REPLACEMENT SET!
DISCARD the Haskell set you received on January 10!

Functional Programming with Haskell

CSC 372, Spring 2018
The University of Arizona
William H. Mitchell
whm@cs
Paradigms
Thomas Kuhn's *The Structure of Scientific Revolutions* (1962) describes a *paradigm* as a scientific achievement that is...

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

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

Examples of 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?
The procedural paradigm, continued

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 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
Multiple paradigms (?)

Paradigms in a field of science are often incompatible.
   Example: geocentric vs. heliocentric model of the universe

Imperative programming is used both with procedural and object-oriented programming.
   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?
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.
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);
}
```

```c
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
Value, type, and side effect

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

Here are some Java expressions:

```java
'x'
i + j * k
f(args.length * 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

1 < 2

"abc".charAt(1)

s = 3 + 4 + "5"

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

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

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

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

3 + 4

1 < 2

"abc".charAt(1)

s = 3 + 4 + "5"

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

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

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

\[ x + 3 \times y \]

\[ x += 3 \times y \]

\[ s.length() > 2 \ | \ | \ s.charAt(1) == '#' \]
Value, type, and side effect, continued

More expressions to consider wrt. side effects:

```
"testing".toUpperCase()

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

System.out.println("Hello!")

window.checkSize()
```
The hallmark of imperative programming

Side effects are the hallmark of imperative programming.

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

Recall:

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

Can we program without side effects?
The Functional Paradigm
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
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.
Haskell resources

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](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

Haskell 2010 Report (I'll call it H10.)
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.
Interacting with Haskell

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.
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)
The ~/.ghci file

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

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

:set prompt "> "
import Text.Show.Functions

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

*The second line is very important:*
- It loads a module that lets functions be printed.
- Prints <function> for function values.
- Without it, lots of examples in these slides won't work!
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 output and put it in a plain text file, `eca1.txt`. No need for your name, NetID, etc. in the file. No need to edit out errors.

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

[Link to Haskell by Observation](cs.arizona.edu/classes/cs372/spring18/cle-haskell-obs.html)
Functions and function types
In Haskell, *juxtaposition* indicates a function call:

```haskell
> negate 3
-3

> even 5
False

> pred 'C'
'B'

> signum 2
1
```

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

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

\begin{verbatim}
> signum negate 2
<interactive>:1|1: error:
  • Non type-variable argument ...
...

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

\begin{verbatim}
> signum (negate 2)
-1
\end{verbatim}
\end{verbatim}
Function call has higher precedence than any operator.

> negate 3+4

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

> negate (3 + 4)
-7

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

> toUpper 'a'

> ord 'A'

> chr 66

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

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

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

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

The text

```haskell
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

> :t ord

> :t chr
What is the type of the following Java methods?

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

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

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

Important:
• Java: common to think of a method's return type as the method's type
• Haskell: function's type has both type of argument(s) and return type
Type consistency

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

Here is an example of an inconsistency:

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

> :t chr
chr :: Int -> Char

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

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

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

'\texttt{a}'

\texttt{isUpper}

\texttt{isUpper 'a'}

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

\begin{align*}
\texttt{'a'} &:: \texttt{Char} \\
\texttt{chr} &:: \texttt{Int} \rightarrow \texttt{Char} \\
\texttt{digitToInt} &:: \texttt{Char} \rightarrow \texttt{Int} \\
\texttt{intToDigit} &:: \texttt{Int} \rightarrow \texttt{Char} \\
\texttt{isUpper} &:: \texttt{Char} \rightarrow \texttt{Bool} \\
\texttt{not} &:: \texttt{Bool} \rightarrow \texttt{Bool} \\
\texttt{ord} &:: \texttt{Char} \rightarrow \texttt{Int} \\
\texttt{toUpper} &:: \texttt{Char} \rightarrow \texttt{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
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?

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

- **Eq**: All except IO, (->)
- **Ord**: All except IO, IOError, (->)
- **Num**: Int, Integer, Float, Double
- **Show**: All except IO, (->)
- **Read**: All except IO, (->)
- **Monad**: IO, [], Maybe
- **MonadPlus**: IO, [], Maybe
- **Bounded**: Int, Char, Bool, (()), Ordering, tuples
- **Real**: Int, Integer, Float, Double
- **Fractional**: Float, Double
- **Floating**: Float, Double
- **RealFloat**: Float, Double
- **Integral**: Int, Integer
- **Enum**: (()), Bool, Char, Ordering, Int, Integer, Float, Double
- **RealFrac**: Float, Double

Adapted from [http://en.wikibooks.org/wiki/Haskell/Classes_and_types](http://en.wikibooks.org/wiki/Haskell/Classes_and_types)
The arrow from Num to Fractional means that a Fractional can be used as a Num.

Given

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

and

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

then

\[
\text{negate} \ 5.0 \ \text{is valid.}
\]
What does the diagram show us other than the relationship between **Num** and **Fractional**?
The Prelude has a `trunc`ate function:

```plaintext
> truncate 3.4
3
```

What does the type of `truncate` tell us?

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

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

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

```haskell
> :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 **Bounded**:

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

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

How can we use them?
The construct \texttt{::type} is an expression type signature.

A usage of it:

\begin{itemize}
    \item \texttt{minBound::Char}
    \item \texttt{maxBound::Int}
    \item \texttt{maxBound::Bool}
    \item \texttt{maxBound::Integer}
\end{itemize}
We can use `:set +t` to direct `ghci` to automatically show types:

> :set +t

> 3
3
it :: Num p => p

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

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

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

```plaintext
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:

> double x = x * 2
double :: Num a => a -> a  (\textit{\texttt{: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:

\textit{function-name parameter} = \textit{expression}

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

Two more functions:

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

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

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

Problem: Write \texttt{isPositive \ x} which returns \texttt{True} iff \texttt{x} is positive.
We can use :: type to constrain a function's type:

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

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

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

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

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

We'll use :: type to simplify 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/
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
```
Sidebar: My usual edit-run cycle

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

```bash
alias hs=ghci
```

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

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

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

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

Lather, rinse, repeat.
Functions with multiple arguments
Here's a function that produces the sum of its two arguments:
   > add x y = x + y :: Int

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

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

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

But what does that mean?
Multiple arguments, continued

Recall our negate function:

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

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

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

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

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

Call \textbf{add} with the value 3, producing a nameless function. Call that nameless function with the value 5.
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
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:

```haskell
> 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?

```haskell
> plusThree 5
```
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:

\[
\begin{align*}
> \text{add } x \ y &= x + y :: \text{Int} \\
\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad -- \text{parens added}
\end{align*}
\]

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

\[
\begin{align*}
> \text{add3 } x \ y \ z &= x + y + z :: \text{Int} \\
\text{add3} :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))
\end{align*}
\]

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?
Another model of partial application

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

> \( f \ x \ y = (y \times x + x) :: \text{Int} \) \quad -- f :: \text{Int} \to \text{Int} \to \text{Int} \\

> \( g = f 3 \) \\
g :: \text{Int} \to \text{Int} \\

-- as if we'd done this: \( g \ y = y \times 3 + 3 \) \\

> \( g 5 \) \\
18 

Everybody: Try it!

> \( f 3 5 \) \\
18
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):
  
  \[
  \text{name } \text{param1 param2 ... paramN } = \text{expression}
  \]

• 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
  
  \text{f x y z}

  means
  
  \text{((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.
A fundamental characteristic of a functional language: **functions are values** that can be used as flexibly as values of other types.

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

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

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

```haskell
> plus = add
```

Either name can be used to reference the function value:

```haskell
> add 3 4
7
> plus 5 6
11
```
Functions as values, continued

What does the following suggest to you?

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

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

... infixl 6 +

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

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

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

> (+) 3 4
7

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

> 3 `add` 4
7

> 11 `rem` 3
2

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

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

Usage:

```
> 100 +% 1
```

```
> 12 +% 25
```

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>!!</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>^, ^^, **</td>
</tr>
<tr>
<td>7</td>
<td>*, /, <code>div</code>, <code>mod</code>, <code>rem</code>, <code>quot</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+, -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>:, ++</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><code>==</code>, <code>/=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>elem</code>, <code>notElem</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></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;=</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td><code>$</code>, <code>!</code>, <code>seq</code></td>
</tr>
</tbody>
</table>

Note: From page 51 in Haskell 2010 report
Type Inferencing
Haskell does type inferencing:
The types of values are inferred based on the operations performed on the values.

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` is inferred to be a `Char`.
Recall `ord` in the `Data.Char` module:

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

What type will be inferred for `f`?

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

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

Because `x` is being compared to 0, Haskell also infers that the type of `x` must be 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 \texttt{x} as both a \texttt{Num} and a \texttt{Char}.

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

What does the error "No instance for (Num Char)" mean?
Type Specifications
Type specifications for functions

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

Here's a file with several functions preceded by their types:

```hs
% 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`?

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

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

Challenge: Without using `::type`, show an expression that works with `add1` but fails with `add2`.
Two pitfalls related to type specifications for functions:

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

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

Recommendation:
Try writing functions without a type specification and see what type gets inferred. If the 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:

```%
% 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:

```
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:

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

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

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

Hint: guards are tried in turn.
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`?

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?
"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 \texttt{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.

> factorial 40

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

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

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]
```

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

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

```haskell
> length []
```

What's the type of `length`?

```haskell
> :type length
```

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

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

> head [3,4,5]

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

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

> tail [3,4,5]

[4,5]

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

**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]

> it ++ reverse(it)

What are the types of ++ and reverse?

> :type (++)

> :type reverse
Haskell has an *arithmetic sequence notation*:

```
> [1..20]

> [-5,-3..20]

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

```haskell
> drop 3 [1..10]
```

```haskell
> take 5 [1.0,1.2..2]
```
Problem: **halves**

Problem:

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

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

> halves([1])
[[],[1]]
```

`halves` will be a little repetitious because we don't have the `where` clause in our toolbox yet.
Solution: halves
The `!!` operator produces a list's Nth element, zero-based:

\[
> [10,20..100] !! 3
\]

\[
> :type (!!)
\]

Speculate: do negative indexes work?

\[
> [10,20..100] !! (-2)
\]

Important:

Much use of `!!` might indicate you're writing a Java program in Haskell!
Haskell lists are values and can be compared as values:

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


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

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

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

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

Example:
> \([1,2,3]\) < \([1,2,4]\)
We can make lists of lists.

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

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

What's the length of `x`?

```haskell
> length x
```
More examples:

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

> head x

> tail x

> x !! 1 !! 2

> head (head (tail (tail x)))
Earlier I showed you this:
\[ \text{length} :: [a] \rightarrow \text{Int} \]

Around version 7.10 \textit{length} was generalized to this:
\[ \text{length} :: \text{Foldable} \ t \Rightarrow t \ a \rightarrow \text{Int} \]

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

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

Pretend this \[ \text{sum} :: \text{Num} \ a \Rightarrow [a] \rightarrow a \]

Instead of \[ \text{minimum} :: (\text{Ord} \ a, \text{Foldable} \ t) \Rightarrow t \ a \rightarrow a \]

Pretend this \[ \text{minimum} :: \text{Ord} \ a \Rightarrow [a] \rightarrow a \]
Strings in Haskell are simply lists of characters.

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

> ['a..'z']

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

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

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

> length asciiLets

> reverse (drop 26 asciiLets)

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

> isAsciiLet c = c `elem` asciiLets
```
The Prelude defines \textbf{String} as \texttt{[Char]} (a \emph{type synonym}).
\begin{verbatim}
> :info String
type String = [Char]
\end{verbatim}

A number of functions operate on \texttt{Strings}. Here are two:
\begin{verbatim}
> :type words
words :: String -> [String]

> :type unwords
unwords :: [String] -> String
\end{verbatim}

What's the following doing?
\begin{verbatim}
> unwords (tail (words "Just some words!"))
\end{verbatim}
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).

\[
\begin{align*}
> & 5 : [10, 20, 30] \\
> & [5, 10, 20, 30]
\end{align*}
\]

What's the type of the cons operator?

\[
\begin{align*}
> & \text{type (:)}
\end{align*}
\]
The `cons` (:) operation forms a new list from a value and a list.

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

> head c
5

> tail c
[10,20,30]

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

A cons node can be referenced by multiple cons nodes.

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

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

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

What are the values of the following expressions?

> 1:[2,3]

> 1:2

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

> []:[]

> [1,2]:[]

> []:[1]

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

```haskell
> a = [5,10,20,30]
> h = head a
> h
5

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

> t2 = tail (tail t)
> t2
[30]
```
A little on performance

What operations are likely fast with cons lists?

What operations are likely slower?

With cons lists, what does list concatenation involve?

> m=[1..10000000]
> length (m++[0])
100000001
The head of a list is a one-element list.

The tail of a list is a list.

The tail of an empty list is an empty list.

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

A cons list is essentially a singly-linked list.

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

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

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

\begin{verbatim}
fromTo 1 3
1 : fromTo (1+1) 3
1 : fromTo 2 3
1 : 2 : fromTo (2+1) 3
1 : 2 : fromTo 3 3
1 : 2 : 3 : fromTo (3+1) 3
1 : 2 : 3 : fromTo 4 3
1 : 2 : 3 : []
\end{verbatim}

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

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

Here's a simple example of a list comprehension:

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

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

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

Chapter 5 in Hutton has some very interesting examples of practical computations with list comprehensions.
A little output
The `putStr` function outputs a string:

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

Here's the type of `putStr`:

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

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

The construction `()` is read as "unit". The unit type has a single value, `unit`. Both the type and the value are written as `()`. 
For the time being, we'll use this approach for functions that produce output:

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

- The top-level function will call the helper function and then call `putStr` with the helper function's result.
A little output, continued

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

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

```haskell
> show 10
```

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

```haskell
> show show
```
Let's write a function to print the integers from 1 to N:

> printN 3
1
2
3

First, let's write a helper, \texttt{printN'}:

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

Solution: (does appear on next slide)
At hand:

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

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

Usage:

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

Let's write the top-level function:

\[
\text{printN} :: \text{Integer} \rightarrow \text{IO} () \quad \\
\text{printN} \ n = \text{putStrLn} (\text{printN'} \ n)
\]
All together, as a file:

```haskell
% 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
```

**printN, continued**
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:

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

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

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

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

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

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

```haskell
charbox' :: Int -> Int -> Char -> String
```

Test:

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

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

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

- 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!