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

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

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

Examples of works that documented paradigms:
- Newton's *Principia*
- Lavoisier's *Chemistry*
- Lyell's *Geology*
Kuhn says a paradigm has:
• A world view
• A vocabulary
• A set of techniques for solving problems

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

Kuhn equates a paradigm shift with a scientific revolution.
The imperative programming paradigm

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

Originated with machine-level programming:
- Instructions change memory locations or registers
- Branching instructions alter the flow of control

Examples of areas of study for those interested in the paradigm:
- Data types
- Operators
- Branching mechanisms and (later) control structures

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

Language-wise, imperative programming requires:

• "Variables"—data objects whose values can change
• Expressions to compute values
• Support for iteration—a “while” control structure, for example.

Support for imperative programming is very common.

• Java
• C
• C++
• Python
• and hundreds more
• but not Haskell

Code inside a Java method or C function is likely imperative.
The procedural programming paradigm

An outgrowth of imperative programming was *procedural programming*:

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

Examples of areas of study:

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

What does a language need to provide to support procedural programming?
Support for procedural programming is very common.
- C
- Python
- Ruby
- and hundreds more

The procedural and imperative paradigms can be combined:
- Procedural programming: the set of procedures
- Imperative programming: the contents of procedures

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

Procedural programming is possible but clumsy in Java.
  - Classes devolve into collections of static methods and data
The object-oriented programming paradigm

The essence of the object-oriented programming paradigm:
Programs are a system of interacting objects.

Dan Ingalls said,
"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."

Examples of areas of study:
• How to model systems as interacting objects
• Managing dependencies between classes
• Costs and benefits of multiple inheritance
• Documentation of object-oriented designs

What does a language need to support OO programming?
The object-oriented paradigm, continued

Brief history of the rise of the object-oriented paradigm:

- Simula 67 recognized as first language to support objects
- Smalltalk created broad awareness of OO programming
  (see https://archive.org/details/byte-magazine-1981-08)
- C++ started a massive shift to OO programming
- Java broadened the audience even further

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

World view:
- Systems are interacting objects

Vocabulary:
- Methods, inheritance, superclass, instances

Techniques:
- Model with classes, work out responsibilities and collaborators, don't have public data, etc.
The object-oriented paradigm, continued

Language support for OOP has grown since mid-1980s.

Many languages support OO programming but don't force it.
  • C++
  • Python
  • Ruby

Java forces at least a veneer of OO programming.

The OO and imperative paradigms can be combined:
  • OO: the set of classes and their methods
  • Imperative: the code inside methods
Paradigms in a field of science are often incompatible. 
   Example: geocentric vs. heliocentric model of the universe

Imperative programming is used both with procedural and object-oriented programming.
   Is imperative programming really a paradigm?

Wikipedia's Programming_paradigm has this:
   Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Are "programming paradigms" really paradigms by Kuhn's definition or are they just characteristics?
The level of a paradigm

Programming paradigms can apply at different levels:

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

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

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

The imperative paradigm applies to programming in the small.

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

The programming paradigms we know affect how we approach problems.

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

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

• If we know only imperative programming, code inside methods and functions will be imperative.
Imperative programming revisited

Recall these language requirements for imperative programming:

- "Variables"—data objects whose values can change
- Expressions to compute values
- Support for iteration—a “while” control structure, for example.

Another:

- Statements are sequentially executed
Here's an imperative solution in Java to sum the integers in an array:

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

    return sum;
}
```

How does it exemplify imperative programming?

- The values of `sum` and `i` change over time.
- An iterative control structure is at the heart of the computation.
Imperative summation, 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?
Non-imperative summation

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

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

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

Which of the three versions is the easiest to believe it is correct?
Background: Value, type, side effect
An expression is a sequence of symbols that can be evaluated to produce a value.

Here are some Java expressions:

- \( 'x' \)
- \( i + j \times k \)
- \( f(\text{args.length} \times 2) + n \)

Three questions to consider about an expression:

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

Mnemonic aid for the three: Imagine you're wearing a vest that's reversed. "vest" reversed is "t-se-v": type/side-effect/value.
What is the value of the following Java expressions?

- \(3 + 4\)  
  \(7\)

- \(1 < 2\)  
  \(true\)

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

- \(s = 3 + 4 + "5"\)  
  "75"

- "a,bb,c3".split("",")  
  A String array with three elements: "a", "bb" and "c3"

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

- "a,bb,c3".split("",")[2].charAt(0) == 'X'  
  false
Value, type, and side effect, continued

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

\[ 3 + 4 \]

*int*

\[ 1 < 2 \]

*boolean*

\"abc\".charAt(1)

*char*

\[ s = 3 + 4 + \"5\" \]

*String*

\"a,bb,c3\".split("",""")

*String []*

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

*String*

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

*boolean*

When we ask,

"What's the type of this expression?"

we're actually asking this:

"What's the type of the value produced by this expression?"
Value, type, and side effect, continued

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

Which of these **Java** expressions have a side effect?

- \( x + 3 \times y \)
  
  *No side effect. A computation was done but no evidence of it remains.*

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

- \( s.length() > 2 \mid \mid s.charAt(1) == '#' \)
  
  *No side effect. A computation was done but no evidence of it remains.*
More expressions to consider wrt. side effects:

"testing".toUpperCase()
   A string "TESTING" was created somewhere but we can't get to it. No side effect.

L.add("x"), where L is an ArrayList
   An element was added to L. Definitely a side-effect!

System.out.println("Hello!")
   Side effect: "Hello!" went somewhere.

window.checkSize()
   We can't tell without looking at window.checkSize()!
The hallmark of imperative programming

Side effects are the hallmark of imperative programing.

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

Recall:

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

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

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

Pure mathematical functions:

- Always produce the same value for given input(s)
- Have no side effects
- Can be easily combined to produce more powerful functions
- Are often specified with cases and expressions
Functional programming, continued

Other characteristics of the functional paradigm:

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

Based on the above, how well would the following languages support functional programming?

- Java?
- Python?
- C?
Haskell basics
What is Haskell?

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

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*.

Is not object-oriented in any way.
Website: haskell.org
All sorts of resources!

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

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

  * Thinking Functionally with Haskell by Richard Bird

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

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

Windows
2. Download Core (64 bit)
3. Install it!
   • Under "Choose Components", deselect "Stack"

macOS
2. Download Core (64 bit)
3. Install it!

The latest version is 8.2.2. Lectura is running 8.0.1 but there should be no significant differences for our purposes.
On macOS and Linux machines like lectura we can interact with Haskell by running `ghci`:

```
% ghci
GHCi, version 8.0.1: ... :? for help
Loaded GHCi configuration from /p1/hw/whm/.ghci

Prelude> 3 + 4
7

Prelude> 1 > 2
False
```

With no arguments, `ghci` starts a read-eval-print loop (REPL): Expressions typed at the prompt (`Prelude>`) are evaluated and the result is printed.
Interacting with Haskell, continued

On Windows there's a choice between ghci:

And WinGHCi:

Suggested WinGHCi options: (File > Options)

Prompt: Just a >

Uncheck Print type after evaluation (for now)
When \texttt{ghci} starts up on macOS or Linux it looks for the file \texttt{~/.ghci} – a .\texttt{ghci} file in the user's home directory.

I have these two lines in my \texttt{~/.ghci} file on both my Mac and on lectura:

\begin{verbatim}
:set prompt "> \\
import Text.Show.Functions
\end{verbatim}

The first line simply sets the prompt to just "> ".

\textit{The second line is very important:}
\begin{itemize}
  \item It loads a module that lets functions be printed.
  \item Prints \texttt{<function>} for function values.
  \item Without it, lots of examples in these slides won't work!
\end{itemize}
Goofy fact: ~/.ghci must not be group- or world-writable!

If you see something like this,
 *** WARNING: /home/whm/.ghci is writable by someone else, IGNORING!
Suggested fix: execute
 'chmod go-w /home/whm/.ghci'

the suggested fix should work.

Details on .ghci and lots more can be found in
 downloads.haskell.org/~ghc/latest/docs/users_guide.pdf
On Windows, ghci and WinGHCi use a different initialization file:

```
%APPDATA%\ghc\ghci.conf
```

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

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

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

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

```
C:\Users\whm\AppData\Roaming\ghc\ghci.conf
```
Extra Credit Assignment 1

For two assignment points of extra credit:

1. Run ghci (or WinGHCi) somewhere and try ten Haskell expressions with some degree of variety. (Not just ten additions, for example!)

2. Demonstrate that you've got import Text.Show.Functions in your ~/.ghci or ghc.conf file, as described on slides 35-37, by showing that typing negate produces <function>, like this:
   
   Prelude> negate
   <function>

3. Capture the interaction (both expressions and results) and put it in a plain text file, eca1.txt. No need for your name, NetID, etc. in the file. No need to edit out errors.

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

   % turnin 372-eca1 eca1.txt

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

Haskell by Observation

cs.arizona.edu/classes/cs372/spring18/cle-haskell-obs.html
Functions and function types
In Haskell, *juxtaposition* indicates a function call:

```
> negate 3
-3

> even 5
False

> pred 'C'
'B'

> signum 2
1
```

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

\[ \text{signum negate 2 means (signum negate) 2} \]

\[
> \text{signum negate 2} \\
<\text{interactive}>:11:1: \text{error:} \\
  \cdot \text{Non type-variable argument ...} \\
  \ldots
\]

We add parentheses to call \text{negate 2} first:

\[
> \text{signum (negate 2)} \\
-1
\]
Function call has higher precedence than any operator.

> negate 3+4
1

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

> negate (3 + 4)
-7

> signum (negate (3 + 4))
-1
The `Data.Char` module

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

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

> isLower 'b'
  True

> toUpper 'a'
  'A'

> ord 'A'
  65

> chr 66
  'B'

> Data.Char.ord 'G'  
    (uses a qualified name)
  71
```
Function types, continued

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

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

The type `Char -> Bool` says that `isLower` is a function that
1. Takes an argument of type `Char`
2. Produces a result of type `Bool`

The text
```
isLower :: Char -> Bool
```
is read as "isLower has type Char to Bool"
Recall:

> toUpper 'a'
'A'

> ord 'A'
65

> chr 66
'B'

What are the types of those three functions?

> :t toUpper
toUpper :: Char -> Char

> :t ord
ord :: Char -> Int

> :t chr
chr :: Int -> Char
Sidebar: Contrast with Java

What is the type of the following Java methods?

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

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

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

**Important:**
- Java: common to think of a method's return type as the method's type
- Haskell: function's type has both type of argument(s) and return type
Like most languages, Haskell requires that expressions be *type-consistent* (or *well-typed*).

Here is an example of an inconsistency:

```haskell
> chr 'x'
<interactive>:1:5: error:
  • Couldn't match expected type `Int' with actual type `Char'
  • In the first argument of `chr', namely "x"

> :t chr
chr :: Int -> Char

> :t 'x'
'x' :: Char
```

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

'\text{a}'
isUpper
isUpper '\text{a}'
not (isUpper '\text{a}')
not not (isUpper '\text{a}')
toUpper (\text{ord} \, 97)
isUpper \, (\text{toUpper} \, (\text{chr} \, \text{\text{a}'}))
isUpper \, (\text{intToDigit} \, 100)

\begin{align*}
\text{'a'} &:: \text{Char} \\
\text{chr} &:: \text{Int} \rightarrow \text{Char} \\
\text{digitToInt} &:: \text{Char} \rightarrow \text{Int} \\
\text{intToDigit} &:: \text{Int} \rightarrow \text{Char} \\
\text{isUpper} &:: \text{Char} \rightarrow \text{Bool} \\
\text{not} &:: \text{Bool} \rightarrow \text{Bool} \\
\text{ord} &:: \text{Char} \rightarrow \text{Int} \\
\text{toUpper} &:: \text{Char} \rightarrow \text{Char}
\end{align*}
Sidebar: Key bindings in ghci

ghci uses the haskeline package to provide line-editing.

A few handy bindings:

- **TAB** completes identifiers
- **^A** Start of line
- **^E** End of line
- **^R** Incremental search through previously typed lines

More:

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

Windows: Use **Home** and **End** for start- and end-of-line
Sidebar: Using a REPL to help learn a language

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

How does a REPL help us learn a language?

Is there a REPL for Java?

Java 9 has `jshell`. There's also `javarepl.com`.

What are some other languages that have a REPL available?

What characteristics does a language need to support a REPL?

If there's no REPL for a language, how hard is it to write one?
Type classes
Recall the `negate` function:

> negate 5
-5

> negate 5.0
-5.0

Speculate: What's the type of `negate`?
"A type is a collection of related values." —Hutton

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

Haskell also has **type classes**.

Type class:
   A collection of types that support a specified set of operations.

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

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

**Important:**
   The names of types and type classes are always capitalized.
The **Num** type class

> :info Num

class Num a where

  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a

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

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

The Prelude defines these types as instances of **Num**
Here's the type of `negate`:

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

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

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

"If you give me an `X`, I'll give you back an `X`."

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

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

`negate` accepts any value whose type is an instance of `Num`. It returns a value of the same type.
What type do integer literals have?

> :type 3

3 :: Num p => p

> :type (-27)  -- Note: Parens needed!

(-27) :: Num p => p

Why are integer literals typed with a class constraint rather than just Int or Integer?
What's the type of a decimal fraction?

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

Will negate 3.4 work?

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

> negate 3.4
-3.4

Speculate: Why does it work?
Haskell type classes form a hierarchy. The Prelude has these:

Adapted from [http://en.wikibooks.org/wiki/Haskell/Classes_and_types](http://en.wikibooks.org/wiki/Haskell/Classes_and_types)
Type classes, continued

The arrow from \texttt{Num} to \texttt{Fractional} means that a \texttt{Fractional} can be used as a \texttt{Num}.

Given

\begin{align*}
\text{negate} & \, :: \, \texttt{Num} \ a \implies a \implies a \\
\text{and} & \, \\
5.0 & \, :: \, \texttt{Fractional} \ a \implies a \\
\text{then} & \, \\
\text{negate} \ 5.0 & \text{ is valid.}
\end{align*}
What does the diagram show us other than the relationship between **Num** and **Fractional**?

It shows us types that are instances of **Num** and **Fractional**.

Do :info **Num** again. Do :info **Fractional**, too.
The Prelude has a `trimc` function:

```haskell
> trimc 3.4
3
```

What does the type of `trimc` tell us?

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

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

Explore the `Integral` and `RealFrac` type classes with `:info`.
:info Type shows the classes that Type is an instance of.

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

Try :info for each of the classes.
In LYAH, *Type Classes 101* has a good description of the Prelude's type classes.

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

Remember:
Haskell's type classes are unrelated to classes in the OO sense.
In essence, `negate :: Num a => a -> a` describes many functions:

- `negate :: Integer -> Integer`
- `negate :: Int -> Int`
- `negate :: Float -> Float`
- `negate :: Double -> Double`

...and more...

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

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

Does Java have polymorphic methods? Does C? Python?
Consider this excerpt from \textit{Bounded}:

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

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

How can we use them?
Polymorphic values

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

A usage of it:

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

```haskell
> maxBound::Int
9223372036854775807
```

```haskell
> maxBound::Bool
True
```

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

\begin{verbatim}
> :set +t

> 3
3
it :: Num p => p

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

> abs
<function>
it :: Num a => a -> a
\end{verbatim}

Use \texttt{:unset +t} to turn off display of types.
Sidebar: LHtLaL—introspective tools

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

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

Some type-related tools in other languages:

Python: type(expr) and repr(expr)

JavaScript: typeof(expr)

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

C: sizeof(expr)

Java: getClass(); /var in jshell.

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

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

Output:

```
class java.lang.Integer
class java.lang.Double
class [C
class [Ljava.lang.String;
class java.util.HashMap  (Note: no String or Integer—type erasure!)
```
More on functions
Writing simple functions

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

```haskell
> double x = x * 2
double :: Num a => a -> a  \(\text{(:set +t is in effect)}\)

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

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

General form of a function definition for the moment:

```plaintext
function-name parameter = expression
```

Function and parameter names must begin with a lowercase letter or an underscore.
Simple functions, continued

Two more functions:

\[
\begin{align*}
> \text{neg } x &= -x \\
\text{neg} :: \text{Num } a => a \to a \\
\end{align*}
\]

\[
\begin{align*}
> \text{toCelsius } \text{temp} &= (\text{temp} - 32) \times 5/9 \\
\text{toCelsius} :: \text{Fractional } a \Rightarrow a \to a \\
\end{align*}
\]

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

Problem: Write \textit{isPositive } x which returns \textbf{True} iff \textbf{x} is positive.

\[
\begin{align*}
> \text{isPositive } x &= x > 0 \\
\text{isPositive} :: (\text{Num } a, \text{Ord } a) \Rightarrow a \to \text{Bool} \\
\end{align*}
\]
We can use :: \textit{type} to constrain a function's type:

\begin{verbatim}
> neg x = -x :: Int
neg :: Int -> Int

> toCelsius temp = (temp - 32) * 5/9 :: Double
toCelsius :: Double -> Double
\end{verbatim}

:: \textit{type} has low precedence; parentheses are required for this:

\begin{verbatim}
> isPositive x = x > (0::Int)
isPositive :: Int -> Bool
\end{verbatim}

Note that :: \textit{type} applies to an expression, not a function.

We'll use :: \textit{type} to simplify some following examples.
We can put function definitions in a file.

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

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

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

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

https://www2.cs.arizona.edu/classes/cs372/spring18/haskell/
/cs/www/classes/cs372/spring18/haskell (on lectura)
Assuming `simple.hs` is in the current directory, we can load it with `:load` and see what we got with `:browse`.

```ghci
% ghci
> :load simple  
(assumes .hs suffix)
[1 of 1] Compiling Main ...
Ok, one module loaded.

> :browse
double :: Int -> Int
neg :: Int -> Int
isPositive :: Int -> Bool
toCelsius :: Double -> Double
```
ghci is clumsy to type! I've got an hs alias in my ~/.bashrc:

```
alias hs=ghci
```

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

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

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

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

Lather, rinse, repeat.
Functions with multiple arguments
Functions with multiple arguments

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

\[
\text{add } x \ y = x + y :: \text{Int}
\]

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

\[
\text{add } 3 \ 5
\]

8

Here is its type:

\[
\text{add} :: \text{Int} \to \text{Int} \to \text{Int}
\]

The operator \(\to\) is right-associative, so the above means this:

\[
\text{add} :: \text{Int} \to (\text{Int} \to \text{Int})
\]

But what does that mean?
Recall our negate function:

\[ \text{neg } x = -x :: \text{Int} \]

\[ \text{neg} :: \text{Int} \rightarrow \text{Int} \]

Here's \texttt{add} again, with parentheses added to show precedence:

\[ \text{add } x y = x + y :: \text{Int} \]

\[ \text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

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

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

Call \texttt{add} with the value 3, \textit{producing a nameless function}. Call that nameless function with the value 5.
Consider the following expression:
\[ r = f \ a \ b + g \ c \ b(a) \]

1. Fully parenthesize it to show the order of operations

2. Write some code to precede it such that \( r \) gets bound to 3.
Collaborative Learning Exercise

Haskell Functions

http://cs.arizona.edu/classes/cs372/spring18/cle-3-functions.html
Partial applications
Partial application

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

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

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

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

What will `plusThree 5` produce?

> plusThree 5
> 8
> it :: Int
Partial application, continued

At hand:

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

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

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

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

At hand:

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

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

> add3 x y z = x + y + z :: Int
add3 :: Int -> (Int -> (Int -> Int))

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

LYAH nails it:

... functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on.
A little history:
- 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?

\[ \log_{2} n \]
Another model of partial application

When an argument is provided to a function...
• The next parameter is dropped
• The argument's value is "wired" into the expression
• The result is a new function with one less parameter

```haskell
> f x y = (y * x + x)::Int -- f :: Int -> Int -> Int

> g = f 3
> g :: Int -> Int
  -- as if we'd done this:  g y = y * 3 + 3

> g 5
18

> f 3 5
18
```

Everybody: Try it!
Consider this function:
\[ f(x, y, z) = x + y + y \times z \]

\[ f_1 = f \ 3 \]
is equivalent to
\[ f_1(y, z) = 3 + y + y \times z \]

\[ f_2 = f_1 \ 5 \]
is equivalent to
\[ f_2(z) = 3 + 5 + 5 \times z \]

\[ \text{val} = f_2(7) \]
is equivalent to
\[ \text{val} = f \ 3 \ 5 \ 7 \]

and
\[ \text{val} = f_1 \ 5 \ 7 \]

Another model, continued

When an argument is provided to a function...

- A parameter is dropped
- The argument's value is "wired" into the expression
- The result is a new function with one less parameter
Some key points about functions

- The *general form* of a function definition (for now):
  \[
  \text{name param} \, 1 \, \text{param} \, 2 \, \ldots \, \text{param} \, N = \text{expression}
  \]

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

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

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

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

• There's really nothing special about a partial application—it's just another function.
Functions are values

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

The following creates a function \texttt{add} and binds the name \texttt{add} to it.
\[
> \texttt{add x y = x + y}
\]

The following binds the name \texttt{plus} to the expression \texttt{add}.
\[
> \texttt{plus = add}
\]

Either name can be used to reference the function value:
\[
> \texttt{add 3 4}
7
> \texttt{plus 5 6}
11
\]
What does the following suggest to you?

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

> :info +
class Num a where
  (+) :: a -> a -> a

...  
  infixl 6 +

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

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

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

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

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

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

```haskell
> 11 `rem` 3
2
```

Speculate: do `add` and `rem` have precedence and associativity?
Haskell lets us define custom operators.

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

Usage:
> 100 +% 1
 101.0
> 12 +% 25
 15.0

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

Haskell's standard modules define LOTS of custom operators.
## Reference: Operators from the Prelude

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Left associative operators</th>
<th>Non associative operators</th>
<th>Right associative operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td><code>!!</code></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td><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>\texttt{\textbackslash div}</code>, <code>\texttt{\textbackslash mod}</code>, <code>\texttt{\textbackslash rem}</code>, <code>\texttt{\textbackslash 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>\texttt{\textbackslash elem}</code>, <code>\texttt{\textbackslash notElem}</code></td>
<td></td>
<td></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;</code>, <code>&gt;&gt;&gt;</code></td>
<td></td>
<td><code>$</code>, <code>$!</code>, <code>\texttt{\textbackslash seq}</code></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: From page 51 in Haskell 2010 report
Squeezed in for flip, to avoid adding a slide.

```haskell
> :set +t
> add x y = x + y
add :: Num a => a -> a -> a
```

What has Haskell inferred (figured out)?

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

Type Inferencing
Haskell does type inferencing:
- The types of values are inferred based on the operations performed on the values.
- Inferences based on assumption that there are no errors.

Example:

```haskell
> isCapital c = c >= 'A' && c <= 'Z'

isCapital :: Char -> Bool
```

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

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

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

What type will be inferred for `f`?

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

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

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

Let's try it:

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

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

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

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

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

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

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

The function below uses \( x \) as both a \textbf{Num} and a \textbf{Char}.

\[
g \ x \ y = x > 0 \land x > '0'
\]

<interactive>:1:13: error:

  • No instance for (Num Char) arising from the literal ‘0’
  • In the second argument of ‘(>)’, namely ‘0’
    In the first argument of ‘(&&)’, namely ‘x > 0’
    In the expression: x > 0 \land x > '0'

What does the error "\textbf{No instance for (Num Char)}" mean?
\textbf{Char} is not an instance of the \textbf{Num} type class.
(:info Num shows instance Num Int, instance Num Float, etc.)
Type Specifications
Type specifications for functions

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

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

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

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

isPositive :: (Num a, Ord a) => a -> Bool
isPositive x = x > 0
```
Sometimes type specifications can backfire.

What's a ramification of the difference between the types of `add1` and `add2`?

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

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

`add1` can operate on `Nums` but `add2` requires `Integers`.

Challenge: Without using `::type`, show an expression that works with `add1` but fails with `add2`.
Type specification for functions, continued

Two pitfalls related to type specifications for functions:

• Specifying a type, such as \texttt{Integer}, rather than a type class, such as \texttt{Num}, may make a function's type needlessly specific, like \texttt{add2} on the previous slide.

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

Recommendation:

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

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

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

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

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

These weaving declarations are poor style but are valid:

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

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

% ghci indent2
...
indent2.hs:3:1: error:
  parse error (possibly incorrect indentation ...)
  |
  3 | x + y
    ^

Note that 3:1 indicates line 3, column 1.
Guards
Recall this characteristic of mathematical functions:
"Are often specified with cases and expressions."

This function definition uses guards to specify three cases:

\[
\text{sign } x \mid \begin{array}{l}
  x < 0 = -1 \\
  x == 0 = 0 \\
  \text{otherwise} = 1 
\end{array}
\]

Notes:
- This definition would be found in a file, not typed in ghci.
- \textit{sign } x \textit{ appears just once}. First guard might be on next line.
- The guards appear \textit{between | and =}, and produce \texttt{Bools}.
- What is \textit{otherwise}?
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}
Problem: Write a function \texttt{weather} that classifies a given temperature as hot if 80+, else nice if 70+, and cold otherwise.

\begin{verbatim}
> weather 95
"Hot!"
> weather 32
"Cold!"
> weather 75
"Nice"
\end{verbatim}

Hint: guards are tried in turn.

Solution:

\begin{verbatim}
weather temp | temp >= 80 = "Hot!"
            | temp >= 70 = "Nice"
            | otherwise = "Cold!"
\end{verbatim}
if-else
Here's an example of Haskell's if-else:

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

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

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

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

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

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

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

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

`3 if 1 < 2 else 4`
"A statement changes the state of the program while an expression wants to express itself."

— Victor Nguyen, CSC 372, Spring 2014
Guards vs. \texttt{if-else}

Which of the versions of \texttt{sign} below is better?

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

\begin{verbatim}
  sign x = if x < 0 then -1
          else if x == 0 then 0
          else 1
\end{verbatim}

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

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

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

Write factorial in Haskell. (p.s. 0! is 1)

What is the type of `factorial`?

```haskell
:type factorial
factorial :: (Eq a, Num a) => a -> a
```
One way to manually trace through a recursive computation is to underline a call, then rewrite the call with a textual expansion.

\[
\text{factorial } n \\
\begin{align*}
| \quad & n == 0 = 1 \\
| \text{ otherwise } = n * \text{factorial} (n - 1)
\end{align*}
\]

factorial 4
4 * factorial 3
4 * 3 * factorial 2
4 * 3 * 2 * factorial 1
4 * 3 * 2 * 1 * factorial 0
4 * 3 * 2 * 1 * 1
Lists
In Haskell, a list is a sequence of values of the same type.

Here's one way to make a list.

```haskell
> [7, 3, 8]
[7,3,8]

it :: Num a => [a]
```

```haskell
> ['x', 10]

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

It is said that lists in Haskell are homogeneous.
The function \texttt{length} returns the number of elements in a list:

\begin{verbatim}
> length [3,4,5]
3

> length []
0
\end{verbatim}

What's the type of \texttt{length}?

\begin{verbatim}
> :type length
length :: [a] -> Int  (Note: A white lie, to be fixed!)
\end{verbatim}

With no class constraint specified, \([a]\) indicates that \texttt{length} operates on lists containing elements of any type.
The **head** function returns the first element of a list.

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

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

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

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

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

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

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

**Important:** **head** and **tail** are good for learning about lists but we'll almost always use *patterns* to access list elements!
The ++ operator concatenates two lists, producing a new list.

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

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

What are the types of ++ and reverse?

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

> :type reverse
reverse :: [a] -> [a]
Haskell has an *arithmetic sequence notation*:

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

it :: (Enum a, Num a) => [a]
```

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

```haskell
> [10..5]
[]
```
Here are drop and take:

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

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

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

\begin{verbatim}
> halves([1..10])
[[1,2,3,4,5],[6,7,8,9,10]]

> halves([1])
[[],[1]]
\end{verbatim}

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

\[
\ln = \text{length} \\
\text{halves } \text{lst} = [\text{take } (\ln \text{lst} \div 2) \text{lst}, \text{drop } (\ln \text{lst} \div 2) \text{lst}]
\]
List basics, continued

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

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

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

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

Important:
Much use of `!!` might indicate you're writing a Java program in Haskell!
Comparing lists

Haskell lists are values and can be compared as values:

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

True

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

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

In Haskell we'll write complex expressions using lists (and more) as freely as a Java programmer might write

\[ f(x) \times a = g(a,b) + c. \]
Comparing lists, continued

Lists are compared \textit{lexicographically}:
\begin{itemize}
  \item Corresponding elements are compared until an inequality is found.
  \item The inequality determines the result of the comparison.
\end{itemize}

Example:
\begin{align*}
  > \ [1,2,3] & < \ [1,2,4] \\
  \text{True}
\end{align*}

Why: The first two elements are equal, and $3 < 4$. 
We can make lists of lists.

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

\texttt{x :: Num a \Rightarrow [[a]]}

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

What's the length of \texttt{x}?

> length x

3
Lists of lists, continued

More examples:

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

\[
> \text{head } x
\]
\[
[1]
\]

\[
> \text{tail } x
\]
\[
[[2,3,4],[5,6]]
\]

\[
> x !! 1 !! 2
\]
\[
4
\]

\[
> \text{head (head (tail (tail x)))}
\]
\[
5
\]
Earlier I showed you this:

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

Around version 7.10 `length` was generalized to this:

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

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

Instead of `sum :: (Num a, Foldable t) => t a -> a`

Pretend this `sum :: Num a => [a] -> a`

Instead of `minimum :: (Ord a, Foldable t) => t a -> a`

Pretend this `minimum :: Ord a => [a] -> a`
Strings in Haskell are simply lists of characters.

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

> length asciiLets
52

> reverse (drop 26 asciiLets)
"zyxwvutsrqponmlkjihgfedcba"

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

> isAsciiLet c = c `elem` asciiLets
isAsciiLet :: Char -> Bool
```
The Prelude defines `String` as `[Char]` (a `type synonym`).

> :info String

type String = [Char]

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

> :type words

words :: String -> [String]

> :type unwords

unwords :: [String] -> String

What's the following doing?

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

"some words!"
"cons" lists

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

A "cons" list has two parts:
   head: a value
   tail: a list of values (possibly empty)

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

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

What's the type of the cons operator?

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

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

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

> head c
5

> tail c
[10,20,30]

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

A cons node can be referenced by multiple cons nodes.

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

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

> f = 1:c
[1, 5, 10, 20, 30]
```
"cons" lists, continued

What are the values of the following expressions?

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

> 1:2
...error...

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

> []:[]
[[]]

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

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

cons is right associative
chr 97:(chr 98:(chr 99:[]))
It's important to understand that \texttt{tail} does not create a new list. Instead it simply returns an existing cons node.

\begin{center}
\begin{tikzpicture}
\tikzset{font={\scriptsize}}
\node[draw,rectangle] (a) at (0,0) {5};
\node[draw,rectangle] (b) at (0,-2) {10};
\node[draw,rectangle] (c) at (0,-4) {20};
\node[draw,rectangle] (d) at (0,-6) {30};
\node (e) at (0,-8) {5};
\node (f) at (0,-10) {10};
\node (g) at (0,-12) {20};
\node (h) at (0,-14) {30};
\draw (a) -- (b);
\draw (b) -- (c);
\draw (c) -- (d);
\draw (e) -- (f);
\draw (f) -- (g);
\draw (g) -- (h);
\end{tikzpicture}
\end{center}

\begin{itemize}
\item > \texttt{a = [5,10,20,30]}
\item > \texttt{h = head a}
\item > \texttt{h}
\item 5
\item > \texttt{t = tail a}
\item > \texttt{t}
\item [10,20,30]
\item > \texttt{t2 = tail (tail t)}
\item > \texttt{t2}
\item [30]
\end{itemize}
A little on performance

What operations are likely fast with cons lists?
- Get the head of a list
- Get the tail of a list
- Make a new list from a head and tail ("cons up a list")

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

With cons lists, what does list concatenation involve?
> m=[1..10000000]
> length (m++[0])
10000001
True or false?

The head of a list is a one-element list.
   False, unless...
      ...it's the head of a list of lists that starts with a one-element list
The tail of a list is a list.
   True
The tail of an empty list is an empty list.
   It's an error!
length (tail (tail x)) == (length x) – 2
   True (assuming what?)
A cons list is essentially a singly-linked list.
   True
A doubly-linked list might help performance in some cases.
   Hmm...what's the backlink for a multiply-referenced node?
Changing an element in a list might affect the value of many lists.
   Trick question! We can't change a list element. We can only
   "cons up" new lists and reference existing lists.
Here's a function that produces a list with a range of integers:

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

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

Problem:

Write a recursive version of `fromTo` that uses the cons operator to build up its result.
One solution:

\[
\text{fromTo} \text{ first } \text{ last}
\]
\[
| \text{ first } > \text{ last } = []
\]
\[
| \text{ otherwise } = \text{ first } : \text{ fromTo } (\text{first}+1) \text{ last}
\]

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

\[
\text{fromTo 1 3}
\]
\[
1 : \text{fromTo (1+1) 3}
\]
\[
1 : \text{fromTo 2 3}
\]
\[
1 : 2 : \text{fromTo (2+1) 3}
\]
\[
1 : 2 : \text{fromTo 3 3}
\]
\[
1 : 2 : 3 : \text{fromTo (3+1) 3}
\]
\[
1 : 2 : 3 : \text{fromTo 4 3}
\]
\[
1 : 2 : 3 : []
\]

The \text{Enum} type class has \text{enumFromTo} and more.
Do `:set +s` to get timing and memory information, and make some lists. Try these:

```
fromTo 1 10
f = fromTo
f 1 1000
f = fromTo 1
f 1000
x = f 1000000
length x
take 5 (f 1000000)

-- So we can type f instead of fromTo
-- Note partial application
```
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]
```

In English:

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

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

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

What are other languages with list comprehensions?
REPLACEMENTS! Discard 150+ in the old set

A little output
What can you tell me about `show`?

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

`show` produces a string representation of a value.

```haskell
> show 10
"10"

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

> show show
"<function>"
```

Important: `show` does not produce output!

What's the Python analog for `show`?
Challenge: Write a Java analog for `show`.
The `putStr` function outputs a string:

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

Type:

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

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

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

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

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

- The top-level function uses `putStr` to print that string returned by the helper.
A Java analog to our approach for functions that produce output:

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

Why `print` instead of `println`?
If the output should end with a newline, `computeOutput` includes it.
Let's write a function to **print** the integers from 1 to N:

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

First, write a helper, `printN'`:

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

Solution: (does appear on next slide)

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

At hand:

\[
\text{printN} :: \text{Integer} \to \text{String} \quad -- \text{Covered in flip!}
\]

\[
\text{printN} \ n
\]

\[
\begin{align*}
| & n == 0 = "" \\
| & \text{otherwise} = \text{printN} \ (n-1) ++ \text{show} \ n ++ "\n"
\end{align*}
\]

Usage:

\[
> \text{printN} \ 10
\]

"1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n"

Let's write the top-level function:

\[
\text{printN} :: \text{Integer} \to \text{IO} ()
\]

\[
\text{printN} \ n = \text{putStrLn} \ (\text{printN} \ n)
\]
All together in a file:
% cat printN.hs
printN::Integer -> IO ()
printN n = putStrLn (printN' n)

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

% ghci printN
...
> printN 3
1
2
3
Let's write `charbox`:

```haskell
> charbox 5 3 '*'

*****
*****
*****

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

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

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

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

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

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

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

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

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

Test:

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

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

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

- Should we have used a helper function `charrow rowLen char`?
- How does this approach contrast with how we'd write it in Java?
Patterns
Motivation: Summing list elements

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

> sumElems [1..10]
55

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

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

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

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

\[
> x
10
\]

\[
> y
20
\]

\[
> [\text{inner}] = [[2,3]]
\]

\[
> \text{inner}
[2,3]
\]

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

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

```haskell
> h
10
```

```haskell
> t
[20,30]
```

What values get bound by the following pattern?

```haskell
> a:b:c:d = [10,20,30]
> [c,b,a]        -- Why in a list?
[30,20,10]
```

```haskell
> d
[]            -- Why did I do [c,b,a] instead of [d,c,b,a]?
```
Patterns, continued

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

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

No binding is done for the wildcard pattern.

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

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

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

```
> a:b:[] = [1]
> a
**** Exception: Irrefutable pattern failed for pattern a : b : []
```
Describe in English what must be on the right hand side for a successful match.

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

let [x:xs] = ...
    A list whose only element is a non-empty list.
    Does words "a test" match?
    [words "a test"] ?
    [[]] ?
    [[[]]] ?
Patterns in function definitions

Recall our non-idiomatic `sumElems`:

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

How could we redo it using patterns?

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

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

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

Do the types of the two versions differ?

```haskell
(Eq a, Num a) => [a] -> a  -- with head/tail
Num a => [a] -> a          -- with pattern
```
Here's a buggy version of `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
At hand:

\[
\text{buggySum } [x] = x \\
\text{buggySum } (h:t) = h + \text{buggySum } t
\]

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

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

Suggestion: add a bash alias! (See us if you don't know how to.) 
\texttt{alias ghci="ghci -fwarn-incomplete-patterns"} \\
Todo: Find a Windows analog.
What's a little silly about the following list-summing function?

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

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

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

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

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

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

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

Can it be improved?

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

The term "as pattern" perhaps comes from Standard ML, which uses an "as" keyword for the same purpose.
Patterns, then guards, then if-else

Good coding style in Haskell:
  Prefer patterns over guards
  Prefer guards over if-else

Patterns—first choice!
  \[\text{sumElems} \; [\;] = 0\]
  \[\text{sumElems} \; (h:t) = h + \text{sumElems} \; t\]

Guards—second choice...
  \[\begin{align*}
  \text{sumElems} \; \text{list} \\
  &| \; \text{list} == [] = 0 \\
  &| \; \text{otherwise} = \text{head list} + \text{sumElems} \; (\text{tail list})
  \end{align*}\]

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

And, these comparisons imply that list's type must be an Eq!
"Throughout the assignment I tried to keep in mind that I should use patterns first then guards if patterns didn't work.

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

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

—Kelsey McCabe, Spring 2016, a3/observations.txt
Patterns, then guards, then if-else

Recall this example of guards:

\[
\text{weather temp} \mid \text{temp} \geq 80 = "Hot!"
\mid \text{temp} \geq 70 = "Nice"
\mid \text{otherwise} = "Cold!"
\]

Can we rewrite `weather` to have three clauses with patterns?
No.
The pattern mechanism doesn't provide a way to test ranges.

Design question: should patterns and guards be unified?
Revision: the general form of a function

An earlier general form of a function definition:

\[ \text{name \ param\_1 \ param\_2 \ldots \ param\_N} = \text{expression} \]

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

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

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

If values in a call match the pattern(s) for a clause and a guard is true, the corresponding expression is evaluated.
At hand, a more general form for functions:

\[
\text{function-name \ pattern1 pattern2 ... patternN} \\
\{ | \text{guard-expression1} \} = \text{result-expression1} \\
\ldots \\
\{ | \text{guard-expressionN} \} = \text{result-expressionN}
\]

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

- \text{x} and \text{y} are trivial patterns
- \text{add} has one clause, which has no guard
Pattern/guard interaction

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

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

Usage:

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

> f []
"empty"

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

How many clauses does \( f \) have?

4

What if 2\(^{nd}\) and 3\(^{rd}\) clauses swapped?

3\(^{rd}\) clause would never be matched!

What if 4\(^{th}\) clause is removed?

Warning re "non-exhaustive patterns" exception on \( f [] \) (if -fwarn-incomplete-patterns specified).
REPLACEMENTS! Discard 179+ in the old set

Recursive functions on lists
Simple recursive list processing functions

Problem: Write \texttt{len} \texttt{x}, which returns the length of list \texttt{x}.

\begin{verbatim}
> len []
0

> len "testing"
7
\end{verbatim}

Solution:

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

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

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

Handy: \texttt{odd :: Integral a ==> a -\rightarrow Bool}

Solution:
\begin{verbatim}
odds [] = []
odds (h:t)
  | odd h = h:odds t
  | otherwise = odds t
\end{verbatim}
Simple list functions, continued

Problem: write \texttt{isElem x vals}, like \texttt{elem} in the Prelude.
\begin{verbatim}
 > isElem 5 [4,3,7]
 False

 > isElem 'n' "Bingo!"
 True

 > "quiz" `isElem` words "No quiz today!"
 True
\end{verbatim}

Solution:
\begin{verbatim}
isElem _ [] = False  -- Why a wildcard?
isElem x (h:t)
  | x == h = True
  | otherwise = x `isElem` t
\end{verbatim}
Problem: Write a function that returns a list's maximum value.

> maxVal "maximum" 'x'

> maxVal [3,7,2]
7

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

Recall that the Prelude has \texttt{max :: Ord a => a -> a -> a}

One solution:
\[
\begin{align*}
\text{maxVal } [x] &= x \\
\text{maxVal } (x:xs) &= \text{max } x \text{ (maxVal } xs) \\
\text{maxVal } [] &= \text{error } "\text{empty list}" \\
\end{align*}
\]
Sidebar: C and Python challenges

C programmers:
• Write `strlen` in C in a functional style. (No loops or assignments.)
• Do `strcmp` and `strchr`, too!
• Mail us!

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

> (1, "two", 3.0)
(1,"two",3.0)
it :: (Num a, Fractional c) => (a, [Char], c)

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

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

Note that we can't create analogous lists for the above tuples, due to the mix of types. Lists must be homogeneous.
A function can return a tuple:

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

What's the type of `pair`?

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

Usage:

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

> pair (3,4)
<function>

> it 5
((3,4),5)
```
The Prelude has two functions that operate on 2-tuples.

\[
\begin{align*}
> & \ p = \text{pair} \ 30 \ "forty"

> & \ p

(30, "forty")

> & \ \text{fst} \ p

30

> & \ \text{snd} \ p

"forty"
\end{align*}
\]
Recall: patterns used to bind names to list elements have the same syntax as expressions to create lists.

Patterns for tuples have the same syntax as expressions to create tuples.

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

```haskell
> middle ("372", "ILC 119", "Mitchell")
"ILC 119"
```

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

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

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

Solution:

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

What's the type of `middle`?

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

Will the following call work?

```haskell
> middle(1,[(2,3)],4)
[(2,3)]
```
Problem: Write a function `swap` that behaves like this:

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

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

Solution:

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

What is the type of `swap`?

```markdown
swap :: (b, a) -> (a, b)
```
Here's the type of `zip` from the Prelude:

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

Speculate: What does `zip` do? *(Pythonistas: Silence please!)*

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

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

What's especially interesting about the second example?  

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

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

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

Solution:

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

elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps
```
What's wrong below?

```haskell
> x = ((1,2),(3,4,5))
> fst x
(1,2)

> snd x
(3,4,5)

> fst (snd x)
<interactive> error: Couldn't match expected type '(a, b0)' with actual type '(Integer, Integer, Integer)'
```

What's wrong with `fst (snd x)`?

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

- We can write a function that handles a list of arbitrary length.
- We can't write a function that operates on a tuple of arbitrary "arity".*
The \textbf{Eq} type class and tuples

\texttt{:info Eq} shows many lines like this:

\begin{verbatim}
...  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)
\underline{instance (Eq a, Eq b, Eq c) => Eq (a, b, c)}
instance (Eq a, Eq b) => Eq (a, b)
\end{verbatim}

Speculate: What's being specified by the above?

One of them:

\begin{verbatim}
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
\end{verbatim}

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

The \textbf{Ord} and \textbf{Bounded} type classes have similar instance declarations.
Lists vs. tuples

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

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

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

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

If there were *Head First Haskell*, it would no doubt have an interview with List and Tuple, each arguing their own merit.
Sidebar: To curry or not to curry?

Consider these two functions:

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

Usage:

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

Which is better, \textit{add}_c or \textit{add}_t?

**Important:** Note the difference in types!
The *where* clause
The **where** clause

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

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

\[
\text{f } \ x \\
\quad | \ x < 0 = g \ a + g \ b \\
\quad | \ a > b = g \ b \\
\quad | \ \text{otherwise} = c + 10
\]

**where** { 
\[
\begin{align*}
  a &= x \times 5; \\
  b &= a \times 2 + x; \\
  g \ t &= \log t + a; \\
  c &= a \times 3;
\end{align*}
\]
}

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

Like variables defined in a method or block in Java, \(a, b, c\) and \(g\) are not visible outside the function \(f\).
The `where` clause, continued

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

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

Binding `semi` in a `where` clause avoids lots of repetition.

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

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

halves lst =

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

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

Solution:

  halves lst = (take halflen lst, drop halflen lst)

  where {
    halflen = (length lst `div` 2)
  }
The \textit{layout rule}
The *layout rule* for *where* (and more)

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

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

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

We can take advantage of Haskell's *layout rule* and write it like this instead:

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

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

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

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

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

Note that the declaration of \texttt{g} is continued onto a second line; if the minus sign were at or left of the line, it would be an error.
The layout rule, continued

Don't confuse the layout rule with indentation-based continuation of declarations! (See slides 106-108.)

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

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

Usage:

\[
\begin{align*}
  > f\ 1 \\
  10 \\
  > f\ 3 \\
  3
\end{align*}
\]

For contrast, with guards:

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

Remember: Patterns are tried in the order specified.
Here's a function that classifies characters as parentheses (or not):

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

Could we improve it by using patterns instead of guards?

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

Which is better?

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

> :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
A pattern can be:

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

Note the recursion.

Patterns can be arbitrarily complex.

3.17.1 in H10 shows the full syntax for patterns.
Larger examples
Imagine a function that counts occurrences of even and odd numbers in a list.

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

-- one even, two odds

Code:

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

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

Here's one way to picture this recursion:

\[
\begin{align*}
\text{countEO} & \ [10,20,25] \text{ returns } (2,1) \text{ (result of } (1 + 1,1)) \\
\text{countEO} & \ [20,25] \text{ returns } (1,1) \text{ (result of } (1 + 0,1)) \\
\text{countEO} & \ [25] \text{ returns } (0,1) \text{ (result of } (0,0 + 1)) \\
\text{countEO} & \ [\] \text{ returns } (0,0)
\end{align*}
\]
Imagine a robot that travels on an infinite grid of cells. Movement is directed by a series of one character commands: n, e, s, and w.

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

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

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

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

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

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

> travel "nnessw"
"Got home"

How can we approach this problem?
One approach:
1. Map letters into integer 2-tuples representing X and Y displacements on a Cartesian plane.
2. Sum the X and Y displacements to yield a net displacement.

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

Another:
Argument value: "nnessw"
Mapped to tuples: \((0,1) (0,1) (1,0) (0,-1) (0,-1) (-1,0)\)
Sum of tuples: \((0,0)\)
First, let's write a helper function to turn a direction into an \((x,y)\) displacement:

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

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

Usage:

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

- `> mapMove 'w'
  (-1,0)
Next, a function to sum x and y displacements in a list of tuples:

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

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

Implementation:

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

```haskell
travel :: [Char] -> [Char]
travel s
  | disp == (0,0) = "Got home"
  | otherwise = "Got lost; " ++ show (abs x + abs y) ++
                " from home"

where
  tuples = makeTuples s
  disp@(x,y) = sumTuples tuples -- note "as pattern"

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

As is, `mapMove` and `sumTuples` are at the top level but `makeTuples` is hidden inside `travel`. How should they be arranged?
travel s
  | disp == (0,0) = "Got home"
  | otherwise = "Got lost; " ...
where
  tuples = makeTuples s
  disp = sumTuples tuples

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

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

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

Sidebar: top-level vs. hidden functions

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

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

Note: Types are not shown, to save space.
Consider a function `tally` that counts character occurrences in a string:

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

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

How can this problem be approached?

In a nutshell: `[(a',3),(b',2),( ',2),(g',1),(n',1),(e',1)]`
Let's start by writing `incEntry c` tuples, which takes a list of `(character, count)` tuples and produces a new list of tuples that reflects the addition of the character `c`.

\[
\text{incEntry} :: \text{Char} \rightarrow \left[ \left( \text{Char}, \text{Int} \right) \right] \rightarrow \left[ \left( \text{Char}, \text{Int} \right) \right]
\]

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

\[
\begin{align*}
&\text{> incEntry 't' []} \\
&\quad [ ('t',1)] \\
\end{align*}
\]

\[
\begin{align*}
&\text{> incEntry 'o' it} \\
&\quad [ ('t',1), ('o',1)] \\
\end{align*}
\]

\[
\begin{align*}
&\text{> incEntry 'o' it} \\
&\quad [ ('t',1), ('o',2)] \\
\end{align*}
\]
{- incEntry c tups

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

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

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

incEntry :: Char -> [(Char, Int)] -> [(Char, Int)]
incEntry c [] = [(c, 1)]
incEntry c ((char, count):entries)
  | c == char = (char, count+1) : entries
  | otherwise = (char, count) : incEntry c entries
Next, let's write \texttt{mkentries s}. It calls \texttt{incEntry} for each character in the string \texttt{s} in turn and produces a list of \texttt{(char, count)} tuples.

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

Usage:

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

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

Code:

\begin{verbatim}
mkentries :: [Char] -> [(Char, Int)]
mkentries s = mkentries' s []
    where
    mkentries' [ ] entries = entries
    mkentries' (c:cs) entries =
        mkentries' cs (incEntry c entries)
\end{verbatim}
{- insert, isOrdered, and sort provide an insertion sort -} insert v [ ] = [v]
insert v (x:xs)
    | isOrdered (v,x) = v:x:xs
    | otherwise = x:insert v xs

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

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

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

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

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

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

> tally "cocoon"
  o 3
c 2
n 1

- How does this solution exemplify functional programming? (slide 28+)
Running **tally** from the command line

Let's run it on lectura...

```
% code=/cs/www/classes/cs372/spring18/haskell

% cat $code/tally.hs

... *everything we've seen before and now a main:*
main = do
  bytes <- getContents  -- reads all of standard input
  tally bytes

% echo -n cocoon | runghc $code/tally.hs

  o 3
  c 2
  n 1
```
**tally** from the command line, continued

`$code/genchars N` generates `N` random letters:

```
% $code/genchars 20
KVQaVPEmClHRbgdkmMsQ
```

Let's tally a million letters:
```
% $code/genchars 1000000 |
time runghc $code/tally.hs >out
```

```
21.79user 0.24system 0:22.06elapsed
```

```
% head -3 out
s 19553
V 19448
J 19437
```
tally from the command line, continued

Let's try a compiled executable.

% cd $code
% ghc --make -rtsopts tally.hs
% ls -l tally
-rw-rw-r-x 1 whm whm 1118828 Jan 26 00:54 tally

% ./genchars 1000000 > 1m
% time ./tally < 1m > out
real 0m7.367s
user 0m7.260s
sys 0m0.076s
Here are user CPU times for implementations of *tally* in several languages. The same one-million letter file was used for all timings.

<table>
<thead>
<tr>
<th>Language</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>7.260</td>
</tr>
<tr>
<td>Ruby</td>
<td>0.548</td>
</tr>
<tr>
<td>Icon</td>
<td>0.432</td>
</tr>
<tr>
<td>Python 2</td>
<td>0.256</td>
</tr>
<tr>
<td>C w/ gcc -O3</td>
<td>0.016</td>
</tr>
</tbody>
</table>

However, our *tally* implementation is very simplistic. An implementation of *tally* by an expert Haskell programmer, Chris van Horne, ran in 0.008 seconds. (See `spring18/haskell/tally-cwvh[12].hs`.)

Then I revisited the C version (*tally2.c*) and got to 3x faster than Chris' version with a one-billion character file.
Real world problem: "How many lectures?"

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

"How many lectures will there be?"

How should we answer that question?

Google for a course planning app?
No! Let's write a Haskell program! 😊
One approach:

> classdays ...arguments...

#1 H 1/15  (for 2015...)
#2 T 1/20
#3 H 1/22
#4 T 1/27
#5 H 1/29

...

What information do the arguments need to specify?

First and last day
Pattern, like M-W-F or T-H
How about holidays?
Let's start with something simple:

> classdays (1,15) (5,6) [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
>

The first and last days are represented with \((month,day)\) tuples.

The third argument shows the pattern of class days: the first is a Thursday, and it's five days to the next class. The next is a Tuesday, and it's two days to the next class. Repeat!
Date handling

There's a `Data.Time.Calendar` module but writing two minimal date handling functions provides good practice.

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

> fromOrdinal 32
(2,1)   -- The 32\textsuperscript{nd} day of the year is February 1.
```

What's a minimal data structure that could help us?

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

(1,31) *The last day in January is the 31\textsuperscript{st} day of the year*

(7,212) *The last day in July is the 212\textsuperscript{th} day of the year*
toOrdinal and fromOrdinal

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

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

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

> toOrdinal (12,31)
   365

> fromOrdinal 32
   (2,1)
Recall:

> classdays (1,15) (5,6) [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
...

Ordinal dates for (1,15) and (5,6) are 15 and 126, respectively.

With the Thursday-Tuesday pattern we'd see the ordinal dates progressing like this:

15, 20, 22, 27, 29, 34, 36, 41, ...

\[ \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \uparrow \downarrow \] ...

+5  +2  +5  +2  +5  +2  +5  ...
Imagine this series of calls to a helper, `showLecture`:

```
showLecture 1 15 'H'
showLecture 2 20 'T'
showLecture 3 22 'H'
showLecture 4 27 'T'
...
showLecture 32 125 'T'
```

Desired output:

```
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
```

What computations do we need to transform
```
showLecture 1 15 'H'
```
into
```
"#1 H 1/15\n"
"
We have: `showLecture 1 15 'H'`
We want: "#1 H 1/15"

Let's write `showOrdinal :: Integer -> [Char]`

```
> showOrdinal 15
"1/15"
```

```
showOrdinal ordDay = show month ++ "/" ++ show day
where
  (month, day) = fromOrdinal ordDay
```

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

```
showLecture 1 15 'H'
showLecture 2 20 'T'
...
showLecture 32 125 'T'
```

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

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

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

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

Result:

\[
["#1 H 1/15\n","#2 T 1/20\n", ..., "#33 H 5/5\n"]
\]

Now let's write \texttt{showLectures}: 

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

```haskell
classdays first last pattern = putStrLn (concat result)
  where
      result =
        showLectures 1 (toOrdinal first) (toOrdinal last) pattern
```

Usage:

```haskell```
```text
> classdays (1,15) (5,6) [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
...
#31 H 4/30
#32 T 5/5
```

Full source is in `spring18/haskell/classdays.hs`
Errors
What syntax errors do you see in the following file?

```haskell
% cat synerrors.hs

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

where
    g x:xs = x
    y =
    g [x] + 5
    g2 x = 10
```
What syntax errors do you see in the following file?

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

% cat synerrors.hs

- Function name starts with cap.
- Use /= for inequality.
- Continuation should be indented.
- No = before guards.
- =, not == before result.
- Missing | before otherwise.
- Needs parens: (x:xs).
- Violates layout rule.
Type errors

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

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

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

> chr 'x'
  Couldn't match expected type `Int' with actual
type `Char'
In the first argument of 'chr', namely 'x'
In the expression: chr 'x'
In an equation for 'it': it = chr 'x'
```
Type errors, continued

Code and error:

\[
\begin{align*}
f x y &= \frac{}{\begin{array}{c} x == 0 = [] \\
\text{otherwise} = f x \end{array}} \\
\end{align*}
\]

Couldn't match type 'p0 -> [a]' with '[a]'
Expected type: t -> [a]
Actual type: t -> p0 -> [a]

The first clause implies that \( f \) returns \([a]\) but the second clause returns a partial application, of type \( p0 -> [a] \), a contradiction.
Type errors, continued

Code:

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

Error:

```haskell
Couldn't match expected type '(a1, b)'
  with actual type '[a] -> (a1, b)'
Probable cause: countEO is applied to too few arguments
  In the expression: countEO
```

What's the problem?

It's expecting a tuple, \((a1, b)\) but it's getting a function, \([a] \rightarrow (a1, b)\)

Typically, instead of errors about too few (or too many) function arguments, you get function types popping up in unexpected places.
Type errors, continued

Here's an example of omitting an operator:

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

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

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

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

Try it: See if a type declaration for `add3` leads to a better error.
Is there an error in the following?

\[ f \[\] = [\] \]
\[ f \[x\] = x \]
\[ f \langle x : xs \rangle = x : f \langle xs \rangle \]

\textbf{Occurs check: cannot construct the infinite type:} \( a \sim [a] \)
\textbf{Expected type:} [a]
\textbf{Actual type:} [[a]]  
\textbf{("a is a list of as"--whm)}

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

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

\textbf{Technique:} Comment out clauses (and/or guards) to find the troublemaker, or incompatibilities between them.
Recall $\text{ord} :: \text{Char} \rightarrow \text{Int}$.

Note this error:

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

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

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

instance Num Char doesn't appear
Debugging
Debugging in general

My advice in a nutshell:
  Don't need to do any debugging in Haskell!

My usual development process in Haskell:
  1. Work out expressions at the ghci prompt.
  2. Write a function using those expressions and put it in a file.
  3. Test that function at the ghci prompt.
  4. Repeat with the next function.

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

With Haskell I might write a half-dozen lines of code before trying them out.
The `trace` function

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

Observe:

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

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

What's happening?

- `trace string value` returns `value` but also outputs `string` as a side-effect. (!)
  - Great for debugging!
  - Completely subverts Haskell's isolation of the side-effects of output.
Here's a trivial function:
\[
\begin{align*}
    f \ 1 &= 10 \\
    f \ n &= n \times 5 + 7
\end{align*}
\]

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

Execution:
\[
\begin{align*}
    > f \ 1 \\
    f: \text{first case} \\
    10
\end{align*}
\]
\[
\begin{align*}
    > f \ 3 \\
    f: \text{default case} \\
    22
\end{align*}
\]
Let's add `trace` calls to `sumElems`:

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

Execution:

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

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

```haskell
import Debug.Trace

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

result = countEO xs
(evens, odds) =
  trace ("countEO " ++ show xs ++ " --> " ++ show result) result
```

Execution:

```
> countEO [3,2,4]
(countEO [] --> (0,0)
countEO [4] --> (1,0)
countEO [2,4] --> (2,0)
2,1)
```

Before tracing the `where` was:
```
(evens, odds) = countEO xs
```
Recall this clause for \texttt{buildingAtHeight}:
\[
\text{buildingAtHeight} \ (width, \ height, \ ch) \ n = \ \\
\text{replicate width } \ (\text{if } n \ > \ height \ \text{then} \ ' ' \ \text{else} \ ch)
\]

Outputting \texttt{width}, \texttt{height}, and \texttt{ch} with labels is tedious:
\[
\text{buildingAtHeight} \ (width, \ height, \ ch) \ n = \ \\
\text{trace } ("width " ++ \textbf{show width} ++ " , height: " ++ \textbf{show height} ++ " , ch: " ++ \textbf{show ch})
\]
\[
\text{replicate width } \ (\text{if } n \ > \ height \ \text{then} \ ' ' \ \text{else} \ ch)
\]

Example of trace output: \texttt{width: 3, height: 2, ch: 'x'}

We can use a tuple to simplify the \texttt{trace} call:
\[
\text{buildingAtHeight} \ (width, \ height, \ ch) \ n = \ \\
\text{trace } (\textbf{show } ("width:", \text{width}, \"height\", \text{height}, \"ch\", \text{ch}))
\]
\[
\text{replicate width } \ (\text{if } n \ > \ height \ \text{then} \ ' ' \ \text{else} \ ch)
\]

Example of trace output: ("width:",3,"height",2,"ch:','x")
Icon has a built-in tracing mechanism.

Here's `sumElems` in Icon:

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

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

I know of no better out-of-the-box tracing facility in any language.
ghci does have some debugging support but debugging is expression-based. Here's some simple interaction with it on `countEO`:

```
> :step countEO [3,2,4]
Stopped at countEO.hs:(1,1)-(6,29)
_result :: (t, t1) = _
> :step
Stopped at countEO.hs:3:7-11
_result :: Bool = _
x :: Integer = 3
> :step
Stopped at countEO.hs:3:15-29
_result :: (t, t1) = _
evens :: t = _
odds :: t1 = _
> :step
(Stopped at countEO.hs:6:20-29
_result :: (t, t1) = _
xs :: [Integer] = [2,4]
```

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

_result shows type of current expression

Arbitrary expressions can be evaluated at the > prompt (as always).
Excursion:
A little bit with infinite lists
and lazy evaluation
Infinite lists

Here's a way we've seen to make an infinite list:

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

In essence, what does the following bind \( f \) to?

```
> f = (!!) [1,3..]
f :: (Num a, Enum a) => Int -> a
```

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

- Yes, we could say \( f \ n = (n*2)+1 \) but that wouldn't be nearly as much fun! (This is functional programming!)

- I want you to be cognizant of performance but don't let concerns about performance stifle creativity!
Lazy evaluation

Consider the following binding. Why does it complete?
> fives = [5,10..]

Haskell uses *lazy evaluation*. Values aren't computed until needed. (Simplistic; we'll refine it.)

How will the following expression behave?
> take (head fives) fives
[5,10,15,20,25]
Lazy evaluation, continued

Here is an expression that is said to be non-terminating:

\[
> \text{length fives}
\]

...when tired of waiting...\(^C\) Interrupted.

The value of \text{length fives} is said to be \bot ("bottom").

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

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

\text{numFives} :: \text{Int}

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

What does the following do?

\[
> \text{numFives}
\]

...after a while...\(^C\) Interrupted.
Lazy evaluation, continued

We can use :print to explore lazy evaluation:

```haskell
google 8.2.2 behavior
```

```haskell
> fives = [5,10..]
> :print fives
fives = (_t1::{[Integer]})  -- ghci 7.8.3 behavior
fives = (_t1::{(Enum a, Num a) => [a])  -- 8.2.2
```n

```haskell
> take 3 fives
[5,10,15]
```

What do you think :print fives will now show?

```haskell
> :print fives
fives = 5 : 10 : 15 : (_t3::{[Integer]})  -- ghci 7.8.3
fives = (_t1::{(Enum a, Num a) => [a])  -- 8.2.2
```
Lazy vs. non-strict

In fact, Haskell doesn't fully meet the requirements of lazy evaluation. The word "lazy" appears only once in the Haskell 2010 Report.

What Haskell does provide is *non-strict evaluation*:
Function arguments are not evaluated until a value is needed.

Consider this function:
\[
\text{f } x \ y \ z = \text{if even } x \text{ then } y \text{ else } z
\]

What does the following expression produce?
\[
> \ f \ 4 \ 10 \ (\text{length} \ [1..])
10
\]

Why does it complete?
Because 4 is even, the value of z isn't needed and never computed.
Lazy vs. non-strict

At hand:

\[ f \times y \ z = \text{if even } x \text{ then } y \text{ else } z \]

How will the following behave?

```haskell
> a = f 4 (length [1..]) 100
> b = a + 1
> c = [1,b]
> length c
2
> head c
1
> c
[1, ^CInterrupted.
```

See [wiki.haskell.org/Lazy vs. non-strict](http://wiki.haskell.org/Lazy_vs._non-strict) for the fine points of lazy evaluation vs. non-strict evaluation. Google for more, too.
How is the following Java expression evaluated?
\[ x = f(g(x+3), h()) \]
A model:
1. \( t1 = x+3 \)
2. \( t2 = g(t1) \)
3. \( t3 = h() \)
4. \( x = f(t2, t3) \)

- Java uses strict evaluation
- Java guarantees left to right evaluation of argument lists (JLS 15.7)
- Contrast: C does not guarantee L-to-R evaluation of argument lists.

What's a case in which Java's L-to-R guarantee makes a difference?
- If \( g \) and \( h \) do output.
- Output is a side-effect!

What's a non-output case where order would make a difference?
Speculate: Can infinite lists be concatenated?
   > values = [1..] ++ [5,10..] ++ [1,2,3]
   >

What will the following do?
   > values > [1,2,3,5]
   False

False due to lexicographic comparison of fourth elements: 4 < 5

How far did evaluation of values progress?
   > :print values
   values = 1 : 2 : 3 : 4 : (_t2::[Integer]) -- ghci 7.8.3
What does the following expression mean?

\[ \text{threes} = 3 : \text{threes} \]

\text{threes} is a list whose head is 3 and whose tail is \text{threes}!

\[ > \text{take 5 threes} \]
\[ [3,3,3,3,3] \]

How about the following?

\[ > \text{xys} = ['x','y'] ++ \text{xys} \]

\[ > \text{take 5 xys} \]
\[ "xyxyx" \]

\[ > \text{xys !! 100000000} \]
\[ 'x' \]

One more:

\[ > x = 1 + x \]
\[ > x \]
\[^C\text{Interrupted}.\]
Problem: write a function \texttt{intsFrom} that produces the integers from a starting value. (No, you can't use \texttt{[n..]}!)

\begin{itemize}
  \item $\texttt{intsFrom 1}$
    \begin{itemize}
      \item \texttt{[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...]}
    \end{itemize}
  \item $\texttt{intsFrom 1000}$
    \begin{itemize}
      \item \texttt{[1000,1001,1002,1003,1004,1005,1006,1007,1008,...]}
    \end{itemize}
  \item $\texttt{take 5 (intsFrom 1000000)}$
    \begin{itemize}
      \item \texttt{[1000000,1000001,1000002,1000003,1000004]}
    \end{itemize}
\end{itemize}

Solution:

\texttt{intsFrom \ n = \ n : intsFrom (n + 1)}

Does \texttt{length (intsFrom (\texttt{minBound::Int})}} terminate?
Collaborative Learning Exercise

Infinite experimentation

[link to webpage]

Note to self: do push-cle 4
Higher-order functions
Recall this fundamental characteristic of a functional language: Functions are values that can be used as flexibly as values of other types.

Here are some more examples of that. What do the following do?

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

> funcs = (tail, (:) 100)

> nums = [1..10]

> fst funcs nums
[2,3,4,5,6,7,8,9,10]

> snd funcs nums
[100,1,2,3,4,5,6,7,8,9,10]
```
Lists of functions

Is the following valid?
> [take, tail, init]

Couldn't match type `[a2]' with `Int'
  Expected type: Int -> [a0] -> [a0]
  Actual type: [a2] -> [a2]
  In the expression: init

What's the problem?
  \texttt{take} does not have the same type as \texttt{tail} and \texttt{init}.

Puzzle: Make \texttt{[take, tail, init]} valid by adding two characters.
Can functions be compared?

> add == plus

No instance for (Eq (Integer -> Integer -> Integer))
  arising from a use of `=='
In the expression: add == plus

You might see a proof based on this in CSC 473:
If we could determine if two arbitrary functions perform the same computation, we could solve *the halting problem*, which is considered to be unsolvable.

Because functions can't be compared, this version of `length` won't work for lists of functions: (Its type: `(Num a, Eq t) => [t] -> a`)

```
len list@(_h:t)
  | list == [] = 0
  | otherwise = 1 + len t
```
A simple higher-order function

Definition: A higher-order function is a function that (and/or)
• Has one or more arguments that are functions
• Returns a function

twice is a higher-order function with two arguments: f and x
\[
\text{twice } f \ x = f \ (f \ x)
\]

What does it do?
> \text{twice tail [1,2,3,4,5]}
[3,4,5]

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

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

Let's make the precedence explicit:

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

Consider a partial application...

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

\[
> \text{twice } f \ x = f (f \ x)
\]
\[
> \text{twice } \text{tail} \ [1,2,3,4,5]
[3,4,5]
\]

Let's give \texttt{twice} a partial application!

\[
> \text{twice } (\text{drop } 2) \ [1..5]
[5]
\]

Let's make a partial application with a partial application!

\[
> \text{twice } (\text{drop } 5)
<\text{function}>
> \text{it } ['a'..'z']
"klmnopqrstuvwxyz"
\]

Try these!

\[
\text{twice } (\text{twice } (\text{drop } 3)) \ [1..20]
\text{twice } (\text{twice } (\text{take } 3)) \ [1..20]
\]
At hand:
\[
twice \ f \ x = f \ (f \ x)
\]
What's the the type of \texttt{twice}?
\[
> \ :t \ twice \\
\texttt{twice :: (t -> t) -> t -> t}
\]
Parentheses added to show precedence:
\[
\texttt{twice :: (t -> t) -> (t -> t)}
\]
\[
\texttt{twice f x = f (f x)}
\]
What's the correspondence between the elements of the clause and the elements of the type?

A \textit{higher-order function} is...
a function that (1) has one or more arguments that are functions and/or (2) returns a function.
The **map** function
Recall \( \text{double } x = x \times 2 \)

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

\[
\begin{aligned}
&> \text{map double [1..5]} \\
&\quad [2,4,6,8,10] \\
&> \text{map length (words "a few words"')} \\
&\quad [1,3,5] \\
&> \text{map head (words "a few words"')} \\
&\quad "afw"
\end{aligned}
\]

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

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

What is its type?
\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

What's the relationship between the length of \( \text{map} \)'s input and output lists?
The lengths are \textbf{always} the same.
Mapping (via `map`) is applying a transformation (a function) to each of the values in a list, always producing a new list of the same length.

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

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

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

```
> map head (map show it) -- Note: show True is "True"
"FTFFFTFFFTFFF"
```
Sidebar: `map` can go parallel

Here's another map:

```haskell
> map weather [85,55,75]
["Hot!","Cold!","Nice"]
```

This is equivalent:

```haskell
> [weather 85, weather 55, weather 75]
["Hot!","Cold!","Nice"]
```

- If functions have no side effects, we can immediately turn a mapping into a parallel computation.
- We might start each function call on a separate processor and combine the values when all are done.
map and partial applications

What's the result of these?

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

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

```haskell
> map (replicate 5) "abc"
["aaaaa","bbbbbb","cccccc"]
```
map and partial applications, cont.

What's going on here?
> f = map double
> f [1..5]
[2,4,6,8,10]

> map f [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]

Here's the above in one step:
> map (map double) [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]

Here's one way to think about it:
[(map double) [1..3],(map double) [10..15]]
Sections

Instead of using `map (add 5)` to add 5 to the values in a list, we should use a `section` instead: (it's the idiomatic way!)

```haskell
> map (5+) [1,2,3]
[6,7,8]  -- [5+ 1, 5+ 2, 5+ 3]
```

More sections:

```haskell
> map (10*) [1,2,3]
[10,20,30]

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

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

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

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

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

\[\text{map } (\leqslant) \ [1..4] \quad [3 \leqslant 1, 3 \leqslant 2, 3 \leqslant 3, 3 \leqslant 4] \quad \text{[False,False,True,True]}\]

\[\text{map } (\leqslant 3) \ [1..4] \quad [1 \leqslant 3, 2 \leqslant 3, 3 \leqslant 3, 4 \leqslant 4] \quad \text{[True,True,True,False]}\]

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

\[\text{twice } (+5) \ 3 \quad 13\]
travel, revisited
Now that we're good at recursion...

Some of the problems on the next assignment will encourage working with higher-order functions by prohibiting you from writing any recursive functions!

Think of it as isolating muscle groups when weight training.

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

_Pretend that you don't understand recursion!_

_What's a base case? Is it related to baseball?_
_Why would a function call itself? How's it stop?_
_Is a recursive plunge refreshing?_

If you were UNIX machines, I'd do _chmod 0_ on an appropriate section of your brains.
Recall our traveling robot: (slide 214+)

> travel "nnee"
"Got lost"

> travel "nnss"
"Got home"

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

How can we solve it without writing any recursive functions?
Recall:

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

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

Now what?

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

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

```
> disps = map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
```

We want: (2,3)

Any ideas?

```
> :t fst
fst :: (a, b) -> a
```

```
> map fst disps
[0,0,1,1,0]
```

```
> map snd disps
[1,1,0,0,1]
```
We have:

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

We want: (2,3)

Ideas?

\[
\text{sum} :: \text{Num } a => [a] \rightarrow a
\]

\[
\text{sum (map fst disps), sum (map snd disps))} \\
(2,3)
\]
travel :: [Char] -> [Char]

travel s
  | totalDisp == (0,0) = "Got home"
  | otherwise = "Got lost"

where
  disps = map mapMove s
  totalDisp = (sum (map fst disps),
               sum (map snd disps))

Did we have to understand recursion to write this version of travel?
  No.

Did we write any recursive functions?
  No.

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

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

```
> filter isDigit "(800) 555-1212"
"8005551212"
```

What's `filter f list` doing?
Producing the values in `list` for which `f` returns `True`.

Note: Think of `filter` as filtering in, not filtering out.

What is the type of `filter`?

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

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

More...

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

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

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

*Note that (`elem` ...) is a section.*

```
elem :: Eq a => a -> [a] -> Bool
```
At hand:

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

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

Problem: Write filter!

filter _ [] = []
filter f (x:xs)
  | f x = x : filteredTail
  | otherwise = filteredTail
where
  filteredTail = filter f xs
Prelude functions that use predicates

Several Prelude functions use predicates. Here are two:

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

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

How could we find other Prelude functions that use predicates?

```
% grep "(a -> Bool)" prelude-8.0.1.txt
```
For reference:

> map double [1..10]
[2,4,6,8,10,12,14,16,18,20]

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

**map:**
transforms a list of values
length *input* == length *output*

**filter:**
selects values from a list
0 <= length *output* <= length *input*

*map* and *filter* are in Python and JavaScript, to name two of many languages having them. (And, they're trivial to write!)
Anonymous functions
Imagine that for every number in a list we'd like to double it and then subtract five.

Here's one way to do it:

\[
> f \ n = n \times 2 - 5
\]

\[
> \text{map } f \ [1..5]
\]
\[-3, -1, 1, 3, 5\]

We could instead use an *anonymous function* to do the same thing:

\[
> \text{map } (\\ n \rightarrow n \times 2 - 5) \ [1..5]
\]
\[-3, -1, 1, 3, 5\]

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

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

vs.
\[
\text{map}(\lambda n \rightarrow n \times 2 - 5)[1..5]
\]

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

Anonymous functions...
- Directly associate a function's definition with its only use.
- Let us avoid the need to think up a good name for a function! 😊
- Can be likened to not using an intermediate variable:
  \[
  \begin{align*}
  \text{int } t &= a \times 3 + g(a+b); \quad // \text{Java} \\
  \text{return } f(t);
  \end{align*}
  \]
  vs.
  \[
  \text{return } f(a \times 3 + g(a+b));
  \]
Anonymous functions, continued

The general form of an anonymous function:

\( \text{pattern}_1 \ldots \text{pattern}_N \rightarrow \text{expression} \)

Simple syntax suggestion: enclose the whole works in parentheses.

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

These terms are synonymous with "anonymous function":

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

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

What will ghci say?

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

\(\lambda x \; y \to x + y \times 2\) is an expression whose value is a function.

Here are three ways to bind the name `double` to a function that doubles a number:

```haskell
double x = x * 2

double = \x -> x * 2

double = (*2)
```
Anonymous functions are commonly used with higher order functions such as `map` and `filter`.

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

```haskell
> map (\c -> "{" ++ [c] ++ "}") "anon."
["{a"","{n"","{o"","{n"","{.}""
```

```haskell
> filter (\x -> head x == last x) (words "pop top suds")
["pop","suds"]
```
From the previous slide:

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

A rough Java analogy: (spring18/haskell/javamap.java)

```java
ArrayList<Object[]> result = new ArrayList<>();

for (String s: "a test now".split(" "))
    result.add(new Object[]{s.length(), s});
```

An anonymous function given to `map` is a bit like the body of an enhanced `for` in Java.

Challenge: Rewrite in Java the other two examples on the previous slide.
Sidebar: Three languages

A simple anonymous function in Haskell...

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

Python...

```python
>>> lambda s: s + '-' + str(len(s))
<function <lambda> at 0x10138af28>
>>> _('abc')  # underscore is like Haskell's it
'abc-3'
```

and JavaScript...

```javascript
> f = function (s) { return s + '-' + s.length; }
> f("abc")
"abc-3"
```
Larger example: longest
Example: longest line(s) in a file

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

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

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

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

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

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

To avoid wading into input yet, let's focus on a function that operates on the full contents of a file as a single string:

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

\[
\text{> bytes = "data\nto\ntest\n"}
\]

\[
\text{> lns = lines bytes}
\]

\[
\text{["data","to","test"]}
\]

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

Let's use \texttt{zip3} and \texttt{map length} to create (length, line-number, line) triples:

\[
\text{> triples = zip3 (map length lns) [1..] lns}
\]

\[
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
\]
We have (length, line-number, line) triples at hand:

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

Let's use `Data.List.sort :: Ord a => [a] -> [a]` on them:

\[
> \text{sortedTriples} = \text{reverse} (\text{Data.List.sort} \text{triples})
\]

\[
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
\]

Note that by having the line length first, the triples are sorted first by line length. Ties are resolved by line number, which is second.

We reverse the list to put the tuples in descending order.

If line length weren't first in the tuple, we could instead use `Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]`

What is `sortBy`'s first argument?
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
> first (len, _, _) = len
> maxLength = first (head sortedTriples)
4
```

We'll be using `first` elsewhere but if we weren't, we'd bind `maxLength` using a pattern:

```haskell
(maxLength,_,_):_ = sortedTriples
```
At hand:

> sortedTriples

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

> maxLength

4

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

> maxTriples = takeWhile

(\triple -> first triple == maxLength) sortedTriples

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

anonymous function for `takeWhile`
At hand:

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

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

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

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

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

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

Look, Ma! No conditional code!
At hand:

> longest "data\nto\ntest\n" "l: data\n3: test\n"

Let's add a `main` that handles command-line args and does I/O:

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

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

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

Execution:

```
$ runghc longest.hs /usr/share/dict/words # lectura
39886:electroencephalograph's
```
Composition
Given two functions \( f \) and \( g \), the \textit{composition} of \( f \) and \( g \) is a function \( c \) that for all values of \( x \), \((c \ x)\) equals \((f \ (g \ x))\).

Here is a function that applies two functions in turn:

\[
\text{compose } f \ g \ x = f \ (g \ x)
\]

How many arguments does \texttt{compose} have?

Its type:

\[
(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

\[
> \text{compose init tail [1..5]} \\
> \text{[2,3,4]}
\]

\[
> \text{compose signum negate 3} \\
> -1
\]
Haskell binds the symbolic variable dot to a "compose" function:

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

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

> numwords = length . words

> numwords "just testing this"
3

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

From previous slide: compose :: (b -> c) -> (a -> b) -> a -> c
Composition, continued

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

\[
\texttt{ntl} = \text{head} \ . \ \text{tail} \ . \ \text{reverse}
\]

Two solutions:

\[
\texttt{ntl} = \text{last} \ . \ \text{init}
\]

Problem: Recall \texttt{twice \ f \ x = f (f \ x)}. Define \texttt{twice} as a composition.

\[
\texttt{twice \ f = f \ . \ f}
\]
Problem: Create a function to remove the digits from a string:
  > rmdigits "Thu Feb  6 19:13:34 MST 2014"
  "Thu Feb :: MST "

Solution:
  > rmdigits = filter (not . isDigit)

Given the following, describe \( f \):
  > f = (*2) . (+3)

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

Would an anonymous function be a better choice for \( f \)'s computation?
Given the following, what's the type of `numwords`?

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

length :: [a] -> Int

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.

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
```
Consider the following:
> s = "It's on!"
> map head (map show (map not (map isLetter s)))
"FFTFTFFFT"

Can we use composition to simplify it?
> map (head . show . not . isLetter) s
"FFTFTFFFT"

Question: Is
map f (map g x)
equivalent to the following?
map (f . g) x

If f and g did output, how would the output of the two cases differ?
What would be a better name for the following function?

\[
f_2 = f \cdot f
\]

where \( f = \text{reverse} \cdot \text{dropWhile isSpace} \)

Credit: Eric Normand on Stack Overflow
Sidebar: A little Standard ML

- explode;
val it = fn : string -> char list

- implode;
val it = fn : char list -> string

- rev;
val it = fn : 'a list -> 'a list

Problem: Write `revstr s`, which reverses the string `s`.

- revstr "backwards";
val it = "sdrawkcab" : string

Solution:

- val revstr = implode o rev o explode;
val revstr = fn : string -> string

`sml` runs Standard ML on lectura
Point-free style
(video)
Recall \texttt{rmdigits}:
\begin{verbatim}
> rmdigits "Thu Feb 6 19:13:34 MST 2014"
"Thu Feb :: MST 
\end{verbatim}

What is the difference between these two bindings for \texttt{rmdigits}?
\begin{verbatim}
rmdigits s = filter (not . isDigit) s
\end{verbatim}

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

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

A point-free binding of a function \texttt{f} has NO parameters!
Point-free style, continued

I think of point-free style as a natural result of fully grasping partial application and operations like composition.

Although it was nameless, we've already seen examples of point-free style, such as these:

\[
\text{nthOdd} = (!!) \ [1,3..] \\
\text{t2} = \text{twice tail} \\
\text{numwords} = \text{length . words} \\
\text{ntl} = \text{head . tail . reverse}
\]

There's nothing too special about point-free style but it does save some visual clutter. It is commonly used.

The term "point-free" comes from topology, where a point-free function operates on points that are not specifically cited.
Point-free style, continued

Problem: Using point-free style, bind \texttt{len} to a function that works like the Prelude's \texttt{length}.

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

> const 10 20
   10

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

Solution:
\begin{verbatim}
len = sum . map (const 1)
\end{verbatim}

See also: \textit{Tacit programming} on Wikipedia