5)

>>> type(f)
<type 'function'>

>>> map(f, [10,20,30])
[15, 25, 35]
```

# Another mystery function

Here's another mystery function:

```
> let m f x y = f y x

> :type m
m :: (t1 -> t2 -> t) -> t2 -> t1 -> t
```

Can you devise a call to `m`?

```
> m add 3 4
7

> m (++) "a" "b"
"ba"
```

What is `m` doing? What could `m` be useful for?

# flip

At hand:

```
m f x y = f y x
```

m is actually a Prelude function named flip:

```
> :t flip
flip :: (a -> b -> c) -> b -> a -> c

> flip take [1..10] 3
[1,2,3]

> let ftake = flip take
> ftake [1..10] 3
[1,2,3]
```

Any ideas on how to use it?

# flip, continued

At hand:

```
flip f x y = f y x

> map (flip take "Haskell") [1..7]
["H","Ha","Has","Hask","Haske","Haskel","Haskell"]
```

Problem: write a function that behaves like this:

```
> f 'a'
["a","aa","aaa","aaaa","aaaaa",...
```

Solution:

```
f x = map (flip replicate x) [1..]
```

# **flip**, continued

From assignment 3:

```
> splits "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

Some students have noticed the Prelude's **splitAt**:

```
> splitAt 2 [10,20,30,40]
([10,20],[30,40])
```

Problem: Write **splits** using higher order functions but no explicit recursion.

Solution:

```
splits list = map (flip splitAt list) [1..(length list - 1)]
```

# The $ operator

$ is the "application operator". Note what :info shows:

```
> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $      -- right associative infix operator with very
                -- low precedence
```

The following binding of $ uses an infix syntax:

```
f $ x  =  f x        -- Equivalent: ($) f x = f x
```

Usage:

```
> negate $ 3 + 4
-7
```

What's the point of it?

# The $ operator, continued

$ is a low precedence, right associative operator that applies a function to a value:

```
f $ x = f x
```

Because + has higher precedence than $, the expression

```
negate $ 3 + 4
```

groups like this:

```
negate $ (3 + 4)
```

How does the following expression group?

```
filter (>3) $ map length $ words "up and down"
```

```
filter (>3) (map length (words "up and down"))
```

Don't confuse $ with . (composition)!

# Currying the uncurried

Problem: We're given a function whose argument is a 2-tuple but we wish it were curried so we could use a partial application of it.

```
g :: (Int, Int) -> Int
g (x,y) = x^2 + 3*x*y + 2*y^2

> g (3,4)
77
```

Solution: Curry it with **curry** from the Prelude!

```
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Your problem: Write **curry**!

# Currying the uncurried, continued

At hand:

```
> g (3,4)
77
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Here's **curry**, and use of it:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)

> let cg = curry g
> :type cg
cg :: Int -> Int -> Int

> cg 3 4
77
```

# Currying the uncurried, continued

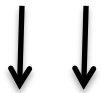
At hand:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

The key: **(curry g 3)** is a partial application of **curry**!

Call: **curry g 3**



Dcl: **curry f x y = f (x, y)**

**= g (3, y)**

# Currying the uncurried, continued

At hand:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
```
*(parentheses added)*

```
curry f x y = f (x, y)
```

```
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Let's get **flip** into the game!

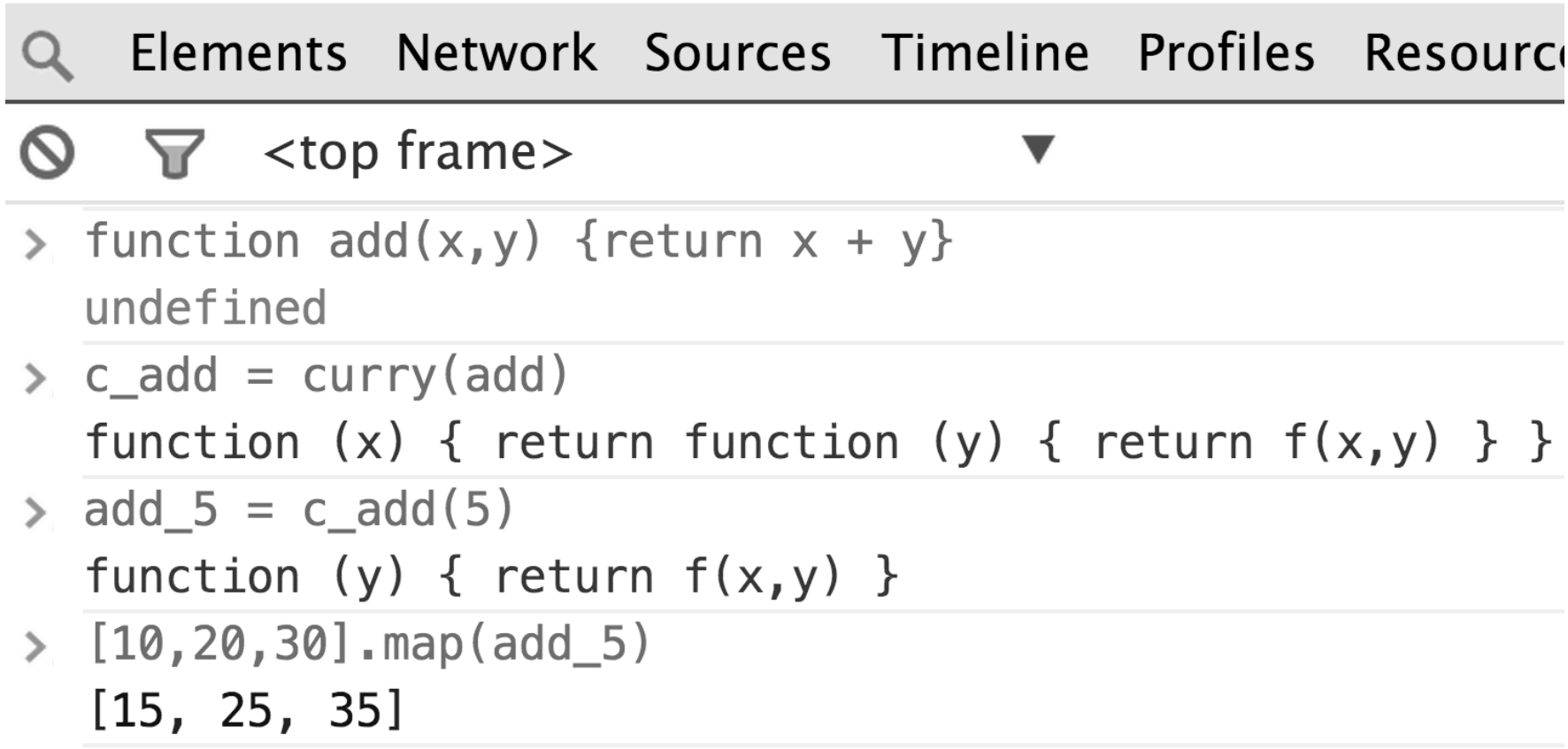
```
> map (flip (curry g) 4) [1..10]
[45,60,77,96,117,140,165,192,221,252]
```

The counterpart of **curry** is **uncurry**:

```
> uncurry (+) (3,4)
7
```

# A **curry** function for JavaScript

```
function curry(f) {
    return function(x) {
        return function (y) { return f(x,y) }
        }
    }
```



# Folding

# Reduction

We can *reduce* a list by a binary operator by inserting that operator between the elements in the list:

`[1,2,3,4]` reduced by `+` is `1 + 2 + 3 + 4`

`["a","bc", "def"]` reduced by `++` is `"a" ++ "bc" ++ "def"`

Imagine a function **`reduce`** that does reduction by an operator.

```
> reduce (+) [1,2,3,4]
10

> reduce (++) ["a","bc","def"]
"abcdef"

> reduce max [10,2,4]
10                                  -- think of 10 `max` 2 `max` 4
```

# Reduction, continued

At hand:

```
> reduce (+) [1,2,3,4]
10
```

An implementation of **reduce**:

```
reduce _ [] = undefined
reduce _ [x] = x
reduce op (x:xs) = x `op` reduce op xs
```

Does **reduce + [1,2,3,4]** do

```
((1 + 2) + 3) + 4
```

or

```
1 + (2 + (3 + 4))
```

?

In general, when would the grouping matter?

If the operation is non-associative, like division.

# foldl1 and foldr1

In the Prelude there's no reduce but there is foldl1 and foldr1.

```
> foldl1 (+) [1..4]
10

> foldl1 max "maximum"
'x'

> foldl1 (/) [1,2,3]
0.16666666666666666    -- left associative: (1 / 2) / 3

> foldr1 (/) [1,2,3]          -- right associative: 1 / (2 / 3)
1.5
```

The types of both foldl1 and foldr1 are (a -> a -> a) -> [a] -> a.

# foldl1 vs. foldl

Another folding function is foldl (no 1). Let's compare the types of the two:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
```

What's different between them?

First difference: foldl requires one more argument:

```
> foldl (+) 0 [1..10]
55

> foldl (+) 100 []
100

> foldl1 (+) []
*** Exception: Prelude.foldl1: empty list
```

# foldl1 vs. foldl, continued

Again, the types:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Second difference:

foldl can fold a list of values into a different type! (This is BIG!)

Examples:

```
> foldl f1 0 ["just","a","test"]
3                       -- folded strings into a number

> foldl f2 "stars: " [3,1,2]
"stars: ******"     -- folded numbers into a string

> foldl f3 0 [(1,1),(2,3),(5,10)]
57                      -- folded two-tuples into a sum of products
```

# foldl

For reference:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Here's another view of the type: (acm_t stands for accumulator type)

```
foldl :: (acm_t -> elem_t -> acm_t) -> acm_t -> [elem_t] -> acm_t
```

foldl takes three arguments:

1. A function that takes an accumulated value and an element value and produces a new accumulated value
2. An initial accumulated value
3. A list of elements

Recall:

```
> foldl f1 0 ["just","a","test"]
3

> foldl f2 "stars: " [3,1,2]
"stars: ******"
```

Recall:

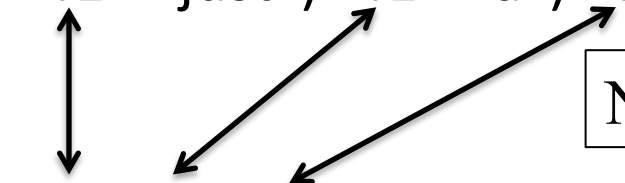
```
> foldl f1 0 ["just","a","test"]
3
```

Here are the computations that foldl did to produce that result

```
> f1 0 "just"
1
> f1 it "a"
2
> f1 it "test"
3
```

Let's do it in one shot, and use backquotes to infix f1:

```
> ((0  `f1`  "just")  `f1`  "a")  `f1`  "test"
3
```



Note the parallels between these two.

1 + 2 + 3 + 4 is the reduction we started this section with.

# foldl, continued

At hand:

```
> f1 0 "just"
1
> f1 it "a"
2
> f1 it "test"
3
```

For reference:

```
> foldl f1 0 ["just","a","test"]
3
```

Problem: Write a function f1 that behaves like above.

Starter:

```
f1 :: acm_t -> elem_t -> acm_t
f1 acm elem =  acm + 1
```

Congratulations!  You just wrote a *folding function*!

Recall:

```
> foldl f2 "stars: " [3,1,2]
"stars: ******"
```

Here's what foldl does with f2 and the initial value, "stars: ":

```
> f2 "stars: " 3
"stars: ***"
> f2 it 1
"stars: ****"
> f2 it 2
"stars: ******"
```

Write f2, with this starter:

```
f2 :: acm_t -> elem_t -> acm_t
f2 acm elem = acm ++ replicate elem '*'
```

Look! Another folding function!

Folding abstracts a common pattern of computation: a series of values contribute one-by-one to a result that accumulates.

The challenge of folding is to envision a function that takes nothing but an accumulated value (acm) and a single list element (elem) and produces a result that reflects the contribution of elem to acm.

```
f2 acm elem = acm ++ replicate elem '*'
```

It's important to recognize that the folding function never sees the full list!

We then call foldl with that folding function, an appropriate initial value and a list of values.

```
foldl f2 "stars: " [3,1,2]
```

foldl orchestrates the computation by making the appropriate series of calls to the folding function.

```
> (("stars: " `f2` 3) `f2` 1) `f2` 2
"stars: ******"
```

# foldl, continued

Recall:

```
> foldl f3 0 [(1,1),(2,3),(5,10)]
57
```

Here are the calls that foldl will make:

```
> f3 0 (1,1)
1
> f3 it (2,3)
7
> f3 it (5,10)
57
```

Problem: write f3!

```
f3 acm (a,b) = acm + a * b
```

# foldl, continued

Remember that

```
foldl f 0 [10,20,30]
```

is like

```
((0 `f` 10) `f` 20) `f` 30
```

Here's an implementation of foldl:

```
foldl f acm [] = acm
foldl f acm (elem:elems) =  foldl f (acm `f` elem) elems
```

We can implement foldl1 in terms of foldl:

```
foldl1 f (x1:xs) = foldl f x1 xs
foldl1 _ [] = error "empty list"
```

# A non-recursive countEO

Let's use folding to implement our even/odd counter non-recursively.

```
> countEO [3,4,7,9]
(1,3)
```

Often, a good place to start on a folding is to figure out what the initial accumulator value should be. What should it be for countEO?

```
(0,0)
```

What will be the calls to the folding function?

```
> f (0,0) 3
(0,1)
> f it 4
(1,1)
> f it 7
(1,2)
> f it 9
(1,3)
```

Now we're ready to write countEO:

```
countEO nums = foldl f (0,0) nums
    where
        f (evens, odds) elem
            | even elem = (evens + 1, odds)
            | otherwise = (evens, odds + 1)
```

# Folds with anonymous functions

If a folding function is simple, an anonymous function is typically used.

Let's redo our three earlier folds with anonymous functions:

```
> foldl (\acm _ -> acm + 1) 0 ["just","a","test"]
3

> foldl (\acm elem -> acm ++ replicate elem '*') "stars: " [3,1,2]
"stars: ******"

> foldl (\acm (a,b) -> acm + a * b) 0 [(1,1),(2,3),(5,10)]
57
```

# foldr

The counterpart of foldl is foldr. Compare their meanings:

```
foldl f zero [e1, e2, ..., eN] == (...((zero `f` e1) `f` e2) `f`...)`f` eN

foldr f zero [e1, e2, ..., eN] == e1 `f` (e2 `f` ... (eN `f` zero)...)
```

"zero" represents a computation-specific initial value. Note that with foldl, zero is leftmost; but with foldr, zero is rightmost.

Their types, with long type variables:

```
foldl :: (acm -> val -> acm) -> acm -> [val] -> acm
foldr :: (val -> acm -> acm) -> acm -> [val] -> acm
```

Mnemonic aid:

foldl's folding function has the accumulator on the left

foldr's folding function has the accumulator on the right

# foldr, continued

Because cons (:) is right-associative, folds that produce lists are often done with foldr.

Imagine a function that keeps the odd numbers in a list:

```
> keepOdds [5,4,2,3]
[5,3]
```

Implementation, with foldr:

```
keepOdds list = foldr f [] list
    where
        f elem acm
            | odd elem = elem : acm
            | otherwise = acm
```

Here are calls to the folding function:

```
> f 3 [] -- rightmost first!
[3]
> f 2 it
[3]
> f 4 it
[3]
> f 5 it
[5,3]
```

# filter and map with folds?

keepOdds could have been written using filter:

```
keepOdds = filter odd
```

Can we implement filter as a fold?

```
filter predicate list = foldr f [] list
    where
        f elem acm
            | predicate elem = elem : acm
            | otherwise = acm
```

How about implmenting map as a fold?

```
map f = foldr (\elem acm -> f elem : acm) []
```

Is folding One Operation to Rule Them All?

# **paired** with a fold

Recall **paired** from assignment 3:

```
> paired "((())())"
True
```

Can we implement **paired** with a fold?

```
counter (-1) _ = -1
counter total '(' = total + 1
counter total ')' = total - 1
counter total _ = total

paired s = foldl counter 0 s == 0
```

**paired** is a fold with a simple wrapper, to test the result of the fold.

Point-free:

```
paired = (0==) . foldl counter 0
```

# Folding, continued

**Data.List.partition** partitions a list based on a predicate:

```
> partition isLetter "Thu Feb 13 16:59:03 MST 2014"
("ThuFebMST","  13 16:59:03  2014")

> partition odd [1..10]
([1,3,5,7,9],[2,4,6,8,10])
```

Problem: Write **partition** using a fold.

```
sorter f val (pass, fail)
    | f val = (val:pass, fail)
    | otherwise = (pass, val:fail)

partition f = foldr (sorter f) ([],[])
```

# A progression of folds

Let's do a progression of folds related to finding vowels in a string.

First, let's write a fold that counts vowels in a string:

```
> foldr (\val acm ->
        acm + if  val `elem` "aeiou" then 1 else 0)  0  "ate"
2
```

Now let's produce both a count and the vowels themselves:

```
> foldr (\val acm@(n, vows) ->
          if  val `elem` "aeiou" then (n+1, val:vows)
                                 else acm) (0,[]) "ate"
(2,"ae")
```

# A progression of folds, continued

Finally, let's write a function that produces a list of vowels and their positions:

```
> vowelPositions "Down to Rubyville!"
[('o',1),('o',6),('u',9),('i',13),('e',16)]
```

Solution:

```
vowelPositions s = reverse result
  where (_, result, _) =
    foldl (\acm@(n, vows,pos) val ->
      if  val `elem` "aeiou" then (n, (val,pos):vows,pos+1)
                             else (n,vows,pos+1)) (0,[],0) s
```

Note that **vowelPositions** uses **foldl** to produce a 3-tuple whose middle element is the result, in reverse order. (This is another function that's a fold with a wrapper, like **paired** on 348).

# **map** vs. **filter** vs. folding

**map**:

transforms a list of values

length *input* == length *output*

**filter**:

selects values from a list

0 <= length *output* <= length *input*

folding

Input: A list of values and an initial value for accumulator

Output: A value of any type and complexity

True or false?

Any operation that processes a list can be expressed in a terms of a fold, perhaps with a simple wrapper.

# We can fold a list of anythings into anything!

Far-fetched folding:

Refrigerators in Gould-Simpson to
((grams fat, grams protein, grams carbs), calories)

Keyboards in Gould-Simpson to
[("a", #), ("b", #), ..., ("@2", #), ("CMD", #)]

[Backpack] to
(# pens, pounds of paper,
[(title, author, [page #s with the word "computer")])

[Furniture]
to a structure of 3D vertices representing a *convex hull* that could hold any single piece of furniture.

# User-defined types

# A **Shape** type

A new type can be created with a **data** declaration.

Here's a simple **Shape** type whose instances represent circles or rectangles:

```
data Shape =
   Circle Double |          -- just a radius
   Rect Double Double       -- width and height
        deriving Show
```

The shapes have dimensions but no position.

**Circle** and **Rect** are *data constructors*.

"**deriving Show**" declares **Shape** to be an instance of the **Show** type class, so that values can be shown using some simple, default rules.

**Shape** is called an *algebraic type* because instances of **Shape** are built using other types.

# Shape, continued

Instances of Shape are created by calling the data constructors:

```
> let r1 = Rect 3 4
> r1
Rect 3.0 4.0

> let r2 = Rect 5 3

> let c1 = Circle 2

> let shapes = [r1, r2, c1]

> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]
```

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

Lists must be homogeneous—why are both Rects and Circles allowed in the same list?

# Shape, continued

The data constructors are just functions—we can use all our function-fu with them!

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

```
> :t Circle
Circle :: Double -> Shape

> :t Rect
Rect :: Double -> Double -> Shape

> map Circle [2,3] ++ map (Rect 3) [10,20]
[Circle 2.0,Circle 3.0,Rect 3.0 10.0,Rect 3.0 20.0]
```

# Shape, continued

Functions that operate on algebraic types use patterns based on the type's data constructors.

```
area (Circle r) = r ** 2 * pi
area (Rect w h) = w * h
```

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

Usage:

```
> r1
Rect 3.0 4.0

> area r1
12.0

> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]

> map area shapes
[12.0,15.0,12.56637061435917]

> sum $ map area shapes
39.56637061435917
```

# Shape, continued

Let's make the **Shape** type an instance of the **Eq** type class.

What does **Eq** require?

```
> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Default definitions from **Eq**:

```
(==) a b = not $ a /= b
(/=)  a b = not $ a == b
```

We'll say that two shapes are equal if their areas are equal.

```
instance Eq Shape where
   (==) r1 r2 = area r1 == area r2
```

Usage:

```
> Rect 3 4 == Rect 6 2
True

> Rect 3 4 == Circle 2
False
```

# **Shape**, continued

Let's see if we can find the biggest shape:

```
> maximum shapes
 No instance for (Ord Shape) arising from a use of
`maximum'
   Possible fix: add an instance declaration for (Ord
Shape)
```

What's in **Ord**?

```
> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

**Eq a => Ord a** requires would-be **Ord** classes to be instances of **Eq**. (Done!)

Like == and /= with **Eq**, the operators are implemented in terms of each other.

# Shape, continued

Let's make Shape an instance of the Ord type class:

```
instance Ord Shape where
    (<) r1 r2 = area r1 < area r2          -- < and <= are sufficient
    (<=) r1 r2 = area r1 <= area r2
```

Usage:

```
> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]

> map area shapes
[12.0,15.0,12.566370614359172]

> maximum shapes
Rect 5.0 3.0

> Data.List.sort shapes
[Rect 3.0 4.0,Circle 2.0,Rect 5.0 3.0]
```

Note that we didn't need to write functions like **sumOfAreas** or **largestShape**—we can express those in terms of existing operations

# Shape all in one place

Here's all the **Shape** code: (in **shape.hs**)

```
data Shape =
   Circle Double |
   Rect Double Double deriving Show

area (Circle r) = r ** 2 * pi
area (Rect w h) = w * h

instance Eq Shape where
   (==) r1 r2 = area r1 == area r2

instance Ord Shape where
   (<) r1 r2 = area r1 < area r2
   (<=) r1 r2 = area r1 <= area r2
```

What would be needed to add a **Figure8** shape and a **perimeter** function?

How does this compare to a **Shape/Circle/Rect** hierarchy in Java?

# The type Ordering

Let's look at the **compare** function:

```
> :t compare
compare :: Ord a => a -> a -> Ordering
```

**Ordering** is a simple algebraic type, with only three values:

```
> :info Ordering
data Ordering = LT | EQ | GT
```

```
> [r1,r2]
[Rect 3.0 4.0,Rect 5.0 3.0]
```

```
> compare r1 r2
LT
```

```
> compare r2 r1
GT
```

# What is **Bool**?

What do you suppose **Bool** really is?

**Bool** is just an algebraic type with two values:

```
> :info Bool
data Bool = False | True
```

**Bool** is an example of Haskell's extensibility. Instead of being a primitive type, like **boolean** in Java, it's defined in terms of something more basic.

# A binary tree

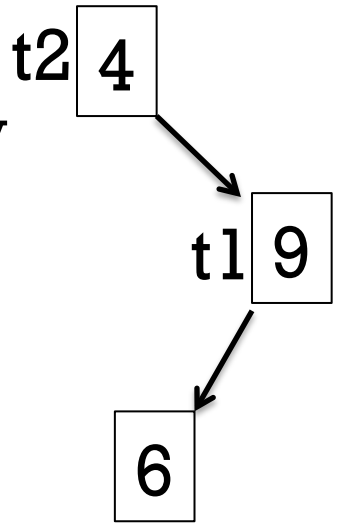
Here's an algebraic type for a binary tree: (in **tree.hs**)

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
              deriving Show
```

The **a** is a type variable. Our **Shape** type used **Double** values but **Tree** can hold values of any type!

```
> let t1 = Node 9 (Node 6 Empty Empty) Empty
> t1
Node 9 (Node 6 Empty Empty) Empty

> let t2 = Node 4 Empty t1
> t2
Node 4 Empty (Node 9 (Node 6 Empty Empty) Empty)
```



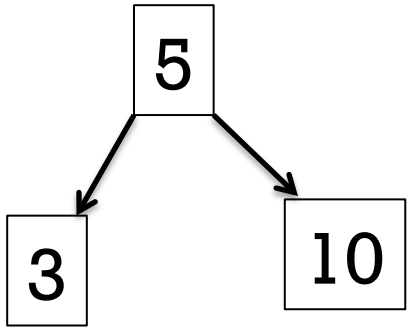
# Tree, continued

Here's a function that inserts values, maintaining an ordered tree:

```
insert Empty v = Node v Empty Empty
insert (Node x left right) value
    | value <= x = (Node x (insert left value) right)
    | otherwise = (Node x left (insert right value))
```

Let's insert some values...

```
> let t = Empty
> insert t 5
Node 5 Empty Empty
```




```
> insert it 10
Node 5 Empty (Node 10 Empty Empty)

> insert it 3
Node 5 (Node 3 Empty Empty) (Node 10 Empty Empty)
```

Note that each insertion rebuilds some portion of the tree!

# Tree, continued

Here's an in-order traversal that produces a list of values:

```
inOrder Empty = []
inOrder (Node val left right) =
    inOrder left ++ [val] ++ inOrder right
```

What's an easy way to insert a bunch of values?

```
> let t = foldl insert Empty [3,1,9,5,20,17,4,12]
> inOrder t
[1,3,4,5,9,12,17,20]

> inOrder $ foldl insert Empty "tim korb"
" bikmort"

> inOrder $ foldl insert Empty [Rect 3 4, Circle 1, Rect 1 2]
[Rect 1.0 2.0,Circle 1.0,Rect 3.0 4.0]
```

# Maybe

Here's an interesting type:

```
> :info Maybe
data Maybe a = Nothing | Just a
```

Speculate: What's the point of it?

Here's a function that uses it:

```
> :t Data.List.find
Data.List.find :: (a -> Bool) -> [a] -> Maybe a
```

How could we use it?

```
> find even [3,5,6,8,9]
Just 6

> find even [3,5,9]
Nothing

> case (find even [3,5,9]) of { Just _ -> "got one"; _ -> "oops!"}
"oops!"
```

# A little I/O

# Sequencing

Consider this function declaration

```
f2 x = a + b + c
   where
      a = f x
      b = g x
      c = h x
```

```
a = f x
c = h x
b = g x
```

```
c = h x
b = g x
a = f x
```

Haskell guarantees that the order of the **where** clause bindings is inconsequential—those three lines can be in any order.

What enables that guarantee?

> (Pure) Haskell functions depend only on the argument value. For a given value of **x**, **f x** always produces the same result.

You can shuffle the bindings of any function's **where** clause without changing the function's behavior! (Try it with **longest**, slide 291.)

# I/O and sequencing

Imagine a **getInt** function, which reads an integer from standard input (e.g., the keyboard).

Can the **where** clause bindings in the following function be done in any order?

```
f x = r
   where
      a = getInt
      b = getInt
      r = a * 2 + b + x
```

The following is not valid syntax but ignoring that, is it reorderable?

```
greet name = ""
   where
      putStr "Hello, "
      putStr name
      putStr "!\n"
```

# I/O and sequencing, continued

One way we can specify that operations are to be performed in a specific sequence is to use a **do**:

```
% cat io2.hs
main = do
   putStrLn "Who goes there?"
   name <- getLine
   let greeting = "Hello, " ++ name ++ "!"
   putStrLn greeting
```

Interaction:

```
% runghc io2.hs
Who goes there?
whm (typed)
Hello, whm!
```

# Actions

Here's the type of **putStrLn**:

```
putStrLn :: String -> IO ()
```
*("unit", (), is the no-value value)*

The type **IO x** represents an interaction with the outside world that produces a value of type **x**. Instances of **IO x** are called *actions*.

When an action is evaluated the corresponding outside-world activity is performed.

```
> let hello = putStrLn "hello!"
hello :: IO ()

> hello
hello!
it :: ()
```
*(Note: no output here!)*

*(Type of **hello** is an action.)*

*(Evaluating **hello**, an action, caused output.)*

# Actions, continued

The value of **getLine** is an action that reads a line:

```
getLine :: IO String
```

We can evaluate the action, causing the line to be read, and bind a name to the string produced:

```
> s <- getLine
testing

> s
"testing"
```

Note that **getLine** is not a function!

# Actions, continued

Recall **io2.hs**:

```
main = do
   putStrLn "Who goes there?"
   name <- getLine
   let greeting = "Hello, " ++ name ++ "!"
   putStrLn greeting
```

Note the type: **main :: IO ()**. We can say that **main** is an action. Evaluating **main** causes interaction with the outside world.

```
> main
Who goes there?
hello? (I typed)
Hello, hello?!
```

# Is it pure?

A pure function (1) always produces the same result for a given argument value, and (2) has no side effects.

Is this a pure function?

```
twice :: String -> IO ()
twice s = do
   putStr s
   putStr s
```

`twice "abc"` will always produce the same value, an action that if evaluated will cause "`abcabc`" to be output.

# The Haskell solution for I/O

We want to use pure functions whenever possible but we want to be able to do I/O, too.

In general, evaluating an action produces side effects.

Here's the Haskell solution for I/O in a nutshell:

> Actions can evaluate other actions and pure functions but pure functions don't evaluate actions.

Recall **longest.hs**:

```
longest bytes = result where ...lots...
main = do
    args <- getArgs -- gets command line arguments
    bytes <- readFile (head args)
    putStr (longest bytes)
```

# In conclusion...

# If we had a whole semester...

If we had a whole semester to study functional programming, here's what might be next:

- More with infinite data structures (like `x = 1:x`)
- How lazy/non-strict evaluation works
- Implications and benefits of referential transparency (which means that the value of a given expression is always the same).
- Functors (structures that can be mapped over)
- Monoids (a set of things with a binary operation over them)
- Monads (for representing sequential computations)
- Zippers (a structure for traversing and updating another structure)
- And more!

Jeremiah Nelson and Jordan Danford are great local resources for Haskell!

# Even if you never use Haskell again...

Recursion and techniques with higher-order functions can be used in most languages. Some examples:

JavaScript, Python, PHP, all flavors of Lisp, and lots of others:
Functions are "first-class" values; anonymous functions are supported.

C
Pass a function pointer to a recursive function that traverses a data structure.

C#
Excellent support for functional programming with the language itself, and LINQ, too.

Java 8
Lambda expressions are in!

OCaml
"an industrial strength programming language supporting functional, imperative and object-oriented styles" – **OCaml.org**
**http://www.ffconsultancy.com/languages/ray_tracer/comparison.html**

# Killer Quiz!