Functional Programming with Haskell

CSC 372, Spring 2023
The University of Arizona
William H. Mitchell
whm@cs

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."

Examples of books that documented paradigms:
- Newton's *Principia*
- Lavoisier's *Elementary Treatise on Chemistry*
- Lyell's *Principles of Geology*

Paradigms, continued

Kuhn says a paradigm has:

- A world view
- A vocabulary
- A set of techniques for solving problems

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

Kuhn equates a paradigm shift with a scientific revolution.

The imperative programming paradigm

Imperative programming is a very early paradigm that's still used.

Originated with machine-level programming:

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

Examples of areas of study for those interested in the paradigm:

- Data types
- Operators
- Branching mechanisms and (later) control structures

Imperative programming fits well with the human mind's ability to describe and understand processes as a series of steps.

The imperative paradigm, continued

Language-wise, imperative programming requires:

- "Variables"—data objects whose values can be changed
- Expressions to compute values
- Support for iteration—a “while” control structure, for example.
- Statements are sequentially executed

Support for imperative programming is very common.

- Java
- C
- Python
- and hundreds more
- but not Haskell

Typically, code in a Java method or Python function is imperative.

The procedural programming paradigm

An outgrowth of imperative programming was *procedural programming*:

- Programs are composed of bodies of code (procedures) that manipulate individual data elements or structures.
- Procedures encapsulate complexity.

Examples of areas of study:

- How to decompose a computation into procedures and calls
- Parameter-passing mechanisms in languages
- Scoping of variables and nesting of procedures
- Visualization of procedural structure

The procedural paradigm, continued

Support for procedural programming is very common.

- Java
- Python
- C
- and lots more

The procedural and imperative paradigms can be combined:

- Procedural programming: the set of procedures
- Imperative programming: the contents of procedures

Devising the set of functions for a C program is an example of procedural programming.

Procedural programming is possible in Java but classes devolve into collections of static methods and data

The object-oriented paradigm

Object-oriented programming fits Kuhn's paradigm criteria well:

World view:
Systems are interacting objects. Pillars of OOP are abstraction, encapsulation, inheritance, polymorphism.

Vocabulary:
Methods, instances, constructors, super/subclasses, and more

Techniques:
Model with classes, work out responsibilities and collaborators, don't have public data, etc.

The object-oriented paradigm, continued

Many languages support OO programming but don't force it.

- C++
- Python
- Ruby

Java forces at least a veneer of OO programming.

The OO and imperative paradigms can be combined:

- OO: the set of classes and their methods
- Imperative: the code inside methods

Multiple paradigms(?)

Paradigms in a field of science are often incompatible.

Example: geocentric vs. heliocentric model of the universe

Imperative programming is used both with procedural and object-oriented programming.

Wikipedia's **Programming_paradigm** has this:

> *Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.*

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

The level of a paradigm

Programming paradigms can apply at different levels:

- Making a choice between procedural and object-oriented programming fundamentally determines the nature of 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.

Do co-existing paradigms imply they're solving fundamentally different types of problems?

The influence of paradigms

The programming paradigms 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.

Imperative programming revisited

Recall these language requirements for imperative programming:

- "Variables"—data objects whose values can be changed
- Expressions to compute values
- Support for iteration—a “while” control structure, for example.
- Statements are sequentially executed

Imperative summation

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;
}
```

How does it exemplify imperative programming?

- The values of **sum** and **i** change over time.
- An iterative control structure is at the heart of the computation.
- Statements are executed in sequence

Imperative summation, continued

With Java's "enhanced **for**" 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?

Non-imperative summation

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);
}
```

Which of the three versions is the easiest to believe as correct?
(simple for-loop, enhanced for-loop, or icky recursion)

Non-imperative summation, cont.

A recursive solution is far simpler in Python:

```
def sumnums(nums):
    if len(nums) == 0:
        return 0
    else:
        return nums[0] + sumnums(nums[1:])
```

Any loops or assignments?

What feature of Python enables this cleaner solution?

Could we do better with recursion in Java by using `java.util.List`?

Challenge: If you know C, write a non-imperative version of `strlen`.

Expressions:
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
```

Here are three questions we can ask about an expression:

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

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

Value

What is the value of the following expressions?

```
3 + 4                   # Java
    7

[1][-1]                 # Python
    1

s = 3 + 4 + "5"         # Java
    "75"

"a,bb,c3".split(",")    # Java
```

A **String** array with three elements: "a", "bb" and "c3"

```
list({1:2,3:4,5:6})     # Python
    [1, 3, 5]
```

Type

What is the type of each of the following expressions?

3 + 4 # Java
int

`[(1,2)][-1]` # Python
tuple

s = 3 + 4 + "5" # Java
String

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

['x'].append(3) # Python
None

When we ask,
"What's the type of this expression?"

we're actually asking,
"What's the type of the value produced by this expression?"

How can we determine the type of an expression in Python? In Java?

Sidebar: A litmus test for expressions?

Q: What's an experiment to test if something is an expression?
A: See if we can pass it as an argument!

```
System.out.println(i = 7)     // works!

>>> print(i = 7)
TypeError: 'i' is an invalid keyword argument for
print()

>>> print(3 in range(5))
True

>>> print(1 if 2 < 3 else 4)
1
```

Side effects

Evaluating some expressions causes other things to happen in addition to computing a value.

```
jshell> ArrayList<Integer> L = new ArrayList<>();

jshell> L
L ==> []

jshell> L.add(7)
$2 ==> true

jshell> L
L ==> [7]    // L.add(7) has the side effect of adding 7 to L
```

Side effects, continued

A side effect is something that happens in addition to the computation of an expression's value. It must be "observable".

"A side effect is a change you can witness."
—Jasmine Ying, Fall '22 Original Thought

Examples of side effects of expressions:

- The value of a variable was 5 but now is 6
- A line of output appeared
- A pixel changed color
- A list has one more element
- A file is gone
- A table in a database has one more row

Side effects, continued

What is the value, type, and side effect of these expressions?

n = 5 # Java, given **int n = 3;**

Value: 5

Type: **int**

Side effect: **n** changed to **5**

```
>>> print(1,2,3)        # Python
1 2 3
>>>
```

```
>>> print(print(1), print(2))
1
2
None None
```

Side effects, continued

What is the value of the following Java expression?

`i++`

The value of `i++` is `i`, whatever `i` is.

Does `i++` have a side effect?

Evaluating `i++` has a side effect of incrementing `i`.

Let's experiment with JShell!

Side effects, continued

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.

Side effects, continued

Some Python to ponder wrt. side effects:

`"testing".upper()`

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

`print("ok!")`

Output is surely a side effect!

`sys.stdin.readline()`

Nothing is done with the resulting string, but a line was consumed and that is a side effect.

`base[n].launch_missles()`

The method name implies a significant side effect, but...

The hallmark of imperative programming

Side effects are the hallmark of imperative programing.

Code written in an imperative style is 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

What is Functional Programming?

"Functional programming is so called because its fundamental operation is the application of functions to arguments."
—John Hughes, *Why Functional Programming Matters*

"Generally speaking, however, functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments."
—Graham Hutton, *Programming in Haskell*

It seems that a competing name years ago was "applicative programming".

The term "function-oriented programming" crosses my mind.

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
- Are often specified with 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.

Haskell basics

What is Haskell?

Haskell is a lazy and pure functional programming language.
Lazy: Only evaluates expressions when needed
Pure: Expressions never have any side effects
But, I/O is performed with *monadic effects*

Haskell is statically typed, with a very elaborate type system.

Haskell is not object-oriented in any way.

Designed by a committee, formed in 1987, 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. Here is the Haskell 2010 Report, which I'll call "H10".

http://haskell.org/definition/haskell2010.pdf

Haskell resources

Website: haskell.org

Here are three books I can recommend:

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

Haskell: The Craft of Functional Programming, 3rd edition, by Simon Thompson. (2nd edition is pretty good, too.)

Programming in Haskell, 2e by Graham Hutton. (First edition here.)

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

There's a pile of stuff at haskell.org/documentation, but it's a big pile!

For the curious: A History of Haskell: Being Lazy With Class

Getting Haskell

Haskell 8.6.5 is installed on lectura and if you wish, you can simply work with **ghci** there.

To get Haskell for your machine, start at **haskell.org/ghcup**

The GHCup page shows copy-and-paste command lines for UNIX-like platforms (**curl ... | sh**), and for Windows PowerShell that start an installer.

The installer will offer options to install **HLS** and **stack**. HLS can be handy if you're using VSCode, **vim**, or Emacs. You won't need **stack** for what we're doing. On Windows, you'll also be asked about **MSys2**, and it appears the installation won't proceed without it.

The latest version of Haskell appears to be 9.4.1 but 8.10.7 gets installed by default and that seems to be what's recommended by the Haskell folks.

Interacting with Haskell

We'll usually interact with Haskell by running **ghci** in a terminal window on UNIX-like machines, or in a PowerShell or **cmd.exe** window on Windows.

```
% ghci
GHCi, version 8.6.5...
Prelude> 3 + 4
7

Prelude> ^D (control-D to quit)
%
```

With no arguments, **ghci** starts a read-eval-print loop (REPL):

Expressions typed at the prompt (**Prelude>**) are evaluated and the result is printed.

The **~/.ghci** file

When **ghci** starts up on UNIX-like systems 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 "> "
import Text.Show.Functions
```

The first line simply sets the prompt to "**>** " and that's just my preference.

The second line is very important:

- It loads a module that lets function values be shown as **<function>**, instead of producing an error.
- Without it, lots of examples in these slides won't work!

~/.ghci, continued

Fact: ~/.ghci must not be group- or world-writable!

If you see something like this,

```
*** WARNING: /home/whm/.ghci is writable by
someone else, IGNORING!
Suggested fix: execute
    'chmod go-w /home/whm/.ghci'
```

the suggested fix should work.

Details on .ghci can be found by Googling for "the .ghci file" but much of what turns up is quite old.

~/.ghci, continued

On Windows, instead of looking for a **~/.ghci** file, **ghci** looks for **ghc\ghci.conf** in your "app data" directory.

If you're using **cmd.exe**, do this to see where your app data is:

```
C:\>set appdata
APPDATA=C:\Users\whm\AppData\Roaming
```

If you're using PowerShell, do this:

```
% $env:APPDATA
C:\Users\whm\AppData\Roaming
```

Combing the two paths, the full path to the file <u>for me</u> is

```
C:\Users\whm\AppData\Roaming\ghc\ghci.conf
```

Extra Credit Assignment 1

For two assignment points of extra credit:

1. Run **ghci** somewhere and try ten Haskell expressions with some degree of variety and not simply the ones on the following slide.

2. Demonstrate that you've got **import Text.Show.Functions** in your **~/.ghci** or **ghc.conf** file, as described on slide 40, by showing that typing **negate** produces **<function>,** like this:

```
Prelude> negate
<function>
```

3. Capture the interaction (both expressions and results) 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.

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

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

Due: At the start of the next lecture after the lecture in which I present this slide.

Haskell by Observation

Let's see what we can learn about Haskell by trying some expressions:

```
3 + 4
3 * 4.5
it + it
it /= 3
3 > 4 || 5 < 7
not 3 < 4
2^200
2**0.5
"abc" + 3
```

```
"ab" ++ "xy"
it!!3
replicate 5 '.'
words "U of A"
map length it
[1..10]
map (*10) [1,3..10]
(+) 3 4
:help
```

Functions and function types

Calling functions

In Haskell, *juxtaposition* indicates a function call:

```
> negate 3
-3

> even 5
False

> pred 'C'
'B'

> signum 2
1
```

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>:11:1: error:
    • Non type-variable argument ...
    ...
```

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

```
> signum (negate 2)
-1
```

Calling functions, continued

Function call has higher precedence than any operator.

```
> negate 3+4
1
```

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

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

> signum (negate (3 + 4))
-1
```

The **Data.Char** module

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

```
> import Data.Char      (import the Data.Char module)

> isLower 'b'
True

> toUpper 'a'
'A'

> ord 'A'
65

> chr 66
'B'

> Data.Char.ord 'G'     (uses a qualified name)
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
```

The type **Char -> Bool** says that **isLower** is a function that

1. Takes an argument of type **Char**
2. Produces a result of type **Bool**

The text

```
isLower :: Char -> Bool
```

is read as "**isLower** <u>**has type**</u> **Char** <u>**to**</u> **Bool**"

LHtLaL discernment: **:type** is part of **ghci**, not Haskell!

Function types, continued

Recall:

```
> toUpper 'a'
'A'
> ord 'A'
65
> chr 66
'B'
```

What are the types of those three functions?

```
> :t toUpper
toUpper :: Char -> Char

> :t ord
ord :: Char -> Int

> :t chr
chr :: Int -> Char
```

Sidebar: Contrast with Java

What are the types of the following Java methods?

```
jshell> Character.isLetter('4')
$1==> false

jshell> Character.toUpperCase('a')
$2 ==> 'A'
```

```
% javap java.lang.Character | grep "isLetter(\|toUpperCase("
  public static boolean isLetter(char);
  public static boolean isLetter(int);
  public static char toUpperCase(char);
  public static int toUpperCase(int);
```

Important:

- Java: common to think of a method's return type as the method's type
- Haskell: a function's type includes both the type of argument(s) and the return type

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>:1:5: error:
   • Couldn't match expected type ‘Int’ with actual type ‘Char’
   • In the first argument of ‘chr’, namely ‘'x'’
     ...

> :t chr
chr :: Int -> Char

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

For reference:

```
'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
`^L`	Clear the screen
`^R`	Incremental search through previously typed lines

Windows: Use **Home** and **End** for start- and end-of-line

More:

https://github.com/judah/haskeline/wiki/KeyBindings

Sidebar: Using a REPL to help learn a language

`ghci` provides a REPL (read-eval-print loop) for Haskell.

How does a REPL help us learn a language?

What are some other languages that have a REPL available?

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

> negate 5.0
-5.0
```

Speculate: What's the type of `negate`?

Type classes

"A type is a collection of related values." —Hutton

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

Haskell also has *type classes*.

Type class:

A collection of types that support a specified set of operations.

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

Important:

The names of types and type classes are always capitalized.

Haskell's type classes are unrelated to classes in the OO sense.

The **Num** type class

```
> :info Num
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

A type must support all of these operations to be an instance of **Num**

```
instance Num Word
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

These five types are all instances of **Num**: If we need a **Num**, we can use a value of type **Word**, or of type **Integer**, or of type **Int**, or **Float** or **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 **X**, I'll give you back an **X**."*

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's the type of a decimal fraction?

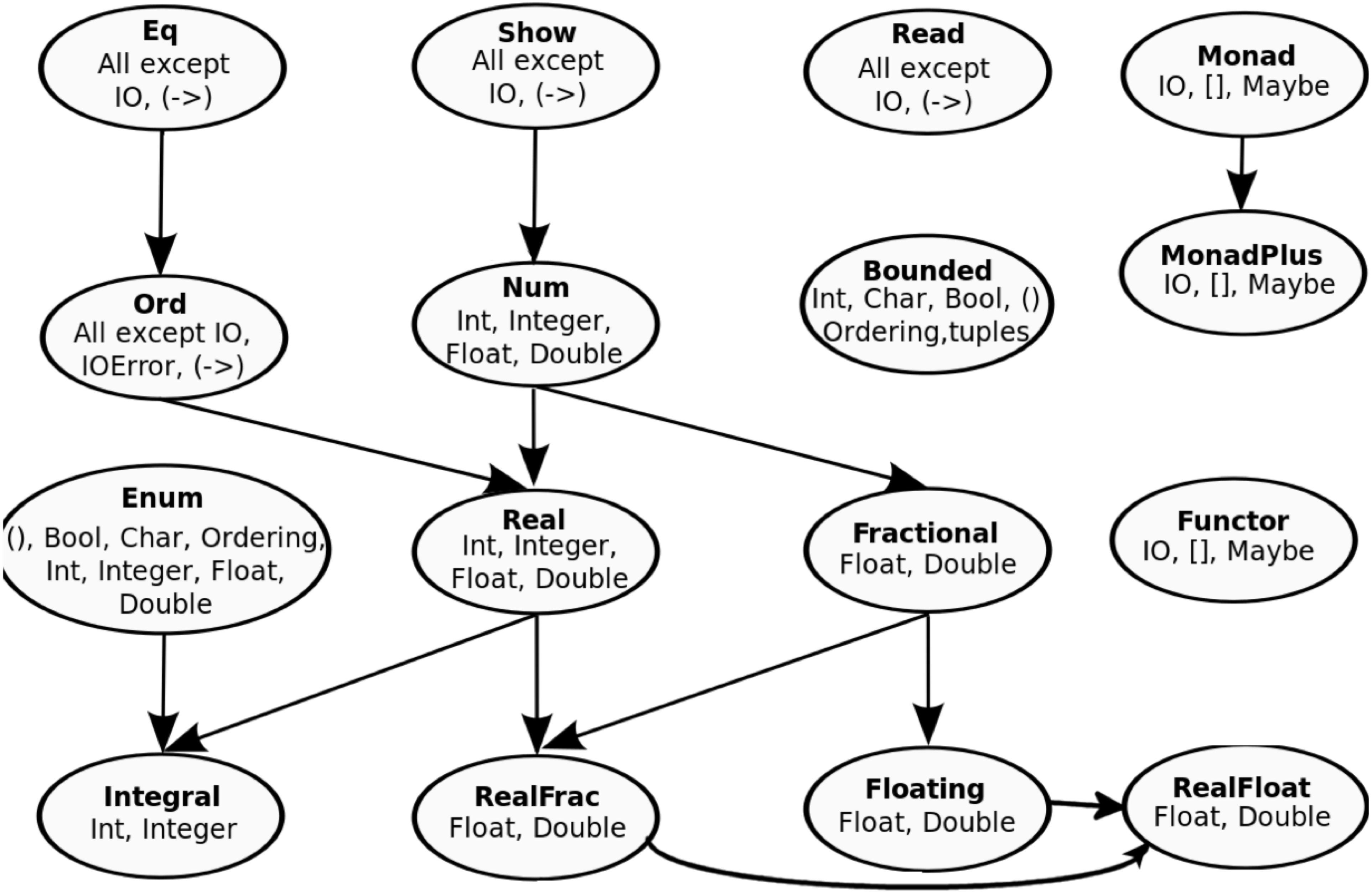
```
> :type 3.4
3.4 :: Fractional a => a
```

negate works with decimal fractions.

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

> negate 3.4
-3.4
```

Speculate: Why does it work?

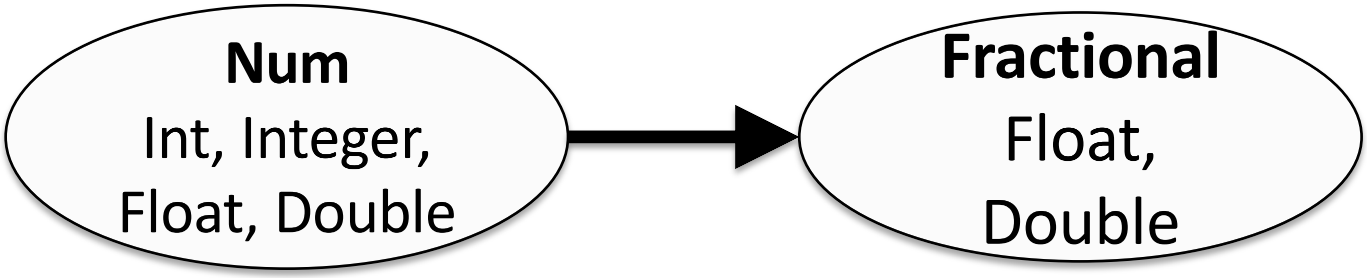
There is a hierarchy of type classes



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

Type classes, continued

Excerpt:


Num
Int, Integer,
Float, Double

Fractional
Float,
Double

The arrow indicates that types that are instances of `Fractional` can be used where types that are instances of `Num` are required.

Given

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

and

`3.4 :: Fractional a => a`

then

`negate 3.4` is valid.

How does the direction of the arrow relate to inheritance in UML?

Type classes, continued

:info *Type* shows the classes that *Type* is an instance of.

```
> :info Int
type Int :: *
data Int = GHC.Types.I# GHC.Prim.Int#
instance Eq Int
instance Ord Int
instance Show Int
instance Read Int
instance Enum Int
instance Num Int
instance Real Int
instance Bounded Int
instance Integral Int
```

Contrast **Int** with **Num**:

```
> :info Num
...
instance Num Word
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

Try **:info** for each of the classes (**Eq**, **Ord**, etc.)

Type classes, continued

The Prelude has a truncate to integer function:

```
> truncate 7.999
7
```

What does the type of `truncate` tell us?

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

`truncate` accepts any type that is an instance of `RealFrac`
`truncate` returns a type that is an instance of `Integral`

Explore the `Integral` and `RealFrac` type classes with `:info`.

Type classes, continued

Note that the following expressions are described by type classes, not types:

```
> :type 3
3 :: Num p => p

> :t 3.4 + 75
3.4 + 75 :: Fractional a => a

> :t 2^100000
2^100000 :: Num a => a

> :t 2**100000.1
2**100000.1 :: Floating a => a
```

Type classes, continued

In LYAH, Type Classes 101 has a good description of the Prelude's type classes.

Note:

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

Remember:

Haskell's type classes are unrelated to classes in the OO sense.

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 is a polymorphic function.

Does Java have polymorphic methods? Does Python? C?

A mystery

Consider this excerpt from **Bounded**:

```
> :info Bounded
class Bounded a where
  minBound :: a
  maxBound :: a
  ...
```

What sort of things are **minBound** and **maxBound**?

Polymorphic values!

How can we use them?

Polymorphic values

The construct *::type* is an *expression type signature*.

A usage of it:

```
> minBound::Char
'\NUL'

> maxBound::Int
9223372036854775807

> maxBound::Bool
True

> maxBound::Integer
<interactive>:9:1: error:
    • No instance for (Bounded Integer)
```

:set +t

We can use **:set +t** to direct **ghci** to automatically show types:

```
> :set +t

> 3
3
it :: Num p => p

> 3 + 4.5
7.5
it :: Fractional a => a

> abs
<function>
it :: Num a => a -> a
```

Use **:unset +t** to turn off display of types.

Sidebar: LHtLaL—introspective tools

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

<u>When learning a language, look for such tools early on.</u>

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()**; **/var** in **jshell**.

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[]) {
      showtype(3 + 'a');
      showtype(3 + 4.0);
      showtype("(2<F".toCharArray());
      showtype("a,b,c".split(","));
      showtype(new HashMap<String,Integer>());
   }
   private static void showtype(Object o) {
      System.out.println(o.getClass());
   }}
```

Output:

```
class java.lang.Integer
class java.lang.Double
class [C
class [Ljava.lang.String;
class java.util.HashMap
```

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

Sidebar: `~/notes/haskell.txt`

LHtLAL: Start accumulating a file of brief notes on Haskell.

```
$ cat ~/notes/haskell.txt
#faq
import Data.Char

#ghci
:type EXPR
:set +t -- shows types of all expressions
:info TYPE or TYPECLASS
   "instance Ord Int" means "[an] instance [of] Ord [is] Int"
Use -ignore-dot-ghci ... to suppress loading of .ghci

#misc
REPL is read-eval-print loop
function call has higher precedence than any operator
isLower :: Char -> Bool is read as "isLower has type Char to Bool"
type class hierarchy:
  https://www2.cs.arizona.edu/classes/cs372/spring23/haskell.pdf#page=63
```

More on functions

Writing simple functions

A function can be defined at the REPL prompt. Example:

```
> double x = x * 2
double :: Num a => a -> a        (:set +t is in effect)

> double 5
10
it :: Num a => a

> double 2.7
5.4
it :: Fractional a => a
```

General form (i.e. syntax) of a function definition for the moment:

function-name parameter = expression

Function and parameter names must begin with a lowercase letter or an underscore.

Simple functions, continued

Two more functions:

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

(:set +t is in effect)

```
> 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!)

Problem: Write **isPositive x** which returns **True** iff **x** is positive. Predict the function's type, too.

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

Simple functions, continued

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

```
> neg x = -x :: Int
neg :: Int -> Int

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

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

```
> isPositive x = x > (0::Int)
isPositive :: Int -> Bool
```

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

We'll use :: *type* to simplify the types of some functions that follow.

Sidebar: loading functions from a file

We can put function definitions in a file.

The file **simple.hs** has four function definitions:

```
% cat simple.hs
double x = x * 2 :: Int
neg x = -x :: Int
isPositive x = x > (0::Int)
toCelsius temp = (temp - 32) * 5/9 :: Double
```

We'll use the extension **.hs** for Haskell source files.

Generally, code from the slides will be (poorly organized) here:

https://www2.cs.arizona.edu/classes/cs372/spring23/haskell

/cs/www/classes/cs372/spring23/haskell (on lectura)

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                    (assumes .hs suffix)
[1 of 1] Compiling Main ...
Ok, one module loaded.

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

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
[1 of 1] Compiling Main           ( simple.hs, interpreted )
Ok, one module loaded.
> ... experiment ...
```

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

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

If you don't see "Compiling", the file hasn't changed!

Lather, rinse, repeat.

Functions with multiple arguments

Functions with multiple arguments

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

```
add x y = x + y :: Int
```

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

```
> add 3 5
8
```

Problem: Use `add` to compute the sum of 5, 3, 9 and 4.

Exercise

The Prelude has a **min** function:

```
> min 6 2
2
```

Problem: Define a function **min3** that computes the minimum of three values.

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

Solution:

```
min3 a b c = min a (min b c)
```

Does **min3** exemplify functional programming?

The type of `add`

Recall `add`:

```
add x y = x + y :: Int
```

Here is its type:

```
> :type add
add :: Int -> Int -> Int
```

`Int -> Int -> Int` is a *type expression*. It describes a type.

The operator `->` is *right-associative*. Let's add parentheses:

```
Int -> (Int -> Int)
```

But what does that mean?

Multiple arguments, continued

For reference, here's **`add`** and its type, with parentheses added:

```
> add x y = x + y :: Int
add :: Int -> (Int -> Int)
```

`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.

What does **`add (add 5 3) (add 9 4)`** mean?

```
(add ((add 5) 3)) ((add 9) 4)
```

The type of **min3**

Recall **min3**, but let's restrict it to **Int**s:

```
> min3 a b c = min a (min b c) :: Int
```

What's the type of **min3**?

```
> :t min3
min3 :: Int -> Int -> Int -> Int
```

How should the type expression be parenthesized to reflect associativity?

```
Int -> (Int -> (Int -> Int))
```

What does **min3 7 4 9** mean?

```
 > ((min3 7) 4) 9
4
```

Partial application

Partial application

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

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

```
> plusThree = add 3
plusThree :: Int -> Int
```

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

What will **plusThree 5** produce?

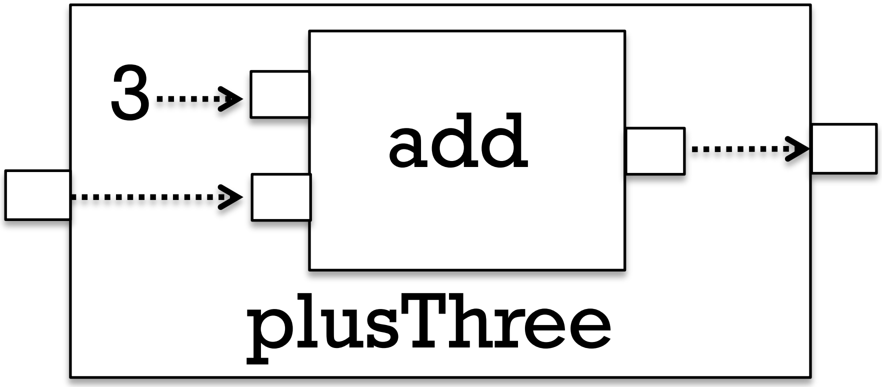
```
> plusThree 5
8
it :: Int
```

Partial application, continued

At hand:

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

> plusThree = add 3
plusThree :: Int -> Int
```

Imagine **add** and **plusThree** as machines with inputs and outputs:



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

Examples!

```
> p7 = add 7
p7 :: Int -> Int
> m3 = add (-3)
> p7 5
12
> m3 it
9
> add 4
<function>
> :type it
it :: Int -> Int
> it 10
14
> add it
<function>
```

At hand:

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

A little physics

Formula for displacement (*s*) of a falling object:

$s = \frac{1}{2}at^2$ (*a* is acceleration due to gravity, *t* is time)

Haskell:

```
> s a t = 0.5 * a * t^2
> s 32 1        # one second of falling towards earth
16.0            # 16 feet
> s 32 2        # two seconds...
64.0
> s 5.31 2      # two seconds of falling towards the moon
10.62
```

How can we make some use partial application?

(i.e., How can we use our brand new shiny tool?!)

A little physics, continued

At hand: (in **gravity.hs**)

```
> s a t = 0.5 * a * t^2
```

And...

```
> sEarth = s 32      # sEarth is a partial application
                     # 32 is "wired-in" for a
> sMoon = s 5.31
> sEarth 1
16.0
> sEarth 2
64.0
> sMoon 1
2.655
> sMoon 2
10.62
```

Another peek ahead!

Recall **map**:

```
> words "a test for words"
["a","test","for","words"]

> map length it
[1,4,3,5]

> map sEarth [1..5]
[16.0,64.0,144.0,256.0,400.0]

> map sMoon [1..5]
[2.655,10.62,23.895,42.48,66.375]

> map (s 80) [1..5]
[40.0,160.0,360.0,640.0,1000.0]
```

Another example

```
> hwrap t s = "<" ++ t ++ ">" ++ s ++ "</" ++ t ++ ">"

> hwrap "code" "print(3)"
"<code>print(3)</code>"

> bold = hwrap "b"

> uline = hwrap "u"

> bold "test"
"<b>test</b>"

> bold "Not" ++ " again, " ++ bold (uline "never!")
"<b>Not</b> again, <b><u>never!</u></b>"
```

A model of partial application

Given

```
wrap c s = c ++ s ++ c
```

and the following binding, what does **f** look like?

```
f = wrap "*"
```

Process:

Replace RHS with eqn. for function to partially apply (FtPA):

```
f = wrap c s = c ++ s ++ c
```

Remove **=**, name of FtPA (**wrap**), and first parameter (**c**):

```
f s = c ++ s ++ c
```

Replace occurrences of **c** with FtPA's argument ("*")

```
f s = "*" ++ s ++ "*"
```

Let's try **f**:

```
> f "test"
"*test*"
```

Partial application, continued

Consider:

```
> wrap c s = c ++ s ++ c
wrap :: [a] -> [a] -> [a]
> min3 x y z = min x (min y z)
min3 :: Ord a => a -> a -> a -> a
```

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

LYAH nails it:

> *... functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on.*

Partial application, continued

A little history:

- The idea of partially applying a function was first described by Moses Schönfinkel. (?)
- It was further developed by Haskell B. Curry.
- Both worked with David Hilbert in the 1920s.

What prior use have you made of partially applied functions?

$$log_2 N$$

Some key points about functions

- The *general form* of a function definition (for now):
  name param1 param2 ... paramN = expression

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

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

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.

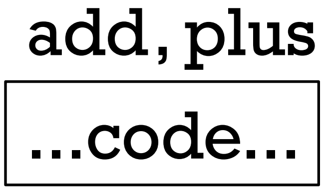
Functions are values

A fundamental characteristic of a functional language:

Functions are values that can be used as flexibly as values of other types.

The following creates a function and binds the name **add** to it.

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



The following *binds* the name **plus** to the expression **add**.

```
> plus = add
```

Either name can be used to reference the function value:

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

Functions are values in Python, too!

```
>>> def add(x,y): return x + y

>>> add
<function add at 0x7fda4efb68c8>

>>> add(3,4)
7

>>> plus = add
>>> plus(5,10)
15

>>> [plus][0](3,4)
7
```

An interesting thing in Python

```
>>> f = "just a test".split

>>> f()
['just', 'a', 'test']

>>> f('t')
['jus', ' a ', 'es', '']

>>> f
<built-in method split of str object at 0x7fc26703a230>

>>> str.split
<method 'split' of 'str' objects>
```

First-class values

Functions are said to be *first-class values* if a language allows them to be used in all* contexts where other values are allowed.

Wikipedia:

> In programming language design, a **first-class citizen** (also **type**, **object**, **entity**, or **value**) in a given programming language is an entity which supports all the operations generally available to other entities. ... (8/30/2022)

"Functions are values"

I consider "functions are values" to be synonymous with "functions are first-class *whatevers*".

If a language treats functions as values, then you can do some amount of functional programming in that language.

Python: Yes!
C: Yes!
Racket: Yes!
JavaScript: Yes!
Java: No! (Yes, there are lambdas, but still I say "No!")
Bash: Long answer...

Function/operator equivalence

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 +` indicates that the token `+` 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, continued

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

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

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

```
> 3 `add` 4
7

> 11 `rem` 3
2
```

Explore: Do `add` and `rem` have precedence and associativity?

Sidebar: Custom operators

We can define new operators in Haskell!

```
% cat plusper.hs
infixl 6 +%
x +% percentage = x + x * percentage / 100
```

Usage:

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

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

Haskell's standard modules define LOTS of 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 H10

Type Inferencing

Observation and inference

Imagine we're observing someone in a foreign land.

What do we infer if they pick up a bottle of something and ...

- ...squirt it on their food?
- ...squirt it into their mouth?
- ...squirt it onto some gears?
- ...squirt it onto a pile of wood and then sets it on fire?

What did Haskell infer?

Here's ***add*** again:

```
> add x y = x + y
> :t add
add :: Num a => a -> a -> a
> :info +
class Num a where
  (+) :: a -> a -> a
  ...
```

What did Haskell infer?

- Both arguments must have same type.
- That type must be an instance of the ***Num*** class.
- A value of that same type is returned.

Type inferencing

Haskell does type inferencing:

- The types of values are inferred based on the operations performed on the values.
- Inferences are based on an assumption of no errors.

Example:

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

Process:

1. `c` is being compared to `'A'` and `'Z'`
2. `'A'` and `'Z'` are of type `Char`
3. `c` must be a `Char`
4. The result of `&&`, of type `Bool`, is returned

Type inferencing, continued

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

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

What type will be inferred for **f** below?

```
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:

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

Type inferencing, continued

Recall this example:

```
> 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
  ...
```

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

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

Type inferencing, continued

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

The function below compares **x** to both a **Num** and a **Char**.

```
> g x y = x > 0 && x > '0'
<interactive>:1:13: error:
    • No instance for (Num Char) arising from the literal ‘0’
    • In the second argument of ‘(>)’, namely ‘0’
      In the first argument of ‘(&&)’, namely ‘x > 0’
      In the expression: x > 0 && x > '0'
```

What does the error "**No instance for (Num Char)**" mean?

Char is not an instance of the **Num** type class.

(**:info Num** shows **instance Num Int**, **instance Num Float**, etc.)

Type Specifications for Functions

Type specifications for functions

Even though Haskell has type inferencing, a common practice is to specify the types of functions.

Here's a file with several functions, each preceded by its type:

```
% cat typespecs.hs
min3::Ord a => 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 a ramification of the difference between the types of `add1` and `add2`?

```
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 `add2` requires `Integer`s.

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

Type specification for functions, continued

Two pitfalls related to type specifications for functions:

- 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.
- 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 inferred type looks reasonable, and the function works as expected, add a specification for that type.
- Toggling a type spec with a comment sometimes reveals problems.

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.

Indentation

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
```

Rule: A declaration can be continued across multiple lines by indenting subsequent lines more than the first line of the declaration.

These two weaving declarations are poor style but are valid:

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

Indentation, continued

Rule: A line that starts in the same column as did 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: error:
    parse error (possibly incorrect indentation ...)
  |
3 | x + y
  | ^
```

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

Guards

Guards

Recall this characteristic of mathematical functions:
"Are often specified with cases and expressions."

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

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

Notes:

- This definition would be found in a file, not typed in `ghci`.
- `sign x` appears just once. First guard might be on next line.
- The guards appear between `|` and `=`, and produce `Bool`s.
- 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"
```

Hint: guards are tried in turn.

Solution:

```
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 it compare to Java's **if-else**?

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

Java's **if-else** is a statement. It cannot be used where a value is required. `System.out.println(if (1 < 2) 3; else 4;); // (?)`

Does Java have an analog to Haskell's **if-else**?

The conditional operator: `1 < 2 ? 3 : 4`

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*.

Remember Python's *conditional expression*, too:

`"test"[1 if 2 < 3 else -1]`

An Original Thought

"A statement changes the *state* of the program while an expression wants to *express* itself. "

— Victor Nguyen, CSC 372, Spring 2014

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.
- For now, prefer guards over **if-else**.

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.

```
> factorial 40
815915283247897734345611269596115894272000000000
```

Write factorial in Haskell. Recall that `0!` is 1.

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

What is the type of **factorial**?

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

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

Lists

List basics

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

Here's one way to make a list.

```
> [7, 3, 8]
[7,3,8]
it :: Num a => [a]

> ['x', 10]
<interactive>:3:7:
   No instance for (Num Char) arising from the literal `10'
```

We can say that Haskell lists 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
```

(Note: A white lie, to be fixed!)

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 parts of a list!

List basics, continued

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

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

> it ++ reverse(it)
[3,4,10,20,30,30,20,10,4,3]
```

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

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

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

List basics, continued

Haskell has an *arithmetic sequence notation*:

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

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

> [10..5]
[]
```

List basics, continued

Here are **sum** and **product**:

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

> product [1..5]
120
```

Problem: Write a factorial function.

Solution:

```
factorial n = product [1..n]
```

List basics, continue

Here are ***drop*** and ***take***:

```
> drop 3 [1..10]
[4,5,6,7,8,9,10]

> take 5 [1.0,1.2..2]
[1.0,1.2,1.4,1.5999999999999999,1.7999999999999998]
```

Problem: **halves**

Problem:

Write **halves lst** that returns a list with the two halves of **lst**, a list. If **lst**'s length is odd, the second "half" is longer.

```
> halves [1..10]
[[1,2,3,4,5],[6,7,8,9,10]]

> halves [1]
[[],[1]]
```

halves will be a little repetitious because we don't have the *where clause* in our toolbox yet.

Solution: **halves**

Solution: (**halves.hs**)

```
halves lst =
    [take (length lst `div` 2) lst,
      drop (length lst `div` 2) lst]
```

List basics, continued

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

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

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

Speculate: do negative indexes work?

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

Important:

Much use of !! might indicate you're writing a Java, Python, C, etc. 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
```

Comparing lists, continued

Lists are compared *lexicographically*:

- Corresponding elements are compared until an inequality is found.
- The inequality determines the result of the comparison.
- If `L1` is a proper prefix of `L2`, then `L1` < `L2`.
- (Same as Python...)

Example:

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

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

Lists of Lists

We can make lists of lists.

```
> x = [[1], [2,3,4], [5,6]]
x :: Num a => [[a]]
```

Note the type: **`x`** is a list of **`Num a => [a]`** lists.

What's the length of **`x`**?

```
> length x
3
```

Wait! Is **`x`** homogeneous?

Lists of lists, continued

More examples:

```
> 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
```

I lied!

Earlier I showed you this:

length :: [a] -> Int

Around version 7.10 **length** was generalized to this:

length :: Foldable t => t a -> Int

We're going to think of **Foldable t => t a** as meaning **[a]**.

Instead of	**sum :: (Num a, Foldable t) => t a -> a**
Pretend this	**sum :: Num a => [a] -> a**

Instead of	**minimum :: (Ord a, Foldable t) => t a -> a**
Pretend this	**minimum :: Ord a => [a] -> a**

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!

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

> length asciiLets
52

> reverse (drop 26 asciiLets)
"zyxwvutsrqponmlkjihgfedcba"

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

> 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]
```

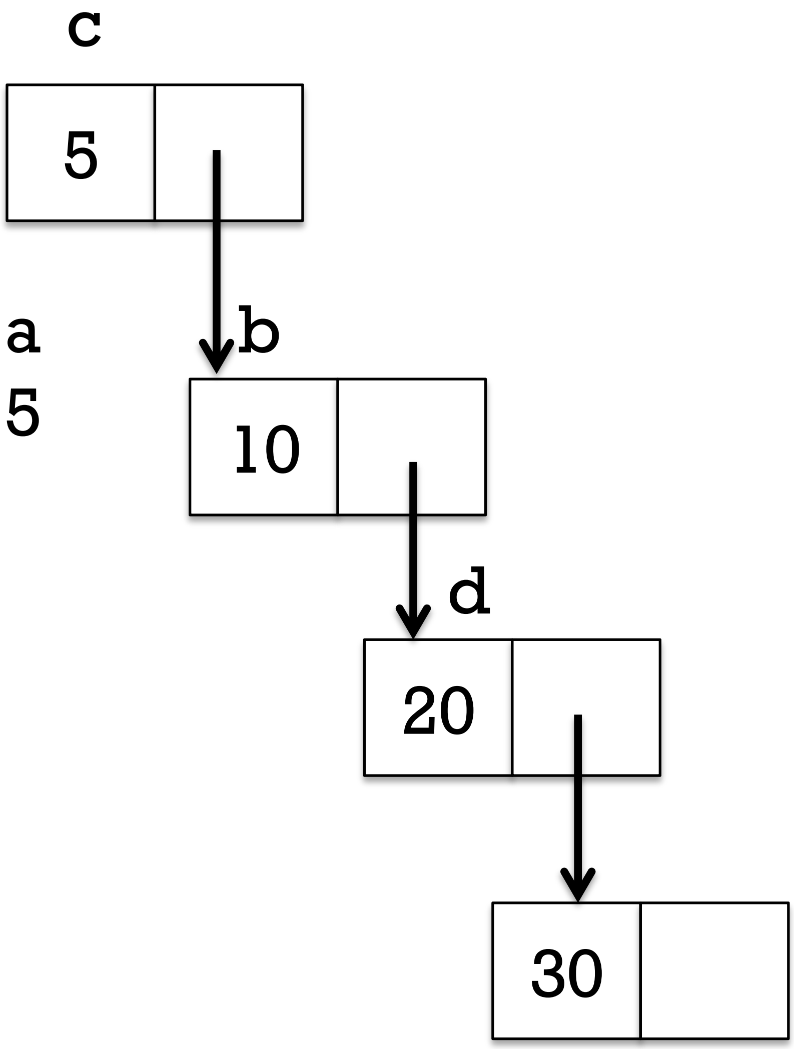
What's the type of the cons operator?

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

"cons" lists, continued

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

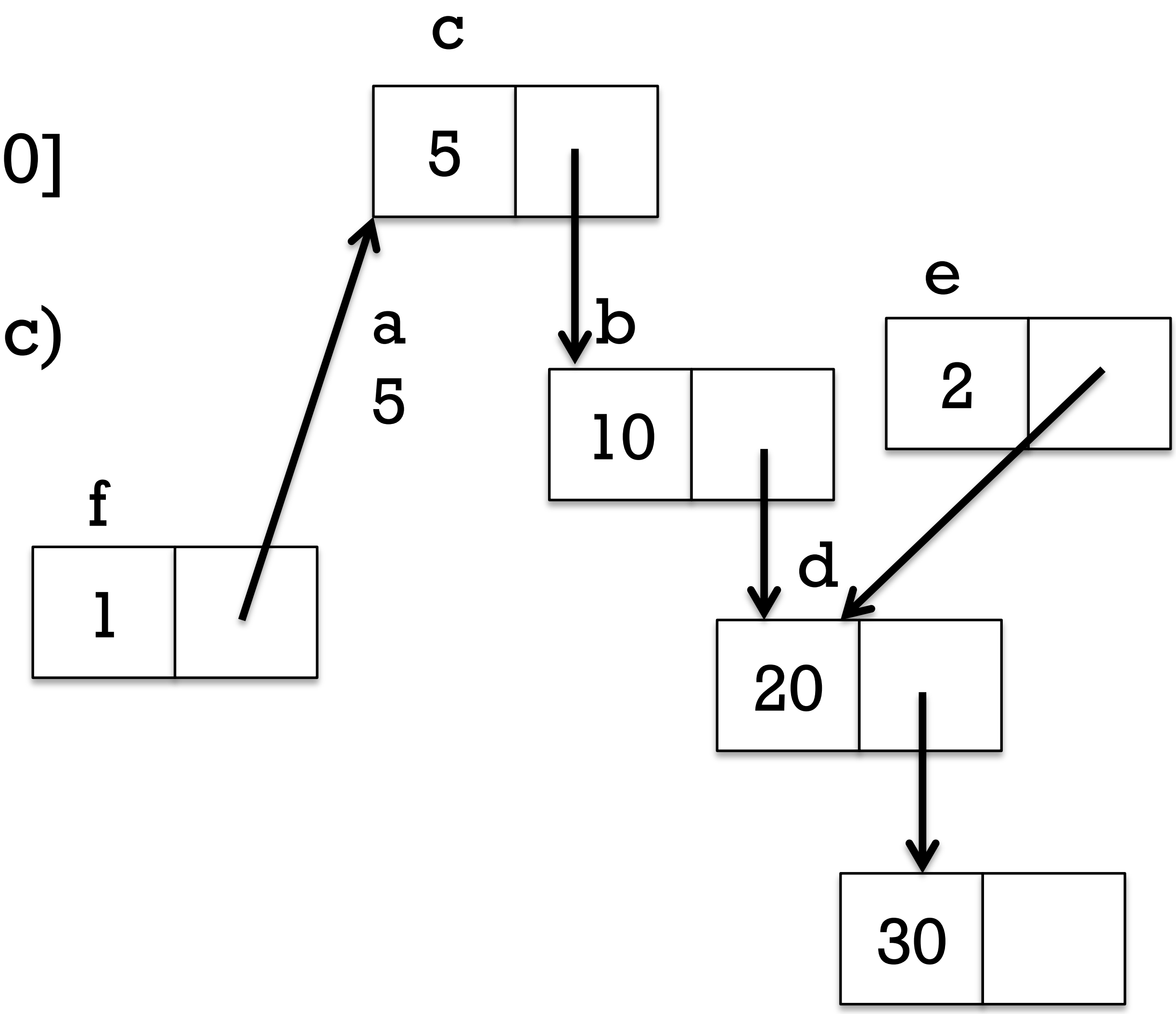
```
> a = 5
> b = [10,20,30]
> c = a:b
[5,10,20,30]

> head c
5

> tail c
[10,20,30]

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



"cons" lists, continued

A cons node can be referenced by multiple cons nodes.

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

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

> 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...
```

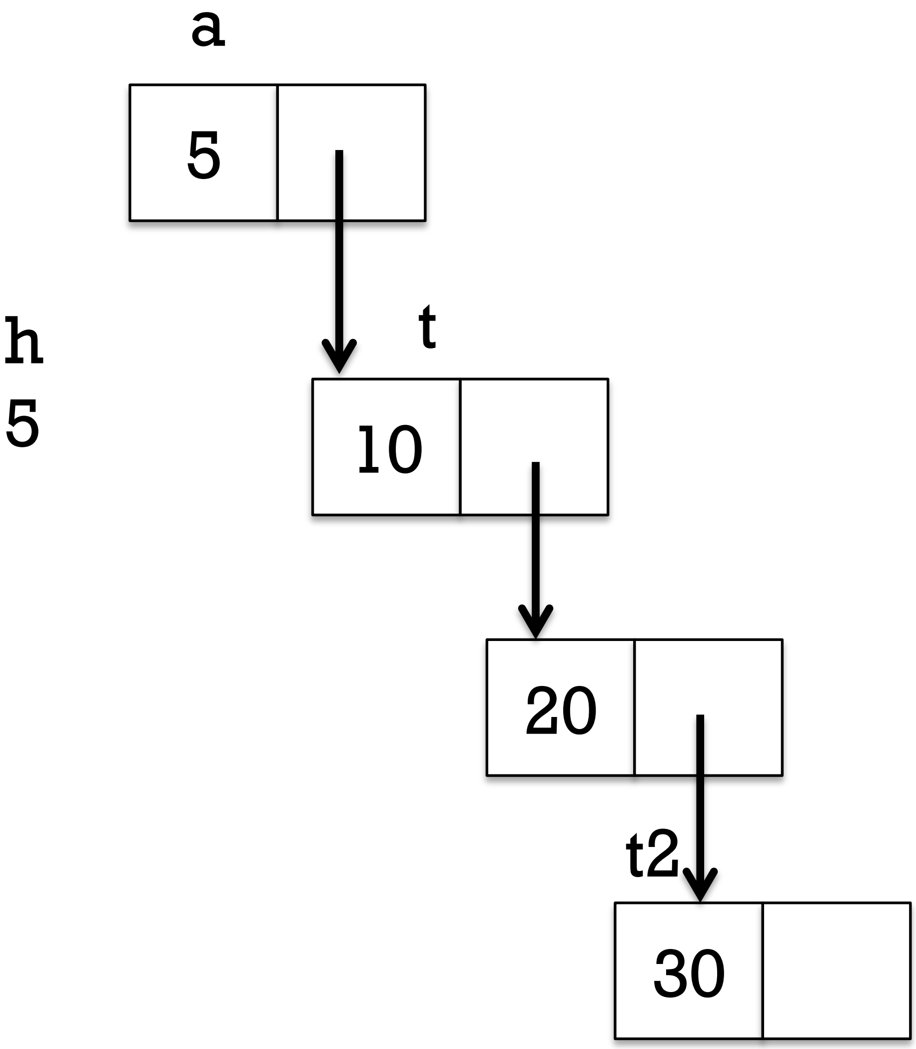
cons is right associative

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

head and **tail** visually

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

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

> h = head a
> h
5

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

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



A little on performance

Some things that are fast (*O(1)*) 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")

Some things that are slow (*O(n)*) with cons lists:

- Get the Nth element of a list
- Get the length of a list
- Concatenate two lists

  `[1,2,3] ++ [4,5]` turns into `1 : 2 : 3 : [4,5]`

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:

```
> 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
f = fromTo          -- So we can type f instead of fromTo
f 1 1000
f = fromTo 1        -- Note partial application
f 1000
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]
```

In English:

Make 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

Handy: the **show** function

What can you tell me about **show**?

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

show produces a string representation of a value.

```
> show 10
"10"

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

> show show
"<function>"
```

Important: **show** does not produce output!

What's the Python analog for **show**?

Is there a Java analog for **show**?

A little output

The **putStr** function outputs a string:

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

Type:

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

- `IO ()`, the type returned by `putStr`, is an *action*.
- An action is an interaction with the outside world.
- An interaction with the outside world is a side effect.
- An action can hold/produce a value. (simplistic)
- The construct `()` is read as "unit".
- The unit type has a single value, unit.
- Both the type and the value are written as `()`.
- Contrast: `getChar :: IO Char`

Our approach

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

- A helper function produces a ready-to-print string that represents all the output to be produced by the function.
    - We'll often use **show** to create pieces of the string.
    - The string will often have embedded newlines.

- The top-level function calls the helper function to get a string.

- The top-level function uses **putStr** to print the string returned by the helper.

Our approach, continued

Here's Java analog for our approach for functions that produce output:

```
public class output {
   public static void main(String args[]) {
      System.out.print(computeOutput(args));
      }

   public String computeOutput(String args[]) {
  ...builds a string that is the entire output for this run...
     }
   }
```

We use **print** instead of **println** so that **computeOutput** has control over whether the output should include a newline.

printN

Let's write a function to <u>print</u> the integers from 1 to N:

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

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

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

Solution for **printN'**:

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

printN, continued

At hand:

```
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 now write the top-level function:

```
> printN n = putStr (printN' n)
> :t printN
printN :: (Eq a, Num a, Show a) => a -> IO ()
```

printN, continued

All together in a file:

```
% cat printN.hs
printN n = putStr (printN' n)

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

% ghci printN
> printN' 3
"1\n2\n3\n"

> printN 3
1
2
3
```

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)
```

- Should we have used a helper function **charrow rowLen char**?
- How does this approach contrast with how we'd write it in Java?

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.

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

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

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.

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

> h
10

> t
[20,30]
```

What values get bound by the following pattern?

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

> d
[]
```

-- *Why in a list?*

-- *Why did I do* **[c,b,a]** *instead of* **[d,c,b,a]**?

Patterns, continued

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

```
> _ :( 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.

```
> a:a:[] = [3,3]
<interactive>: error: Multiple declarations of ‘a’
```

A failed pattern isn't manifested until we try to see what's bound to a name.

```
> a:b:[] = [1]
> a
*** Non-exhaustive patterns in a : b : []
```

Practice

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

`a:b:c = ...`

A list containing at least two elements.

Does `[[10,20]]` match?

`[20,30]` ?

`"abc"` ?

`[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)
```

Idiomatic:

```
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        -- with head/tail
       Num a => [a] -> a        -- with pattern
```

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:  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)
```

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

if-else—third choice...

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

Students wrote...

"Throughout the assignment I tried to keep in mind that I should use patterns first then guards if patterns didn't work.

"However, as I was doing the assignment, I realized that sometimes I couldn't see the patterns until I had written them as guards, so I would go back and change them.

"As I continued with the assignment, this happened less because the more code I wrote the more I was able to see patterns before I had them written as guards."

—Kelsey McCabe, Spring 2016, **`a3/observations.txt`**

"...there were multiple cases where I solved a problem with guards and failed multiple test cases, only to replace the logic with patterns and have it work."

—Ryan Smith, Fall 2022, **`a3/observations.txt`**

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

An earlier *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 ***name*** is the *function binding* for ***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 re "non-exhaustive patterns" exception on **f []** (if **-fwarn-incomplete-patterns** specified).

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"
```

Recall 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. (No loops or assignments.)
- Do **strcmp** and **strchr**, too!
- Could you do **strcpy**, too?
- Mail us!

Python programmers:

- In a functional style write **size(x)**, which returns the number of elements in the string, list, or range **x**.
  Restriction: You may not use **type()** or **len()**.
- Mail us!

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 :: (Num a, Fractional c) => (a, [Char], c)

> (3 < 4, it)
(True,(1,"two",3.0))
it :: (Num a, Fractional c) => (Bool, (a, [Char], c))

> (head, tail, [words], putStr)
(<function>,<function>,[<function>],<function>)
it ::  ([a1] -> a1, [a2] -> [a2], [String -> [String]], String -> IO ())
```

Of course, 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:

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

What's the type of `pair`?

```
pair :: a -> b -> (a, b)
```

Usage:

```
> 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.

```
> p = pair 30 "forty"

> 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 have the same syntax as expressions to create tuples.

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

```
> middle ("372", "GS 906", "Mitchell")
"GS 906"

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

(Solution on next slide. Don't peek! This means **<u>you</u>**!)

Tuples, continued

At hand:

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

Solution:

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

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

```
middle :: (a, b, c) -> b
```

Will the following call work?

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

Tuples, continued

Problem: Write a function ***swap*** that behaves like this:

```
> swap ('a',False)
(False,'a')

> swap (1,(2,3))
((2,3),1)
```

Solution:

```
> swap (x,y) = (y,x)
```

What is the type of ***swap***?

```
swap :: (b, a) -> (a, b)
```

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',1
0), ...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.

```
[('a',0),('b',1),('c',2),('d',3),('e',4),...]
```

Solution:

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

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

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)
```

Here's one of them. What does it mean?

```
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.

Using a tuple lets type-checking ensure that an exact number of values is being aggregated, even if all values have the same type.

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

In our Haskell we can't write functions that operate on tuples of arbitrary arity.

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

Sidebar: To curry or not to curry?

Consider these two functions:

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

> 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
```

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

Important: Note the difference in types!

The **where** clause

The **where** clause

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

Here's a declaration that shows the syntax; the computation is not meaningful.

```
f x
     | x < 0 = g a + g b
     | a > b = g b
     | otherwise = c + 10
   where {
     a = x * 5;
     b = a * 2 + x;
     g t = log t + a;
     c = a * 3;
     }
```

The ***where*** *clause* specifies bindings that may be needed when evaluating the guards and their associated expressions.

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

The **where** clause, continued

A Computer Science Tapestry by Owen Astrachan shows an interesting way to raise a number to a power:

```
power base expo
    | expo == 0 = 1.0
    | even expo = semi * semi
    | otherwise = base * semi * semi
   where {
      semi = power base (expo `div` 2)
      }
```

Binding **semi** in the **where** clause avoids lots of repetition.

Exercise for the mathematically inclined: Figure out how it works.

Problem: **halves**

Recall:

```
> halves ['a'..'z']
("abcdefghijklm","nopqrstuvwxyz")

halves lst =
  [take (length lst `div` 2) lst, drop (length lst `div` 2) lst]
```

Problem: Rewrite **halves** to be less repetitious. Also, have it return a tuple instead of a list.

Solution:

```
halves lst = (take halflen lst, drop halflen lst)
    where {
        halflen = (length lst `div` 2)
        }
```

The *layout rule*

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 Haskell's *layout rule* and write it like this instead:

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

Look Mom, no braces! (No semicolons, either.)

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 <u>absence of a brace</u> after **where** activates the layout rule.

The column position of the <u>first token after **where**</u> establishes the column in which declarations in 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 123-124.)

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.

Literals in patterns

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
```

For contrast, with guards:

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

Usage:

```
> f 1
10

> f 3
3
```

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

What's an important quality of the definition above?

Patterns can be arbitrarily complex.

3.17.1 in H10 shows the full syntax for patterns.

Errors

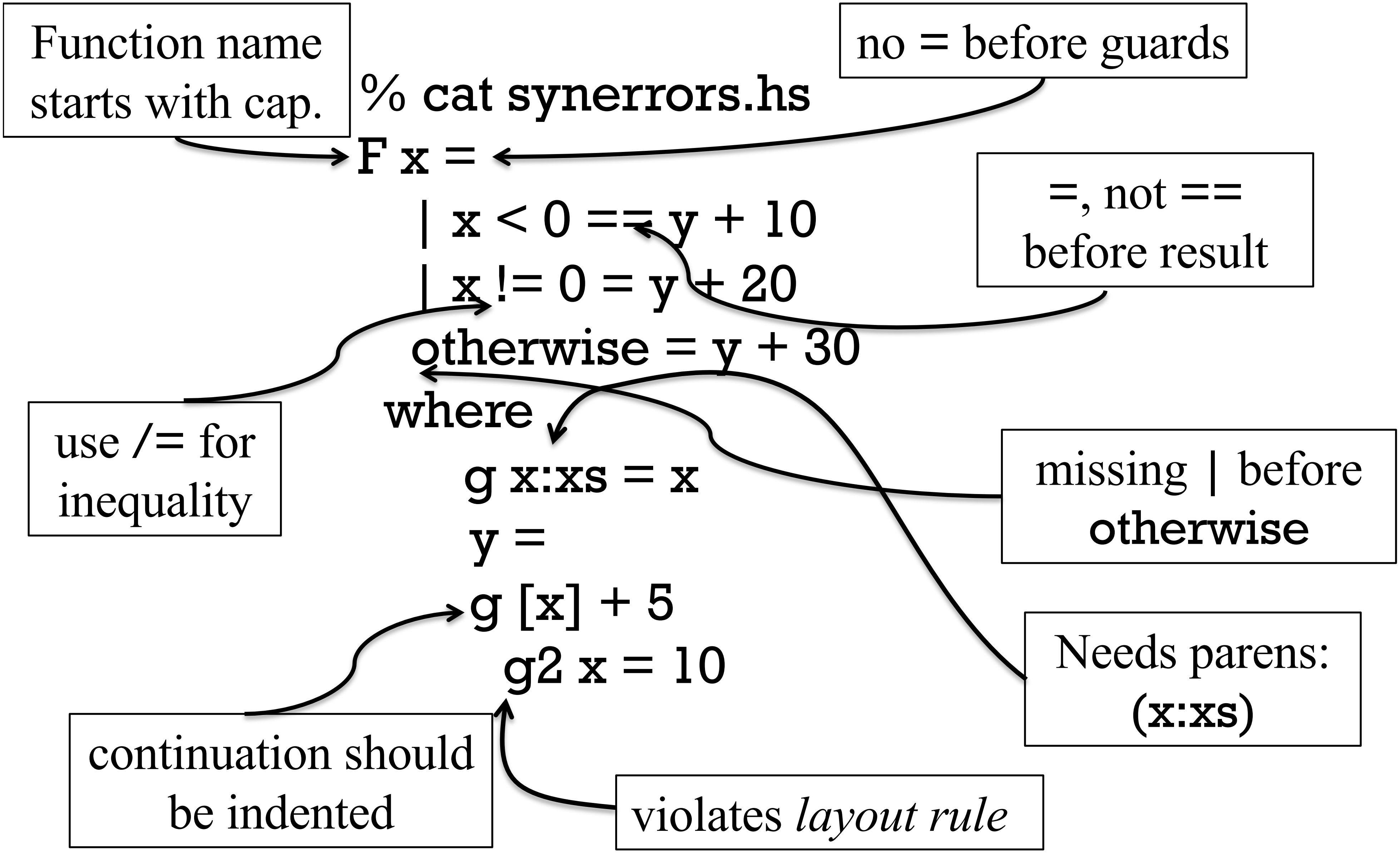
Syntax errors

What syntax errors do you see in the following file?

```
% cat -n haskell/synerrors.hs
     1	F x =
     2	    | x < 0 == y + 10
     3	    | x != 0 = y + 20
     4	    otherwise = y + 30
     5	  where
     6	       g x:xs = x
     7	       y =
     8	       g [x] + 5
     9	          g2 x = 10
```

Syntax errors, continued

What syntax errors do you see in the following file?



```
% cat synerrors.hs
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
```

Function name starts with cap.

no = before guards

=, not == before result

use /= for inequality

missing | before **otherwise**

Needs parens: **(x:xs)**

continuation should be indented

violates *layout rule*

Type errors

In my opinion, producing understandable messages for type errors is what **ghci** is worst at.

If no polymorphic functions are involved, type errors are typically easy to understand.

```
> :type chr
chr :: Int -> Char

> 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 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 '(a1, b)'
                with actual type '[a] -> (a1, b)'
Probable cause: countEO is applied to too few arguments
  In the expression: countEO
```

What's the problem?

It's expecting a tuple, `(a1,b)` but it's getting a function, `[a] -> (a1,b)`

Typically, instead of errors about too few (or too many) function arguments, you get <u>function types popping up in unexpected places</u>.

Type errors, continued

Here's an example of omitting an operator:

```
> add3 x y z = x + y z
> add3 4 5 6
<interactive>:9:1: error:
Non type variable argument in the constraint:
Num (t -> a) (Use FlexibleContexts to permit this)
```

Looking at the type of `add3` sheds some light on the problem:

```
> :t add3
add3 :: Num a => a -> (t -> a) -> t -> a
```

A function type unexpectedly being inferred for **y** suggests we should look at how **y** is being used.

Try it: See if a type declaration for `add3` leads to a better error.

Type errors, continued

Is there an error in the following?

```
f [] = []
f [x] = x
f (x:xs) = x : f xs
```

A simple way to produce an infinite type:

```
x = head x
```

```
Occurs check: cannot construct the infinite type: a ~ [a]
    Expected type: [a]
    Actual type: [[a]]
In the expression: x : f xs
In an equation for 'f': f (x : xs) = x : f xs
```

("a is a list of as"--whm)

The second and third clauses are fine by themselves but together they create a contradiction.

<u>Technique: Comment out clauses (and/or guards) to find the troublemaker, or incompatibilities between them.</u>

Type errors, continued

Recall **ord :: Char -> Int.**

Note this error:

```
> ord 5
  No instance for (Num Char) arising from the literal `5'
```

The error "**No instance for (*TypeClass Type*)**" means that ***Type*** (**Char**, in this case) is not an instance of ***TypeClass*** (**Num**).

```
> :info Num
....
instance Num Word
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

instance Num Char doesn't appear

Really?

```
> (mod 17 3) / 2          -- Thanks to Freya Barber for this one!
<interactive>:11:1: error:
    • Ambiguous type variable ‘a0’ arising from a use of ‘print’
      prevents the constraint ‘(Show a0)’ from being solved.
      Probable fix: use a type annotation to specify what ‘a0’ should be.
      These potential instances exist:
        instance Show Ordering -- Defined in ‘GHC.Show’
        instance Show Integer -- Defined in ‘GHC.Show’
        instance Show a => Show (Maybe a) -- Defined in ‘GHC.Show’
        ...plus 23 others
        ...plus 13 instances involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In a stmt of an interactive GHCi command: print it
> (mod 17 3)
2
> :type it
it :: Integral a => a
> it / 2        -- produces the error
```

Works: `(mod 17 2) `div` 2`

Sidebar: LHtLaL—Start an error collection!

If a language [implementation] has obscure error messages, collect examples of errors and the message(s) produced.

You might also intentionally create error cases and catalog the errors produced.

Where could we keep our error collection?

`~/notes/haskell.txt`

"For this invention will produce forgetfulness in the minds of those who learn to use it, because they will not practice their memory. ..."—Socrates on writing

LHtLaL, continued

How can the following Python errors be produced?

```
KeyError: 3
    >>> d={}; d[3]
    >>> {}[3]
```

```
TypeError: 'int' object is not callable
    >>> f=3; f(4)
    >>> 3(4)
    >>> range(1,10).start()
```

If you see the following, what does it mean?

```
    <built-in method join of str object at 0x10457dc70>
    >>> ",".join
```

Debugging

Debugging in general

My general strategy for debugging Haskell code:

AVOID THE NEED TO DO ANY DEBUGGING IN HASKELL!

A good process for Haskell beginners when writing a function:

1. Work out expressions at the **ghci** prompt as shown on 176.
2. Write a single clause for the function using those expressions and put it in a file.
3. Load the file with **ghci** and test that one clause.
4. Repeat with the next clause for function. Etc.

With conventional languages I might write dozens of lines of code before running them.

With Haskell I might write a half-dozen lines of code before running them.

The **trace** function

The **Debug.Trace** module has a **trace** function.

Observe:

```
> import Debug.Trace  -- put it in your ghci config file
> :t trace
trace :: String -> a -> a

> trace "a tuple" (True, 'x')
a tuple
(True,'x')
```

What's happening?

trace *string expr* returns *expr* but also outputs *string* as a side-effect. (!)

- Great for debugging!
- Completely subverts Haskell's isolation of the side-effects of output.

trace, continued

Here's 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
```

trace, continued

Let's add **trace** calls to **sumElems**:

```
sumElems [] = trace "sumElems []" 0
sumElems lst@(h:t) =
    trace ("sumElems " ++ show lst) h + sumElems t
```

Execution:

```
> sumElems [5,1,4,2,3]
sumElems []
sumElems [3]
sumElems [2,3]
sumElems [4,2,3]
sumElems [1,4,2,3]
sumElems [5,1,4,2,3]
15
```

Unfortunately, due to Haskell's lazy evaluation, the output's order is the opposite of what we'd expect. But it does show "progression".

trace, continued

Code for **buildingAtHeight** in **street.hs**, an old 372 problem:

```
buildingAtHeight (width, height, ch) n =
    replicate width (if n > height then ' ' else ch)
```

Outputting **width**, **height**, and **ch** with labels is tedious:

```
buildingAtHeight (width, height, ch) n =
  trace ("width " ++ show width ++ ", height: " ++
     show height ++ ", ch: " ++ show ch)
   replicate width (if n > height then ' ' else ch)
```

Example of trace **output**: **width: 3, height: 2, ch: 'x'**

Use a tuple to simplify the **trace** call:

```
buildingAtHeight (width, height, ch) n =
   trace (show ("width:", width, "height", height, "ch", ch))
    replicate width (if n > height then ' ' else ch)
```

Example of **trace** output: **("width:",3,"height",2,"ch:",'x')**

Sidebar: Tracing in Icon

Icon has a built-in tracing mechanism.

Here's **sumElems** in Icon:

```
% cat -n sumElems.icn
     1  procedure main()
     2      sumElems([5,1,4,2,3])
     3  end
     4
     5  procedure sumElems(L)
     6      if *L = 0 then
     7          return 0
     8      else
     9          return L[1] + sumElems(L[1:-1])
    10  end
```

Sidebar, continued

Execution:

```
% TRACE=-1 icont sumElems.icn -x
...
             :       main()
sumElems.icn :     2  | sumElems(list_1 = [5,1,4,2,3])
sumElems.icn :     9  | | sumElems(list_2 = [5,1,4,2])
sumElems.icn :     9  | | | sumElems(list_3 = [5,1,4])
sumElems.icn :     9  | | | | sumElems(list_4 = [5,1])
sumElems.icn :     9  | | | | | sumElems(list_5 = [5])
sumElems.icn :     9  | | | | | | sumElems(list_6 = [])
sumElems.icn :     7  | | | | | | sumElems returned 0
sumElems.icn :     9  | | | | | sumElems returned 5
sumElems.icn :     9  | | | | sumElems returned 10
sumElems.icn :     9  | | | sumElems returned 15
sumElems.icn :     9  | | sumElems returned 20
sumElems.icn :     9  | sumElems returned 25
sumElems.icn :     3  main failed
```

I know of no better out-of-the-box tracing facility in any language.

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).

Larger examples

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:

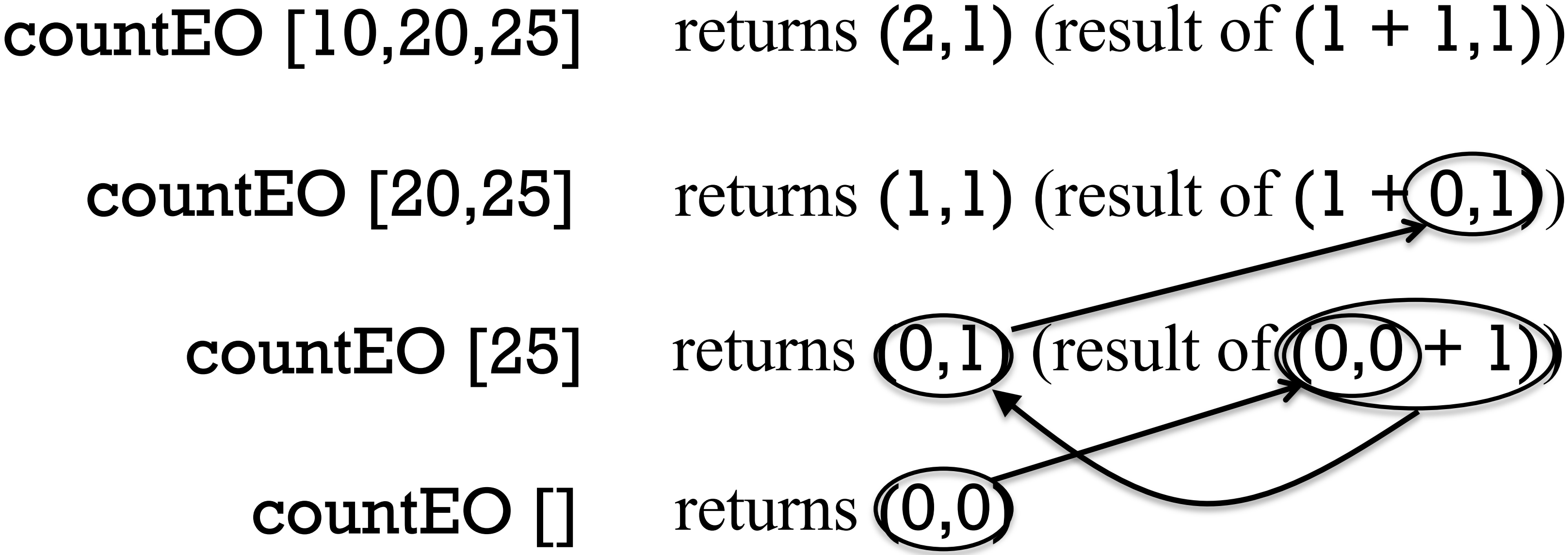
```
countEO [] = (0,0)          -- no odds or evens in []
countEO (x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
  where
    (evens, odds) = countEO xs     -- do counts for tail first!
```

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)

countEO with **trace**

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)
```

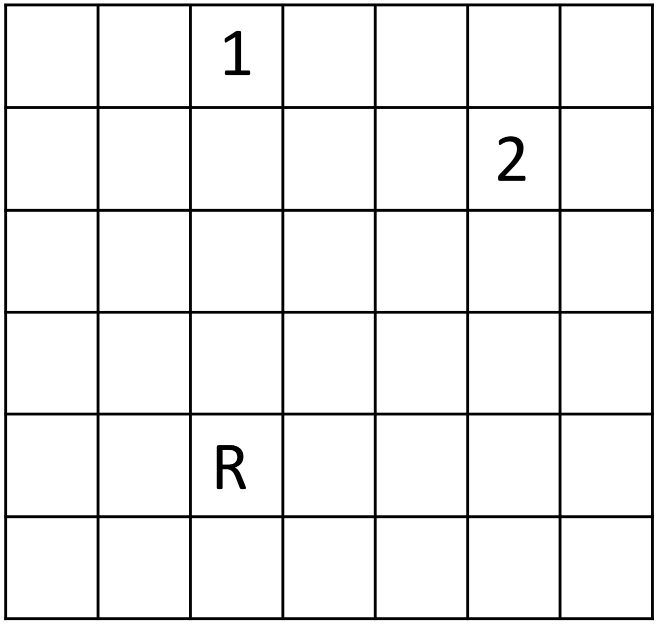
Before tracing the **where** was:

```
(evens,odds) = countEO xs
```

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"
```

		1				
					2	
		R				

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])
```

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, with **makeTuples** in a **where**

```
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)]
```

```haskell
{-  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?

Running **tally** from the command line

Let's run it on lectura...

```
% code=/cs/www/classes/cs372/spring23/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
-rwxrwx--- 1 whm whm 940968 Sep 13 12:09 tally

% ./genchars 1000000 > 1m
% time ./tally < 1m > out
real    0m5.554s
user    0m5.393s
sys     0m0.100s
```

tally performance in other languages

Here are user CPU times for implementations of **tally** in several languages. The same ten million letter file was used for all timings.

Language	Time in seconds; mean of two or more runs
Haskell	`57.284`
Ruby	`18.589` (v2.7.0; much slower than 2.2.4 (or 1.9.3?))
Icon	`8.248`
Python 3	`1.131`
Python 2	`0.824`
C w/ `gcc -O3`	`0.031` (2.97 for one **billion** letters)

Java?

Our **tally** implementation is very simplistic. An implementation of **tally** by an expert Haskell programmer, Chris van Horne, ran in 1.71 seconds for one **billion** letters. (See **spring23/haskell/tally-cwvh[12].hs**.)

Then I revisited the C version (**tally2.c**) and processed one **billion** letters in 0.59 seconds.

Real world problem: "How many lectures?"

Here's an early question when planning a course for a particular semester:

> "How many lectures will there be, and on what dates?"

How should <u>we</u> answer that question?

- Do it on paper?
  - No!
- Google for a course planning app?
  - No!
- Let's write a Haskell program!
  - Cool!

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),(8,2
43),(9,273),(10,304),(11,334),(12,365)]

toOrdinal (month, day) = days + day
   where
      (_,days) = offsets!!(month-1)
```

```
> toOrdinal (12,31)
365
```

```
> fromOrdinal 32
(2,1)
```

```
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?"
```

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"**

1 is lecture #1; 15 is 15th day of year

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 **spring23/haskell/classdays.hs**

Higher-order functions

Remember: Functions are values

Recall this 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?

```
> (if 3 < 4 then head else last) "abc"
'a'

> funcs = (tail, (:) 100)

> nums = [1..10]

> 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

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 different characters.

Comparing functions

Can functions be compared?

```
> add == (+)
• No instance for (Eq (Integer -> Integer -> Integer))
      arising from a use of '=='
```

You might see a proof based on this in CSC 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: (`len`'s type: `(Num a, Eq t) => [t] -> a`)

```
len list@(_:t)
    | list == [] = 0
    | otherwise = 1 + len t
```

A simple *higher-order function*

Definition: A *higher-order function* is a function that (and/or)

- Has one or more arguments that are functions
- Returns a function

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:

```
> twice f x = f (f x)
> twice tail [1,2,3,4,5]
[3,4,5]
```

Let's make the left-associativity explicit:

```
> (twice tail) [1,2,3,4,5]
[3,4,5]
```

Consider a partial application...

```
> t2 = twice tail          -- like t2 x = tail (tail x)
> t2
<function>
it :: [a] -> [a]
```

twice, continued

At hand:

```
> 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:

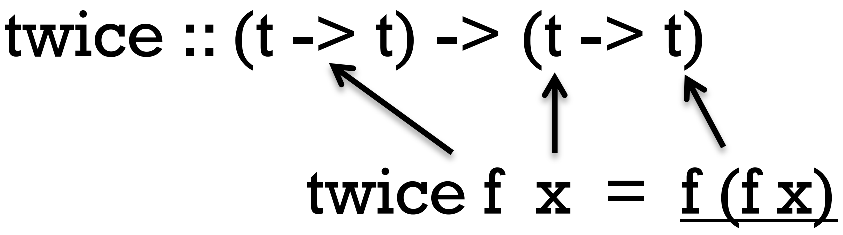
```
twice f x = f (f x)
```

> A *higher-order function* is...
> a function that (1) has one or more arguments that are functions and/or (2) returns a function.

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

```
> :t twice
twice :: (t -> t) -> t -> t
```

Parentheses added to show precedence:



```
twice :: (t -> t) -> (t -> t)
```

```
twice f  x  =  f (f x)
```

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! (Why?)

Its first argument is a function.

map, continued

At hand:

```
> map double [1..5]
[2,4,6,8,10]
```

Problem: Write **map**!

```
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 **map**'s input and output lists?

The lengths are always the same.

map, continued

More mapping:

```
> 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"
```

Problem: Write a function **f** such that **map f values** "removes" the odd numbers from the list **values**.

Sidebar: **map** can go parallel

Another mapping:

```
> map windSpeed [loc1, loc2, loc3, ...]
[8.7,12.3,10.2,...]
```

Equivalent:

```
[windSpeed loc1, windSpeed loc2, windSpeed loc3, ...]
```

- Because functions have no side effects we can immediately turn a mapping into a parallel computation.
- If a machine has 64 CPUs we might process a thousand-element list with sixteen(+/-) batches of 64-element **map**s.

See *Parallel and Concurrent Programming in Haskell* by Marlow

Google for **MapReduce**

`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?

```
> 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
```

map in Python

Python **2**:

```
>>> map(len,"map in Python".split())
[3, 2, 6]

>>> map
<built-in function map>
```

Python **3**:

```
>>> map(len,"map in Python".split())
<map object at 0x11418d240>

>>> list(map(len,"map in Python".split()))
[3, 2, 6]

>>> map
<class 'map'>
```

map in Python, continued

```
>>> map(print, range(1,6))
<map object at 0x114187fd0>

>>> list(_)   # _ in the Python REPL is like it in ghci
1
2
3
4
5
[None, None, None, None, None]

>>> map(print, range(1,1000000000000000000000000))
<map object at 0x114187fd0>
```

More: docs.python.org/3/library/functools.html

Bird's-eye view of higher order functions

"[Higher-order functions] allow common programming patterns to be encapsulated as functions." —Hutton, 2e

"..we can think of higher-order functions as control structures which we can define ourselves." —Thompson, 3e

Contrast:

- *Design Patterns,* by the "Gang of Four", provides a textual recipe for approaching common programming problems.
  "Making C++ Suck Less"—Vlissides

- Higher-order functions provide encapsulated units for common programming problems.
  (Instead of "write it this way...", it's "call this function".)

travel, revisited

Now that we're good at recursion...

Some of the problems on assignment 4 will encourage working with higher-order functions by prohibiting you from <u>writing</u> any recursive functions!

Think of it as isolating muscle groups when weight training.

Here's a simple way to avoid what's prohibited:

<u>Pretend that you don't understand recursion!</u>

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 systems, I'd do **`chmod 0`** on an appropriate section of your brains.

travel revisited

Recall our traveling robot: (slide 251+)

```
> 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 without <u>writing</u> any recursive functions?

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 the tuples with **map**?

No!

travel, continued

We have:

```
> 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:

```
> 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 know of recursion to write this version of **travel**?

No.

Did we <u>write</u> any recursive functions?

No.

Did we <u>use</u> any recursive functions?

<u>Maybe</u>. But <u>using</u> 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 f list** doing?

Producing the values in **list** for which **f** returns **True**.

Note: Think of **filter** as filtering <u>in</u>, not filtering <u>out</u>.

What is the type of **filter**?

```
filter :: (a -> Bool) -> [a] -> [a]
```

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.

More...

```
>  filter (<= 5) (filter odd [1..10])
[1,3,5]

> map (filter isDigit) ["br549", "24/7"]
["549","247"]
```

For following, note that **(\`elem\` ...)** is a section.

```
> filter (`elem` "aeiou") "some words here"
"oeoee"
```

filter, continued

At hand:

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

> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Problem: Write **filter**!

```
filter _ [] = []
filter f (x:xs)
    | f x = x : filteredTail
    | otherwise = filteredTail
  where
    filteredTail = filter f xs
```

Prelude functions that use predicates

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
"  "
```

How could we find other Prelude functions that use predicates?

```
% grep "(a -> Bool)" prelude.txt
```

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*

Python has **filter**, too. Ditto for JavaScript and many other languages. And, most higher-order functions are easy to write; a language simply needs to treat functions as values.

filter in Python

Here is **filter** in Python, along with two predicates:

```
% cat haskell/filter.py
def filter(p, L):
    result = []
    for e in L:
        if p(e):
            result.append(e)
    return result

def odd(n): return n % 2 == 1

def short(x): return len(x) < 4

def filter2(p,L): return [e for e in L if p(e)]
```

filter in Python, continued

Usage:

```
% python -i haskell/filter.py     # -i loads the source file
                                  # and starts the REPL
>>> filter1(odd, [3,1,4,6,9])
[3, 1, 9]

>>> filter1(short, "here are the words".split())
['are', 'the']

>>> filter1(bool, ["abc", "", 1, 0, [False], 2 < 3])
['abc', 1, [False], True]
```

There's a built-in **filter**, too!

Anonymous functions

Anonymous functions

Imagine that for every number in a list we'd like to double it and then subtract five.

Here's one way to do it:

```
> 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]
```

What benefits does the anonymous function provide?

Anonymous functions, continued

At hand:

```
f n = n * 2 - 5
map f [1..5]
```

vs.

```
map (\n -> n * 2 - 5) [1..5]
```

an anonymous function

The most common use case for an anonymous function: (my speculation)
Supply a simple "one-off" function to a higher-order function.

Anonymous functions...

- Directly associate a function's definition with its only use.
- Let us avoid the need to think up a good name for a function! ☺
- Can be likened to not using an intermediate variable:

```
int t = a * 3 + g(a+b);      // Java
return f(t);
```

vs.

```
return f(a * 3 + g(a+b));
```

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]
```

These terms are synonymous with "anonymous function":

Lambda abstraction (H10)
Lambda expression
Just *lambda* (LYAH).

The \ character was chosen due to its similarity to λ (Greek lambda), used in the *lambda calculus*, another system for expressing computation.

Anonymous functions, continued

What will **ghci** say?

```
> \x y -> x + y * 2
<function>
> it 3 4
11
```

\x y -> x + y * 2 is an <u>expression</u> whose value is a function.

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"]
```

Sidebar: Three languages

A simple anonymous function in Haskell...

```
> \s -> s ++ "-" ++ show (length s)
<function>
> it "abc"
"abc-3"
```

Python...

```
>>> lambda s: s + '-' + str(len(s))
<function <lambda> at 0x10138af28>
>>> _('abc')
'abc-3'
```

and JavaScript...

```
> f = function (s) { return s + '-' + s.length }
> f("abc")
"abc-3"
```

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 $s23/web2
72632:formaldehydesulphoxylate
140339:pathologicopsychological
175108:scientificophilosophical
200796:tetraiodophenolphthalein
203042:thyroparathyroidectomize
```

```
% head .../web2
A
a
aa
aal
aalii
aam
Aani
aardvark
...
```

Imagining that we don't understand recursion, how can we approach it in Haskell?

longest, continued

Let's work with a small file for development purposes:

```
% 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"
```

Let's have a **longest** function that operates on a single string that represents the 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:

```
> bytes = "data\nto\ntest\n"

> lns = lines bytes
~~> lns~~
["data","to","test"]
```

Note: To save space in this example, we'll show the value bound immediately after each binding.

Let's use **zip3** and **map length** to create (length, line-number, line) triples:

```
> 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 **Data.List.sort :: Ord a => [a] -> [a]** on them:

```
> sortedTriples = reverse (Data.List.sort triples)
[(4,3,"test"),(4,1,"data"),(2,2,"to")]a
```

Tuples are sorted based on their first value, with the second value resolving any ties, etc. (Just like Python.)

Why do we reverse the list?

"The **sort** *function [...] is a special case of* **sortBy***, which allows the programmer to supply their own comparison function."*

longest, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
```

Let's make a helper function to get the first element of a 3-tuple:

```
> first (len, _, _) = len
```

Let's get the length of the longest word:

```
> maxLength = first (head sortedTriples)
4
```

We have a tie for the longest word! What to do?

longest, continued

The Prelude's **takeWhile** has this type:

```
(a -> Bool) -> [a] -> [a]
```

Speculate: What does **takeWhile** do?

Let's experiment!

```
> :t odd
odd :: Integral a => a -> Bool

> takeWhile odd [9, 13, 5, 12, 7]
[9,13,5]

> takeWhile (>5) [9, 13, 5, 12, 7]
[9,13]
```

longest, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]

> maxLength
4
```

```
> maxTriples = takeWhile
    (\triple -> first triple  == maxLength) sortedTriples
[(4,3,"test"),(4,1,"data")]
```

anonymous function for **takeWhile**

Should we have just used **filter** instead?

```
maxTriples = filter
      (\triple -> first triple  == maxLength) sortedTriples
```

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:

```
> linesWithNums =
      map (\(_,num,line) -> show num ++ ":" ++ line)
              maxTriples
["3:test","1:data"]
```

We can now produce a ready-to-print result:

```
> result = unlines (reverse linesWithNums)
"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 (Data.List.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
```

Look! No conditional code!

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  -- 'do' "sequences" its expressions
   args <- getArgs  -- Get command line args as list
   bytes <- readFile (head args)
   putStr (longest bytes)
```

Execution:

```
% runghc longest /usr/share/dict/words
42702:electroencephalograph's
```

Larger example: **printCoords**

printCoords

Let's write a function to print one-based row and column coordinates for a grid with some number of rows and columns:

```
> printCoords 3 5
(1,1) (1,2) (1,3) (1,4) (1,5)
(2,1) (2,2) (2,3) (2,4) (2,5)
(3,1) (3,2) (3,3) (3,4) (3,5)
```

How can we decompose this into functions?

A function to make a string for a specific row
A function to make a string with all rows (and newlines)
A function to print that string with all rows

printCoords, continued

Handy:

```
> :t repeat
repeat :: a -> [a]

> take 5 (repeat 7)
[7,7,7,7,7]
```

Next step?

```
> zip (repeat 3) [1..5]
[(3,1),(3,2),(3,3),(3,4),(3,5)]
```

printCoords, continued

At hand:

```
> zip (repeat 3) [1..5]
[(3,1),(3,2),(3,3),(3,4),(3,5)]
```

Let's use that **zip** as the core of a **makeRow** function:

```
makeRow numCols row = line -- NOTE order of arguments!
    where
        tuples = zip (repeat row) [1..numCols]
        coordStrs = map show tuples
        line = unwords coordStrs -- instead of concat, to get
                                  -- blanks between coords.
```

Usage:

```
> makeRow 5 3
"(3,1) (3,2) (3,3) (3,4) (3,5)"
```

printCoords, continued

Let's use **makeRow** to write **makeGrid**:

```
makeGrid numRows numCols =
   map (makeRow numCols) [1..numRows]
```

Usage:

```
> makeGrid 3 5
["(1,1) (1,2) (1,3) (1,4) (1,5)","(2,1) (2,2) (2,3) (2,4) (2,5)",
"(3,1) (3,2) (3,3) (3,4) (3,5)"]
```

Key technique:

The partial application **makeRow numCols** supplies the number of columns. The **map** applies that partial application to each row number in turn.

Ultimately, **makeGrid 3 5** is equivalent to this:

```
[makeRow 5 1, makeRow 5 2, makeRow 5 3]
```

printCoords, continued

Everything, including **printCoords** itself: (in **printCoords.hs**)

```
makeRow numCols row = line -- NOTE order of arguments!
   where
      tuples = zip (repeat row) [1..numCols]
      coordStrs = map show tuples
      line = unwords coordStrs

makeGrid numRows numCols =
   map (makeRow numCols) [1..numRows]

printCoords numRows numCols =
   putStr (unlines (makeGrid numRows numCols))
```

Usage:

```
> printCoords 3 5
(1,1) (1,2) (1,3) (1,4) (1,5)
(2,1) (2,2) (2,3) (2,4) (2,5)
(3,1) (3,2) (3,3) (3,4) (3,5)
```

Composition

Function composition

Definition:

The *composition* of functions **f** and **g** is a function **c** such that (**c x**) equals **(f (g x))**

Here is a function that applies two functions in turn:

```
compose f g x = f (g x)
```

How many arguments does **compose** have?

Usage:

```
> compose init tail [1..5]
[2,3,4]

> compose (:[]) chr 42
"*"
```

Composition, continued

The Prelude binds the symbolic variable dot to a "compose" function:

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Dot is an operator whose operands are functions. Its result is a function.

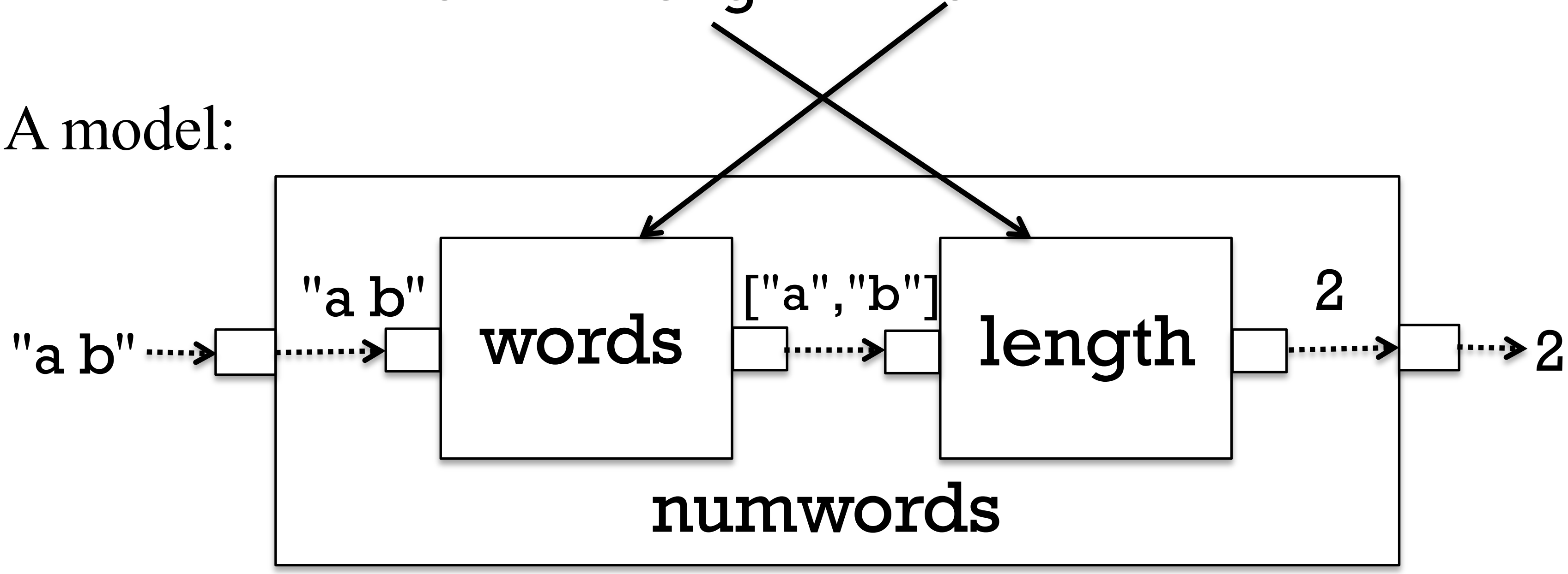
```
> numwords = length  .  words

> numwords
<function>

> numwords "just testing this"
3

> map numwords ["a test", "up & down", "done"]
[2,3,1]
```

Composition, continued

At hand:

```
numwords = length . words
```

A model:



Usage:

```
> numwords "a b"
2
```

Composition, continued

At hand:

```
numwords = length  .  words
```

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

```
> :t (.)
(.) :: (b -> c)   ->   (a -> b)   ->   a   ->   c

         [t] -> Int    String -> [String]
          (length)           (words)

> :t numwords
numwords :: String -> Int
```

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
```

Composition, continued

Problem: Using composition, create a function that reverses the words in a string:

```
> f "flip these words around"
"pilf eseht sdrow dnuora"
```

Hint: **unwords** is the inverse of **words**.

Solution:

```
f = unwords . (map reverse) . words
```

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:

```
> rmdigits = filter (not . isDigit)
```

Composition, continued

Recalling the following, what's the type of **f?**

```
head :: [a] -> a
length :: [a] -> Int
words :: String -> [String]
show :: Show a => a -> String

f = head . show . length . words
```

Simple rule:

If a composition is valid, the type of the resulting function is based only on the input of the rightmost function and the output of the leftmost function.

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

```
String -> Char
```

Composition, continued

Consider the following:

```
> 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"
```

Question: Is

```
map f (map g x)
```

always equivalent to the following?

```
map (f . g) x
```

If **f** and **g** did output, how would the output of the two cases differ?

FOR THE CURIOUS!

Point-free style

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*.

<u>A point-free binding of a function **f** has NO parameters!</u>

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

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:

```
> 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. They 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 at a future time.

Sidebar: Syntactic sugar

Peter Landin coined the term "syntactic sugar" in 1964.

"Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language 'sweeter' for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer."—Wikipedia, Feb '23

Some examples of syntactic sugar in Haskell:

- We can say `| otherwise = ...` instead of `| True = ...`
- We can say `[1,3..10]` instead of `enumFromThenTo 1 3 10`
- We can say `"A#1"` instead of `['A','#','1']`
- ...instead of `'A':'#':'1':[]`

"Syntactic sugar causes cancer of the semicolon." —Alan J. Perlis.

Sidebar, continued

Is Java's `for` an example of syntactic sugar?

`for (int i = 0; i < n; i++) ...`

Three examples of syntactic sugar in C:

- `"abc"` is equivalent to the address of a `char` array initialized with `{'a', 'b', 'c', '\0'}`
- `a[i]` is equivalent to `*(a + i)`
- `p->x` is equivalent to `(*p).x`

Try Googling for "syntactic sugar in Python".

DIY currying in Python

```
>>> def add(x): return lambda y: x + y

>>> add(3)(4)
7

>>> f = add(5)

>>> type(f)
<type 'function'>

>>> list(map(f,[10,20,30]))
[15, 25, 35]

>>> list(map(add("*"),"a new test".split()))
['*a', '*new', '*test']
```

Another mystery function

Here's another mystery function:

```
> 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?

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
```

Recall **take :: Int -> [a] -> [a]**

```
> flip take [1..10] 3
[1,2,3]

> ftake = flip take
> ftake [1..10] 3
[1,2,3]
```

flip, continued

From assignment 3:

```
> splits "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

Some students have already noticed the Prelude's **splitAt**:

```
> splitAt 2 [10,20,30,40]
([10,20],[30,40])
```

Problem: Write a non-recursive version of **splits**.

Solution:

```
splits list = map (flip splitAt list) [1..length (tail list)]
```

Spring '18 solution:

```
splits list = map (flip splitAt list) [1..(length list - 1)]
```

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",...infinitely...
```

Solution:

```
f x = map (flip replicate x) [1..]
```

The **$** operator

$ is the "application operator".

```
> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $      -- right associative infix operator with lowest
                -- possible precedence
```

The Prelude's source code for **$** uses an infix syntax:

```
f $ x  =  f x       -- Equivalent: ($) f x = f x
```

Usage:

```
> negate $ 3 + 4
-7
```

What's this operator good for?

The $ operator, continued

Because + has higher precedence than $, the expression

```
negate $ 3 + 4
```

groups like this:

```
negate $ (3 + 4)
```

Problem: Rewrite the following to take advantage of $:

```
filter (>3) (map length (words "up and down"))

filter (>3) $ map length $ words "up and down"
```

Common mistake: Confusing $ 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 map 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 `g` 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`! (And don't peek ahead!)

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**:

```
curry f x y = f (x,y)
```

Usage:

```
> 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`!

Currying the uncurried, continued

Let's get **flip** into the game!

```
> map (flip (curry g) 4) [1..5]
[45,60,77,96,117]
```

Note that

```
flip (curry g)
```

effectively turns

```
g (x,y) = x^2 + 3*x*y + 2*y^2
```

into

```
g y x    = x^2 + 3*x*y + 2*y^2
```

A **curry** for Python

This Python function returns a curried version of its argument:

```
def curry(f):
    return lambda x: lambda y: f(x,y)
```

Usage:

```
>>> c_print = curry(print)

>>> pt = c_print("Testing")  # partial application of print
>>> pt(7)
Testing 7

>>> pt("this")
Testing this

>>> r2 = curry(pow)(2)
>>> r2(10)
1024
```

uncurry

There's an **uncurry**, too. Here's one way to write it:

```
uncurry f (x,y) = f x y
```

Usage:

```
> uncurry add (3,4)
7

> :t uncurry replicate
uncurry replicate :: (Int, a) -> [a]

> map (uncurry replicate) $ zip [1..] "abcde"
["a","bb","ccc","dddd","eeeee"]
```

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 _ [] = error "emptyList"
reduce _ [x] = x
reduce op (x:xs) = x `op` reduce op xs
```

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   -- behaves like left associative: (1 / 2) / 3

> foldr1 (/) [1,2,3]             -- behaves like 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 foldl1 and foldl:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
```

What's different between them? (No peeking—eyes on the screen!)

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: $\underbrace{***}\underbrace{*}\underbrace{**}$"   -- 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 expression, using backquotes to infix f1:

```
> ((0  `f1`  "just")  `f1`  "a")  `f1`  "test"
3
```

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! You wrote another folding function!

Folding abstracts a common pattern of computation:

A series of values contribute one-by-one to an accumulating result.

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 '*'
```

We then call foldl with (1) the folding function, (2) an appropriate initial value, and (3) a list of values.

```
foldl f2 "stars: " [3,1,2]
```

foldl orchestrates the computation by making a series of calls to the folding function.

```
> (("stars: " `f2` 3) `f2` 1) `f2` 2
"stars: ******"
```

SUPER IMPORTANT: A folding function **NEVER** sees the list!

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 "emptyList"
```

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)
```

Given countEO [3,4,7,9], 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)
```

Problem: Finish the folding function

```
f (evens, odds) elem
    | even elem = (evens + 1, odds)
    | otherwise = (evens, odds + 1)
```

Problem: Write countEO as a foldl with f

```
countEO nums = foldl f (0,0) nums
```

Folds with anonymous functions

Anonymous functions are often used for folds.

Here are 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 the computation-specific initial accumulated 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
```

What are the 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 defined 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
```

Problem: Implement map as a fold

```
map f = foldr (\elem acm -> f elem : acm) []
```

Is folding One Operation to Implement Them All?

paired with a fold

Can **a3**'s **paired** be done with a fold?

```
> paired "((())())"
True
```

Sure!

```
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.

A progression of folds

Let's do a progression of folds related to finding vowels in a string.

First, let's count vowels in a string with a fold:

```
> foldr (\val acm ->
          acm + if  val `elem` "aeiou" then 1 else 0)  0  "ate"
2
```

anonymous folding function

Next, let's produce both a count and the vowels themselves:

Note: **s/val/letter/g**

```
> foldr (\letter acm@(n, vows) ->
          if  letter `elem` "aeiou" then (n+1, letter: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 "Now for some Prolog!"
[('o',1),('o',5),('o',9),('e',11),('o',15),('o',17)]
```

Solution:

```
vowelPositions s = reverse result
  where (result, _) =
    foldl (\acm@(vows, pos) letter ->
          if  letter `elem` "aeiou" then ((letter,pos):vows,pos+1)
                                   else (vows,pos+1))
          ([],0) s
```

The **foldl** produces a 2-tuple whose first element is the result, a list, but in reverse order.

This is another function that's a fold with a *wrapper*, like **paired**.

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: An initial accumulator value and a list of values

Output: <u>A value of any type and complexity</u>

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 foldings:

Refrigerators in Gould-Simpson to
((grams fat, grams protein, grams carbs), calories)

Keyboards in Gould-Simpson to
[("a", # of "a" keys), ("b", #), ..., ("$", #), ("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.

In conclusion...

If we had a whole semester...

If we had a whole semester to study functional programming, here's what might be next:

```
data Shape =
   Circle Double |
   Rect Double Double
     deriving Show
s = [Rect 3 4, Circle 2]
```

- Algebraic types
- Exploration of lazy/non-strict evaluation
- Infinite data structures, such as `x = 1: 2:x`
- Implications and benefits of referential transparency (which means that the value of a given expression is always the same).
- Monads (for representing sequential computations, including I/O)
- Functors (structures that can be mapped over)
- Monoids (a set of things with a binary operation over them)
- Zippers (a structure for traversing and updating another structure)
- And LOTS more!

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 tree and applies the function to each node.

C#
  Excellent support for functional programming with the language itself, and LINQ, too. There's F#, too!

Java
  Lambda expressions were added in Java 8, released in 2014.

OCaml
  "an industrial strength programming language supporting functional, imperative and object-oriented styles" – **OCaml.org**
  http://www.ffconsultancy.com/languages/ray_tracer/comparison.html