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
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?
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?
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
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:

\[
\begin{align*}
&'x' \\
i + j \times k \\
f(\text{args.length} \times 2) + n
\end{align*}
\]

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?

- $3 + 4 = 7$
- $1 < 2 = true$
- $"abc".charAt(1) = 'b'$
- $s = 3 + 4 + "5" = "75$"
- $"a,bb,c3".split("\,").split("\,")[2] = "c3$"
- $"a,bb,c3".split("\,").charAt(0) == 'X' = false$
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?"
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.
Value, type, and **side effect**, continued

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
#include <stdio.h>

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

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 (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.
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.
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)
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,

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

the suggested fix should work.

Details on .ghci 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
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:
   ```haskell
   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:
   ```bash
   % 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:

```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.
Calling functions, continued

Function call with juxtaposition is left-associative.

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

\begin{verbatim}
> signum negate 2
<interactive>:11:1: error:
  • Non type-variable argument ... 
  ...
\end{verbatim}

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

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

```haskell
> negate 3+4
1

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

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

```haskell
> 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  \textit{(import the Data.Char module)}

> isLower 'b'
True

> toUpper 'a'
'A'

> ord 'A'
65

> chr 66
'B'

> Data.Char.ord 'G' \textit{(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.

\[
\begin{align*}
'a' &:: \text{Char} \\
isUpper &:: \text{Char} -> \text{Bool} \\
isUpper ('a') &:: \text{Bool} \\
\text{not} (\text{isUpper} ('a')) &:: \text{Bool} \\
\text{not not} (\text{isUpper} ('a')) &:: \text{bool} \\
toUpper (\text{ord} 97) &:: \text{Char} \\
isUpper (\text{toUpper} (\text{chr} 'a')) &:: \text{bool} \\
isUpper (\text{intToDigit} 100) &:: \text{Char}
\end{align*}
\]
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
What's the type of `negate`?

Recall the `negate` function:

> negate 5
-5

> negate 5.0
-5.0

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

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

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

Haskell also has **type classes**.

Type class:

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

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

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

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

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

The Prelude defines these types as instances of **Num**
Here's the type of \texttt{negate}:

\begin{verbatim}
> :type negate
negate :: Num a => a -> a
\end{verbatim}

The type of \texttt{negate} is specified using a \textit{type variable}, \texttt{a}.

The portion \texttt{a -> a} specifies that \texttt{negate} returns a value having the same type as its argument.

"\textit{If you give me an X, I'll give you back an X.}"

The portion \texttt{Num a =>} is a \textit{class constraint}. It specifies that the type \texttt{a} must be an instance of the type class \texttt{Num}.

How can we state the type of \texttt{negate} in English?

\texttt{negate} \textit{accepts any value whose type is an instance of Num.}

\textit{It returns a value of the same type.}
What type do integer literals have?

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

> :type (-27) -- Note: Paren 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?
Type classes, continued

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

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

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

and

```
    5.0 :: Fractional a => a
```

then

```
    negate 5.0 is valid.
```
What does the diagram show us other than the relationship between \texttt{Num} and \texttt{Fractional}?

It shows us types that are instances of \texttt{Num} and \texttt{Fractional}.

Do \texttt{:info Num} again. Do \texttt{:info Fractional}, too.
The Prelude has a `truncate` function:

```
> truncate 3.4
3
```

What does the type of `truncate` tell us?

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

`truncate` accepts any type that is an instance of `RealFrac`

`truncate` returns a type that is an instance of `Integral`

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

\texttt{:info Type} shows the classes that \texttt{Type} is an instance of.

\begin{verbatim}
> :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
\end{verbatim}

Try \texttt{: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`:

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

What sort of things are `minBound` and `maxBound`?
Polymorphic values!

How can we use them?
Polymorphic values

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

A usage of it:

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

> maxBound :: Int
  9223372036854775807

> maxBound :: Bool
  True

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

> :set +t

> 3
3
it :: Num p => p

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

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

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

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

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:

\[
> \text{double } x = x \ast 2
\]
\[
\text{double :: Num } a \Rightarrow a \rightarrow a \quad (\text{\texttt{:\texttt{set } +t \texttt{ is } in \texttt{ effect)}})
\]

\[
> \text{double } 5
10
\]
\[
it :: \text{Num } a \Rightarrow a
\]

\[
> \text{double } 2.7
5.4
\]
\[
it :: \text{Fractional } a \Rightarrow a
\]

General form of a function definition for the moment:

\[
\text{function-name parameter } = \text{expression}
\]

Function and parameter names must begin with a lowercase letter or an underscore.
Two more functions:

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

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

\[ \text{toCelsius temp } = (\text{temp } - 32) \times \frac{5}{9} \]

\[ \text{toCelsius :: Fractional } a \Rightarrow a \rightarrow a \]

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

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

\[ \text{isPositive } x = x > 0 \]

\[ \text{isPositive :: (Num } a, \text{ Ord } a \Rightarrow a \rightarrow \text{ Bool} \]
Simple functions, continued

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.
Sidebar: loading functions from a file

We can put function definitions in a file.

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

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

> :browse
double :: Int -> Int
neg :: Int -> Int
isPositive :: Int -> Bool
toCelsius :: Double -> Double
```
Sidebar: My usual edit-run cycle

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

```
alias hs=ghci
```

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

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

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

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

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

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

> add x y = x + y :: Int

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

> add 3 5
8

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?
Recall our negate function:

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

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

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

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

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

Call \texttt{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
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:

```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
8
it :: Int
```
Partial application, continued

At hand:

> \texttt{add }x \ y = x + y :: \texttt{Int}
\texttt{add :: Int -> (Int -> Int) -- parens added}

> \texttt{plusThree = add 3}
\texttt{plusThree :: Int -> Int}

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

Analogy: \texttt{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.
Partial application, continued

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...
- A parameter is dropped
- The argument's value is "wired" into the expression
- The result is a new function with one less parameter

\[ f(x, y) = (y \times x + x) :: \text{Int} \quad \text{-- } f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ g = f \, 3 \]
\[ g :: \text{Int} \rightarrow \text{Int} \]
\[ \text{-- as if we'd done this: } g(y) = y \times 3 + 3 \]

\[ g \, 5 \]
\[ 18 \]

\[ f \, 3 \, 5 \]
\[ 18 \]

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

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

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

\[ \text{val} = f_2(7) \]
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):
  \[ \text{name param1 param2 \ldots paramN} = \text{expression} \]

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

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

\[
\begin{align*}
\% \text{ cat plusper.hs} \\
\text{infixl 6 +\%} \\
x +\% \text{ percentage} &= x + x \times \text{ percentage} / 100
\end{align*}
\]

Usage:
\[
\begin{align*}
> 100 +\% 1 \\
101.0 \\
> 12 +\% 25 \\
15.0
\end{align*}
\]

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

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

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Left associative operators</th>
<th>Non associative operators</th>
<th>Right associative operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td><code>!!</code></td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>^, ^^, **</td>
<td></td>
<td>^, ^^, **</td>
</tr>
<tr>
<td>7</td>
<td><code>*</code>, <code>/</code>, <code>div</code>, <code>mod</code>, <code>rem</code>, <code>quot</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>+</code>, <code>-</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>:</code>, <code>++</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>==</code>, <code>/=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>elem</code>, <code>notElem</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>&amp;&amp;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td><code>&gt;&gt;, &gt;&gt;=</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><code>$</code>, <code>!</code>, <code>seq</code></td>
<td></td>
<td></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 \( \text{Char} \)
3. \( c \) is inferred to be a \( \text{Char} \).
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:

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

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

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

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 $x$ as both a **Num** and a **Char**.

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

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

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

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

```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 \texttt{add1} and \texttt{add2}? 

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

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

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

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

Two pitfalls related to type specifications for functions:

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

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

Recommendation:

Try writing functions without a type specification and see what type gets inferred. If the 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
```
Indentation, continued

Rule: A line that starts in the same column as did the previous
declaration ends that previous declaration and starts a new one.

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

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

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

This function definition uses guards to specify three cases:

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

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

```
> smaller 7 10
7

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

Solution:

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

> 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
Haskell's if-else, continued

What's the type of these expressions?

> \texttt{:type if 1 < 2 then 3 else 4}
\texttt{if 1 < 2 then 3 else 4 :: Num a => a}

> \texttt{:type if 1 < 2 then '3' else '4'}
\texttt{if 1 < 2 then '3' else '4' :: Char}

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

> \texttt{if 1 < 2 then 3}
\texttt{<interactive>:4:16:}
\texttt{parse error (possibly incorrect indentation or mismatched brackets)}
Guards vs. 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.
A Little Recursion
A recursive function is a function that calls itself either directly or indirectly.

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

```
> factorial 40
8159152832478977343456112695961158942720000000000
```

Write factorial in Haskell. What is its type?

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

```
> :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.

\[
\begin{align*}
\text{factorial } 4 \\
4 & \times \text{factorial } 3 \\
4 & \times 3 \times \text{factorial } 2 \\
4 & \times 3 \times 2 \times \text{factorial } 1 \\
4 & \times 3 \times 2 \times 1 \times \text{factorial } 0 \\
4 & \times 3 \times 2 \times 1 \times 1 \\
\end{align*}
\]

Recursion, continued

\[
\text{factorial } n \\
\begin{align*}
| & \text{n == 0} = 1 \\
| & \text{otherwise} = n \times \text{factorial } (n - 1) \\
\end{align*}
\]
Consider repeatedly dividing a number until the quotient is 1:

```
> 28 `quot` 3
9
> it `quot` 3
3
> it `quot` 3
1
```

Problem: Write a recursive function `numDivs divisor x` that computes the number of times `x` must be divided by `divisor` to reach a quotient of 1.

```
> numDivs 3 28
3
> numDivs 2 7
2
```
A solution:

\[
\text{numDivs divisor } x = \\
\begin{cases} 
(x \ `\text{quot}` \ divisor) < 1 = 0 \\
\text{otherwise} = \\
1 + \text{numDivs divisor} \ (x `\text{quot}` \ divisor)
\end{cases}
\]

What is its type?
\[
\text{numDivs :: (Integral a, Num a1) => a -> a -> a1}
\]

Will \text{numDivs 2 3.4} work?
\[
> \text{numDivs 2 3.4} \\
<\text{interactive}>:93:1: \\
\quad \text{No instance for (Integral a0) arising from a use of `numDivs'}
\]
Let's compute two partial applications of \texttt{numDivs}, using \texttt{let} to bind them to identifiers:

\begin{verbatim}
> f = numDivs 2
> g = numDivs 10
> f 9
3
> g 1001
3
\end{verbatim}

What are more descriptive names than \texttt{f} and \texttt{g}?

\begin{verbatim}
> floor_log2 = numDivs 2
> floor_log2 1000
9

> floor_log10 = numDivs 10
> floor_log10 1000
3
\end{verbatim}
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
> [1.3, 10, 4, 9.7] -- note mix of literals
[1.3,10.0,4.0,9.7]

it :: Fractional 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 **length** returns the number of elements in a list:

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

```haskell
> length [] 
0
```

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

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

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

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

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

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

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

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

```haskell
> 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.

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

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

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

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

What are the types of ++ and reverse?

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

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