# CS457/557
# Functional Languages
## Spring 2018
## Lecture 1: Course Introduction

Andrew Tolmach
Portland State University

(with thanks to Mark P. Jones)

# Goals of this course

- Introduce the beautiful ideas of functional programming
- Explain new strategies for building and verifying programs
- Demonstrate that functional programming has real-world utility

# Important Underlying Themes

- Computing by calculating
- Recursive algorithms and types
- Type-driven programming
- Abstraction over values, functions, types
- Programming by composition
- Reasoning about programs

# Specific Topics (subject to revision)

- Haskell programming language
- Programming with lists
- Programming with algebraic data types
- Polymorphism and type classes
- Higher-order functions

# More specific topics (subject to revision)

- Functions as data
- Monads
- Laziness
- Parallelism
- Implementation

# What this course is not

- An advanced course in the details of Haskell (and its many non-standard extensions)
- A detailed tour of the Haskell library
- A comparative study of functional languages
- A good course to take if you don’t really like programming much

# What should you bring?

- Your brain, prepared by these prerequisites
  - CS 202,311 are formally required (for 457)
  - CS320 is useful, but not essential
    - Good background in programming (but not FP)
- Your well-charged laptop!
  - This is a hands-on course, and we will be doing lab work towards the end of each class meeting

# Administrivia

- Instructor: Andrew Tolmach
  - Office hours: Tu 1-2pm or by appointment
- TA: Chris Chak
  - Office hours: M 4-5 (tentative)
- Course web page www.cs.pdx.edu/~apt/cs457
  - For all homework assignments, lectures notes, etc.
- Course mailing list cs457list@cs.pdx.edu
  - For helpful announcements and for you to ask questions
- Course homework submission address cs457acc@pdx.edu
  - For homework submission only!  Don't get the two mailing lists confused!

# Course format

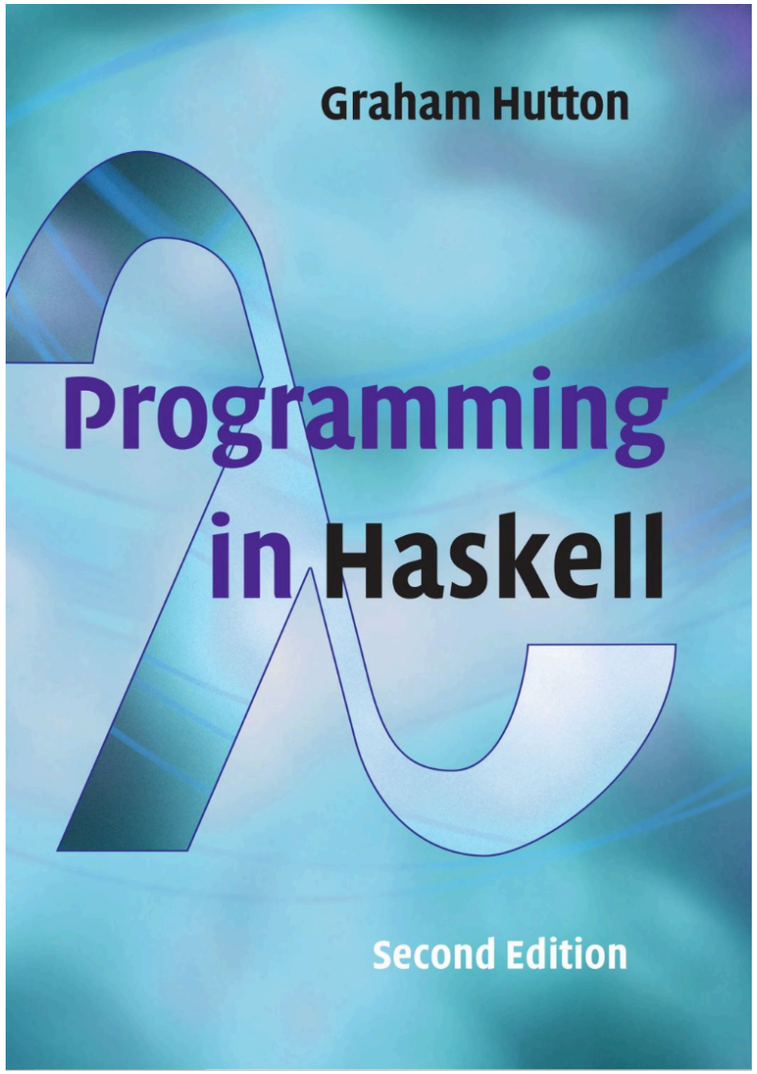
- Meet Mon/Wed 2-3:50pm
  - Start each class with lecture
  - Finish with problem-solving lab (bring your laptop!)
- Weekly homeworks due Wednesdays
  - Essential part of course — 55% of grade
- In-class midterm on May 7 — 20% of grade
- Final programming project due June 6 — 25% of grade
  - Can be done individually or in a team of 2
- NO final exam

# Policies

- By default, late work is not accepted
  - Contact us if you feel an extension is justified
- Work individually on the homeworks
  - Discussion is good
  - But anything you turn should be your own, individual work
- Don’t cheat!

# Resources

- Syllabus
- Textbook
- Lecture slides
- Huge amount of on-line material, starting at www.haskell.org/documentation
  - But beware of unnecessary complexity!



# What is Functional Programming?

- A style of programming that emphasizes evaluation of expressions, rather than execution of commands

- Expressions are formed by using **functions** to combine basic values

- Functions are **first-class** values
  - They can be stored in data structures
  - They can be passed as arguments or returned as results of other functions

- A **functional language** is one that supports and encourages programming in a functional style

# Pure Functional Programming

- No mutation!  Everything (variables, data structures, …) is **immutable**

- Expressions have **no side-effects**, like updates to global variables or output to the screen

- Function results depend only on input values
  - **Deterministic**, like functions in mathematics
- Makes programs much more **compositional**
  - **Refactoring** and **parallelizing** are much easier

# The functional language landscape

- Impure, strict evaluation, dynamic typing:
  - Lisp, Scheme, Racket, Erlang, Clojure, …
- Impure, strict evaluation, static typing:
  - Standard ML (SML), OCaml, F#, Scala, …
- Pure, lazy evaluation, static typing:
  - Haskell, Miranda, Orwell, …
- Other combinations relatively unexplored…

# Haskell

- By far the most important pure, lazy FL
- Developed by committee of academics in late 80's
  - Combined and standardized several earlier languages
- Current stable version is "Haskell 2010"
- Dominant implementation is "Glasgow Haskell" (ghc)
  - Includes many experimental extensions (which we will mostly avoid)

Write a program to add up the numbers from 1 to 10.

# In C, C++, Java, C#, …

```
int tot = 0;
for (int i=1; i<11; i++)
   tot = tot + i;
```

initialization

initialization

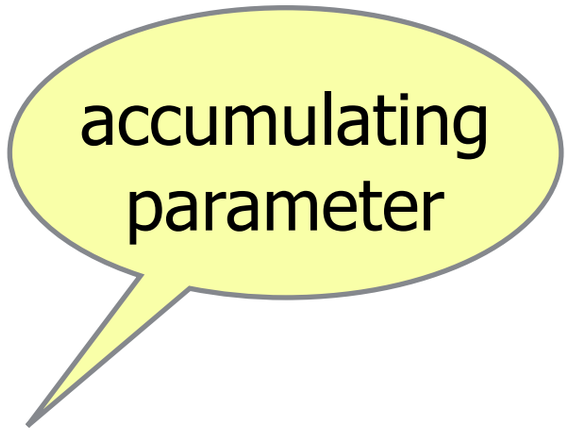
iteration

update

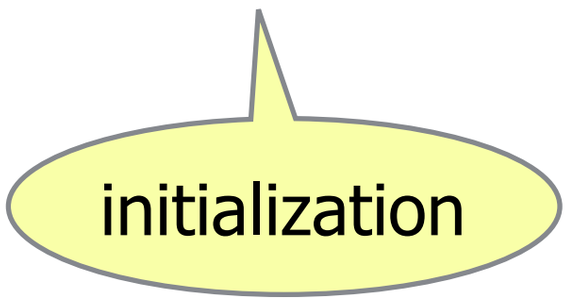
update

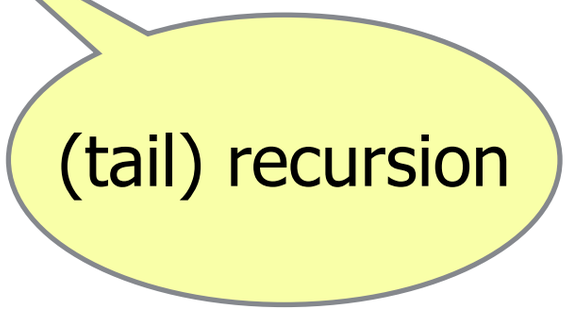
implicit result returns in the variable `tot`

# In OCaml




```
let rec sum i tot =
      if i > 10
      then tot
      else sum (i+1) (tot+i)
in sum 1 0
```




result is the value of the expression

# In Haskell

```
sum [1..10]
```

combining function

the list of numbers to add

result is the value of the expression

# Was that too simple?

- Tried to give "idiomatic" solutions in each language
- This example makes Haskell look good, partly because `sum` function is already in standard library
  - An objective comparison between languages should account for library code as well as main program
- Here's an alternative solution using somewhat less specialized library functions

```
foldr (+) 0 [1..10]
```

# We can write OCaml in Haskell

```
let sum i tot | i > 10 = tot
              | otherwise =
                  sum (i+1) (tot+1)
in sum 1 0
```

and sometimes we will **need** to write explicit recursions like this

but we will try to avoid them when we can

# We can write C in Haskell!

```
import Data.IORef
main =
 do tot <- newIORef 0
    i <- newIORef 1
    let loop =
          do ival <- readIORef i
             if ival <= 11 then
               do modifyIORef tot (+ival)
                  modifyIORef i (+1)
                  loop
             else
               return ()
    loop
    totval <- readIORef tot
    print totval
```

we almost never need to do this!

result is printed by main program

# What makes a good program?

- Correctness
- Maintainability (Clarity, Conciseness, Modularity, …)
- Performance

# Raising the level of abstraction

- "If you want to reduce [design time], you have to stop thinking about something you used to have to think about"
  (Joe Stoy, quoted on the Haskell mailing list)

- Example: memory allocation and deallocation
- Example: data representation
- Example: order of evaluation
- Example: (restrictive) type specifications

# Computing by Calculating

- In high school algebra, we learn to rearrange and simplify numeric expressions to obtain answers
  - Pocket calculators automate details of calculation
- In pure functional programming, we can work with program expressions in much the same way
  - With multiple primitive data types, lists, functions, user-defined types
  - Ability to name (abstract over) values and operations
  - Functional language evaluators automate calculation

# Example calculation

- In pure functional language, we can perform computations by replacing defined symbols with their definitions

Given

```
a = 10
b = 7
diff x y = if x <= y then y-x else x-y
```

Can calculate

```
diff a b ⟹
if a <= b then b-a else a-b ⟹
if 10 <= 7 then 7-10 else 10-7 ⟹
if False then 7-10 else 10-7 ⟹ 10-7 ⟹ 3
```

# Haskell Pragmatics

- Glasgow Haskell ecosystem
  - ghc — native code compiler
  - ghci — interpreter
  - hackage — package database
  - cabal, stack — package managers
  - Haskell Platform — convenient single download
- Other implementations exists (Hugs, …)

# Starting ghci

```
user$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

The most important commands:

| | |
|---|---|
| :q | quit |
| :l file | load file |
| :e file | edit file |
| expr | evaluate expression |

The REPL (read-eval-print loop):

1. Enter expression at prompt
2. Hit return
3. Expression is read, checked, and evaluated
4. Result (or error) is displayed
5. Repeat from step 1

# Simple expressions

- The usual arithmetic operations

```
1 + 2 * 3          (1 + 2) * 3
```

- Comparisons

```
1 == 2             'a' < 'z'
```

- Boolean operators

```
True && False      not False
```

- Standard library functions on numbers

```
odd 2   odd (2+1)    sqrt 4.0 + 2.0     sqrt (4.0 + 2.0)
```

- Lists and library functions on them

```
[1,2,3]     length [True,True,False]     sum [1..10]
```

# Expressions have Types

- The type of an expression tells you what kind of value the expression evaluates to

- In Haskell, read "::" as "has type"

- Examples:

  - **`1 :: Int`**  **`'a'::Char`**  **`True :: Bool`**
    **`1.2 :: Float`**

- You can ask ghci to tell you the type of an expression by entering **`:t expr`**

# Type Errors in ghci

```
Prelude> 'a' && True

<interactive>:7:1: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘Char’
    • In the first argument of ‘(&&)’, namely ‘'a'’
      In the expression: 'a' && True
      In an equation for ‘it’: it = 'a' && True
```

```
Prelude> odd 1 + 2

<interactive>:8:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: odd 1 + 2
      In an equation for ‘it’: it = odd 1 + 2
```

# Definitions and Scripts

- So far, have just been evaluating expressions
- What if we want to
  - Define a new constant (i.e. give a name to the result of an expression)?
  - Define a new function?
  - Define a type?
- We place definitions in a script file with a **`.hs`** suffix that can be loaded into ghci

# Simple Script

- Place the following test in a file “defs.hs”

```
square x = x * x
fact n   = product [1..n]
diff x y = if x <= y then y-x else x-y
a        = 10
```

# Simple Script

- Pass the filename as a command line argument to ghci, or use the :l command from inside ghci

```
user$ ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Prelude> :l defs.hs
[1 of 1] Compiling Main                ( defs.hs, interpreted )
Ok, one module loaded.
*Main> square 12
144
*Main> fact 32
263130836933693530167218012160000000
*Main> diff 1 a
9
*Main> diff a 1
9
*Main>
```

# Let’s get things running

- Get to a position where you can run ghci, by either
  - installing it on your machine; or
  - starting a remote shell on linuxlab.cs.pdx.edu
- Download this file from the course web page:
  - http://www.cs.pdx.edu/~apt/cs457/hw0.hs
- Start ghci, load hw0.hs, and evaluate the following expression:

  **`idme`** “*`your-name-here`*” “*`envvar`*” where the first string is your name and the second string is the name of the shell environment variable containing your username, i.e. **“`USER`”** on *nix and **“`username`”** on Windows

- This should produce an output file in your current directory called **`my_identity.txt`**
- Send mail to the homework account cs457acc@pdx.edu with **`my_identity.txt`** as an attached file