Functional Programming with Haskell
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*
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.
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
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.
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
Here's an imperative solution in Java to sum the integers in an array:

```java
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.
With Java's "enhanced for" we can avoid array indexing:

```java
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!

```c
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)
A recursive solution is far simpler in Python:

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

Here are some Java expressions:

\[ i + j * k \]
\[ \text{f(args.length * 2)} + n \]

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.
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]
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

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

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

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

```java
int n = 3;
```

# Java, given `int n = 3;

Value: 5
Type: `int`
Side effect: `n` changed to 5

```python
>>> print(1,2,3)  # Python
1 2 3
>>> print(print(1), print(2))
1
2
None None
```
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!
Which of these Java expressions have a side effect?

\[ x + 3 \times y \]

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

\[ x += 3 \times y \]

*Side effect: \(3 \times y\) is added to \(x\).*

\[ s.length() > 2 \lor s.charAt(1) == '#' \]

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

Code written in an imperative style is essentially an orchestration of side effects.

Recall:

```java
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
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
Here are three books I can recommend:

*Learn You a Haskell for Great Good!,* by Miran Lipovača

[http://learnyouahaskell.com](http://learnyouahaskell.com) (Known as LYAH.)


*Programming in Haskell, 2e* by Graham Hutton. (First edition [here](http:).)

*Real World Haskell*, by O'Sullivan, Stewart, and Goerzen


There's a pile of stuff at [haskell.org/documentation](http://haskell.org/documentation), but it's a big pile!

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

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

If you see something like this,

```plaintext
*** 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.
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 for me 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:
   ```haskell
   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:
   ```sh
   % turnin 372-eca1 eca1.txt
   ```

Due: At the start of the next lecture after the lecture in which I present this slide.
Let's see what we can learn about Haskell by trying some expressions:

- $3 + 4$
- $3 * 4.5$
- `it + it`
- `it /= 3`
- $3 > 4 \text{ || } 5 < 7$
- `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
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.
Function call with juxtaposition is left-associative.

\texttt{signum negate 2 means (signum negate) 2}

\begin{verbatim}
> signum negate 2
<interactive>:11:1: error:
  • Non type-variable argument ...
...
We add parentheses to call \texttt{n negate 2} first:
> signum (n negate 2)
-1
\end{verbatim}
Function call has higher precedence than any operator.

\[
\text{negate}\ 3 + 4 \quad \text{means} \quad (\text{negate}\ 3) + 4 .
\]

Use parens to force + first:

\[
\text{negate}\ (3 + 4) \quad -7
\]

\[
\text{signum}\ (\text{negate}\ (3 + 4)) \quad -1
\]
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.

```haskell
> 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:

```haskell
> :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` has type `Char` to `Bool`"

LHtLalL discernment: :type is part of ghci, not Haskell!
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
What are the types of the following Java methods?

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

```java
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:

```haskell
> 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.

For reference:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>Char</td>
</tr>
<tr>
<td>isUpper 'a'</td>
<td>Char</td>
</tr>
<tr>
<td>not (isUpper 'a')</td>
<td>Bool</td>
</tr>
<tr>
<td>not not (isUpper 'a')</td>
<td>Bool</td>
</tr>
<tr>
<td>toUpper (ord 97)</td>
<td>Char</td>
</tr>
<tr>
<td>isUpper (toUpper (chr 'a'))</td>
<td>Bool</td>
</tr>
<tr>
<td>isUpper (intToDigit 100)</td>
<td>Char</td>
</tr>
<tr>
<td>isUpper</td>
<td>Char</td>
</tr>
<tr>
<td>chr :: Int -&gt; Char</td>
<td>Int -&gt; Char</td>
</tr>
<tr>
<td>digitToInt :: Char -&gt; Int</td>
<td>Int -&gt; Char</td>
</tr>
<tr>
<td>intToDigit :: Int -&gt; Char</td>
<td>Char</td>
</tr>
<tr>
<td>isUpper :: Char -&gt; Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>not :: Bool -&gt; Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>ord :: Char -&gt; Int</td>
<td>Int</td>
</tr>
<tr>
<td>toUpper :: Char -&gt; Char</td>
<td>Char</td>
</tr>
</tbody>
</table>
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](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
Recall the `negate` function:

```haskell
> negate 5
-5

> negate 5.0
-5.0
```

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

```haskell
> :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**.
Here's the type of `negate`:

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

Given

\[
\text{negate :: Num } a \Rightarrow a \rightarrow a
\]

and

\[
3.4 :: \text{Fractional } a \Rightarrow a
\]

then

\[
\text{negate 3.4 is valid.}
\]

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.

```haskell
> :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`:

```haskell
> :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.)
The Prelude has a truncate to integer function:

```haskell
greaterThan 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.
In essence, `negate :: Num a => a -> a` describes many functions:
- `negate :: Integer -> Integer`
- `negate :: Int -> Int`
- `negate :: Float -> Float`
- `negate :: Double -> Double`
- \textit{...and more...}

`negate` is a \textit{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?
Consider this excerpt from \textbf{Bounded}:

\begin{verbatim}
> :info Bounded
class Bounded a where
  minBound :: a
  maxBound :: a
...
\end{verbatim}

What sort of things are \texttt{minBound} and \texttt{maxBound}?
Polymorphic values!

How can we use them?
The construct `::type` is an *expression type signature*.

A usage of it:

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

> maxBound :: Int
9223372036854775807

> maxBound :: Bool
True

> maxBound :: Integer
<interactive>:9:1: error:
  • No instance for (Bounded Integer)
We can use `:set +t` to direct `ghci` to automatically show types:

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

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

Some type-related tools in other languages:
- Python: `type(expr)` and `repr(expr)`
- JavaScript: `typeof(expr)`
- PHP: `var_dump(expr1, expr2, ...)`
- C: `sizeof(expr)`
- Java: `getClass();` /var in jshell.

What's a difference between ghci's :type and Java's `getClass()`?
Here's a Java program that makes use of the "boxing" mechanism to show the type of values, albeit with wrapper types for primitives.

```java
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!)
```
LHtLAL: Start accumulating a file of brief notes on Haskell.

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

```ghci
#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
#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:

```haskell
> 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.
Two more functions:

\[
\text{neg } x = -x \quad \text{neg} :: \text{Num } a \Rightarrow a \rightarrow a \quad (\text{:set } +t \text{ is in effect})
\]

\[
\text{toCelsius } \text{temp} = (\text{temp} - 32) \times \frac{5}{9} \\
\text{toCelsius} :: \text{Fractional } a \Rightarrow a \rightarrow a
\]

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

Problem: Write \textit{isPositive } x \text{ which returns True iff } x \text{ is positive.} Predict the function's type, too.

\[
\text{isPositive } x = x > 0 \\
isPositive :: (\text{Num } a, \text{Ord } a) \Rightarrow a \rightarrow \text{Bool}
\]
We can use `:: type` to constrain a function's type:

```haskell
> 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:

```haskell
> 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.
We can put function definitions in a file.

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

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

```haskell
% 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:

```bash
alias hs=ghci
```

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

```bash
% 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.

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

Lather, rinse, repeat.

If you don't see "Compiling", the file hasn't changed!
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.
The Prelude has a \texttt{min} function:
\begin{verbatim}
> min 6 2
2
\end{verbatim}

Problem: Define a function \texttt{min3} that computes the minimum of three values.
\begin{verbatim}
> min3 5 2 10
2
\end{verbatim}

Solution:
\[
\texttt{min3 a b c} = \texttt{min a (min b c)}
\]

Does \texttt{min3} exemplify functional programming?
Recall `add`:
\[
\text{add } x \ y = x + y :: \text{Int}
\]

Here is its type:
\[
> :: \text{type add}
\]
\[
\text{add} :: \text{Int} \to \text{Int} \to \text{Int}
\]

\text{Int} \to \text{Int} \to \text{Int} is a \textit{type expression}. It describes a type.

The \textit{operator} \to is \textit{right-associative}. Let's add parentheses:
\[
\text{Int} \to (\text{Int} \to \text{Int})
\]

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)`
Recall \texttt{min3}, but let's restrict it to \texttt{Int}s:
\[
> \texttt{min3 a b c = min a (min b c) :: Int}
\]

What's the type of \texttt{min3}?
\[
> :t \texttt{min3} \\
\texttt{min3 :: Int -> Int -> Int -> Int}
\]

How should the type expression be parenthesized to reflect associativity?
\[
\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))
\]

What does \texttt{min3 7 4 9} mean?
\[
> ((\texttt{min3 7}) 4) 9 \\
4
\]
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:

\[ > \text{add } x \ y = x + y :: \text{Int} \]
\[ \text{add :: Int} \to (\text{Int} \to \text{Int}) \quad -- \text{parens added} \]

\[ > \text{plusThree } = \text{add } 3 \]
\[ \text{plusThree :: Int} \to \text{Int} \]

Imagine \text{add} and \text{plusThree} as machines with inputs and outputs:

Weak analogy: \text{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>
Formula for displacement \( (s) \) of a falling object:
\[
s = \frac{1}{2}at^2 \quad (a \text{ is acceleration due to gravity, } t \text{ is time})
\]

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?!)

At hand: (in \texttt{gravity.hs})
\[
> \texttt{s a t} = 0.5 * a * t^2
\]

And...
\[
> \texttt{sEarth} = \texttt{s 32} \quad \# \texttt{sEarth} \text{ is a partial application} \\
  \quad \# 32 \text{ is "wired-in" for } a
\]
\[
> \texttt{sMoon} = \texttt{s 5.31} \\
> \texttt{sEarth 1} \\
  16.0 \\
> \texttt{sEarth 2} \\
  64.0 \\
> \texttt{sMoon 1} \\
  2.655 \\
> \texttt{sMoon 2} \\
  10.62
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

```haskell
> 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

\[ \text{wrap } c \; s = c \; ++ \; s \; ++ \; c \]

and the following binding, what does \( f \) look like?

\[ f = \text{wrap } "*" \]

Process:

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

\[ f = \text{wrap } c \; s = c \; ++ \; s \; ++ \; c \]

Remove =, name of FtPA (\text{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*"
Consider:

\[ \text{wrap } c \; s = c \; ++ \; s \; ++ \; c \]
\[
\text{wrap} \; :: \; [a] \rightarrow [a] \rightarrow [a]
\]
\[ \text{min3 } x \; y \; z = \text{min } x \; (\text{min } y \; z) \]
\[
\text{min3} \; :: \; \text{Ord } a \rightarrow a \rightarrow a \rightarrow a
\]

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):
  \[ \text{name} \ param_1 \ param_2 \ldots \ param_N = \text{expression} \]

- A function with a type like \text{Int} -\rightarrow \text{Char} -\rightarrow \text{Char} takes two arguments, an \text{Int} and a \text{Char}. It produces a \text{Char}.

- Remember that -\rightarrow is a right-associative type operator.
  \text{Int} -\rightarrow \text{Char} -\rightarrow \text{Char} means \text{Int} -\rightarrow (\text{Char} -\rightarrow \text{Char})

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

```python
>>> 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

```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>
```
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)
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...
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.

\texttt{infixl 6 +} indicates that the token + can be used as a infix operator that is left associative and has precedence level 6.

Use \texttt{:info} to explore these operators: ==, >, +, *, |, |, ^, ^\,^, and **.
To use an operator like a function, enclose it in parentheses:
> (+) 3 4
 7

Conversely, we can use a function like an operator by enclosing it in backquotes:
> 3 `add` 4
 7

> 11 `rem` 3
 2

Explore: Do add and rem have precedence and associativity?
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

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Left associative operators</th>
<th>Non associative operators</th>
<th>Right associative operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td><code>!!</code></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td><code>^, ^^, **</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>*, /, </code>div<code>, </code>mod<code>, </code>rem<code>, </code>quot`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>+,-</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td><code>::, ++</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>==, /=, &lt;, &lt;=, &gt;, &gt;=, </code>elem<code>, </code>notElem`</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td><code>&amp;&amp;</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><code>&gt;&gt;, &gt;&gt;=</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td><code>$, $!, </code>seq``</td>
<td></td>
</tr>
</tbody>
</table>

Note: From page 51 in H10
Type Inferencing
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 \texttt{add} again:

\begin{verbatim}
> add x y = x + y
> :t add
add :: Num a => a -> a -> a
> :info +
class Num a where
  (+) :: a -> a -> a
...
\end{verbatim}

What did Haskell infer?

- Both arguments must have same type.
- That type must be an instance of the \texttt{Num} class.
- A value of that same type is returned.
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:

```haskell
> 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
Recall `ord` in the `Data.Char` module:

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

What type will be inferred for `f` below?

```haskell
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:

```haskell
> f x y = ord x == y
f :: Char -> Int -> Bool
```
Recall this example:

```haskell
> isPositive x = x > 0

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

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

```haskell
> :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.
If a contradiction is reached during type inferencing, it's an error.

The function below compares \( x \) to both a \texttt{Num} and a \texttt{Char}.

\[
> g \ x \ y = x > 0 && x > '0' \\
<\text{interactive}>:1:13: \text{error:} \\
\quad \begin{aligned}
\quad & \text{• No instance for (Num Char) arising from the literal ‘0’} \\
\quad & \text{• In the second argument of ‘(>)’, namely ‘0’} \\
\quad & \quad \text{In the first argument of ‘(&&)’, namely ‘x > 0’} \\
\quad & \quad \text{In the expression: x > 0 && x > '0'}
\end{aligned}
\]

What does the error "No instance for (Num Char)" mean? \texttt{Char} is not an instance of the \texttt{Num} type class.

(:\text{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
```
Sometimes type specifications can backfire.

What's a ramification of the difference between the types of \texttt{add1} and \texttt{add2}?

\begin{verbatim}
add1::Num a => a -> a -> a
add1 x y = x + y

add2::Integer -> Integer -> Integer
add2 x y = x + y
\end{verbatim}

\texttt{add1} can operate on \texttt{Nums} but \texttt{add2} requires \texttt{Integers}.

Challenge: Without using ::\texttt{type}, show an expression that works with \texttt{add1} but fails with \texttt{add2}. 
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
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
```
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 -&gt; Integer -&gt; 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
Recall this characteristic of mathematical functions: "Are often specified with cases and expressions."

This function definition uses guards to specify three cases:

\[
\text{sign } x \mid x < 0 = -1 \\
\mid x == 0 = 0 \\
\mid \text{otherwise} = 1
\]

Notes:
- This definition would be found in a file, not typed in ghci.
- \text{sign } x \text{ appears just once. } First guard might be on next line.
- The guards appear \text{between} \mid and =, and produce \text{Bools}.
- What is \text{otherwise}?
Problem: Using guards, define a function `smaller`, like `min`:

```haskell
> smaller 7 10
7

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

Solution:

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

```haskell
weather temp | temp >= 80 = "Hot!"
| temp >= 70 = "Nice"
| otherwise = "Cold!"
```
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?
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]
```
"A statement changes the state of the program while an expression wants to express itself."

— Victor Nguyen, CSC 372, Spring 2014
Which of the versions of `sign` below is better?

\[
\text{sign } x \\
| \ x < 0 = -1 \\
| \ x == 0 = 0 \\
| \ \text{otherwise} = 1
\]

\[
\text{sign } x = \text{if } x < 0 \text{ then } -1 \text{ else if } x == 0 \text{ then } 0 \text{ else } 1
\]

- We'll later see that patterns add a third possibility for expressing cases.
- For now, prefer guards over if-else.
A Little Recursion
A recursive function is a function that calls itself either directly or indirectly.

Computing the factorial of an integer (N!) is a classic example of recursion.

```
> factorial 40
8159152832478977343456112695961158942720000000000
```

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

\[
\text{factorial } n \\
\begin{cases}
    n == 0 & \Rightarrow 1 \\
    \text{otherwise} & \Rightarrow n \times \text{factorial} (n - 1)
\end{cases}
\]
Lists
In Haskell, a list is a sequence of values of the same type.

Here's one way to make a list.

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

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

We can say that Haskell lists are *homogeneous*. 
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.
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!
The ++ operator concatenates two lists, producing a new list.

\[
\text{> } [3,4] ++ [10,20,30] \\
[3,4,10,20,30]
\]

\[
\text{> it ++ reverse(it)} \\
[3,4,10,20,30,30,20,10,4,3]
\]

What are the types of ++ and reverse?

\[
\text{> :type (++)} \\
(++) :: \text{[a] -> [a] -> [a]}
\]

\[
\text{> :type reverse} \\
\text{reverse :: [a] -> [a]}
\]
Haskell has an arithmetic sequence notation:

```haskell
> [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]
[]
```
Here are `sum` and `product`:

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

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

Problem: Write a factorial function.

Solution:

```
factorial n = product [1..n]
```
Here are *drop* and *take*:

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

```haskell
> take 5 [1.0,1.2..2]
[1.0,1.2,1.4,1.5999999999999999,1.7999999999999998]
```
Problem: halves

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

\[
> \texttt{halves \ [1..10]} \\
\texttt{[[1,2,3,4,5],[6,7,8,9,10]]}
\]

\[
> \texttt{halves \ [1]} \\
\texttt{[[],\[1\]]}
\]

\texttt{halves} will be a little repetitious because we don't have the \texttt{where clause} in our toolbox yet.
Solution: (halves.hs)

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

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!
Haskell lists are **values** and can be compared as values:

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

True

> tail (tail [3,4,5,6]) == [last [4,5]] ++ [6]
True
```
Lists are compared \textit{lexicographically}:

- Corresponding elements are compared until an inequality is found.
- The inequality determines the result of the comparison.
- If $L_1$ is a proper prefix of $L_2$, then $L_1 < L_2$.
- (Same as Python...)

Example:

\begin{itemize}
  \item $> [1,2,3] < [1,2,4]$  
    True  
  \item $> [1,2,3] > [1,2]$  
    True
\end{itemize}
We can make lists of lists.

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

```haskell
x :: Num a => [[a]]
```

Note the type: \( x \) is a list of \( \text{Num} \ a \Rightarrow [a] \) lists.

What's the length of \( x \)?

```haskell
> length x
```

```haskell
3
```

Wait! Is \( x \) homogeneous?
More examples:

```haskell
> 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
```
Earlier I showed you this:

\[
\text{length} :: [a] \rightarrow \text{Int}
\]

Around version 7.10 \textbf{length} was generalized to this:

\[
\text{length} :: \text{Foldable } t \Rightarrow t\ a \rightarrow \text{Int}
\]

We're going to think of \textbf{Foldable } t \Rightarrow t\ a \text{ as meaning } [a].

Instead of \textbf{sum} :: (\text{Num } a, \text{Foldable } t) \Rightarrow t\ a \rightarrow a
Pretend this \textbf{sum} :: \text{Num } a \Rightarrow [a] \rightarrow a

Instead of \textbf{minimum} :: (\text{Ord } a, \text{Foldable } t) \Rightarrow t\ a \rightarrow a
Pretend this \textbf{minimum} :: \text{Ord } a \Rightarrow [a] \rightarrow a
Strings in Haskell are simply lists of characters.

```haskell
gtasteing"ntesting"
ity :: [Char]

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

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

What's the beauty of this?
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
The Prelude defines **String** as \([\text{Char}]\) (a *type synonym*).

\[
\text{> :info String} \\
\text{type String = [Char]}
\]

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

\[
\text{> :type words} \\
\text{words :: String -> [String]}
\]

\[
\text{> :type unwords} \\
\text{unwords :: [String] -> String}
\]

What's the following doing?

\[
\text{> unwords (tail (words "Just some words!")))} \\
\text{"some words!"}
\]
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]
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]
What are the values of the following expressions?

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

> 1:2
...error...

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

cons is right associative
chr 97:(chr 98:(chr 99:[]))

> []:[]
[[]]

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

> []:[1]
...error...
It's important to understand that `tail` does not create a new list. Instead it simply returns an existing cons node.

```haskell
> a = [5,10,20,30]
> h = head a
> h
5
> t = tail a
> t
[10,20,30]
> t2 = tail (tail t)
> t2
[30]
```
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
    
    $\text{[1,2,3]} \text{ ++ } \text{[4,5]}$ turns into $1 : 2 : 3 : [4,5]$
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!
\texttt{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.
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.
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.
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)
```
Here's a simple example of a list comprehension:

```haskell
> [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
What can you tell me about `show`?

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

`show` produces a string representation of a value.

```haskell
> show 10
"10"
```

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

```haskell
> show show
"<function>"
```

Important: `show` does not produce output!

What's the Python analog for `show`?

Is there a Java analog for `show`?
The `putStr` function outputs a string:

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

Type:

```haskell
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`
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:

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

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

First, write a helper, `printN'`:

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

Solution for `printN'`:

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

At hand:

\[
\text{printN'} \ n \\
\quad | \ n == 0 = ""
\quad | \ \text{otherwise} = \text{printN'} (n-1) + \text{show} \ n + "\n"
\]

Usage:

\[
> \text{printN'} \ 10 \\
"1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n"
\]

Let's now write the top-level function:

\[
> \text{printN} \ n = \text{putStrLn} \ (\text{printN'} \ n) \\
> :t \text{printN} \\
\text{printN} :: (\text{Eq} \ a, \text{Num} \ a, \text{Show} \ a) \Rightarrow \ a \to \text{IO} ()
\]
All together in a file:

```haskell
% cat printN.hs
printN n = putStrLn (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
```
Let's write `charbox`:

```haskell
> charbox 5 3 '
```

```
*****
*****
*****
```

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

How can we approach it?
Let's work out a sequence of computations with ghci:

```haskell
> 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"

> putStrLn it
*****
*****
Let's write `charbox'`:

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

Test:

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

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

```haskell
charbox::Int -> Int -> Char -> IO ()
charbox w h c = putStrLn (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
Imagine a function that computes the sum of a list's elements.

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

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

Implementation:

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

- It works but it's not idiomatic Haskell.
- We should use *patterns* instead!
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]  -- Why in a list?
[30,20,10]
```

```
> d
[]  -- Why did I do [c,b,a] instead of [d,c,b,a]?
```
If some part of a structure is not of interest, we indicate that with an underscore, known as the \textit{wildcard pattern}.

\begin{verbatim}
> (_, [b]) : c = [[1], [2, 3], [4]]
> a
2
> b
3
> c
[[4]]
\end{verbatim}

No binding is done for the wildcard pattern.

The pattern mechanism is completely general—patterns can be arbitrarily complex.
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 : []
Describe in English what must be on the right hand side for a successful match.

\[ a:b:c = \ldots \]
A list containing at least two elements.
Does \([10,20]\) match?
\([20,30]\) ?
"abc" ?

\[ x:xs = \ldots \]
A list whose only element is a non-empty list.
Does \textbf{words} "a test" match?
\([\textbf{words} "a test"]\) ?
\([[]]\) ?
\([[[[]]]]\) ?
Patterns in function definitions

Recall our non-idiomatic `sumElems`:

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

Idiomatic:

```haskell
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?

```haskell
(Eq a, Num a) => [a] -> a -- with head/tail
Num a => [a] -> a -- with pattern
```
Here's a buggy version of \texttt{sumElems}:

\begin{verbatim}
buggySum [x] = x
buggySum (h:t) = h + buggySum t
\end{verbatim}

What's the bug?

\begin{verbatim}
> buggySum [1..100]
5050
> buggySum []
*** Exception: Non-exhaustive patterns in function buggySum
\end{verbatim}
At hand:

\[
\begin{align*}
\text{buggySum} [x] &= x \\
\text{buggySum} (h:t) &= h + \text{buggySum} t
\end{align*}
\]

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

\[
% \texttt{ghci -fwarn-incomplete-patterns buggySum.hs}
\]

\[
\texttt{buggySum.hs:1:1: Warning:}
\]

\[
\text{Pattern match(es) are non-exhaustive}
\]

\[
\text{In an equation for ‘buggySum’: Patterns not matched: []}
\]

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

\[
\text{alias ghci="ghci -fwarn-incomplete-patterns"}
\]
What's a little silly about the following list-summing function?

\[
\text{sillySum} \; [] = 0 \\
\text{sillySum} \; [x] = x \\
\text{sillySum} \; (h:t) = h + \text{sillySum} \; t
\]

The second clause isn't needed.
Consider a function that duplicates the head of a list:

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

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

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

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

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

Can it be improved?

```haskell
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!
  \[
  \text{sumElems} \; [] = 0 \\
  \text{sumElems} \; (h:t) = h + \text{sumElems} \; t
  \]

Guards—second choice...
  \[
  \text{sumElems} \; \text{list} \\
  \quad \mid \text{list} = [] \Rightarrow 0 \\
  \quad \mid \text{otherwise} = \text{head list} + \text{sumElems} \; (\text{tail list})
  \]

if-else—third choice...
  \[
  \text{sumElems} \; \text{list} = \\
  \quad \text{if list} = [] \; \text{then } 0 \\
  \quad \text{else head list} + \text{sumElems} \; (\text{tail list})
  \]

And, these comparisons imply that \text{list}'s type must be an \textbf{Eq}!
"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
Recall this example of guards:

```haskell
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.
At hand, a more general form for functions:

\[
\text{function-name pattern}_1 \ \text{pattern}_2 \ \ldots \ \text{pattern}_N \\
\{ \ | \ \text{guard-expression}_1 \ \} = \ \text{result-expression}_1 \\
\ldots \\
\{ \ | \ \text{guard-expression}_N = \ \text{result-expression}_N \ \} \\
\]

How does \( \text{add} \ x \ y = x + y \) conform to the above specification?

- \( x \) and \( y \) are trivial patterns
- \text{add} has one clause, which has no guard
If the patterns of a clause match but all guards fail, the next clause is tried. Here's a contrived example:

\[
\begin{align*}
  f (h:_) \mid h < 0 &= "negative head" \\
  f \text{ list} \mid \text{length list} > 3 &= "too long" \\
  f (_,_) &= "ok" \\
  f [] &= "empty"
\end{align*}
\]

Usage:

\[
\begin{align*}
  > f [-1,2,3] \\
  "negative head"
\end{align*}
\]

\[
\begin{align*}
  > f [] \\
  "empty"
\end{align*}
\]

\[
\begin{align*}
  > f [1..10] \\
  "too long"
\end{align*}
\]

How many clauses does \( f \) have?

4

What if 2\textsuperscript{nd} and 3\textsuperscript{rd} clauses swapped?

3\textsuperscript{rd} clause would never be matched!

What if 4\textsuperscript{th} 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 \texttt{len x}, which returns the length of list \texttt{x}.

\begin{verbatim}
> len []
0

> len "testing"
7
\end{verbatim}

Solution:

\begin{verbatim}
len [] = 0
len (_:t) = 1 + len t  -- since head isn't needed, use _
\end{verbatim}
Problem: Write \texttt{odds x}, which returns a list having only the odd numbers from the list \texttt{x}.

\begin{verbatim}
> odds [1..10]
[1,3,5,7,9]

> take 10 (odds [1,4..100])
[1,7,13,19,25,31,37,43,49,55]
\end{verbatim}

Handy: \texttt{odd :: Integral a \Rightarrow a \rightarrow Bool}

Solution:
\begin{verbatim}
odds [] = []

odds (h:t)
  | odd h = h:odds t
  | otherwise = odds t
\end{verbatim}
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
```
Problem: Write a function that returns a list's maximum value.

```haskell
> 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:

```haskell
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.

```haskell
> (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.
A function can return a tuple:

\[
\text{pair } \mathbf{x} \ \mathbf{y} = (\mathbf{x}, \mathbf{y})
\]

What's the type of \text{pair}?

\[
\text{pair} :: \mathbf{a} \rightarrow \mathbf{b} \rightarrow (\mathbf{a}, \mathbf{b})
\]

Usage:

\[
> \text{pair } 3 \ 4 \\
(3,4)
\]

\[
> \text{pair } (3,4) \\
<\text{function}>
\]

\[
> \text{it } 5 \\
((3,4),5)
\]
The Prelude has two functions that operate on 2-tuples.

```haskell
> p = pair 30 "forty"

> p
(30, "forty")

> fst p
30

> snd p
"forty"
```
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 \texttt{middle}, to extract a 3-tuple's second element.

\begin{verbatim}
> middle ("372", "GS 906", "Mitchell")
"GS 906"
\end{verbatim}

\begin{verbatim}
> middle (1, [2], True)
[2]
\end{verbatim}

(Solution on next slide. Don't peek! This means \textbf{you}!)
At hand:

\[
\text{middle} \ (1, \ [2], \ \text{True}) \\
\text{[2]}
\]

Solution:

\[
\text{middle} \ (_{\text{m, parental}}) = \text{m}
\]

What's the type of `middle`?

```
\text{middle} :: (a, b, c) \rightarrow b
```

Will the following call work?

```
\text{middle}(1, [(2,3)], 4) \\
[(2,3)]
```
Problem: Write a function \texttt{swap} that behaves like this:

\begin{verbatim}
> swap ('a',False)
(False,'a')

> swap (1,(2,3))
((2,3),1)
\end{verbatim}

Solution:

\begin{verbatim}
> swap (x,y) = (y,x)
\end{verbatim}

What is the type of \texttt{swap}?

\texttt{swap} :: (b, a) -> (a, b)
Here's the type of `zip` from the Prelude:

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

Speculate: What does `zip` do?

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

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

What's especially interesting about the second example?

[1..] is an infinite list! `zip` stops when either list runs out.
Problem: Write `elemPos`, which returns the zero-based position of a value in a list, or `-1` if not found.

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

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

Solution:

```haskell
(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 guarantees 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:

\[
> \text{add\_c } x \ y = x + y \quad -- \text{c for curried arguments}
\]

\[
> \text{add\_c} :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a
\]

\[
> \text{add\_t } (x,y) = x + y \quad -- \text{t for tuple argument}
\]

\[
> \text{add\_t} :: \text{Num } a \Rightarrow (a, a) \rightarrow a
\]

Usage:

\[
> \text{add\_c } 3 \ 4
7
\]

\[
> \text{add\_t } (3,4)
7
\]

Important: Note the difference in types!

Which is better, \text{add\_c} or \text{add\_t}?
The *where* clause
Intermediate values and/or helper functions can be defined using an optional \textit{where} clause for a function.

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

\begin{verbatim}
  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;
  }
\end{verbatim}

The \textit{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, \texttt{a}, \texttt{b}, \texttt{c} and \texttt{g} are not visible outside the function \texttt{f}.
**A Computer Science Tapestry** by Owen Astrachan shows an interesting way to raise a number to a power:

```haskell
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.
Recall:

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

```haskell
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:

```haskell
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:

```haskell
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:

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

Look Mom, no braces! (No semicolons, either.)
At hand:
\[
f \ x = a + b + g \ a \\
\text{where}
\]
\[
| a = 1 \\
| b = 2 \\
| g \ x = \\
| -x
\]

Another example:
\[
f \ x = a + b + g \ a \text{ where } a = 1 \\
\text{where}
\]
\[
| b = 2 \\
| g \ x = \\
| -x
\]

The absence of a brace after \texttt{where} activates the layout rule.

The column position of the first token after \texttt{where} establishes the column in which declarations in the \texttt{where} must start.

Note that the declaration of \texttt{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
Literal values can be part or all of a pattern. Here's a 3-clause binding for $f$:

$$
\begin{align*}
  f\ 1 &= 10 \\
  f\ 2 &= 20 \\
  f\ n &= n
\end{align*}
$$

Usage:

```
> f 1
10
```

```
> f 3
3
```

Remember: Patterns are tried in the order specified.

For contrast, with guards:

```hs
f\ n
  | n == 1 = 10
  | n == 2 = 20
  | otherwise = n
```
Here's a function that classifies characters as parentheses (or not):

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

Could we improve it by using patterns instead of guards?

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

Which is better?

Remember: Patterns, then guards, then `if-else`. 
not is a function:

\[
\begin{align*}
> &: \text{type } \text{not} \\
\text{not} &:: \text{Bool} -> \text{Bool} \\
\\
> \text{not True} & \text{False} \\
\text{False} \\
\end{align*}
\]

Problem: Using literals in patterns, define not.

Solution:

\[
\begin{align*}
\text{not True} &= \text{False} \\
\text{not } _ &= \text{True} \quad \text{-- Using wildcard avoids comparison}
\end{align*}
\]
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
What syntax errors do you see in the following file?

```haskell
% 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
```
What syntax errors do you see in the following file?

```haskell
% 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.
- % cat synerrors.hs
- no = before guards
- =, not == before result
- use /= for inequality
- missing | before otherwise
- continuation should be indented
- violates layout rule
- Needs parens: (x:xs)
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.

```haskell
> :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:

```haskell
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 function types popping up in unexpected places.
Here's an example of omitting an operator:

```haskell
> 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:

```haskell
> :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.
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:**
\[ x = \text{head} \ x \]

**Occurs check:** cannot construct the infinite type: \( a \sim [a] \)
- Expected type: \([a]\)
- Actual type: \([[a]]\)

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

In the expression: \( x : f \ xs \)
In an equation for 'f': \( f \ (x : xs) = x : f \ xs \)

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

**Technique:** Comment out clauses (and/or guards) to find the troublemaker, or incompatibilities between them.
Recall \texttt{ord :: Char -> Int}.

Note this error:
\begin{verbatim}
> ord 5
No instance for (Num Char) arising from the literal `5'
\end{verbatim}

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

\begin{verbatim}
> :info Num
....
instance Num Word
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
\end{verbatim}

\texttt{instance Num Char} doesn't appear
> (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
How can the following Python errors be produced?

**KeyError: 3**

```python
>>> d={}; d[3]
>>> {}[3]
```

**TypeError: 'int' object is not callable**

```python
>>> f=3; f(4)
>>> 3(4)
>>> range(1,10).start()
```

If you see the following, what does it mean?

```python
<built-in method join of str object at 0x10457dc70>
>>> ",".join
```
Debugging
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.
Here's a trivial function:
\[
\begin{align*}
f 1 &= 10 \\
f n &= n \times 5 + 7
\end{align*}
\]

Let's augment it with tracing:
\[
\begin{align*}
\text{import Debug.Trace} \\
f 1 &= \text{trace "f: first case" 10} \\
f n &= \text{trace "f: default case" n \times 5 + 7}
\end{align*}
\]

Execution:
\[
\begin{align*}
> f 1 \\
f: \text{first case} \\
10
\end{align*}
\]

\[
\begin{align*}
> f 3 \\
f: \text{default case} \\
22
\end{align*}
\]
Let's add `trace` calls to `sumElems`:

```haskell
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".
Code for `buildingAtHeight` in `street.hs`, an old 372 problem:

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

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

```haskell
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:

```haskell
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'`
Icon has a built-in tracing mechanism.

Here's `sumElems` in Icon:

```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
```
Execution:

```plaintext
% 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 does have some debugging support but debugging is *expression-based*. Here's some simple interaction with it on `countEO`:

```haskell
> :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]
```

```haskell
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
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:

```haskell
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!
```
At hand:

\[
\begin{align*}
\text{countEO} & \quad [] = (0,0) \\
\text{countEO} & \quad (x:xs) \\
& \quad | \quad \text{odd } x = (\text{evens}, \text{odds} + 1) \\
& \quad | \quad \text{otherwise} = (1+ \text{evens}, \text{odds}) \\
\text{where} & \quad (\text{evens}, \text{odds}) = \text{countEO} \; \text{xs}
\end{align*}
\]

Here's one way to picture this recursion:

- \(\text{countEO} \; [10,20,25] \) returns \((2,1)\) (result of \((1 + 1,1))\)
- \(\text{countEO} \; [20,25] \) returns \((1,1)\) (result of \((1 + 0,1))\)
- \(\text{countEO} \; [25] \) returns \((0,1)\) (result of \((0,0 + 1))\)
- \(\text{countEO} \; [] \) returns \((0,0)\)
Here's `countEO` with tracing:

```haskell
import Debug.Trace

countEO [] = (0,0)

countEO list@(x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)

where

result = countEO xs
(newEvens, newOdds) =
    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)
```
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.
Usage:

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

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

> travel "nnessw"
"Got home"

How can we approach this problem?
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)
First, let's write a helper function to turn a direction into an \((x,y)\) displacement:

\[
\text{mapMove} :: \text{Char} \rightarrow (\text{Int}, \text{Int})
\]

\[
\begin{align*}
\text{mapMove 'n'} &= (0,1) \\
\text{mapMove 's'} &= (0,-1) \\
\text{mapMove 'e'} &= (1,0) \\
\text{mapMove 'w'} &= (-1,0) \\
\text{mapMove c} &= \text{error ("Unknown direction: " ++ [c])}
\end{align*}
\]

Usage:

\[
\begin{align*}
> \text{mapMove 'n'} \\
& (0,1) \\
> \text{mapMove 'w'} \\
& (-1,0)
\end{align*}
\]
Next, a function to sum \( x \) and \( y \) displacements in a list of tuples:

\[
\begin{align*}
\text{> sumTuples } &\quad [(0,1),(1,0)] \\
&\quad (1,1)
\end{align*}
\]

\[
\begin{align*}
\text{> sumTuples } &\quad [\text{mapMove } 'n', \text{mapMove } 'w'] \\
&\quad (-1,1)
\end{align*}
\]

Implementation:

\[
\begin{align*}
\text{sumTuples} &\quad :: [(\text{Int},\text{Int})] \rightarrow (\text{Int},\text{Int}) \\
\text{sumTuples} &\quad [] = (0,0) \\
\text{sumTuples} &\quad ((x,y):ts) = (x + \text{sumX}, y + \text{sumY}) \\
\text{where} &\quad (\text{sumX}, \text{sumY}) = \text{sumTuples} \ ts
\end{align*}
\]
travel itself, with makeTuples in a where

```haskell
tavel :: [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?
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

Sidebar: top-level vs. hidden functions

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.
Consider a function `tally` that counts character occurrences in a string:

```haskell
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)]`
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`.

\[
\text{incEntry} :: \text{Char} \to [(\text{Char}, \text{Int})] \to [(\text{Char}, \text{Int})]
\]

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

> incEntry 't' []

\[
[(\text{'t'}, 1)]
\]

> incEntry 'o' it

\[
[(\text{'t'}, 1), (\text{'o'}, 1)]
\]

> incEntry 'o' it

\[
[(\text{'t'}, 1), (\text{'o'}, 2)]
\]
incEntry c tups

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

```
[('b',1),('a',2)]
```

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

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

```haskell
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 \texttt{mkentries} \texttt{s}. It calls \texttt{incEntry} for each character in the string \texttt{s} in turn and produces a list of \textit{(char, count)} tuples.

\texttt{mkentries} :: \texttt{[Char]} -> \texttt{[(Char, Int)]}

\textbf{Usage:}

\begin{verbatim}
> mkentries "tupple"
[('t',1),('u',1),('p',2),('l',1),('e',1)]

> mkentries "cocoon"
[('c',2),('o',3),('n',1)]
\end{verbatim}

\textbf{Code:}

\begin{verbatim}
mkentries :: [Char] -> [(Char, Int)]

mkentries s = mkentries' s []
    where
    mkentries' [ ] entries = entries
    mkentries' (c:cs) entries =
        mkentries' cs (incEntry c entries)
\end{verbatim}
{- 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)]
```haskell
{- fmtEntries prints (char,count) tuples one per line -}
fmtEntries [] = ""
fmtEntries ((c, count):es) = 
    [c] ++ " " ++ show count ++ "\n" ++ fmtEntries es

{- top-level function -}
tally s = putStrLn (fmtEntries (sort (mkentries s)))
```

> tally "cocoon"

```
o 3
c 2
n 1
```

- How does this solution exemplify functional programming?
Let's run it on lectura...

\[
% \text{code}=/cs/www/classes/cs372/spring23/haskell
\]

\[
% \text{cat $\text{code/tally.hs}}
\]

... everything we've seen before and now a main:
\[
\text{main} = \text{do}
\]
\[
\quad \text{bytes} <- \text{getContents} -- \text{reads all of standard input}
\]
\[
\quad \text{tally bytes}
\]

\[
% \text{echo -n cocoon | runghc $\text{code/tally.hs}}
\]

\[
\text{o 3}
\]
\[
\text{c 2}
\]
\[
\text{n 1}
\]
tally from the command line, continued

$\texttt{code/genchars } N \texttt{ generates } N \texttt{ random letters:}$

\%
\texttt{code/genchars 20}
KVQaVPEmClHRbgdkmMsQ

Lets tally a million letters:

\%
\texttt{code/genchars 1000000 } \texttt{|}
\texttt{time runghc code/tally.hs >out}
\texttt{21.79user 0.24system 0:22.06elapsed}
\%
\texttt{head -3 out}
\texttt{s 19553}
\texttt{V 19448}
\texttt{J 19437}
Let's try a compiled executable.

% cd $code  
% ghc --make -rtsopts tally.hs  
% ls -l tally  
-rw-rw-r--  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
Here are user CPU times for implementations of \texttt{tally} in several languages. The same ten million letter file was used for all timings.

<table>
<thead>
<tr>
<th>Language</th>
<th>Time in seconds; mean of two or more runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>57.284</td>
</tr>
<tr>
<td>Ruby</td>
<td>18.589 (v2.7.0; much slower than 2.2.4 (or 1.9.3?))</td>
</tr>
<tr>
<td>Icon</td>
<td>8.248</td>
</tr>
<tr>
<td>Python 3</td>
<td>1.131</td>
</tr>
<tr>
<td>Python 2</td>
<td>0.824</td>
</tr>
<tr>
<td>C w/ gcc -O3</td>
<td>0.031 (2.97 for one \texttt{billion} letters)</td>
</tr>
</tbody>
</table>

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

Then I revisited the C version (\texttt{tally2.c}) and processed one \texttt{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 we answer that question?
   Do it on paper?
      No!
   Google for a course planning app?
      No!
   Let's write a Haskell program!
      Cool!
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?
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!
There's a `Data.Time.Calendar` module but writing two minimal date handling functions provides good practice.

```haskell
> toOrdinal (12,31)
365    -- 12/31 is the last day of the year

> fromOrdinal 32
(2,1)   -- The 32\textsuperscript{nd} 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 31\textsuperscript{st} day of the year
(7,212) The last day in July is the 212\textsuperscript{th} day of the year
```
toOrdinal and fromOrdinal

offsets =
[(0,0),(1,31),(2,59),(3,90),(4,120),(5,151),(6,181),(7,212),(8,243),(9,273),(10,304),(11,334),(12,365)]

> toOrdinal (12,31)
365

> fromOrdinal 32
(2,1)

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

fromOrdinal ordDay =
  fromOrdinal' (reverse offsets) ordDay
where
  fromOrdinal' ((month,lastDay):t) ordDay
    | ordDay > lastDay = (month + 1, ordDay - lastDay)
    | otherwise = fromOrdinal' t ordDay
fromOrdinal' [] _ = error "invalid month?"
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 ...

CSC 372 Spring 2023, Haskell Slide 273
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: \textit{showLecture 1 15 'H'}
We want: "#1 H 1/15"

Let's write \textit{showOrdinal :: Integer -> [Char]}

\texttt{> showOrdinal 15}
"1/15"

\texttt{showOrdinal ordDay = show month ++ "/" ++ show day}
where
\texttt{(month,day) = fromOrdinal ordDay}

Now we can write \textit{showLecture}:
\texttt{showLecture lecNum ordDay dayOfWeek =}
"#" ++ show lecNum ++ " " ++ [dayOfWeek] ++ " " ++ showOrdinal ordDay ++ "\n"
Recall:

\[
\begin{align*}
&\text{showLecture } 1 \ 15 \ 'H' \\
&\text{showLecture } 2 \ 20 \ 'T' \\
&\ldots \\
&\text{showLecture } 32 \ 125 \ 'T'
\end{align*}
\]

Let's "cons up" a list out of the results of those calls...

\[
> \text{showLecture } 1 \ 15 \ 'H' : \\
\text{showLecture } 2 \ 20 \ 'T' : \\
"...more..." : -- I literally typed "...more..." \\
\text{showLecture } 32 \ 125 \ 'T' : []
\]\[
["#1 H 1/15\n", \\
"#2 T 1/20\n", "...more...", "#32 T 5/5\n"]
\]

How close are the contents of that list to what we need?
Now let's imagine a recursive function \texttt{showLectures} that builds up a list of results from \texttt{showLecture} calls:

\begin{verbatim}
showLectures 1 15 126 [('H',5),('T',2)]  "#1 H 1/15
showLectures 2 20 126 [(T',2),('H',5)]  "#2 T 1/20
...
showLectures 32 125 126 [(T',2),('H',5)]  "#32 T 5/5
showLectures 33 127 126 [('H',5),(T',2)]
\end{verbatim}

Result:

\begin{verbatim}
["#1 H 1/15","#2 T 1/20", ... ,"#33 H 5/5"]
\end{verbatim}

Now let's write \texttt{showLectures}:

\begin{verbatim}
showLectures lecNum thisDay lastDay
    (pair@(dayOfWeek, daysToNext):pairs)
| thisDay > lastDay = []
| otherwise = showLecture lecNum thisDay dayOfWeek
          : showLectures (lecNum+1) (thisDay + daysToNext)
          lastDay (pairs ++ [pair])
\end{verbatim}
Finally, a top-level function to get the ball rolling:

```haskell
classdays first last pattern = putStrLn (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
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?

```haskell
> (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]
```
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.
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)

\[
\text{len list@(_:t)} \\
| \text{list == [] = 0} \\
| \text{otherwise = 1 + len t}
\]
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 *two* arguments: \( f \) and \( x \)

\[
\text{twice } f \ x = f \ (f \ x)
\]

What does it do?

\[
> \text{twice tail } [1,2,3,4,5] \\
[3,4,5]
\]

\[
> \text{tail } (\text{tail } [1,2,3,4,5]) \\
[3,4,5]
\]
At hand:

\[
\begin{align*}
&> \text{twice } f \ x = f \ (f \ x) \\
&> \text{twice tail } [1,2,3,4,5] \\
&\quad [3,4,5]
\end{align*}
\]

Let's make the left-associativity explicit:

\[
\begin{align*}
&> (\text{twice tail}) \ [1,2,3,4,5] \\
&\quad [3,4,5]
\end{align*}
\]

Consider a partial application...

\[
\begin{align*}
&> t2 = \text{twice tail} \quad \text{-- like } t2 \ x = \text{tail} \ (\text{tail} \ x) \\
&> t2 \\
&\quad \text{<function>} \\
&\quad \text{it :: } [a] \rightarrow [a]
\end{align*}
\]
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]
```
At hand:
\[
twice \ f \ x = f (f x)
\]

What's the type of \( twice \)?

\[
> :t \ twice
\]
\[
twice :: (t \rightarrow t) \rightarrow t \rightarrow t
\]

Parentheses added to show precedence:
\[
twice :: (t \rightarrow t) \rightarrow (t \rightarrow t)
\]
\[
twice f \ x = f (f x)
\]

What's the correspondence between the elements of the clause and the elements of the type?

A higher-order function is...

a function that (1) has one or more arguments that are functions and/or (2) returns a function.
The `map` function
Recall \( \text{double } x = x \times 2 \)

\textbf{map} is a Prelude function that applies a function to each element of a list, producing a new list:

\[ > \text{map double [1..5]} \]
\[ [2,4,6,8,10] \]

\[ > \text{map length (words "a few words" )} \]
\[ [1,3,5] \]

\[ > \text{map head (words "a few words")} \]
\[ "afw" \]

Is \textbf{map} a higher order function?
Yes! (Why?)
Its first argument is a function.
At hand:
  >>> map double [1..5]
  [2,4,6,8,10]

Problem: Write \texttt{map}!
  
  \begin{align*}
  \text{map} & \ [\ ] = \ [\ ] \\
  \text{map} \ f \ (x:xs) & = f \ x : \text{map} \ f \ xs
  \end{align*}

What is its type?
  
  \texttt{map} :: (a -> b) -> [a] -> [b]

What's the relationship between the length of \texttt{map}'s input and output lists?
  
  The lengths are always the same.
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`. 
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 maps.

See *Parallel and Concurrent Programming in Haskell* by Marlow

Google for MapReduce
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]]
Instead of using \texttt{map (add 5)} to add 5 to the values in a list, we should use a \texttt{section} instead: (it's the idiomatic way!)

\begin{verbatim}
> map (5+) [1,2,3]
[6,7,8] -- [5+1, 5+2, 5+3]
\end{verbatim}

More sections:

\begin{verbatim}
> map (10*) [1,2,3]
[10,20,30]

> map (++"*") (words "a few words")
["a*", "few*", "words*"]

> map ("*"++) (words "a few words")
["*a", "*few", "*words"]
\end{verbatim}
Sections have one of two forms:

- \((\text{infix-operator value})\)
  - Examples: \((+5), (/10)\)

- \((\text{value infix-operator})\)
  - Examples: \((5\ast), ("x"+++)\)

Iff the operator is commutative, the two forms are equivalent.

> map \((3\leq=)\) [1..4]
[False,False,True,True]

> map \((\leq=3)\) [1..4]
[True,True,True,False]

Sections aren't just for \texttt{map}; they're a general mechanism.

> twice \((+5)\) 3
13
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

```python
>>> 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(1,10000000000000000000000))
<map object at 0x114187fd0>

More: docs.python.org/3/library/functools.html
"[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 writing any recursive functions!

Think of it as isolating muscle groups when weight training.

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

\[\textit{Pretend that you don't understand recursion!}\]

- What's a base case? Is it related to baseball?
- Why would a function call itself? How's it stop?
- Is a recursive plunge refreshing?

If you were UNIX systems, I'd do \texttt{chmod 0} on an appropriate section of your brains.
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 writing any recursive functions?
Recall:

\[
> :t \text{mapMove} \\
\text{mapMove :: Char -> (Int, Int)}
\]

\[
> \text{mapMove 'n'} \\
(0,1)
\]

Now what?

\[
> \text{map mapMove "nneen"} \\
[(0,1),(0,1),(1,0),(1,0),(0,1)]
\]

Can we sum the tuples with \text{map}?

No!
We have:

\[
> \text{disps} = \text{map mapMove } "\text{nneen}"
\text{[(0,1),(0,1),(1,0),(1,0),(0,1)]}
\]

We want: (2,3)

Any ideas?

\[
> :t \text{fst}
\]

\[
\text{fst} :: (a, b) \to a
\]

\[
> \text{map fst disps}
\text{[0,0,1,1,0]}
\]

\[
> \text{map snd disps}
\text{[1,1,0,0,1]}
\]
We have:

```haskell
> 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?

```haskell
> :t sum
sum :: Num a => [a] -> a
> (sum (map fst disps), sum (map snd disps))
(2,3)
```
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 write any recursive functions?
No.

Did we use any recursive functions?
Maybe. But using recursive functions doesn't violate the prohibition at hand.
Filtering
Another higher order function in the Prelude is `filter`:

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

```haskell
> 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 in, not filtering out.

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...

```haskell
> 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.

```haskell
> filter (`elem` "aeiou") "some words here"
"oeoee"
```
At hand:

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

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

Problem: Write filter!

\[
\text{filter }_\_ \; [] = []
\]
\[
\text{filter } f \; (x:xs)
\]
\[
\begin{align*}
& | \quad f \; x = x : \text{filteredTail} \\
& | \quad \text{otherwise} = \text{filteredTail}
\end{align*}
\]

where

\[
\text{filteredTail} = \text{filter } f \; xs
\]
Several Prelude functions use predicates. Here are two:

\[
\text{all} :: (a \to \text{Bool}) \to [a] \to \text{Bool}
\]

> all even [2,4,6,8]
True
> all even [2,4,6,7]
False

\[
\text{dropWhile} :: (a \to \text{Bool}) \to [a] \to [a]
\]

> dropWhile isSpace " testing "
"testing "
> dropWhile isLetter it
" "

How could we find other Prelude functions that use predicates?

\%
\text{grep} "(a \to \text{Bool})" \text{prelude.txt}
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.
Here is `filter` in Python, along with two predicates:

```python
% 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)]
```
Usage:

```bash
% 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
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:

```haskell
> 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:

```haskell
> map (\n -> n * 2 - 5) [1..5]
[-3,-1,1,3,5]
```

What benefits does the anonymous function provide?
Anonymous functions, continued

At hand:

\[
\begin{align*}
  f \ n &= n \times 2 - 5 \\
  \text{map } f \ [1..5]
\end{align*}
\]

vs.

\[
\begin{align*}
  \text{map } (\lambda n -> n \times 2 - 5) \ [1..5]
\end{align*}
\]

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:

\[
\begin{align*}
  \text{int } t &= a \times 3 + g(a+b); \\
  &\quad // \text{Java} \\
  \text{return } f(t);
\end{align*}
\]

vs.

\[
\begin{align*}
  \text{return } f(a \times 3 + g(a+b));
\end{align*}
\]
Anonymous functions, continued

The general form of an anonymous function:
\[
\text{pattern 1 ... pattern N} \rightarrow \text{expression}
\]

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 \(\lambda\) (Greek lambda), used in the lambda calculus, another system for expressing computation.
Anonymous functions, continued

What will \texttt{ghci} say?
\begin{verbatim}
> \x y -> x + y * 2
<function>
> it 3 4
11
\end{verbatim}

\(\x y -> x + y * 2\) is an \texttt{expression} whose value is a function.

Here are three ways to bind the name \texttt{double} to a function that doubles a number:
\begin{verbatim}
double x = x * 2

double = \x -> x * 2

double = (*2)
\end{verbatim}
Anonymous functions, continued

Anonymous functions are commonly used with higher order functions such as `map` and `filter`.

```haskell
> map (\w -> (length w, w)) (words "a test now")
[(1,"a"),(4,"test"),(3,"now")]
```

```haskell
> 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
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:scientificphilosophical
200796:tetraiodophenolphthalein
203042:thyroparathyroidectomize

Imagine that we don't understand recursion, how can we approach it in Haskell?
Let's work with a small file for development purposes:

```plaintext
% cat longest.1
data
to
test
```

readFile in the Prelude lazily returns the full contents of a file as a string:

```haskell
> 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:

```haskell
> longest "data nto ntest n"
"1: data n3: test n"
```
Let's work through a series of transformations of the data:

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

```haskell
> lns = lines bytes
```

```haskell
> 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:

```haskell
> triples = zip3 (map length lns) [1..] lns
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
```
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")]

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."
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?
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]
```
At hand:

\[
\text{sortedTriples} = [(4,3,"test"),(4,1,"data"),(2,2,"to")]
\]

\[
\text{maxLength} = 4
\]

\[
\text{maxTriples} = \text{takeWhile}\left(\text{\(\backslash\)triple -> first triple == maxLength}\right)\text{sortedTriples}
\]

\[
[(4,3,"test"),(4,1,"data")]
\]

Should we have just used \text{filter} instead?

\[
\text{maxTriples} = \text{filter}\left(\text{\(\backslash\)triple -> first triple == maxLength}\right)\text{sortedTriples}
\]
At hand:

```haskell
> maxTriples
[(4,3,"test"),(4,1,"data")]
```

Let's map an anonymous function to turn the triples into lines prefixed with their line number:

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

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

```haskell
> result = unlines (reverse linesWithNums)
"l:data\n3:test
"
Let's package up our work into a function:

```haskell
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)
```

```haskell
first (x,_,_) = x
```

Look! No conditional code!
At hand:

> longest "data\nto\ntest\n" "l:da\n3:te\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)
    putStrLn (longest bytes)

Execution:

% runghc longest /usr/share/dict/words
42702:electroencephalograph's
Larger example: printCoords
Let's write a function to print one-based row and column coordinates for a grid with some number of rows and columns:

```haskell
> 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
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)]
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)"
Let's use `makeRow` to write `makeGrid`:

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

Usage:

```haskell
> 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:

```haskell
[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 =
  putStrLn (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
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:
\[
\text{compose} \, f \, g \, x = f \, (g \, x)
\]

How many arguments does \texttt{compose} have?

Usage:
\[
\begin{align*}
> & \text{compose init tail [1..5]} \\
& [2,3,4] \\
> & \text{compose ([]) chr 42} \\
& "*"
\end{align*}
\]
The Prelude binds the symbolic variable dot to a "compose" function:

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

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

```haskell
> numwords = length . words

> numwords
<function>

> numwords "just testing this"
3

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

```haskell
numwords = length . words
```

A model:

```
"a b"  -->  \textbf{words}  \rightarrow  ["a","b"]  \rightarrow  \textbf{length}  \rightarrow  2  \rightarrow  \textbf{numwords}  \rightarrow  2
```

Usage:

```
> numwords "a b"
2
```
At hand:

\[
\text{numwords} = \text{length} \ . \ \text{words}
\]

What's the type of \text{numwords}?

\[
> :t (.)
\]

\[
(\cdot) \colon (b \to c) \to (a \to b) \to a \to c
\]

\[
[t] \to \text{Int} \quad \text{String} \to \text{[String]}
\]

\[
(\text{length}) \quad (\text{words})
\]

\[
> :t \text{numwords}
\]

\[
\text{numwords} :: \text{String} \to \text{Int}
\]
Problem: Using composition create a function that returns the next-to-last element in a list:

```haskell
> ntl [1..5]
4

> ntl "abc"
'b'
```

Two solutions:

```haskell
ntl = head . tail . reverse
ntl = last . init
```
Problem: Using composition, create a function that reverses the words in a string:

\[
> f \ "\text{flip these words around}\"
\ "\text{pilf eseht sdrow dnuora}\"
\]

Hint: \textbf{unwords} is the inverse of \textbf{words}.

Solution:

\[
f = \textbf{unwords} \ . \ (\textbf{map} \ \textbf{reverse}) \ . \ \textbf{words}
\]
Problem: Create a function to remove the digits from a string:

```haskell
> rmdigits "Thu Feb 6 19:13:34 MST 2014"
"Thu Feb :: MST"
```

Solution:

```haskell
> rmdigits = filter (not . isDigit)
```
Recalling the following, what's the type of \( f \)?
- `head :: [a] -> a`
- `length :: [a] -> Int`
- `words :: String -> [String]`
- `show :: Show a => a -> String`

\[
f = \text{head} \ . \ \text{show} \ . \ \text{length} \ . \ \text{words}
\]

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

What's the type of \( f \)?
- `String -> Char`
Consider the following:

\[
> s = \text{"It's on!"} \\
> \text{map head (map show (map not (map isLetter s)))} \\
\text{"FFTFTFFFT"}
\]

Can we use composition to simplify it?

\[
> \text{map (head . show . not . isLetter) s} \\
\text{"FFTFTFFFT"}
\]

Question: Is

\[
\text{map f (map g x)}
\]

always equivalent to the following?

\[
\text{map (f . g) x}
\]

If \( f \) and \( g \) did output, how would the output of the two cases differ?
FOR THE CURIOUS!

Point-free style
Recall \texttt{rmdigits}: 

\begin{verbatim}
> rmdigits "Thu Feb 6 19:13:34 MST 2014"
"Thu Feb :: MST "
\end{verbatim}

What the difference between these two bindings for \texttt{rmdigits}?

\begin{verbatim}
rmdigits s = filter (not . isDigit) s
\end{verbatim}

\begin{verbatim}
rmdigits = filter (not . isDigit)
\end{verbatim}

The latter version is said to be written in \emph{point-free style}.

A point-free binding of a function \texttt{f} has NO parameters!
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.
Problem: Using point-free style, bind \texttt{len} to a function that works like the Prelude's \texttt{length}.

Handy:
\begin{verbatim}
> :t const
const :: a -> b -> a

> const 10 20
10

> const [1] "foo"
[1]
\end{verbatim}

Solution:
\[
\texttt{len} = \texttt{sum} \ . \ \texttt{map} \ (\texttt{const} \ 1)
\]

See also: \textit{Tacit programming} on Wikipedia
Hocus-pocus with higher order functions
What's this function doing?

\[ f \ a = g \]
\[ \text{where} \]
\[ g \ b = a + b \]

Type?

\[ f :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a \rightarrow \ a \]

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

\[
\text{for (int } i = 0; i < n; i++) ...
\]

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

```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']
```
Here's another mystery function:

```haskell
> m f x y = f y x
```

```haskell
> :type m
m :: (t1 -> t2 -> t) -> t2 -> t1 -> t
```

Can you devise a call to \texttt{m}?

```haskell
> m add 3 4
7
```

```haskell
> m (++) "a" "b"
"ba"
```

What is \texttt{m} doing?
At hand:
\[ m \ f \ x \ y = f \ y \ x \]

\( m \) is actually a Prelude function named \textbf{flip}:
\[
> \ :t \ \text{flip} \\
\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c
\]

Recall \( \text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \)
\[
> \text{flip take [1..10] 3} \\
[1,2,3]
\]
\[
> \text{ftake} = \text{flip take} \\
> \text{ftake [1..10] 3} \\
[1,2,3]
\]
From assignment 3:
  > splits "abcd"
  ["a","bcd"],("ab","cd"),("abc","d")

Some students have already noticed the Prelude's \texttt{splitAt}:
  > splitAt 2 \[10,20,30,40\]
  ([10,20],[30,40])

Problem: Write a non-recursive version of \texttt{splits}.

Solution:
  \[
  \text{splits list} = \text{map (flip splitAt list)} \ [1..\text{length (tail list)}]
  \]

Spring '18 solution:
  \[
  \text{splits list} = \text{map (flip splitAt list)} \ [1..(\text{length list} - 1)]
  \]
At hand:
\[ \text{flip } f \ x \ y = f \ y \ x \]

\[ \text{> map (flip take "Haskell") [1..7] } \]
\[ ["H","Ha","Has","Hask","Haske","Haskel","Haskell"] \]

Problem: write a function that behaves like this:
\[ \text{> f 'a'} \]
\[ ["a","aa","aaa","aaaa","aaaaa",...,infinitely...] \]

Solution:
\[ f \ x = \text{map (flip replicate x) [1..]} \]
The $ operator

$ is the "application operator".

```haskell
> :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:

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

Usage:

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

What's this operator good for?
Because + has higher precedence than $, the expression
\[ \text{negate } (3 + 4) \]
group like this:
\[ \text{negate } (3 + 4) \]

Problem: Rewrite the following to take advantage of $:
\[ \text{filter } (>3) (\text{map length } (\text{words } "up and down")) \]
\[ \text{filter } (>3) \text{ $ map length $ words } "up and down" \]

Common mistake: Confusing $ with . (composition)!
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.

\[ g :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \]
\[ g (x,y) = x^2 + 3*x*y + 2*y^2 \]

> g (3,4)
77

Solution: Curry \( g \) with \textbf{curry} from the Prelude!

> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]

Your problem: Write \textbf{curry}! (And don't peek ahead!)
Currying the uncurried, continued

At hand:

\[
\begin{align*}
> & g (3,4) \\
& 77 \\
> & \text{map (curry g 3) [1..10]} \\
& [20, 35, 54, 77, 104, 135, 170, 209, 252, 299]
\end{align*}
\]

Here's \textit{curry}:
\[
\text{curry } f \ x \ y = f (x, y)
\]

Usage:

\[
\begin{align*}
> & \ cg = \text{curry } g \\
> & :\text{type} \ cg \\
& cg :: \text{Int} \to \text{Int} \to \text{Int}
\end{align*}
\]

\[
\begin{align*}
> & cg \ 3 \ 4 \\
& 77
\end{align*}
\]
Currying the uncurried, continued

At hand:

\[
\text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c
\]
\[
\text{curry } f \ x \ y = f \ (x, y)
\]

\[
> \ \text{map (curry } g \ 3) \ [1..10] \\
[20,35,54,77,104,135,170,209,252,299]
\]

The key: \((\text{curry } g \ 3)\) is a partial application of \text{curry}!
Let's get \texttt{flip} into the game!

\begin{verbatim}
> map (flip (curry g) 4) [1..5]
[45,60,77,96,117]
\end{verbatim}

Note that

\texttt{flip} (\texttt{curry} \texttt{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:

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

Usage:

```python
>>> 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
```
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","eeeeee"]
```
Folding
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.

\[ \text{reduce} (+) [1,2,3,4] \]
10

\[ \text{reduce} (++) ["a","bc","def"] \]
"abcdef"

\[ \text{reduce} \text{ max} [10,2,4] \]
10 -- think of 10 `max` 2 `max` 4
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
```
In the Prelude there's no `reduce` but there is `foldl1` and `foldr1`.

```haskell
> 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`. 
Another folding function is \( \text{foldl} \) (no 1). Let's compare the types of \( \text{foldl1} \) and \( \text{foldl} \):

- \( \text{foldl1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \)
- \( \text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \)

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

First difference: \( \text{foldl} \) requires one more argument:

- \( \text{foldl} (+) \ 0 \ [1..10] \)
  \[
  55
  \]

- \( \text{foldl} (+) \ 100 \ [] \)
  \[
  100
  \]

- \( \text{foldl1} (+) [] \)

*** Exception: Prelude.foldl1\: empty list
Again, the types:

\[
\text{foldl1} :: (a \to a \to a) \to [a] \to a
\]
\[
\text{foldl} :: (a \to b \to a) \to a \to [b] \to a
\]

Second difference:

\text{foldl} can fold a list of values into a different type! (This is \text{BIG}!)

Examples:

\[
> \text{foldl } f1 0 \ ["just","a","test"]
3
\] -- folded strings into a number

\[
> \text{foldl } f2 "stars: " \ [3,1,2]
"stars: *****" -- folded numbers into a string
\]

\[
> \text{foldl } f3 0 \ [(1,1),(2,3),(5,10)]
57
\] -- folded two-tuples into a sum of products
For reference:

\[
\text{foldl} :: (a \to b \to a) \to a \to [b] \to a
\]

Here's another view of the type: (\(\text{acm}_t\) stands for accumulator type)

\[
\text{foldl} :: (\text{acm}_t \to \text{elem}_t \to \text{acm}_t) \to \text{acm}_t \to [\text{elem}_t] \to \text{acm}_t
\]

\text{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:

\[
> \text{foldl} \ f1 \ 0 \ ["just","a","test"]
\]

3

\[
> \text{foldl} \ f2 \ "stars: " \ [3,1,2]
"stars: *****"
\]
Recall:
\[
> \text{foldl}\ f\ 0\ ["just","a","test"] \\
3
\]

Here are the computations that \text{foldl} did to produce that result:
\[
> f\ 0\ "just" \\
1 \\
> f\ \text{it}\ "a" \\
2 \\
> f\ \text{it}\ "test" \\
3
\]

Let's do it in one expression, using backquotes to infix \text{f1}:
\[
> ((0 \ `f1` "just") `f1` "a") `f1` "test" \\
3
\]
At hand:
> f1 0 "just"
1
> f1 it "a"
2
> f1 it "test"
3

Problem: Write a function f1 that behaves like above.
Starter:
\[
f1 :: \text{acm\_t} \rightarrow \text{elem\_t} \rightarrow \text{acm\_t}
f1 \text{ acm elem} = \text{acm + 1}
\]

Congratulations! You just wrote a folding function!
Recall:

```haskell
> foldl f2 "stars: " [3,1,2]
"stars: ******"
```

Here's what `foldl` does with `f2` and the initial value, "stars: ":

```haskell
> f2 "stars: " 3
"stars: ***"
> f2 it 1
"stars: ****"
> f2 it 2
"stars: ******"
```

Write `f2`, with this starter:

```haskell
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.

\[ f2 \text{ acm elem} = \text{acm} \:+\:+ \text{replicate elem '***'} \]

We then call `foldl` with (1) the folding function, (2) an appropriate initial value, and (3) a list of values.

\[ \text{foldl f2 "stars: " [3,1,2]} \]

`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!
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
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"
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

\[
f (\text{evens}, \text{odds}) \text{ elem}
\]

| even elem = (evens + 1, odds) |
| otherwise = (evens, odds + 1) |

Problem: Write countEO as a foldl with f

\[
\text{countEO nums} = \text{foldl} f (0,0) \text{ nums}
\]
Anonymous functions are often used for folds.

Here are three earlier folds with anonymous functions:

```haskell
> 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
```
The counterpart of \texttt{foldl} is \texttt{foldr}. Compare their meanings:

\[
\texttt{foldl } f \texttt{ zero } [e_1, e_2, \ldots, e_N] == \ldots((\texttt{zero } `f` e_1) `f` e_2) `f`\ldots)`f` e_N
\]

\[
\texttt{foldr } f \texttt{ zero } [e_1, e_2, \ldots, e_N] == e_1 `f` (e_2 `f` \ldots (e_N `f` \texttt{zero})\ldots)
\]

"zero" represents the computation-specific initial accumulated value. Note that with \texttt{foldl}, zero is leftmost; but with \texttt{foldr}, zero is rightmost.

Their types, with long type variables:

\[
\texttt{foldl} :: (\texttt{acm} \rightarrow \texttt{val} \rightarrow \texttt{acm}) \rightarrow \texttt{acm} \rightarrow [\texttt{val}] \rightarrow \texttt{acm}
\]

\[
\texttt{foldr} :: (\texttt{val} \rightarrow \texttt{acm} \rightarrow \texttt{acm}) \rightarrow \texttt{acm} \rightarrow [\texttt{val}] \rightarrow \texttt{acm}
\]

Mnemonic aid:

\texttt{foldl}'s folding function has the accumulator on the \texttt{left}.
\texttt{foldr}'s folding function has the accumulator on the \texttt{right}. 
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:

```haskell
> keepOdds [5,4,2,3]
[5,3]
```

Implementation, with foldr:

```haskell```
```haskell
keepOdds list = foldr f [] list
  where
    f elem acm
      | odd elem = elem : acm
      | otherwise = acm
```
```haskell```

What are the calls to the folding function?

```haskell```
```haskell
> f 3 [] -- rightmost first!
[3]
> f 2 it
[3]
> f 4 it
[3]
> f 5 it
[5,3]
```
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?
Can a3's `paired` be done with a fold?

```haskell
> paired "(((())())")
True
```

Sure!

```haskell
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.
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:

\[
> \text{foldr (} \lambda v \text{ acm ->} \\text{acm + if } v \text{ `elem` "aeiou" then 1 else 0)} 0 \text{ "ate"} \\
= 2
\]

Next, let's produce both a count and the vowels themselves:

\[
> \text{foldr (} \lambda l \text{ acm@(n, v) ->} \\text{if } l \text{ `elem` "aeiou" then (n+1, l:v) else acm)} 0,[] \text{ "ate"} \\
= (2,"ae")
\]
A progression of folds, continued

Finally, let's write a function that produces a list of vowels and their positions:

```haskell
vowelPositions "Now for some Prolog!"
[("o", 1), ("o", 5), ("o", 9), ("e", 11), ("o", 15), ("o", 17)]
```

Solution:

```haskell
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: A value of any type and complexity

True or false?
  Any operation that processes a list can be expressed in 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 to study functional programming, here's what might be next:

- 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!

```haskell
data Shape = Circle Double | Rect Double Double
  deriving Show

s = [Rect 3 4, Circle 2]
```
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