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 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
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?
The level of a paradigm

Programming paradigms can apply at different levels:

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

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

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

The imperative paradigm applies to programming in the small.

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

The programming paradigms we know affect how we approach problems.

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

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

• If we know only imperative programming, code inside methods and functions will be imperative.
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?
We can use recursion to get rid of loops and assignments, but...ouch!

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

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

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

Here are some Java expressions:

- `x`
- `i + j * 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: 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'
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'

When we ask, "What's the type of this expression?"

we're actually asking this:
"What's the type of the value produced by this expression?"
Value, type, and side effect, continued

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

Which of these Java expressions have a side effect?

\[ x + 3 \times y \]

\[ x += 3 \times y \]

\[ s.length() > 2 \quad || \quad s.charAt(1) == '\#' \]
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 programing.

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  (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.
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,

```markdown
*** 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. 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.

3. On lectura, turn in `eca1.txt` with the following command:
   
   ```
   % turnin 372-eca1 eca1.txt
   ```

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

Haskell by Observation

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

```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.

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

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

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

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

\[ > \text{negate } 3+4 \]

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

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

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

> import Data.Char  
(Import the *Data.Char* module)

> isLower 'b'

> toUpper 'a'

> ord 'A'

> chr 66

> Data.Char.ord 'G'  
(Uses a qualified name)
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** shows 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:

\[
\begin{align*}
&> \text{toUpper } 'a' \\
&\quad 'A' \\
&> \text{ord } 'A' \\
&\quad 65 \\
&> \text{chr } 66 \\
&\quad 'B'
\end{align*}
\]

What are the types of those three functions?

\[
\begin{align*}
&> :t \text{toUpper} \\
&> :t \text{ord} \\
&> :t \text{chr}
\end{align*}
\]
Sidebar: Contrast with Java

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 chartoUpperCase(char);
public static inttoUpperCase(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*. 
State whether each expression is well-typed and if so, its type.

\[
\begin{align*}
'a' & \quad : \text{Char} \\
\text{isUpper} & \quad : \text{Char} -> \text{Bool} \\
\text{isUpper} 'a' & \\
\text{not} (\text{isUpper} 'a') & \\
\text{not not} (\text{isUpper} 'a') & \\
\text{toUpper} (\text{ord} 97) & \\
\text{isUpper} (\text{toUpper} (\text{chr} 'a')) & \\
\text{isUpper} (\text{intToDigit} 100) & \\
\text{chr} & \quad : \text{Int} -> \text{Char} \\
\text{digitToInt} & \quad : \text{Char} -> \text{Int} \\
\text{intToDigit} & \quad : \text{Int} -> \text{Char} \\
\text{isUpper} & \quad : \text{Char} -> \text{Bool} \\
\text{not} & \quad : \text{Bool} -> \text{Bool} \\
\text{ord} & \quad : \text{Char} -> \text{Int} \\
\text{toUpper} & \quad : \text{Char} -> \text{Char}
\end{align*}
\]
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:

http://trac.haskell.org/haskeline/wiki/KeyBindings
As we've seen, ghci provides a REPL (read-eval-print loop) for Haskell.

What are some other languages that have a REPL available?

How does a REPL help us learn a language?

Is there a REPL for Java?

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?