Functional Programming with Haskell
Programming Paradigms
Thomas Kuhn's *The Structure of Scientific Revolutions* describes a *paradigm* as a scientific achievement that is...

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

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

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

Kuhn also wrote,

"[I take] paradigms to be universally recognized scientific achievements that for a time provide model problems and solutions to a community of practitioners."

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

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

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

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

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

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

Java programs with a single class are typically examples of procedural programming.
The object-oriented programming paradigm

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

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

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

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

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

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

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

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

What language features are required to support procedural programming?

• The ability to break programs into procedures.

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

• Ability to define classes that comprise data and methods
• Ability to specify inheritance between classes
Paradigms in a field of science are often incompatible. Example: geocentric vs. heliocentric model of the universe

Can a programming language support multiple paradigms? Yes! We can do procedural programming with Java.

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

Wikipedia's Programming_paradigm cites 60+ paradigms!

But, are "programming paradigms" really paradigms by Kuhn's definition or are they just characteristics?
The imperative programming paradigm
The imperative paradigm has its roots in programming at the machine level, usually via assembly language.

Machine-level programming:
• Instructions change memory locations or registers
• Instructions alter the flow of control

Programming with an imperative language:
• Expressions compute values based on memory contents
• Assignments alter memory contents
• Control structures guide the flow of control
The imperative programming paradigm

Both the procedural and object-oriented paradigms typically make use of the imperative programming paradigm.

Two fundamental characteristics of languages that support the imperative paradigm:

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

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

How can the above solution be improved?
Imperative programming, continued

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

```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?
Imperative programming, continued

We can use recursion instead of a loop, 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);
}
```

Wrt. correctness, which of the three versions would you bet your job on?
Programming paradigms can apply at different levels:

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

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

Java combines the object-oriented and imperative paradigms.

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

The imperative paradigm applies to *programming in the small*. 
Imperative vs. applicative methods in Java

Java methods can be classified as imperative or *applicative*.

- An imperative method changes an object.
  "Change this."

- An applicative method produces a new object.
  "Make me a such and such from this."

In some cases we have an opportunity to choose between the two.
Consider a Java class representing a 2D point:

```java
class Point {
    private int x, y;
}
```

An imperative method to translate by an x and y displacement:

```java
public void translate(int dx, int dy) {
    x += dx; y += dy;
}
```

An applicative translate:

```java
public Point translate(int dx, int dy) {
    return new Point(x + dx, y + dy);
}
```

What are the pros and cons?
Imagine a **Line** class, whose instances are constructed with two **Points**.

What's the following code doing?

```java
Point end = p.clone();
end.translate(10,20);
Line L = new Line(p, end);
```

How about this code? (using an applicative `translate()`)

```java
Line L = new Line(p, p.translate(10,20));
```

Are methods on Java strings imperative or applicative?
Side effects

An expression is a sequence of symbols that can be evaluated to produce a value. Here's a Java expression:

\[ i + j \times k \]

If evaluating an expression also causes a value somewhere to change, we say that expression has a side effect.

Here's a Java expression with a side effect:

\[ i + j++ \times k \]

Do these two expressions have the same value?

What's the side effect?
Which of these Java expressions have a side effect?

\[
dx = 10
\]

\[
\text{p1.translate}(10, 20) \quad \text{// Consider both!}
\]

\[
"testing".toUpperCase()
\]

\[
\text{L.add}("x"), \quad \text{where L is an ArrayList}
\]

\[
\text{System.out.println("Hello!")}
\]

A machine language side effect: Loading a register might set a condition code.
Side effects are a hallmark of imperative programming.

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

Can we program without side effects?
The Functional Paradigm
The functional programming paradigm

One of the cornerstones of the functional paradigm is writing functions that are like pure mathematical functions, which:

• Map values from a domain set to unique values in a range set

• Can be combined to produce more powerful functions

• Have no side effects

Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions.
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 in data structures, passed to functions, returned from functions, and lots of temporary functions are created.

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

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

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

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

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

It is not object-oriented in any way.

My current opinion: it has a relatively large mental footprint.
Website: haskell.org
   All sorts of resources!

Books: (on Safari, too)
   Learn You a Haskell for Great Good!, by Miran Lipovača
   http://learnyouahaskell.com (I'll call it GG.)

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

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

Haskell 2010 Report (I'll call it H10.)
   http://haskell.org/definition/haskell2010.pdf
On lectura we can interact with Haskell by running ghci:

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

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

Note: the standard prompt is Prelude> but I've got
   :set prompt ""> "
in my ~//.ghci file.
Let's try some expressions with ghci:

> 3 + 4
7

> 3 * 4.5
13.5

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

> 2 ^ 200
1606938044258990275541962092341162602522202993782792835301376

> "abc" ++ "xyz"
"abcxyz"
We can use :help to see available commands:
> :help

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

The command :set +t causes types to be shown:
> :set +t
> 3+4
7
it :: Integer

"::" is read as "has type". The value of the expression is "bound" to the name it.
Interacting with Haskell, continued

We can use it in subsequent computations:

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

> it + it * it
56
it :: Integer

> it /= it
False
it :: Bool
```
Getting Haskell
You can either get Haskell for your machine or use Haskell on
lectura.

To work on your own machine, get a copy of the Haskell
Platform for your operating system from haskell.org.

On OS X, I'm using Haskell Platform 2013.2.0.0 for Mac OS
X, 64 bit from www.haskell.org/platform/mac.html

On Windows, use Haskell Platform 2013.2.0.0 for Windows
from http://www.haskell.org/platform/windows.html

You'll need an editor that can create plain text files. Sublime
Text is very popular.
Using Haskell on lectura

To work on lectura from a Windows machine, you might login with PuTTY.

OS X, do ssh YOUR-NETID@lectura.cs.arizona.edu

You might edit Haskell files on lectura with vim, emacs, or nano (ick!), or use something like gedit on a Linux machine in a CS lab.

Alternatively, you might edit on your machine with something like Sublime Text and use a synchronization tool (like WinSCP on Windows) to keep your copy on lectura constantly up to date.

If you go the route of editing on your machine and running on lectura, let us know if you have trouble figuring out how to do automatic synchronization—we can help! It's a terrible waste of time to do a manual copy of any sort in the middle of your edit/run cycle.
Getting and running PuTTY

If you Google for "putty", the first hit should be this:

PuTTY Download Page

• [www.chiark.greenend.org.uk/~sgtatham/putty/download.html](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)

Download putty.exe. It's just an executable—no installer!

<table>
<thead>
<tr>
<th>Binaries</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>The latest release version (beta 0.63).</em> Fixed the bug, before reporting it to me</td>
</tr>
</tbody>
</table>

For Windows on Intel x86

- PuTTY: [putty.exe](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)
- PuTTYtel: [puttytel.exe](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)
- PSCP: [pscp.exe](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)
PuTTY, continued

Click on `putty.exe` to run it. In the dialog that opens, fill in `lec.cs.arizona.edu` for Host Name and click Open.
Login to lectura using your UA NetID. Run ghci, and try some expressions:

```
lectura ~ 5003 $ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
> 3 + 4
7
> "abc" ++ "xyz"
"abcxyz"
> 2 ^ 200
1606938044258990275541962092341162602522202993782792835301376
> (control-D to exit)
Leaving GHCi.
lectura ~ 5004 $
```

Go to [http://cs.arizona.edu/computing/services](http://cs.arizona.edu/computing/services) and use "Reset my forgotten Unix password" if needed.
Functions and function types
In Haskell, *juxtaposition* indicates a function call:

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

> even 5
False
it :: Bool

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

> signum 2
1
it :: Integer
```

Note: These functions and many more are defined in the Haskell "Prelude", which is loaded by default when **ghci** starts up.
Function call with juxtaposition is left-associative.

`signum negate 2` means `(signum negate) 2`. It's an error:

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

We add parentheses to call `negate 2` first:

```
> signum (negate 2)
-1
it :: Integer
```
Function call with juxtaposition has higher precedence than any operator.

\[
> \text{negate } 3+4 \\
1 \quad \text{it :: Integer}
\]

\text{negate } 3 + 4 \text{ means } (\text{negate } 3) + 4. \text{ Use parens to force } + \text{ first:}

\[
> \text{negate } (3 + 4) \\
-7 \quad \text{it :: Integer}
\]

\[
> \text{signum } (\text{negate } (3 + 4)) \\
-1 \quad \text{it :: Integer}
\]
Function types

Haskell's `Data.Char` module has a number of functions for working with characters. It provides some simple examples of function types.

```haskell
> :m Data.Char -- loads module

> isLower 'b'
True
it :: Bool

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

> ord 'A'
65
it :: Int

> chr 65
'A'
it :: Char
```
Function types, continued

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

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

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

What are the types of `toUpper`, `ord`, and `chr`?

We can use :browse Data.Char to see everything in the module.
Like most languages, Haskell requires that expressions be *type-consistent* (or *well-typed*). Here is an example of an inconsistency:

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

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

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

*chr* requires its argument to be an *Int* but we gave it a *Char*. We can say that *chr 'x'* is *ill-typed*. 
State whether each expression is well-typed and if so, its type.

\[
\begin{array}{ll}
\text{'a'} & \text{'a' :: Char} \\
\text{isUpper} & \text{chr :: Int -> Char} \\
\text{isUpper 'a'} & \text{digitToInt :: Char -> Int} \\
\text{not (isUpper 'a')} & \text{intToDigit :: Int -> Char} \\
\text{not not (isUpper 'a')} & \text{isUpper :: Char -> Bool} \\
\text{toUpper (ord 97)} & \text{not :: Bool -> Bool} \\
\text{isUpper (toUpper (chr 'a'))} & \text{ord :: Char -> Int} \\
\text{isUpper (intToDigit 100)} & \text{toUpper :: Char -> Char}
\end{array}
\]
Recall the `negate` function:

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

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

What is the type of `negate`? (Is it both `Integer -> Integer` and `Double -> Double`?)
Type classes
Type classes

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

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

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

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

There are four types in the *Num* type class: *Int* (word-size) *Integer* (unlimited size), *Float* and *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 `Int`, I'll give you back an `Int`."

The portion `Num a` is a class constraint. It specifies that the type `a` must be in an instance of the type class `Num`.
What type do integer literals have?

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

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

Literals are typed with a class constraint of `Num`, so they can be used by any function that accepts `Num a => a`. 
Why does `negate 3.4` work?

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

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

> negate 3.4
-3.4
```
Haskell type classes form a hierarchy. The Prelude has these:

- **Eq**: All except IO, (->)
- **Ord**: All except IO, IOError, (->)
- **Num**: Int, Integer, Float, Double
- **Show**: All except IO, (->)
- **Read**: All except IO, (->)
- **Monad**: IO, [], Maybe
- **Bounded**: Int, Char, Bool, () Ordering, tuples
- **MonadPlus**: IO, [], Maybe
- **Enum**: (), Bool, Char, Ordering, Int, Integer, Float, Double
- **Integral**: Int, Integer
- **Real**: Float, Double
- **RealFloat**: Float, Double
- **Fractional**: Float, Double
- **Functor**: IO, [], Maybe
The arrow from \texttt{Num} to \texttt{Fractional} means that a \texttt{Fractional} can be used as a \texttt{Num}. (What does that remind you of?)

Given
\[
\text{negate} :: \texttt{Num} \ a \Rightarrow a \rightarrow a
\]
and
\[
5.0 :: \texttt{Fractional} \ a \Rightarrow a
\]
then
\[
\text{negate} \ 5.0 \text{ is valid.}
\]
Type classes, continued

What's meant by the type of \texttt{pz}?

\[
\texttt{pz} :: (\text{Bounded } a, \text{Fractional } b) \Rightarrow a \rightarrow b
\]

Would \texttt{pz 'a'} be valid? How about \texttt{pz 5.5}? \texttt{pz 7}?

GG pp. 27-33 has a good description of the Prelude's type classes.

RWH uses the term "typeclasses"—one word!
In essence, `negate :: Num a => a -> a` describes many functions:

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

We can say that `negate` is a polymorphic function. It handles values of many forms.

If a function's type has any type variables, it's a polymorphic function. (not on handout)

How does Java handle this problem? How about C? C++?
Type checking
A fundamental characteristic of a programming language is whether an expression can be checked for type consistency without executing the expression.

Static type checking is checking for type consistency without executing an expression.

A language that primarily or exclusively uses static type checking is said to be statically typed. (Some say strongly typed or type-safe but those terms are debatable.)

Static typing allows us to guarantee that type mismatches do not occur in a body of code.

Is Java statically typed? If so, exclusively or primarily?
Static type checking, continued

Java is statically typed.

Example:
If a class has a method `String f()`, we don't need to execute any code to know that `f() * 3` is invalid.

Reasoning:
(1) `f()` returns a `String`
(2) `String * int` is not a supported operation

Consider `f() + x * 3`. When is it valid?
Recall:

*Static type checking* is checking for type consistency without executing an expression.

With *dynamic type checking*, types are not checked for consistency until execution time.

Python, Ruby, PHP, and Icon are dynamically typed languages.

Here's an expression in Icon:

```
?["abc", 7] * 2
```

It randomly selects an element from the list `"abc", 7` and multiplies it by 2. It either produces 14 or blows up.

Can we write a Java expression that is ill-typed half the time?
Consider this method:

```java
static Object f() {
    if (Math.random() < 0.5)
        return "abc";
    else
        return new Integer(7);
}
```

Given `int x`, does `x = (Integer)f() + 5` pass a static type check?

Does this affect our claim that Java is statically typed?
Variety in type checking

Java is statically typed but casts introduce the possibility of a type error during execution. However, type errors are detected.

C is statically typed but has casts that allow type errors during execution that are never detected.

Ruby, Python, and Icon have no static type checking whatsoever, but type errors in execution are always detected.

An example of a typing-related trade-off in execution time:
• C spends zero time during execution checking types.
• Java checks types only in certain cases.
• Languages with dynamic typing check types on every operation, at least conceptually.
Haskell is statically typed and by design allows no possibility of type errors during execution.

This decision, an aspect of Haskell's philosophy, permeates the language.

Good news:
• An entire class of errors is eliminated, yet with zero execution-time overhead.

Bad news:
• Supporting it substantially increases the mental footprint.
More on functions
Writing simple functions

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

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

> double 5
10
it :: Integer

> double 2.7
5.4
it :: Double

> double (double (double 1111111111111))
8888888888888
it :: Integer
```
Simple functions, continued

More examples:

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

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

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

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

Note: function names must begin with a lowercase letter or _.

CSC 372 Spring 2014, Haskell Slide 65
We can use :: **type** to constrain the type inferred for a function:

> let neg x = -x :: Integer

neg :: Integer -> Integer

> let isPositive x = x > (0::Integer)

isPositive :: Integer -> Bool

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

toCelsius :: Double -> Double

We'll use :: **type** to simplify some following examples.
We can put function definitions in a file. When we do, we leave off the `let`!

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

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

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

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

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

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

We can use a path, like `:load ~/372/hs/simple`, too.
Functions with multiple arguments

Here's a function that produces the sum of its two arguments:
  > let add x y = x + y :: Integer

Here's how we call it: (no commas or parentheses!)
  > add 3 5
  8

Here is its type:
  > :type add
  add :: Integer -> Integer -> Integer

The operator \( \rightarrow \) is right-associative, so the above means this:
  add :: Integer -> (Integer -> Integer)

But what does that mean?
Recall our negate function:

\[
> \text{let } \text{neg } x = -x :: \text{Integer} \\
\text{neg} :: \text{Integer} \to \text{Integer}
\]

Here's \textbf{add} again, with parentheses to reflect precedence:

\[
> \text{let } \text{add } x \ y = x + y :: \text{Integer} \\
\text{add} :: \text{Integer} \to (\text{Integer} \to \text{Integer})
\]

\textbf{add} is a function that takes an integer as an argument and produces a function as its result!

\textbf{add 3 5} means \((\text{add 3})\ 5\)

Call \textbf{add} with the value 3, producing a nameless function. Call that nameless function with the value 5.
When we give a function fewer arguments than it requires, the result is called a *partial application*.

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

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

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

What will `plusThree 5` produce?

```haskell
> plusThree 5
8
```

`it :: Integer`
Partial application, continued

At hand:

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

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

Analogy: **plusThree** is like a calculator where you've clicked 3, then +, and handed it to somebody.
Partial application, continued

At hand:

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

Another:

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

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

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

What prior use have you made of partially applied functions?
Some key points

Key points:

• A function with a type like \texttt{Integer -> Char -> Char} takes two arguments, an \texttt{Integer} and a \texttt{Char}. It produces a \texttt{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 that it requires creates a \textit{partial application}. 
Specifying a function's type

It is common practice to specify the type of a function along with its definition in a file.

What's the ramification of the difference in these two type specifications?

\[
\text{add1} :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a \rightarrow \ a \\
\text{add1} \ x \ y = x + y
\]

\[
\text{add2} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{add2} \ x \ y = x + y
\]
Function/operator equivalence

Haskell operators are simply functions that can be invoked with an infix form.

We can use :info to find out about an operator.

> :info (^)

(^) :: (Num a, Integral b) => a -> b -> a
infixr 8 ^

(Num a, Integral b) => a -> b -> a shows that the first operand must be a number and the second must be an integer.

infixr 8 shows that it is right-associative, with priority 8.

Explore ==, >, +, *, | |, ^ and **.
To use an operator as a function, enclose it in parentheses:

> (+) 3 4
7

We can use a function as an operator by enclosing it in backquotes:

> 3 `add` 4
7

> 11 `rem` 3
2
Haskell lets us define custom operators.

Example: (load from a file)

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

Usage:

```
> 100 +% 1
101.0

> 12 +% 25
15.0
```

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

Modules often define custom operators.
## Reference: Operators from the Prelude

<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><code>^</code>, <code>^^</code>, <code>**</code></td>
<td></td>
<td><code>^</code>, <code>^^</code>, <code>**</code></td>
</tr>
<tr>
<td>7</td>
<td><code>*</code>, <code>/</code>, <code>\div\</code>, <code>\mod\</code>, <code>\rem\</code>, <code>\quot</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>+</code>, <code>-</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>:</code>, <code>++</code></td>
<td></td>
<td><code>:</code>, <code>++</code></td>
</tr>
<tr>
<td>4</td>
<td><code>==</code>, <code>/=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>\elem\</code>, <code>\notElem</code></td>
<td></td>
<td><code>==</code>, <code>/=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>\elem\</code>, <code>\notElem</code></td>
</tr>
<tr>
<td>3</td>
<td><code>&amp;&amp;</code></td>
<td></td>
<td><code>&amp;&amp;</code></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><code>&gt;&gt;, &gt;&gt;=</code></td>
</tr>
<tr>
<td>0</td>
<td><code>$</code>, <code>!</code>, <code>\seq\</code></td>
<td></td>
<td><code>$</code>, <code>!</code>, <code>\seq\</code></td>
</tr>
</tbody>
</table>

Note: From page 51 in Haskell 2010 report
More functions

The general form of a function definition (for now):

\[
\text{let } \text{name arg1 arg2 ... argN} = \text{expression}
\]

Problem: Define a function \text{min3} that computes the minimum of three values. The Prelude has a \text{min} function.

\[
> \text{min3 5 2 10} \\
2
\]

\[
> \text{let min3 a b c = min a (min b c)} \\
\text{min3 :: Ord a} \Rightarrow a \rightarrow a \rightarrow a \rightarrow a
\]

Problem: Define a function \text{eq3} that returns \text{True} if its three arguments are equal, \text{False} otherwise.
Recall this characteristic of functional programming:
"Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions."

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

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

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

Problem: Using guards, define a function \texttt{smaller}, like \texttt{min}:

\begin{verbatim}
> smaller 7 10
7

> smaller 'z' 'a'
'a'
\end{verbatim}

Solution:

\begin{verbatim}
smaller x y
  | x <= y = x
  | otherwise = y
\end{verbatim}
Guards, continued

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

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

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

```
weather temp | temp >= 80 = "Hot!"
| temp >= 70 = "Nice"
| otherwise = "Cold!"
```
Here's an example of Haskell's if-else:

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

How does this compare to the **if-else** in Java?
Sidebar: Java's if-else

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

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

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

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

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

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

Pythoners: What's the if-else situation in Python?
Haskell's if-else, continued

What's the type of these expressions?

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

> :type if 1 < 2 then 3 else 4.0
  if 1 < 2 then 3 else 4.0 :: Fractional a => a

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

> if 1 < 2 then 3
  <interactive>:13:16:
  parse error (possibly incorrect indentation or mismatched brackets)
Which of the versions of \texttt{sign} below is better?

\begin{verbatim}
sign x
| x < 0 = -1
| x == 0 = 0
| otherwise = 1
\end{verbatim}

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

We'll later see that \textit{patterns} add a third possibility for expressing cases.
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. Write it in Haskell (and don't peek below!) What is its type?

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

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

```
> factorial 40
8159152832478977343456112695961158942720000000000
```
One way to manually trace through a recursive computation is to underline a call, then rewrite the call with a textual expansion:

\[
\text{factorial } n \\
| \ n == 0 = 1 \\
| \ otherwise = n * \text{factorial } (n - 1)
\]

factorial 4

4 * factorial 3

4 * 3 * factorial 2

4 * 3 * 2 * factorial 1

4 * 3 * 2 * 1 * factorial 0

4 * 3 * 2 * 1 * 1
Consider repeatedly dividing a number until the quotient is 1:

\[
> 28 \ `\text{quot}` 3 \quad (\text{Note backquotes to use quot as infix op.})
9
\]

\[
> \text{it} \ `\text{quot}` 3 \quad (\text{Remember that it is previous result.})
3
\]

\[
> \text{it} \ `\text{quot}` 3
1
\]

Problem: Write a recursive function \texttt{numDivs divisor x} that computes the number of times \texttt{x} must be divided by \texttt{divisor} to reach a quotient of 1:

\[
> \text{numDivs} 3 28
3
\]

\[
> \text{numDivs} 2 7
2
\]
Recursion, continued

A solution:

\[
\text{numDivs divisor x} \\
\text{\quad \mid (x `quot` divisor) < 1 = 0} \\
\text{\quad \mid otherwise = 1 + numDivs divisor (x `quot` divisor)}
\]

What is its type?

\[\text{numDivs :: (Integral a, Num a1) => a -> a -> a1}\]

Will \text{numDivs 2 3.4} work?

\[> \text{numDivs 2 3.4}\]
\[<\text{interactive}>:93:1: \text{No instance for (Integral a0) arising from a use of `numDivs'}\]
Sidebar: Fun with partial applications

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

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

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

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

> let floor_log10 = numDivs 10
> floor_log10 1000
3
```
Lists
In Haskell, a list is a sequence of values of the same type.

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

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

> [1.3, 10, 4, 9.7]
[1.3,10.0,4.0,9.7]  -- it :: [Double]

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

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

```
> length [3,4,5]
3
> length []
0
```

What does the type of `length` tell us?

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

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

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

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

What's the type of `head`?

```
> :type 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`?

```
> :type tail
tail :: [a] -> [a]
```
The ++ operator concatenates two lists, producing a new list.

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

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

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

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

What are the types of ++ and reverse?
A range of values can be specified with a dot-dot notation:

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

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

> length [-1000..1000]
> 2001

> [10..5]
> []
> it :: [Integer]
The `!!` operator produces a list's Nth element, zero-based:

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

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

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

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

Should that be allowed?
We can make an infinite list in Haskell! Here's one way:

\[
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,^C
\]

Any ideas on how to make use of an infinite list?

What does the following \texttt{let} create?

\[
> \texttt{let nthOdd} = (!!) [1,3..]
nthOdd :: \texttt{Int} \rightarrow \texttt{Integer}
\]

A function that produces the Nth odd number, zero-based.

Yes, we could say \texttt{let nthOdd n = (n*2)+1} but that wouldn't be nearly as much fun! (This is \texttt{function}al programming!)
Consider the following let. Why does it complete?

```haskell
> let fives=[5,10..]
fives :: [Integer]
```

Haskell uses **lazy evaluation**. It only computes as much of a value as it needs to.

The function `take` produces the first N elements of a list

```haskell
> take 5 fives
[5,10,15,20,25]
```

Haskell computes only enough elements of `fives` to satisfy `take 5`. 
Here's an expression that doesn't complete:

\[
\text{> length fives}
\]

\[...\textit{when tired of waiting}...\] \(^\text{^C Interrupted.}\)

But, we can bind \texttt{length fives} to a name:

\[
\text{> let numFives = length fives}
\]

\[
\text{numFives :: Int}
\]

That completes because Haskell hasn't yet needed to compute a value for \texttt{length fives}.

We can hang ourselves again by asking Haskell to print the value of \texttt{numFives}:

\[
\text{> numFives}
\]

\[...\textit{after a while}...\] \(^\text{^C Interrupted.}\)
let capitals =

list of state capitals, ordered by state's admission to the union

Evaluate:
capitals !! 47
Lazy state capitals, continued

Imagine that `capitals` has this binding:

```haskell
let capitals =
    [..., capital "AZ", capital "AK", capital "HI"]
```

Suspend disbelief and further imagine that the `capital` function gets its result by scraping it from the Wikipedia page for the state.

Evaluating `capitals !! 47` requires only `capital "AZ"` to be called!

What does this look like in Java? What's the trade-off?

```java
String cap47th = capitals()[47];

String cap47th = getCapital(47);
```

How does Haskell avoid the trade-off?
Haskell lists are values and can be compared as values:

\[
\begin{align*}
> [3,4] &= [1+2, 2\cdot2] \\
&\text{True}
\end{align*}
\]

\[
\begin{align*}
&\text{True}
\end{align*}
\]

\[
\begin{align*}
> \text{tail (tail [3,4,5,6])} &= [\text{last [4,5]}] ++ [6] \\
&\text{True}
\end{align*}
\]

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

A programmer using a functional language writes complex expression using lists (and more!) as freely as a Java programmer might write \( f(x) \cdot a = g(a,b) + c \).
Comparing lists, continued

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

Example:

```plaintext
> [1,2,3] < [1,2,4]  
True  
Why: The first two elements are equal, and 3 < 4.
```

More examples:

```plaintext
> [1,2,3] < [1,1,1,1]  
False  
> [1,2,3] > [1,2]  
True  
> [1..] < [1,3..]  
-- Comparing infinite lists!  
True
```
Lists of lists

We can make lists of lists.

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

Note the type: \( x \) is a list of \textbf{Integer} lists.

\textbf{length} counts elements at the top level.

```haskell
> length x
3
```

Recall that \textbf{length} :: \([a]\) \rightarrow \textbf{Int} \quad \text{Given that, what's the type of} \quad \text{\textit{a} for length} \quad \text{\textit{x}?}

What's the value of \textbf{length} (\( x ++ x ++ [3] \))?
Let $x = \begin{bmatrix} [1], [2,3,4], [5,6] \end{bmatrix}$

$\texttt{head } x$ 
$[1]$ 

$\texttt{tail } x$ 
$\begin{bmatrix} [2,3,4], [5,6] \end{bmatrix}$ 

$x !! 1 !! 2$ 
$4$ 

Let $y = \begin{bmatrix} [1..], [10,20..] \end{bmatrix} ++ \begin{bmatrix} [2,3] \end{bmatrix}$ 
$\texttt{take } 5 \ (\texttt{head } (\texttt{tail } y))$ 
$\begin{bmatrix} 10,20,30,40,50 \end{bmatrix}$
Strings in Haskell are simply lists of characters.

> "testing"
"testing"

it :: [Char]

> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

it :: [Char]

> ["just", "a", "test"]
["just","a","test"]

it :: [[Char]]

What's the beauty of this?
All list functions work on strings, too!

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

> length asciiLets
52

> reverse (drop 26 asciiLets)
"zyxwvutsrqponmlkjihgfedcba"

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

> let isAsciiLet c = c `elem` asciiLets
isAsciiLet :: Char -> Bool
```
The Prelude defines `String` as `[Char]`.

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

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

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

```haskell
> :type putStr
putStr :: String -> IO () -- an "action" (more later!)
```

What's the following doing?

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

some words!it :: ()
```
What's the following expression computing?

> \texttt{length [(Data.Char.chr 0)..]}

1114112

Another way:

> \texttt{length ([minBound..maxBound]::[Char])}

1114112
Like most functional languages, Haskell's lists are "cons" lists.

A "cons" list can viewed as having two parts:
- head: a value
- tail: a list of values

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

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

What's the type of the cons operator?

```haskell
> :type (:)
(:) :: a -> [a] -> [a]
```
"cons" lists, continued

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

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

> head c
5

> tail c
[10,20,30]

> let d = tail (tail c)
> d
[20,30]
```
"cons" lists, continued

A cons node can be referenced by multiple cons nodes.

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

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

> let f = 1:c
[1,5,10,20,30]
What are the values of the following expressions?

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

> 1:2
...error...

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

> []:[]
[[[]]

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

> []:[1]
...error...
head and tail visually

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

```haskell
> let a = [5,10,20,30]
> let h = head a
> h
5
> let t = tail a
> t
[10,20,30]
> let t2 = tail (tail t)
> t2
[30]
```
A little on performance

What operations are likely fast with cons lists?
  Get the head of a list
  Get the tail of a list
  Making a new list from a head and tail

What operations are likely slower?
  Get Nth element of a list
  Get length of a list

With cons lists, what does list concatenation involve?
  > let m=[1..10000000]
  > length (m++[0])
  100000001
True or false?

The head of a list is a one-element list.
   False, unless...
   ...it's the head of a list of lists that starts with a one-element list

The tail of a list is a list.
   True

The tail of an empty list is an empty list.
   It's an error!

\[ \text{length (tail (tail x))} = (\text{length x}) - 2 \]
   True

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 whose value is a list with a range of integers:

\[
\text{let } \text{mToN} \ m \ n = [m..n]
\]

\[
\text{mToN} \ 10 \ 15
\[
[10, 11, 12, 13, 14, 15]
\]

Problem: Write a recursive \text{mToN} that uses the cons operator to build up its result.
One solution:

\[
\text{mToN } m \ n \\
| \ m > n = [] \\
| \ otherwise = m : \text{mToN } (m+1) \ n
\]

Evaluation of \text{mToN} 1 3 via substitution and rewriting:

\[
\text{mToN} 1 3 \\
1 : \text{mToN} \ (1+1) \ 3 \\
1 : \text{mToN} 2 3 \\
1 : 2 : \text{mToN} \ (2+1) \ 3 \\
1 : 2 : \text{mToN} 3 3 \\
1 : 2 : 3 : \text{mToN} \ (3+1) \ 3 \\
1 : 2 : 3 : \text{mToN} 4 3 \\
1 : 2 : 3 : []
\]
Let's do `:set +s` to get timing and memory information and try making some lists. Try these:

```haskell
mToN 1 10
let f = mToN
mToN 8 10
f 1 1000
let f = mToN 1
f 1000
let x = f 1000000
length x
take 5 (f 1000000)
```
In 1964 Peter Landin coined the term "syntactic sugar".

A language construct that makes something easier to express but doesn't add a new capability is called syntactic sugar. It simply makes the language "sweeter" for human use.

Two examples from C:

"abc" is equivalent to a char array initialized with
\{'a', 'b', 'c', '\0'\}

a[i] is equivalent to *(a + i)

What's an example of syntactic sugar in Java?
In Haskell a list like \([5, 2, 7]\) can be expressed as \(5:2:7:[]\).

Is the square-bracket list literal notation syntactic sugar?

What about \([1..], [1,3..], ['a..'z']\)?

The \textbf{Enum} type class has \texttt{enumFrom}, \texttt{enumFromTo}, etc.

"Syntactic sugar causes cancer of the semicolon."

—Alan J. Perlis.

Another Perlis quote:

"A language that doesn't affect the way you think about programming is not worth knowing."
Sidebar: An interesting expression

What does the following expression mean?

```haskell
let x = 1:x
```

One reading: `x` is a list whose head is `1` and tail is `x`.

Step by step evaluation with rewriting:

```
x
1:x
1:1:x
1:1:1:x
...
```

What are the first 10 elements of `x`?

```haskell
> take 10 x
[1,1,1,1,1,1,1,1,1,1]
```
Problem: write a function \( f \) that generates the integers starting at a given value.

\[
> \text{take 10 (f 1)} \\
[1,2,3,4,5,6,7,8,9,10]
\]

\[
> \text{take 10 (f (-100))} \\
[-100,-99,-98,-97,-96,-95,-94,-93,-92,-91]
\]

One solution:

\[
\text{let } f \ n = n:f \ (n+1)
\]
Here's a peek at Lisp's lists, via \texttt{ESC-x ielm} in Emacs:

\begin{quote}
\texttt{ELISP} > (setq x (cons 1 '(10 "twenty" 30.0)))
(1 10 "twenty" 30.0)
\end{quote}

\begin{quote}
\texttt{ELISP} > (car x) \hspace{1cm} \textit{contents of address part of register}
1
\end{quote}

\begin{quote}
\texttt{ELISP} > (cdr x) \hspace{1cm} \textit{contents of data part of register}
\hspace{1cm} \textit{—say "could-er"}
(10 "twenty" 30.0)
\end{quote}

\begin{quote}
\texttt{ELISP} > (caddr x) \hspace{1cm} \textit{Speculate: What does caddr mean?}
"twenty"
\end{quote}
Example: Summation

Here's a function that computes the sum of a list's elements:

\[
\text{sumElems} \text{ list} \\
| \text{null list} = 0 \\
| \text{otherwise} = \text{head list} + \text{sumElems} (\text{tail list})
\]

Usage:

\[
> \text{:type sumElems} \\
\text{sumElems :: Num a => [a] -> a}
\]

\[
> \text{sumElems} [1..100] \\
5050
\]

It works but it's not idiomatic Haskell. We should use *patterns* instead!
Patterns
In Haskell we can use patterns to assign names to elements of data structures like lists.

> let \([x,y]\) = \([10,20]\)
> x
10
> y
20

> let \([inner]\) = \([[[2,3]]]\)
> inner
\([2,3]\)

Speculate: Given a list like \([10,20,30]\) how could we use a pattern to assign names to the head and tail of the list?
We can use the cons operator in a pattern.

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

> h
10

> t
[20,30]
```

What values get bound by the following pattern?

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

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

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

No binding is done for the wildcard pattern.

This mechanism is completely general—patterns can be arbitrarily complex.
A name can only appear once in a pattern. This is invalid:

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

When using `let` as we are here, a failed pattern isn't manifested until we try to see the value that was bound.

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

Recall our non-idiomatic `sumElems`:

\[ \text{sumElems } \text{list} \]
\[ \quad | \text{null list } = 0 \]
\[ \quad | \text{otherwise } = \text{head list } + \text{sumElems} \text{ (tail list)} \]

Here is an idiomatic version, using patterns:

\[ \text{sumElems} [] = 0 \]
\[ \text{sumElems} (h:t) = h + \text{sumElems} t \]

Note that `sumElems` appears on both lines and that there are no guards.

The parentheses in `(h:t)` are required!!
Here's a buggy version of `sumElems`:

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

What's the bug?

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

If we use `ghci -Wall`, we'll get a warning when `:load`ing:

```
slides.hs:62:1:Warning:
  Pattern match(es) are non-exhaustive
In an equation for `buggySum': Patterns not matched: []
```
Recursive functions on lists
Simple recursive list processing functions

Problem: Write \texttt{len \textit{x}}, which returns the length of list \textit{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} \texttt{x}, which returns a list having only the odd numbers in the list \texttt{x}.

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

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

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

odds (h:t)
  | odd h = h:odds t
  | otherwise = odds t
\end{verbatim}
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
isElem x (h:t)
    | x == h = True
    | otherwise = x `isElem` t
```
Simple list functions, continued

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

```haskell
> maxVal "maximum"
'x'

> maxVal [3,7,2]
7

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

Solution:

```haskell
maxVal [x] = x
maxVal (x1:x2:xs)
  | x1 >= x2 = maxVal (x1:xs)
  | otherwise = maxVal (x2:xs)
maxVal [] = undefined -- added after copies
```
Sidebar: strlen in C

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

Python programmers: In a functional style write `size(x)`, which returns the number of elements in the string or list `x`. restriction: You may not use `type()`.
Tuples
A Haskell tuple is an ordered aggregation of two or more values of possibly differing types.

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

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

What's something we can represent with a tuple that we can't represent with a list?
A function can return a tuple:

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

What's the type of `pair`?

\[
\text{pair :: } t \rightarrow t1 \rightarrow (t, t1)
\]

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

\[
> \text{let } p = \text{pair } 30 \ "\text{forty}\"
\]

\[
p :: (\text{Integer}, \text{[Char]})
\]

\[
> \text{fst } p
\]

\[
30
\]

\[
> \text{snd } p
\]

\[
"\text{forty}\"
\]
Tuples, continued

Recall that patterns used to bind names to list elements have the same syntax as expressions to create lists. Patterns for tuples are like that, too.

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

> middle ("372", "BIOW 301", "Mitchell")
"BIOW 301"

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

Solution:

> let middle (_, x, _) = x  -- What's the type of `middle`?
middle :: (t, t1, t2) -> t1
Here's the type of `zip` from the Prelude:

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

Speculate: What do you think `zip` does?

\[
> \text{zip \{"one","two","three"\} [10,20,30]}
> [("one",10),("two",20),("three",30)]
\]

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

What's especially interesting about the second example?
Problem: Write \texttt{elemPos}, which returns the zero-based position of a value in a list, or -1 if not found.

\begin{verbatim}
> elemPos 'm' ['a'..'z']
12
\end{verbatim}

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

Solution:

\begin{verbatim}
elemPos x vals = elemPos' x (zip vals [0..])

elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps
\end{verbatim}
Consider these two functions:

\[
\begin{align*}
&\text{> let } \text{add}_c \ x \ y = x + y \quad -- \_c \text{ for curried arguments} \\
&\text{add}_c :: \text{Num} \ a =\Rightarrow a \to a \to a
\end{align*}
\]

\[
\begin{align*}
&\text{> let } \text{add}_t (x,y) = x + y \quad -- \_t \text{ for tuple argument} \\
&\text{add}_t :: \text{Num} \ a =\Rightarrow (a, a) \to a
\end{align*}
\]

Usage:

\[
\begin{align*}
&\text{> add}_c 3 \ 4 \\
&7
\end{align*}
\]

\[
\begin{align*}
&\text{> add}_t (3,4) \\
&7
\end{align*}
\]

Which is the better way to define such a function, \texttt{add}_c or \texttt{add}_t?
The **Eq** type class and tuples

`:info Eq` shows many lines like this:

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

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

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

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

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

We can write a function that handles a list of any length but a function that operates on a tuple specifies the arity of that tuple. Example: we can't write an analog for `head`, to return the first element of any an arbitrary tuple.

Even if values are homogeneous, using a tuple lets static type-checking ensure that an exact number of values is being aggregated. Example: A 3D point could be represented with a 3-element list but using a 3-tuple guarantees points have three coordinates.

If there were Head First Haskell it would no doubt have an interview with List and Tuple, each arguing their own merit.
More on patterns and functions
Earlier in the slides the general form of a function definition was shown as this: \textit{name \, arg1 \, arg2 \, ... \, argN = expression}

This is more accurate:

\[
\begin{align*}
\text{name \, pattern1 \, pattern2 \, ... \, patternN} \\
\text{guard1 = expression1} \\
... \\
\text{guardN = expression N}
\end{align*}
\]

For a given \textit{name}, any number of declarations like the above may be specified. The set of declarations for a given name is the \textit{binding} for that name. (See 4.4.3 in H10 for formal details.)

If values in a call match the pattern(s) for a declaration and the guard is true, the corresponding expression is evaluated. (Note that currying muddies the meaning of "a call" but we'll stay clear of that tarpit!)
Literal values can be part or all of a pattern. Example:

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

Usage:

\[
\begin{align*}
  > f\ 1 \\
  10
\end{align*}
\]

\[
\begin{align*}
  > f\ 3 \\
  3
\end{align*}
\]

Patterns are tried in the order specified.

Here's `factorial` with literals in patterns instead of guards:

\[
\begin{align*}
  \text{factorial}\ 0 &= 1 \\
  \text{factorial}\ n &= n \times \text{factorial}\ (n - 1)
\end{align*}
\]
not is a function:
  > :type not
  not :: Bool -> Bool

  > not True
  False

Problem: Using literals in patterns, define not.

Solution:
  not True = False
  not _ = True  -- Using wildcard avoids comparison
Pattern construction

A pattern can be:

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

Note the recursion.

Patterns can be arbitrarily complicated.

3.17.1 in H10 shows the full syntax for patterns.
The \texttt{where} clause for functions

Intermediate values and/or helper functions can be defined using the optional \texttt{where} clause for a function.

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

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

The names \texttt{a} and \texttt{b} are bound to expressions; \texttt{g} is a function binding.

The bindings in the \texttt{where} clause are done first, then the guards are evaluated in turn.

Like variables defined in a method or block in Java, \texttt{a}, \texttt{b}, and \texttt{g} are not visible outside the declaration.
Imagine a function that counts occurrences of even and odd numbers in a list.

```haskell
> countEO [3,4,5]
(1,2)
```

Code:

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

Would it be awkward to write it without using `where`? (Try it!)
Imagine a function that returns every Nth value in a list:

```haskell
> everyNth 2 [10,20,30,40,50]
[20,40]
> everyNth 3 ['a'..'z']
"cfilorux"
```

Can we write this without a helper function?

We could use `zip` to pair elements with positions to know that 30 is the third element, for example.

```haskell
> let everyNth n xs = helper n (zip xs [1..])

[(10,1),(20,2),(30,3),(40,4),(50,5)]
```

To learn a different technique, let's not use `zip`.
Let's write a version of `everyNth` that has an extra parameter: the one-based position of the head of the list:

\[
\text{everyNthWithPos } n \ (x:xs) \ pos \\
\quad \mid \ pos \ `\text{rem}` \ n \ == \ 0 = x : \text{everyNthWithPos } n \ xs \ (pos + 1) \\
\quad \mid \ otherwise = \text{everyNthWithPos } n \ xs \ (pos + 1)
\]

\[
\text{everyNthWithPos _ [] pos = []}
\]

We then write `everyNth`:

\[
\text{everyNth } n \ xs = \text{everyNthWithPos } n \ xs \ 1
\]

everyNth 2 [10,20,30,40,50] would lead to these calls:

- `everyNthWithPos 2 [10,20,30,40,50] 1`
- `everyNthWithPos 2 [20,30,40,50] 2`  -- `2 \rem 2 == 0`
- `everyNthWithPos 2 [30,40,50] 3`
- `everyNthWithPos 2 [40,50] 4`  -- `4 \rem 2 == 0`
- `everyNthWithPos 2 [50] 5`
Let's rewrite using \texttt{where} to conceal \texttt{everyNthWithPos}:

\begin{verbatim}
 everyNth n xs = everyNthWithPos n xs 1
 where {
   everyNthWithPos _ [] pos = [];
   everyNthWithPos n (x:xs) pos
     | pos `rem` n == 0 = x : everyNthWithPos n xs (pos+1)
     | otherwise = everyNthWithPos n xs (pos+1)
 }
\end{verbatim}

Just like a Java private method, \texttt{everyNth} can't be accessed outside the body of \texttt{everyNthWithPos}.

The code works, but it's repetitious! How can we improve it?
everyNth \( n \) \( xs \) = everyNthWithPos \( n \) \( xs \) \( 1 \)
where {
  everyNthWithPos _ [] pos = [];
  everyNthWithPos n (x:xs) pos
    | pos `rem` n == 0 = x : everyNthWithPos n xs (pos+1)
    | otherwise = everyNthWithPos n xs (pos+1) 
}

Let's use another \texttt{where} to bind the recursive call to \texttt{rest}.

everyNth \( n \) \( xs \) = everyNthWithPos \( n \) \( xs \) \( 1 \)
where {
  everyNthWithPos _ [] pos = [];
  everyNthWithPos n (x:xs) pos
    | pos `rem` n == 0 = x : rest
    | otherwise = rest
    where { rest = everyNthWithPos n xs (pos+1) }
}
Continuation with indentation

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

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

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

```
add
  ::
  Int->Int->Int
add x y
  =
    x
    + y
```
A line that starts in the same column as the previous declaration ends that declaration and starts a new one.

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

```hs
% runghc indent2.hs
indent2.hs:3:1: parse error (possibly incorrect indentation or mismatched brackets)
```

Note that 3:1 indicates line 3, column 1.
The *layout rule* for **where** (and more)

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

\[
f \ x = a + b + g \ a \ where \{ \ a = 1; \ b = 2; \ g \ x = -x \} \]

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

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

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

What's different about the second version?
At hand:
\[
f \ x = a + b + g \ a \\
\text{where} \\
\quad a = 1 \\
\quad b = 2 \\
\quad g \ x = \\
\quad \quad -x
\]

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

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

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

Note that the declaration of \textbf{g} is continued onto a second line; if the minus sign were at or left of the line, it would be an error.
Don't confuse the layout rule with indentation-based continuation of declarations!

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

Indentation-based continuation applies

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

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

TAB characters are assumed to have a width of 8.

What other languages have rules of a similar nature?
What's this function doing?

\[
f \ a = g
\]

where

\[
g \ b = a + b
\]

Type?

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

Interaction:

\[
> \ \text{let} \ a' = 10 \\
> f \ a' 20 \\
> 30
\]
Consider this claim:
A function definition in curried form, which is the norm in Haskell, is really just syntactic sugar.

Compare:
add x y = x + y
add a = add' where add' b = a + b

Challenge: Write \( \text{add3} :: \text{Num } t \Rightarrow t \to t \to t \to t \to t \) with this do-it-yourself currying.
Errors
What syntax errors do you see in the following file?

```haskell
% cat synerrors.hs
let f x =
  | x < 0 == y + 10
  | x != 0 = y + 20
  otherwise = y + 30
where
  g x:xs = x
  y =
    g [x] + 5
  g2 x = 10
```
What syntax errors do you see in the following file?

```haskell
% cat synerrors.hs

let f x =
    | x < 0 == y + 10
    | x != 0 = y + 20
    otherwise = y + 30

where
    g x:xs = x
    y =
    g [x] + 5
g2 x = 10
```

- no `let` before functions in files
- no `=` before guards
- `=` is not `==`
- before result
- `|` before `otherwise`
- use `/=` for inequality
- continuation should be indented
- violates off-side rule

Needs parens: `(x:xs)`

CSC 372 Spring 2014, Haskell Slide 171
Syntax errors, continued

Line and column information is included in syntax errors.

% cat synerror2.hs
weather temp  | temp >= 80 = "Hot!"
| temp >= 70   "Nice"
| otherwise = "Cold!"

% ghci synerror2.hs
...
[1 of 1] Compiling Main ( synerror2.hs, interpreted )

synerror2.hs:3:14: parse error on input `|'

3:14 indicates an error has been detected at line 3, column 14.

What's the error?
If only concrete types are involved, type errors are typically easy to understand.

```haskell
> chr 'x'
<interactive>:9:5:
  Couldn't match expected type `Int' with actual type `Char'
  In the first argument of `chr', namely 'x'
  In the expression: chr 'x'
  In an equation for `it': it = chr 'x'

> :type chr
chr :: Int -> Char
```
Type errors, continued

Code:

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

What's the error?

```
Couldn't match expected type `(t3, t4)'
    with actual type `[[t0] -> (t1, t2)]'
```

In the expression: `countEO`

In a pattern binding: `(evens, odds) = countEO`

What's the problem?

It's expecting a tuple, (t3, t4) but it's getting a function, [t0] -> (t1, t2)
How about this one?

> length

No instance for (Show ([a0] -> Int)) arising from a use of `print'

Possible fix: add an instance declaration for (Show ([a0] -> Int))

In a stmt of an interactive GHCi command: print it

> :type print

print :: Show a => a -> IO ()

Typing an expression at the ghci prompt causes it to be evaluated and print called with the result. The (trivial) result here is a function, and functions aren't in the Show type class.
Type errors, continued

Code and error:

```haskell
f x y
  | x == 0 = []
  | otherwise = f x
```

Couldn't match expected type `\[a1\]' with actual type `t0 -> \[a1\]'
In the return type of a call of `f'
Probable cause: `f' is applied to too few arguments
In the expression: f x

The error message is perfect in this case but in general note that an unexpected actual type that's a function suggests too few arguments are being supplied for some function.
Type errors, continued

Is there an error in the following?

\[
\begin{align*}
  f [] &= [] \\
  f [x] &= x \\
  f (x:xs) &= x : f xs
\end{align*}
\]

\textbf{Occurs check: cannot construct the infinite type:} \(a0 = [a0]\) \hspace{1em} ("\textit{a0 is a list of a0s}''--whm)

In the first argument of `(:)', namely `x'

In the expression: \(x : f xs\)

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

Without the second pattern, it turns into an identity function on lists:
\[
f [1,2,3] == [1,2,3]
\]

What's the problem?

Technique: Comment out cases to find the troublemaker.
What's happening here?

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

```haskell
> ord 5
<interactive>:2:5:
    No instance for (Num Char) arising from the literal `5'
    Possible fix: add an instance declaration for (Num Char)
```

Why does that error cite (Num Char)? It seems to be saying that if Char were in the Num type class the expression would be valid.
Larger examples
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).

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```haskell
> travel "nnnn"    -- ends at 1
   "Got lost"

> travel "nenene"  -- ends at 2
   "Got lost"

> 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)
Two helpers:

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

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

```haskell
travel s
    | disp == (0,0) = "Got home"
    | otherwise = "Got lost"
where
    makeTuples [] = []
    makeTuples (c:cs) = mapMove c : makeTuples cs

    tuples = makeTuples s
    disp = sumTuples tuples
```

As is, `mapMove` and `sumTuples` are at the top level but `makeTuples` is hidden inside `travel`. How should they be arranged?
Sidebar: top-level vs. hidden functions

```haskell
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)

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

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

The functions at left are hidden in the `where` block but they can easily be changed to top-level using a shift or two with an editor.
Consider a function `tally` that counts character occurrences in a string:

```haskell
tally "a bean bag"
```

```
> a 3
b 2
g 1
n 1
e 1
```

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

How can this problem be approached?
incEntry c tups

tups is a list of (Char, Int) tuples that indicate how many times a character has been seen.
incEntry produces a copy of tups with the count in the tuple containing the character c incremented by one.
If no tuple with c exists, one is created with a count of 1.

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

incEntry c [] = [(c, 1)]
incEntry c ((char, count):entries)
  | c == char = (char, count+1) : entries
  | otherwise = (char, count) : incEntry c entries

Note that we're including an explicit type specification for this function. What's the merit of it? Should it be more general?
Calls to \texttt{incEntry} with 't', 'o', 'o':

\begin{verbatim}
> incEntry 't' []
[('t',1)]

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

> incEntry 'o' it
[('t',1),('o',2)]
\end{verbatim}
-- mkentries s calls incEntry for each character
-- in the string s

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

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

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

tally, continued

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

> sort it
[('o',3),('c',2),('n',1)]
tally, continued

{- fmt_entries prints (Char, Int) tuples one per line -}
fmt_entries [] = ""
fmt_entries ((c, count):es) =
    [c] ++ " " ++ (show count) ++ "\n" ++ fmt_entries es

{- grand finale -}
tally s = putStrLn (fmt_entries (sort (mkentries s)))

> tally "cocoon"
  o 3
  c 2
  n 1

[Added post-copies]
  • How does this solution exemplify functional programming? (slide 23)

  • How is it like imperative programming?

  • How is it like procedural programming (s. 5)
Running **tally** from the command line

Let's run it on lectura...

```bash
% code=/cs/www/classes/cs372/spring14/code

% cat $code/tally.hs
... everything we've seen before and now a main:
main = do
    bytes <- getContents -- reads all of standard input
    tally bytes

% echo -n cocoon | runghc $code/tally.hs
  o 3
  c 2
  n 1
```
tally from the command line, continued

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

% \$\text{code/genchars} \, 20
KVQaVPEmClHRbgdkmMsQ

Lets tally a million characters:

% \$\text{code/genchars} \, 1000000 \, | \\
\text{time runghc} \, \text{\$code/tally.hs} \, >\text{out}
21.79\text{user} \, 0.24\text{system} \, 0:22.06\text{elapsed}
% \text{head -3 out}
s 19553
V 19448
J 19437
tally from the command line, continued

Let's try a compiled executable.

% ghc --make -rtsopts tally.hs

% ls -l tally
-rwxrwxr-x 1 whm whm 1118828 Feb 1 22:41 tally

% $code/genchars 1000000 | time ./tally +RTS -K40000000 -RTS >out
7.44user 0.29system 0:07.82elapsed 98%CPU

Back to 123 for "syntactic sugar"!

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

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

This describes a list of the squares of \( x \) where \( x \) takes on each of the values from 1 through 10.

The portion \( x <- [1..10] \) is a *generator*.

Multiple generators can be specified:

```plaintext
> [(x,y) | x <- [1..3], y <- "ab"]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Which generator varies most rapidly?
Examples from Hutton

Page 39 (5.2) in *Programming in Haskell* by Hutton has some interesting examples of computations with comprehensions:

```haskell
> let firsts pairs = [x | (x,_) <- pairs]
> firsts [(1, 'x'),(5, 'y'),(10, 'z')]
[1,5,10]

> let len xs = sum [1 | x <- xs]
> len "test"
4

> let concat xss = [x | xs <- xss, x <- xs]
> concat ["just", "a", "test"]
"justatest"
```
This is the general form of a list comprehension:

\[ [ \text{expr} \mid \text{qualifier}_1, \ldots, \text{qualifier}_N ] \]

Qualifiers can be one of three things:

- \( \text{pattern} \leftarrow \text{expr} \) (a generator)
- \text{let declarations}
- \text{guard}

Guards are boolean expressions that act as filters. If a guard is false, the value at hand is discarded and the previous generator produces its next value, if any. Example:

\[
> \text{let justDigits s = [ch | ch <- s, isDigit ch]}
> \text{justDigits "(800) 555-1212"}
"8005551212"
\]
Primes via comprehensions

Hutton has an interesting example of computing primes with list comprehensions. (Section 5.2)

First he defines a function to compute the factors of a number:

```haskell
> let factors n = [x | x <- [1..n], n `mod` x == 0]
> factors 20
[1,2,4,5,10,20]
> factors 7
[1,7]
```

If a number's only factors are 1 and the number, it's a prime.

```haskell
> let prime n = factors n == [1,n]
> prime 7
True
> prime 20
False
```
At hand:

```
factors n = [x | x <- [1..n], n `mod` x == 0]
prime n = factors n == [1,n]
```

Now we can write a function to generate primes starting at any number:
```
> let primes n = [p | p <- [n..], p > 1, prime p]
```
```
> take 15 (primes 2)
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```
```
> take 5 (primes 1000000)
[1000003,1000033,1000037,1000039,1000081]
```

**Added**: On your own, look at quicksort, p. 57 in GG.
Higher-order functions
Functions as values

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

This `let` creates a function value and binds the name `add` to that value.

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

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

```haskell
> let plus = add
```

Either of the names can be used to reference the function value:

```haskell
> add 3 4
7
> plus 5 6
11
```
Sidebar: printing function values

By default, if an expression evaluated at ghci's REPL prompt produces a function, we get an error because function values aren't in the Show type class.

> let add x y = x + y

> add
No instance for (Show (a0 -> a0 -> a0) arising from a use of `print`

> add 3
No instance for (Show (a0 -> a0)) arising from a use of `print`
Sidebar, continued

Contrast: With **Chrome's JavaScript console**, we can see the code for some functions:

```javascript
function add(x,y) { return x + y }
undefined

> typeof(add)
"function"

> add
function add(x,y) { return x + y }

> alert
function alert() { [native code] }
```
Confession: I wish I'd found the `Text.Show.Functions` module sooner. It's a trivial `instance` declaration:

```haskell
instance Show (a -> b) where showsPrec _ _ = showString "<function>"
```

Usage:

```haskell
> :m Text.Show.Functions
> let add x y = x + y
> add
<function>
> it 3
<function>
> it 5
8
```

Sidebar, continued

I've added this to my `~/.ghci`:

```haskell
:mk Text.Show.Functions
```

You should add it to yours, too!

Trouble with `:m Data.Char`!
Functions as values, continued

Line by line, what are the following expressions doing?

> let fs = [head, last]

> fs
[<function>,<function>]

> let ints = [1..10]

> head fs ints
1

> (fs!!1) ints
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 it valid by adding two characters.

> [take 5, tail, init]

[<function>,<function>,<function>]

CSC 372 Spring 2014, Haskell Slide 207
A **higher-order function** is a function that:
- Has one or more arguments that are functions.
- And/or returns a function.

Have we seen any functions thus far that are higher-order functions?

Strictly speaking, any curried function with more than one argument meets the above definition.

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

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

We'll ignore this fine point wrt. the a1 prohibition on higher-order functions!
Here's a simple higher-order function:
\[
\text{twice } f \ x = f \ (f \ x)
\]

What does it do?
\[
> \text{twice tail } [1..5] \\
[3,4,5]
\]

What is its type?
\[
\text{twice } :: (t -> t) -> t -> t
\]

What's going on here?
\[
> \text{twice tail} \\
<\text{function}> \\
> \text{it } [1..5] \\
[3,4,5]
\]

A \textit{higher-order function} is a function that:
- Has one or more arguments that are functions.
- And/or returns a function.

Note: line missing on handout!
A simpler higher-order function

Here's a simpler higher-order function. What does it do?

\[
\text{apply } f \ x = f \ x
\]

What's its type? Translate it to English.

\[
\text{apply :: (} t1 \ \rightarrow \ t \text{)} \rightarrow \ t1 \rightarrow t
\]

Usage:

\[
> \text{apply head } [5,6,7] \\
5
\]

\[
> \text{apply negate it} \\
-5
\]

\[
> \text{apply length } [\text{'a'..'z'}] \\
26
\]

In what other languages could we write \text{apply}?
Recall `double x = x * 2`

`map` is a Prelude function that applies a function to each element of a list, producing a new list:

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

> map length (words "a few words")
[1,3,5]

> map head (words "a few words")
"afw"
```

Let's write `map` ourselves!
Solution:

\[
\text{map } [] = [] \\
\text{map } f (x:xs) = f x : \text{map } f xs
\]

What is its type?

\[
\text{map} :: (t -> a) -> [t] -> [a]
\]

What's the relationship between the length of the argument and the result?

Is \text{map} a higher order function?
A few more maps...

> map chr [97,32,98,105,103,32,99,97,116] "a big cat"

> map isLetter it
[True,False,True,True,True,True,True,True,True,True]

> map not it
[False,True,False,False,False,True,False,False,False,False]

> map head (map show it) -- show True is "True"
"FTFFFTFFFF"

> map weather [85,55,75]
["Hot!","Cold!","Nice"]
map and partial applications

What's the result of these?

> map (add 5) [1..10]
[6,7,8,9,10,11,12,13,14,15]

> map (drop 1) (words "the knot was cold")
["he","not","as","old"]

> map (replicate 5) "abc"
["aaaaa","bbbbbb","cccccc"]

> let f = map double
> f [1..5]
[2,4,6,8,10]

> map f [[1..3],[10..15]] -- same as map (map double) ...
[[2,4,6],[20,22,24,26,28,30]]
Instead of using `map (add 5)` to add 5 to the values in a list, we can use a `section` instead:

```haskell
> map (5+) [1..3]
[6,7,8]
```

More sections:

```haskell
> map (10*) [1..3]
[10,20,30]
```

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

```haskell
> map ("*"++) (words "a few words")
["*a","*few","*words"]
```
Sections have one of two forms:

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

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

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

\[\text{map (3\leq) \[1..5\]}\]
\[\text{[False, False, True, True, True]}\]

\[\text{map (\leq3) \[1..5\]}\]
\[\text{[True, True, True, False, False]}\]

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

\[\text{twice (+5) 3}\]
\[13\]
Another higher order function in the Prelude is **filter**:

\[
\begin{align*}
> \text{filter odd [1..10]} \\
[1,3,5,7,9]
\end{align*}
\]

\[
\begin{align*}
> \text{filter isDigit "(800) 555-1212"} \\
"8005551212"
\end{align*}
\]

What's **filter** doing? What is its type?

\[
\text{filter} :: (a -> \text{Bool}) -> [a] -> [a]
\]

Two more:

\[
\begin{align*}
> \text{filter (`elem` "aeiou") "some words here"} \\
"oeoee" \quad \text{*Note that (`elem` ...) is a section!*}
\end{align*}
\]

\[
\begin{align*}
> \text{filter (5 >=) (filter odd [1..10])} \\
[1,3,5]
\end{align*}
\]
At hand:

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

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

Let's write filter!

myfilter _ [] = []
myfilter f (x:xs)
  | f x = x : filteredTail
  | otherwise = filteredTail
where
  filteredTail = myfilter f xs
filter uses a predicate

filter's first argument (a function) is called a predicate because inclusion of each value is predicated on the result of calling that function with value.

Several Prelude functions use predicates. Here are two:

all :: (a -> Bool) -> [a] -> Bool
> all even [2,4,6,8]
True
> all even [2,4,6,7]
False

dropWhile :: (a -> Bool) -> [a] -> [a]
> dropWhile isSpace " testing 
"testing 
" "
> dropWhile isLetter it
" "

CSC 372 Spring 2014, Haskell Slide 219
Problem: longerThan

Write a non-recursive function `longerThan n lists` that produces the lists in `lists` that have more than `n` elements.

```
> longerThan 2 [[1..3],[10,20],[5..8]]
[[1,2,3],[5,6,7,8]]
```

```
> longerThan 3 (words "up and down it went")
["down","went"]
```

How can we approach it? (non-recursive helpers are allowed)
We could **map** with **length** and then **filter** with a section, but then we lose the lists themselves:

```
> let lists = [[1..3],[10,20],[5..8]]

> map length lists
[3,2,4]

> filter (>2) it
[3,4]

> filter (>3) (map length (words "up and down it went"))
[4,4]
```

"I can't tell what they are but I can tell you how many we got."
Let's write a helper:

\[
\text{isLonger} :: \text{Int} \rightarrow [a] \rightarrow \text{Bool} \\
\text{isLonger} \ n \ \text{list} = \text{length} \ \text{list} > n
\]

But how can we use it? \text{filter} needs an \((a \rightarrow \text{Bool})\) predicate:

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

This is a perfect place for a partial application:

\[
> :\text{type} \ \text{isLonger} \ 3 \\
\text{isLonger} \ 3 :: [a] \rightarrow \text{Bool} \\
> \ \text{filter} (\text{isLonger} \ 3) \ \text{(words "up and down it went"}) \\
\text{["down","went"]}
\]

A final solution:

\[
\text{isLonger} \ n \ \text{list} = \text{length} \ \text{list} > n \\
\text{longerThan} \ n \ \text{lists} = \text{filter} \ (\text{isLonger} \ n) \ \text{lists}
\]
Recall:

\[
isLonger \; n \; list = \text{length} \; list > n
\]
\[
\text{longerThan} \; n \; \text{lists} = \text{filter} \; (\text{isLonger} \; n) \; \text{lists}
\]

A solution that takes advantage of scoping rules:

\[
\text{longerThan2} \; n \; \text{lists} = \text{filter} \; \text{isLonger} \; \text{lists}
\]

\[
\text{where}
\]
\[
isLonger \; \text{list} = \text{length} \; \text{list} > n
\]

In the latter case, \text{isLonger} must be in the \text{where} to have access to the argument \text{n}.
Anonymous functions

We can map a section to double the numbers in a list:

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

Alternatively we could use an \textit{anonymous function}:

\[
> \text{map} \ (\lambda x \rightarrow x \times 2) \ [1..5] \\
[2,4,6,8,10]
\]

What are things we can do with an anonymous function that we can't do with a section?

\[
> \text{map} \ (\lambda n \rightarrow n \times 3 + 7) \ [1..5] \\
[10,13,16,19,22]
\]

\[
> \text{filter} \ (\lambda x \rightarrow \text{head} \ x == \text{last} \ x) \ (\text{words} \ "\text{pop top suds}"") \\
["\text{pop","suds}""]
\]
Anonymous functions, continued

The general form:
\( \backslash \text{pattern1} \ldots \text{patternN} \rightarrow \text{expression} \)

Simple syntax rule: enclose the works in parentheses.

The typical use case for an anonymous function is a single instance of supplying a higher order function with a computation that can't be expressed with a section or partial application.

Anonymous functions are also called *lambda, lambda expressions*, and *lambda abstractions*.

The \ character was chosen due to its similarity to \( \lambda \), used in Lambda calculus, another system for expressing computation.
Example: longest line(s) in a file

Imagine a program to print the longest line(s) in a file, along with their line numbers:

```
% runghc longest.hs /usr/share/dict/web2
72632:formaldehydesulphoxylate
140339:pathologicopsychological
175108:scientificophilosophical
200796:tetraiodophenolphthalein
203042:thyroparathyroidectomize
```

What are some ways in which we could approach it?
Let's work with a shorter file for development testing:

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

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

```haskell
> readFile "longest.1"
"data\nto\ntest\n"
```

To avoid wading into I/O yet, let's focus on a function that operates on a string of characters (the full contents of a file):

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

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

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

Note: To save space, values of `let` bindings are being shown immediately after each `let`. E.g., `> lns` is not shown above.

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

```haskell
> let 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 `sort :: Ord a => [a] -> [a]` on them:
> let sortedTriples = reverse (sort triples)
  [(4,3,"test"), (4,1,"data"), (2,2,"to")]

Note that by having the line length first, triples are sorted first by line length, with ties resolved by line number.

We use `reverse` to get a descending order.

If line length weren't first, we'd instead use

```haskell
Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

and supply a function that returns an `Ordering`.
At hand:

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

We'll handle ties by using `takeWhile` to get all the triples with lines of the maximum length.

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

```haskell
> let first (len, _, _) = len
> let maxLength = first (head sortedTriples)
4
```

`first` will be used in another place but were it not for that we might have used an anonymous function:

```haskell
> let maxLength = (\(len, _, _\) -> len) (head sortedTriples)
```

a pattern: `let (maxLength, _, _) = head sortedTriples`
At hand:

\[
\begin{array}{l}
\text{sortedTriples} = [(4,3,"test"),(4,1,"data"),(2,2,"to")],
\text{maxLength} = 4
\end{array}
\]

Let's use \texttt{takeWhile :: (a -> Bool) -> [a] -> [a]} to get the triples having the maximum length:

\[
\begin{array}{l}
\text{let maxTriples = takeWhile (\triple -> first \triple == maxLength) sortedTriples }
\end{array}
\]

\[
\begin{array}{l}
[(4,3,"test"),(4,1,"data")]
\end{array}
\]
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
> let linesWithNums =
    \(\_\),num,line \to show num ++ ":" ++ line
    \map\ maxTriples
["3:test","1:data"]
```

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

```haskell
> let result = unlines (reverse linesWithNums)
> result
"1: data
3: 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 (sort triples)
        maxLength = first (head sortedTriples)
        maxTriples = takeWhile
            (\triple -> first triple  ==  maxLength)  sortedTriples
        linesWithNums =
            map (\(_,num,line) -> show num ++ "":"" ++ line)
            maxTriples
        result = unlines (reverse linesWithNums)
```

first (x,_,_) = x
At hand:

```
> longest "data\nto\ntest\n"
"1:data\n3:test\n"
```

Let's add a main that does I/O:

```
% cat longest.hs
import System.Environment (getArgs)
import Data.List (sort)

longest bytes = ...from previous slide...

main = do
  args <- getArgs  -- Get command line args as list
  bytes <- readFile (head args)
  putStrLn (longest bytes)
```
Function composition

Given two functions \( f \) and \( g \), the composition of \( f \) and \( g \) is a function \( c \) that for all values of \( x \), \((c \ x)\) equals \((f \ (g \ x))\)

Here is a primitive \texttt{compose} function that applies two functions in turn:

\begin{verbatim}
> let compose f g x = f (g x)

> :type compose
compose :: (t1 -> t) -> (t2 -> t1) -> t2 -> t

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

> compose signum negate 3  
-1
\end{verbatim}
Haskell has a function composition operator. It is a dot (.).

> :t (.)

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

Its two operands are functions, and its result is a function.

> let numwords = length . words

> numwords "just testing this"

3
Problem: Using composition create a function that returns the next-to-last element in list:

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

> ntl "abc"
'b'
```

Here's one solution, but what's another?

```haskell```
> let ntl = head . tail . reverse

> let ntl = head . reverse . init
```

Problem: Create a function to remove the digits from a string:

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

Solution:

> let rmdigits = filter (not . isDigit)

Given the following, describe f:

> let f = (*2) . (+3)

> map f [1..5]
[8,10,12,14,16]

Would an anonymous function be a better choice?
Given the following, what's the type of `numwords`?

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

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

```haskell
> let numwords = length . words
```

Type:

```haskell
numwords :: String -> Int
```

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

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```
Recall `rmdigits`:

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

What the difference between these two declarations?

```haskell
rmdigits s = filter (not . isDigit) s
```

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

The latter declaration is in *point-free style*.

A point-free declaration of a function `f` does not mention the parameter of `f`. (Wording revised after handouts.)

Is the following a point-free function declaration or a partial application?

```haskell
t5 = take 5
```
Problem: Using point-free style, declare a function named \texttt{len} that works like the Prelude's \texttt{length}.

Hint:

\begin{verbatim}
> :t const
const :: a -> b -> a
> const 10 20
 10
> const [1] "foo"
[1]
\end{verbatim}

Solution:

\[
\text{len} = \text{sum} . \text{map} \ (\text{const} \ 1)
\]

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

```haskell
f a = g
  where
    g b = a + b
```

Type?

```haskell
f :: Num a => a -> a -> a
```

Interaction:

```haskell
> let f' = f 10
> f' 20
30
```
Consider this claim:
A function definition in curried form, which is idiomatic in Haskell, is really just syntactic sugar.

Compare these two completely equivalent declarations for \texttt{add}:
\begin{verbatim}
add x y = x + y
add x = add'
  where
  add' y = x + y
\end{verbatim}

The result of the call \texttt{add 5} is essentially this function:
\begin{verbatim}
add' y = 5 + y
\end{verbatim}

The combination of the code for \texttt{add'} and the binding for \texttt{x} is known as a \textit{closure}. It contains what's needed for execution.
DIY currying in JavaScript

JavaScript doesn't provide the syntactic sugar of curried function definitions but we can do this:

```javascript
function add(x) {
    return function (y) { return x + y }
}
```

Try it!

View > Developer > JavaScript Console brings up the console.

Type in the code for `add` on one line.

- `add(5)(3)`
  - 8
- `a5 = add(5)`
  - function (y) { return x + y }
- `[10,20,30].map(a5)`
  - [15, 25, 35]
>>> def add(x):
...     return lambda y: x + y
...

>>> f = add(5)

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

>>> map(f, [10,20,30])
[15, 25, 35]
Here's another mystery function:

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

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

Can you devise a call to `m`?

```haskell
> m add 3 4
7

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

What is it doing? What could it be useful for?
At hand:

\[ m f x y = f y x \]

\( m \) is actually a Prelude function named \texttt{flip}:

\[
\begin{align*}
&\texttt{flip :: (a \to b \to c) \to b \to a \to c} \\
&\texttt{flip take [1..10] 3} \\
&\quad\text{[1,2,3]} \\
&\texttt{let ftake = flip take} \\
&\texttt{ftake [1..10] 3} \\
&\quad\text{[1,2,3]}
\end{align*}
\]

Any ideas on how to use it?
At hand:

\[ \text{flip } f \times y = f \ y \ x \]

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

Problem: write a function that behaves like this:

\[ > f 'a' \]
\[ ["a","aa","aaa","aaaa","aaaaa",...] \]

Solution:

\[ > \text{let } f \ x = \text{map (flip replicate x)} [1..] \]
From assignment 1:

> splits "abcd"

[("a","bcd"),("ab","cd"),("abc","d")]

Many students have noticed the Prelude's `splitAt`:

> splitAt 2 [10,20,30,40]

([10,20],[30,40])

Problem: Write `splits` using higher order functions but no explicit recursion.

Solution:

\[
\text{splits list} = \text{map (flip splitAt list)} [1..(length list - 1)]
\]
The $ operator

$ is the "application operator". Note what :info shows:

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

The following declaration of $ uses an infix syntax:

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

Usage:

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

What's the point of it?
The $ operator, continued

$ is a low precedence, right associative operator that calls a function.

\[ f \; $ \; x = f \; x \]

Because + has higher precedence than $ the expression

\[ \text{negate} \; $ \; (3 + 4) \]

groups like this:

\[ \text{negate} \; $ \; (3 + 4) \]

How does the following expression group?

\[ \text{filter} \; (>3) \; $ \; \text{map length} \; $ \; \text{words} \; "\text{up and down}" \]

\[ \text{filter} \; (>3) \; (\text{map length} \; (\text{words} \; "\text{up and down}")) \]
Currying the uncurried

Problem: We're given a function whose argument is a two-tuple but we wish it were curried so we could use 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 it with \texttt{curry} from the Prelude!

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

Your problem: Write \texttt{curry}!
Currying the uncurried, continued

At hand:
> g (3,4)
77
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]

Here's curry, and use of it:
curry :: ((a, b) -> c) -> (a -> b -> c)  
(latter parens added to help)
curry f x y = f (x,y)

> let cg = curry g
> :type cg
cg :: Int -> Int -> Int

> cg 3 4
77
Currying the uncurried, continued

At hand:

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

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

The key: \text{(curry } g\ 3)\ is a partial application of \text{curry}!

Call: \text{curry } g\ 3

\[
\text{Dcl: curry } f\ x\ y = f\ (x,\ y) \\
= g\ (3,\ y)
\]

CSC 372 Spring 2014, Haskell Slide 255
At hand:

\[
\text{curry} :: ((a, b) \to c) \to (a \to b \to c) \quad (\text{parentheses added})
\]

\[
\text{curry } f \ x \ y = f (x, y)
\]

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

Let's get \textit{flip} into the game!

> map (flip (curry g) 4) [1..10]
[45,60,77,96,117,140,165,192,221,252]

The counterpart of \textit{curry} is \textit{uncurry}:

> uncurry (+) $ (3,4)
7
A **curry** function for JavaScript

```javascript
function curry(f) {
    return function(x) {
        return function (y) { return f(x,y) }
    }
}
```

```javascript
function add(x,y) {return x + y}
undefined

c_add = curry(add)
function (x) { return function (y) { return f(x,y) } }
add_5 = c_add(5)
function (y) { return f(x,y) }
[10,20,30].map(add_5)
[15, 25, 35]
```
Folding
We can *reduce* a list by a binary operator by inserting that operator between the elements in the list:

\[ [1,2,3,4] \text{ reduced by } + \text{ is } 1 + 2 + 3 + 4 \]

\[ ["a","bc","def"] \text{ reduced by } ++ \text{ is } "a" ++ "bc" ++ "def" \]

Imagine a function `reduce` that does reduction by an operator.

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

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

\[ \text{reduce } \text{max} \ [10,2,4] \]
\[ \text{10} \]

\[ \text{map (reduce max) (permutations [10,2,4])} \]
\[ [10,10,10,10,10,10,10] \quad -- \text{permutations is from Data.List} \]
At hand:

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

10

An implementation of `reduce`:

\[
\text{reduce _ [] = undefined}
\]

\[
\text{reduce _ [x] = x}
\]

\[
\text{reduce op (x:xs) = x `op` reduce op xs}
\]

Does `reduce + [1,2,3,4]` do

\[
((1 + 2) + 3) + 4
\]

or

\[
1 + (2 + (3 + 4))
\]

In general, when would the grouping matter?
foldl1 and foldr1

In the Prelude there's no reduce but there is foldl1 and foldr1.

> foldl1 (/) [1,2,3]
0.16666666666666666 -- left associative: (1 / 2) / 3

> foldr1 (/) [1,2,3] -- right associative: 1 / (2 / 3)
1.5

Here's the type of foldr1:
foldr1 :: (a -> a -> a) -> [a] -> a

Here's the type of a related function, foldr (no "1"):
foldr :: (a -> b -> b) -> b -> [a] -> b

What are the differences between the two?
foldr1 vs. foldr

For reference:

foldr1 :: (a -> a -> a) -> [a] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b

Use:

> foldr1 (+) [1..4]
10

> foldr (+) 0 [1..4]
10

> foldr (+) 0 []    -- Empty list is exception with foldr1
0
foldr1 vs. foldr, continued

For reference:

\[
\begin{align*}
\text{foldr1} & : (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\text{foldr} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\end{align*}
\]

The big difference is that \text{foldr} can fold a list of values into a different type!

\[
> \text{foldr } (\lambda \text{elem acm} \rightarrow \text{acm} + \text{elem})\ 0\ [1..4]
10
\]

(acm stands for "accumulated")

\[
> \text{foldr } (\lambda \text{elem acm} \rightarrow \text{show elem ++ acm})\ ""\ [1..4]
"1234"
\]
Folding

Fill in the blank, creating a folding function that can be used to compute the length of a list:

\[
\text{foldr (\ }) 0 \ [10,20,30] \\
3
\]

Solution:

\[
\text{let len = foldr (\ } x \mapsto \text{acm + 1) 0} \\
\text{len ['a..'z']} \\
26
\]

Problem: Define map in terms of foldr.

\[
\text{let mp f = foldr (\ } x \mapsto f x \mapsto x : \text{acm}) [] \\
\text{mp toUpper "test"} \\
"TEST"
\]
Recall our even/odd counter
> countEO [3,4,7,9]
(1,3)

Define it terms of foldr.
> let eo = foldr (\val (e,o) ->
                    if even val then (e+1,o) else (e,o+1)) (0,0)
> eo [3,4,7,9]
(1,3)

> eo []
(0,0)

Strictly FYI: Instead of if/else we could have used Haskell's case:
> let eo = myfoldr (\val (e,o) ->
                    case even val of {True -> (e+1,o); False -> (e,o+1)}) (0,0)
Folding, continued

Here's a definition for foldr. We're using a type specification with multicharacter type variables to help know which is which:

\[
\text{foldr :: (val -> acm -> acm) -> acm -> [val] -> acm}
\]
\[
\text{foldr f acm [] = acm}
\]
\[
\text{foldr f acm (val:vals) = f val $ foldr f acm vals}
\]

When loaded, we see this:
> :t foldr
foldr :: (val -> acm -> acm) -> acm -> [val] -> acm

> foldr (\val acm -> acm ++ val) "?" $ words "a test here"
"?heretesta"

IMPORTANT: There's no connection between the type variable names and the names in functions. We might have done this instead: foldr (\v a -> a ++ v) ...
Problem: Write reverse in terms of a foldr.

Two solutions, but both with issues:

```
rv1 = foldr (\val acm -> acm ++ [val]) []
rv2 = reverse . foldr (:) []  (Oops! This is completely stupid!)
```

The issue: reverse and ++ are relatively expensive wrt. cons.

By definition, foldr operates like this:

```
foldr f zero [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` zero)...)
```

The first application of f is with the last element and the "zero" value, but the first cons would need to be with the first element of the list.
The counterpart of foldr is foldl. Compare their meanings:

\[
\text{foldr } f \text{ zero } [x_1, x_2, \ldots, x_n] = x_1 \cdot f \cdot (x_2 \cdot f \cdot \ldots (x_n \cdot f \cdot \text{zero})\ldots)
\]

\[
\text{foldl } f \text{ zero } [x_1, x_2, \ldots, x_n] = ((\ldots((\text{zero} \cdot f \cdot x_1) \cdot f \cdot x_2) \cdot f \ldots) \cdot f \cdot x_n)
\]

Their types, with long type variables:

\[
\text{foldr} :: (\text{val} \to \text{acm} \to \text{acm}) \to \text{acm} \to [\text{val}] \to \text{acm}
\]

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

Problem: Write reverse in terms of foldl.

\[
> \text{let rev } = \text{foldl } (\backslash \text{acm } \text{val} \to \text{val}:\text{acm}) \; []
\]

\[
> \text{rev } "\text{testing}"
\]

\[
"\text{gnitset}"
\]
Recall \textbf{paired} from assignment 1:
\[
> \text{paired "((())())"}
\]
True

Can we implement \textbf{paired} with a fold?

\[
\begin{align*}
\text{counter} (-1) \_ &= -1 \\
\text{counter total '} &= \text{total} + 1 \\
\text{counter total ')} &= \text{total} - 1 \\
\text{counter total } \_ &= \text{total}
\end{align*}
\]

\[
\text{paired } s = \text{foldl counter 0 } s == 0
\]

Point-free:
\[
\text{paired} = (0==) \cdot \text{foldl counter 0}
\]
Data.List.partition partitions a list based on a predicate:

> partition isLetter "Thu Feb 13 16:59:03 MST 2014"
("ThuFebMST", 13 16:59:03 2014")

> partition odd [1..10]

([1,3,5,7,9],[2,4,6,8,10])

Write it using a fold!

sorter f val (pass, fail) =
    if f val then (val:pass, fail)
    else (pass, val:fail)

partition f = foldr (sorter f) ([],[])
True or false?
  Any operation that processes a list can be expressed in terms of a fold, perhaps with a simple wrapper.

_scans_ are similar to folds but all intermediate values are produced:

\[
> \text{scanl} \ (+) \ 0 \ [1..5] \\
[0,1,3,6,10,15]
\]

\[
> \text{let scanEO} = \text{scanl} \ (\text{\(e,o\)} \ \text{val} \rightarrow \begin{cases} 
\text{if even val then } (e+1,o) & \text{else } (e,o+1) 
\end{cases}) \ (0,0)
\]

\[
> \text{scanEO} \ [1,3,5,6,7,9] \\
[(0,0),(0,1),(0,2),(0,3),(1,3),(1,4),(1,5)]
\]
A little I/O
Consider this function declaration
\[
    f2 \ x = a + b + c
\]

where
\[
    a = f \ x \quad a = f \ x \quad c = h \ x \\
    b = g \ x \quad c = h \ x \quad b = g \ x \\
    c = h \ x \quad b = g \ x \quad a = f \ x
\]

Haskell guarantees that the order of the where clause bindings is inconsequential—those three lines can be in any order.

What enables that guarantee?

(Pure) Haskell functions depend only on the argument value. For a given value of \( x \), \( f \ x \) always produces the same result.

You can shuffle the bindings of any function's where clause without changing the function's behavior! (Try it with longest, slide 233.)
Imagine a `getInt` function, which reads an integer from standard input (e.g., the keyboard).

Can the `where` clause bindings in the following function be done in any order?

```haskell
f x = r
    where
        a = getInt
        b = getInt
        r = a * 2 + b + x
```

The following is not valid syntax but ignoring that, is it reorderable?

```haskell
greet name = ""
    where
        putStr "Hello, "
        putStr name
        putStr "!\n"
```
I/O and sequencing, continued

One way we can specify that operations are to be performed in a specific sequence is to use a `do`:

```hs
% cat io2.hs
main = do
    putStrLn "Who goes there?"
    name <- getline
    let greeting = "Hello, " ++ name ++ "!"
    putStrLn greeting
```

Interaction:

```hs
% runghc io2.hs
Who goes there?
whm (typed)
Hello, whm!
```
Here's the type of `putStrLn`:

```
putStrLn :: String -> IO ()  ("unit", (), is the no-value value)
```

The type `IO x` represents an interaction with the outside world that produces a value of type `x`. Instances of `IO x` are called `actions`.

When an action is evaluated the corresponding outside-world activity is performed.

```
> let hello = putStrLn "hello!"  (Note: no output here!)
hello :: IO ()  (Type of `hello` is an action.)
```

```
> hello
hello!  (Evaluating `hello`, an action, caused output.)
it :: ()
```
The value of `getLine` is an action that reads a line:

```
getLine :: IO String
```

We can evaluate the action, causing the line to be read, and bind a name to the string produced:

```
> s <- getLine
 testing

> s
"testing"
```

Note that `getLine` is not a function!
Recall io2.hs:

```haskell
main = do
  putStrLn "Who goes there?"
  name <- getLine
  let greeting = "Hello, " ++ name ++ "!"
  putStrLn greeting
```

Note the type: `main :: IO ()`. We can say that `main` is an action. Evaluating `main` causes interaction with the outside world.

```
> main
Who goes there?
hello?
Hello, hello?!
```
A pure function (1) always produces the same result for a given argument value, and (2) has no side effects.

Is this a pure function?

```
twice :: String -> IO ()
twice s = do
    putStr s
    putStr s
```

twice "abc" will always produce the same value, an action that if evaluated will cause "abcabc" to be output.
The Haskell solution for I/O

We want to use pure functions whenever possible but we want to be able to do I/O, too.

In general, evaluating an action produces side effects.

Here's the Haskell solution for I/O in a nutshell:

Actions can evaluate other actions and pure functions but pure functions don't evaluate actions.

Recall `longest.hs` from 233-234:

```haskell
longest bytes = result where ...lots...
main = do
  args <- getArgs -- gets command line arguments
  bytes <- readFile (head args)
  putStrLn (longest bytes)
```
User-defined types
A new type can be created with a data declaration.

Here's a simple Shape type whose instances represent circles or rectangles:

```haskell
data Shape =
    Circle Double  -- just a radius
  | Rect Double Double  -- width and height
  deriving Show
```

The shapes have dimensions but no position.

Circle and Rect are data constructors.

"deriving Show" declares Shape to be an instance of the Show type class, so that values can be shown using some simple, default rules.

Shape is called an algebraic type because instances of Shape are built using other types.
Instances of **Shape** are created by calling the data constructors:

```haskell
> let r1 = Rect 3 4
> r1
Rect 3.0 4.0

> let r2 = Rect 5 3

> let c1 = Circle 2

> let shapes = [r1, r2, c1]

> shapes
[Rect 3.0 4.0, Rect 5.0 3.0, Circle 2.0]
```

Lists must be homogeneous—why are both **Rects** and **Circles** allowed in the same list?

```haskell
data Shape =
  Circle Double |
  Rect Double Double
  deriving Show
```
The data constructors are just functions—we can use all our function-fu with them!

```haskell
> :t Circle
Circle :: Double -> Shape

> :t Rect
Rect :: Double -> Double -> Shape

> map Circle [2,3] ++ map (Rect 3) [10,20]
[Circle 2.0,Circle 3.0,Rect 3.0 10.0,Rect 3.0 20.0]
```
Functions that operate on algebraic types use patterns based on their data constructors.

\[
\text{area (Circle } r) = r ** 2 * \pi \\
\text{area (Rect } w \ h) = w * h
\]

Usage:

\[
\begin{align*}
&> \text{ r1} \\
&\text{ Rect 3.0 4.0} \\
&> \text{ area r1} \\
&12.0 \\
&> \text{ shapes} \\
&[\text{Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0}] \\
&> \text{ map area shapes} \\
&[12.0,15.0,12.566370614359172] \\
&> \text{ sum $ map area shapes} \\
&39.56637061435917
\end{align*}
\]
Let's make the **Shape** type an instance of the **Eq** type class.

What does **Eq** require?

```haskell
> :info Eq
class Eq a where
    (==) :: a -> a -> Bool
    (=/=) :: a -> a -> Bool
```

Default definitions from **Eq**:

```haskell
(==) a b = not $ a /= b
(=/=) a b = not $ a == b
```

Let's say that two shapes are equal if their areas are equal. (Iffy!)

```haskell
instance Eq Shape where
    (==) r1 r2 = area r1 == area r2
```

Usage:

```haskell
> Rect 3 4 == Rect 6 2
True

> Rect 3 4 == Circle 2
False
```
Let's see if we can find the biggest shape:
> maximum shapes
No instance for (Ord Shape) arising from a use of `maximum'
Possible fix: add an instance declaration for (Ord Shape)

What's in Ord?
> :info Ord

class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    max :: a -> a -> a
    min :: a -> a -> a

Eq a => Ord a requires would-be Ord classes to be instances of Eq. (Done!)

Like == and /= with Eq, the operators are implemented in terms of each other.
Let's make *Shape* an instance of the *Ord* type class:

```haskell
instance Ord Shape where
  (<) r1 r2 = area r1 < area r2  -- < and <= are sufficient
  (<=) r1 r2 = area r1 <= area r2
```

Usage:

```haskell
> shapes
[Rect 3.0 4.0, Rect 5.0 3.0, Circle 2.0]

> map area shapes
[12.0, 15.0, 12.566370614359172]

> maximum shapes
Rect 5.0 3.0

> Data.List.sort shapes
[Rect 3.0 4.0, Circle 2.0, Rect 5.0 3.0]
```

Note that we didn't need to write functions like `sumOfAreas` or `largestShape`—we can express those in terms of existing operations.
Here's all the Shape code: (in shape.hs)

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

area (Circle r) = r ** 2 * pi
area (Rect w h) = w * h

instance Eq Shape where
    (==) r1 r2 = area r1 == area r2

instance Ord Shape where
    (<) r1 r2 = area r1 < area r2
    (<=) r1 r2 = area r1 <= area r2
```

What would be needed to add a Figure8 shape and a perimeter function?

How does this compare to a Shape/Circle/Rect hierarchy in Java?
Two simple algebraic types

Let's look at the `compare` function:

```haskell
> :t compare
compare :: Ord a => a -> a -> Ordering
```

`Ordering` is a simple algebraic type, with only three values:

```haskell
> :info Ordering
data Ordering = LT | EQ | GT
```

```haskell
> [r1,r2]
[Rect 3.0 4.0,Rect 5.0 3.0]

> compare r1 r2
LT

> compare r2 r1
GT
```

What do you suppose `Bool` really is?

```haskell
> :info Bool
data Bool = False | True
```
Here's an algebraic type for a binary tree:

```haskell
data Tree a = Node a (Tree a) (Tree a)  -- tree.hs
    | Empty
  deriving Show
```

The `a` is a type variable. Our `Shape` type used `Double` values but `Tree` can hold values of any type!

```haskell
> let t1 = Node 9 (Node 6 Empty Empty) Empty
> t1
Node 9 (Node 6 Empty Empty) Empty

> let t2 = Node 4 Empty t1
> t2
Node 4 Empty (Node 9 (Node 6 Empty Empty) Empty)
```
Here's a function that inserts values, maintaining an ordered tree:

```haskell
insert Empty v = Node v Empty Empty
insert (Node x left right) value
    | value <= x = (Node x (insert left value) right)
    | otherwise = (Node x left (insert right value))
```

Let's insert some values...

```haskell
> let t = Empty
> insert t 5
Node 5 Empty Empty

> insert t 10
Node 5 Empty (Node 10 Empty Empty)

> insert t 3
Node 5 (Node 3 Empty Empty) (Node 10 Empty Empty)
```

How many Nodes are constructed by each of the insertions?
Here's an in-order traversal that produces a list of values:

\[
\text{inOrder Empty} = [] \\
\text{inOrder (Node val left right)} = \\
\text{inOrder left ++ [val] ++ inOrder right}
\]

What's an easy way to insert a bunch of values?

\[
\text{> let t = foldl insert Empty [3,1,9,5,20,17,4,12]} \\
\text{> inOrder t} \\
[1,3,4,5,9,12,17,20]
\]

\[
\text{> inOrder $ foldl insert Empty "tim korb" } \\
\text{" bikmort"}
\]

\[
\text{> inOrder $ foldl insert Empty [Rect 3 4, Circle 1, Rect 1 2]} \\
[Rect 1.0 2.0,Circle 1.0,Rect 3.0 4.0]
\]
Here's an interesting type:

```haskell
> :info Maybe
data Maybe a = Nothing | Just a
```

Speculate: What's the point of it?

Here's a function that uses it:

```haskell
> :t Data.List.find
Data.List.find :: (a -> Bool) -> [a] -> Maybe a
```

How could we use it?

```haskell
> find even [3,5,6,8,9]
Just 6

> find even [3,5,9]
Nothing

> case (find even [3,5,9]) of { Just _ -> "got one"; _ -> "oops!"}
"oops!"
```
In conclusion...
If we had a whole semester... If we had a whole semester to study functional programming, here's what might be next:

- Infinite data structures (see slides 125-126 for a tiny bit).
- How lazy evaluation works
- Implications and benefits of referential transparency (which means that the value of a given expression is always the same).
- Functors (structures that can be mapped over)
- Monoids (a set of things with a binary operation over them)
- Monads (for representing sequential computations)
- Zippers (a structure for traversing and updating another structure)
- And more!
Recursion and techniques with higher-order functions can be used in most languages. Some examples:

JavaScript, Python, PHP, all flavors of Lisp, and lots of others: Functions are "first-class" values; anonymous functions are supported.

C
Pass a function pointer to a recursive function that traverses a data structure.

C#
Excellent support for functional programming with the language itself and LINQ, too.

Lambda expressions are slated for Java 8 (2015?)