WHY STUDY PROGRAMMING LANGUAGES?
PROGRAMMING LANGUAGES ARE EVERYWHERE!
STANDARD APPLICATIONS

- Systems and low-level tasks
  - C, C++, Assembly, Rust, Go, ...
- Higher-level/general-purpose
  - Java, C#, OCaml, Haskell, Lisp, Python, Ruby, ...
- Web development and mobile apps
  - Javascript, Swift, Dart, Objective-C, ...
- Scripting
  - Bash, Perl, Awk, Sed, ...
NOT-SO-STANDARD

- Database queries
- Networking and distributed systems
- Typesetting
- Configuration and build systems
- Theorem proving
- Graphics and GPUs, hardware and FPGAs
- Numerical and scientific computing
- Parsing and lexing
- Blockchain and smart contracts
HOW WE TELL COMPUTERS WHAT TO DO

- From human thoughts to precise instructions
  - Enable computers to help us program
  - Spot mistakes, perform optimizations, etc.
PL SHAPES HOW WE THINK

- Programmers think in terms of language abstractions
  - Classes, objects, functions, types, ...
  - Fits complex systems into human brains
WHAT ARE MLS FOR?
WRITING PROGRAMS

- Small one-off scripts
  - Automate some boring task
- Useful applications
  - Notetaking app, web server, ...
- Serious corporate products ($$$)
  - Google, Facebook, Amazon, Apple, ...
- Critical infrastructure
  - Hospitals, power plants, electricity grids, ...
REUSING EXISTING CODE

- Share code between members of a team
- Use built-in standard libraries
- Open-source community, Github
PREVENTING ERRORS

• At compile-time
  ▪ Rule out nonsensical programs
  ▪ Catch common mistakes automatically
  ▪ Check for security vulnerabilities

• Through better design
  ▪ Make certain kinds of errors impossible
  ▪ Ensure programmer handles all cases
“BILLION-DOLLAR MISTAKE”

I call it my billion-dollar mistake [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage.

• Tony Hoare, on inventing null pointers/references
ORGANIZING SOFTWARE

- Software: most complex human-designed thing, ever
- Not limited by laws of physics
  - If you build a 1000 story skyscraper, it will collapse
- Limited by complexity
  - If you produce enough code, you will run out of programmers to fix bugs
- PLs: first line of defense to manage complexity
THERE’S A LOT OF CODE

How much?
A THEORY OF PROGRAMMING LANGUAGES?
A BUNCH OF LANGUAGES?

- Many languages sort of “look the same”
- Every real language has a ton of quirks
  - Historical accidents
  - Specific constraints
- Essential features of PLs often hard to see
“PROGRAMMING PARADIGMS”?

- Popular way of categorizing PLs
  - Objected-oriented (OO)
  - Functional (FP)
  - Imperative
  - Declarative
- Hard to pin down what these paradigms mean
  - Most languages have features from all paradigms
  - A programming style, or a kind of language?
YES: COMMON PL FEATURES

- Many PLs arrived at the same few concepts
  - Examples: variables, functions, loops
- Analyze the essence of each feature
- Understand how different features interact
YES: FORMALIZE LANGUAGES

• Study toy models of programming languages
  ▪ Extremely simplified (not practical)
  ▪ Focus on just a few, essential features
• Formally defined using *mathematics*
  ▪ Clearest way to think about languages
  ▪ Possible to prove things about languages
  ▪ Provides a rigorous foundation to PL
WHAT MAKES A LANGUAGE POPULAR?
“EASE OF USE/ERGONOMICS”

- Depends on things like...
  - What PLs a programmer is familiar with
  - A programmer’s mental model of programs
  - How “readable” programs are
  - Specific details (braces/parentheses, ...)
- Hard to analyze scientifically
SUPPORTING TOOLS

- Development tools
  - IDE, debugger, linter, code formatter, GUI designer
- Standard libraries and documentation
  - Math, data structures, networking, DB, graphics, ...
- "Toolchain": compiler, package manager, runtime
- Requires a lot of development effort ($$$)
SOCIAL FACTORS

- Specific niche
  - iOS apps, scientific computing
- Community
  - Reddit, Stack Overflow, packages on Github
- Industrial influence
  - “Language for NVIDIA GPUs”
- Reputation and stereotypes
  - “Real hackers use C”
- Advertising and marketing
  - Tech talks, conferences, charismatic leaders
WHAT MAKES A LANGUAGE “GOOD”?
SPECIFY WELL-FORMED PROGRAMS

- Language should describe:
  - Which programs are well-formed
  - Which programs are not well-formed

Define what programs look like!
DESCRIBE BEHAVIOR OF PROGRAMS

- Language should describe:
  - How well-formed programs should behave
  - What are acceptable outputs, and what are not
  - Which programs are equivalent, and which are not

Define what programs should do!
MAKE IT EASY TO COMBINE PROGRAMS

- Should be possible to:
  - Understand program by looking at individual parts
  - Put programs together without causing bugs
- Crucial for managing complexity
- Makes language feel elegant and well-designed
MAKE IT HARD TO WRITE BAD PROGRAMS

- Make some errors impossible
  - Null pointer, buffer overflows, forgotten cases, ...
- Catch errors early, at compile time
  - Better not to crash during rocket launch
- Warn when programmer does something dangerous
COURSE PLAN AND
OVERALL GOALS
HANDS-ON EXPERIENCE

- Use cutting-edge programming languages
  - First half: Haskell
    - Functional programming
    - Advanced type system
    - Tight control of effects
  - Second half: Rust
    - Imperative programming
    - Neat memory-management mechanisms
    - Fearless concurrency
EXPLORE PL FEATURES

- Type systems of all kinds
- Typeclasses/traits
- Effect systems
- Mutable and immutable references
- Lifetimes and memory ownership
- ...


FORMALIZE LANGUAGES

- Sprinkled throughout: core lectures
  - Work with toy languages
  - Define program syntax and grammars
  - Set up operational semantics
  - Design type systems
- This part: on paper (no programming)
COURSE FORMAT
WE WILL CARE MORE ABOUT:

- Learning core Haskell and Rust
- Specifying languages precisely
- Specifying type systems precisely
WE WILL CARE LESS ABOUT:

- Implementations: compilers, JITs, runtimes, ...
  - Would require a whole course to cover properly
- Performance (time and space)
  - Lots of tricks and techniques
- Formally proving stuff about programs
  - Not super difficult, but we don’t have time
- Experimental language features
  - Very interesting, but we will steer clear
DETAILS

• Assignments
  ▪ Both programming and written components
  ▪ Roughly 2-3 weeks per assignment
• Midterm exam: in class
• Final exam: take home
• Communications: ask/answer questions on Piazza

https://pages.cs.wisc.edu/~justhsu/teaching/current/cs53
READINGS

- On calendar: references for each lecture
  - **RWH**: Real World Haskell
  - **LYAH**: Learn You a Haskell for Great Good
  - **PFPL**: Practical Foundations for PL
- Very helpful and recommended, but **not required**
EXPERIMENT IN PROGRESS

- This course was first taught last year
- *Everything* is pretty new: format, lectures, HWs
- Haskell and Rust both move fast; we will too
BOTTOM LINE

If something is not working, please let us know ASAP and we will try to fix it.
HOMEWORKS
INSTALLATION

- Instructional machines all have Haskell software
- GHC and HLint should just work
- Don’t waste time fighting installation errors; ask us
WRITTEN EXERCISES

- Type up solutions or scan
- Some parts we won’t cover until next week
PROGRAMMING EXERCISES

• Check out resources page on course site
• You will have to search in the docs
  ▪ Hayoo/Hoogle: searching by type
• GHCi will help you see what your code is doing
COMPILER ERRORS

- Strong type system: Compiler will complain, a lot
- Languages have type inference
  - Pro: usually don’t have to write types
  - Con: harder time reporting error location
SOME ADVICE

- Step 1: Don’t panic!
- Step 2: Take errors one at a time, in order
  - No matter how tempting: never jump ahead!
  - Fixing one error often fixes many others
- Step 3: Try to add type annotations
  - Help compiler narrow down what type you mean
LATE DAY POLICY

- You will each have *6 late days* total
- Can spend *at most 2 late days* per HW
- One day = 24 hours from the due time
- Bonus credit for unused late days
HW1 OUT LATER TONIGHT

- Due in two weeks
- Programming exercises and written exercises
- See full instructions on class site

Start as early as you can!
FUNCTIONAL PROGRAMMING
A BRIEF HISTORY

- Based on lambda calculus by Alonzo Church (1930s)
- First real language: Lisp by John McCarthy (1950s)
- Popularized by many, especially John Backus
- ML developed by Robin Milner at Edinburgh (1973)
- Miranda and Haskell in late 1980s
BUILDING BLOCK: FUNCTIONS

- A function has two components:
  - *Input*: arguments passed to function
  - *Output*: result of running function
- Functions are *first class*: treated like any other value
  - Can be passed into other functions
  - Can be returned from other functions
- Combine functions to build new functions
CONTROL “SIDE-EFFECTS”

- Pure functions fully described by input/output
  - Always return same result on fixed input
- Avoid hidden state
  - Counters, local variables, etc.
- Carefully manage side-effects
  - Printing, reading a file, etc.

Think about programs in isolation
A TASTE OF HASKELL
DECLARING FUNCTIONS

double :: Int -> Int
double n = n + n

- First line: optional type signature/type annotation
  - This one says: function from Int to Int
- Second line: function definition/function body
CALLING FUNCTIONS

• Format: put function name, space(s), argument

myBool  = myFun 42  -- Call with 42
-- NOT: myBool = myFun(42)

myBool' = constFun () -- Call with unit
MULTIPLE ARGUMENTS?

- Type signature: `doublePlus` takes two inputs
- Function calls: `double x` and `double y`
CASE ANALYSIS

Standard if-then-else:

```haskell
doubleIfBig :: Int -> Int
doubleIfBig n = if (n > 100) then n + n else n
```

Cleaner (or for more cases):

```haskell
doubleIfBig' :: Int -> Int
doubleIfBig' n
  | n > 100   = n + n
  | otherwise = n
```
ANOTHER WAY TO MATCH

Use a case expression:

```haskell
listPrinter''' :: [Int] -> String
listPrinter''' l = case l of
  []       -> "Empty list :(
  (x:xs)   -> (show x) ++ " and " ++ (show xs)
```

DECLARING VARIABLES

At the beginning...

```
tripleSecret :: Int
tripleSecret = let secret = mySecretNum
                  other  = myOtherNum
              in 3 * secret + other
```

...or at the end

```
tripleSecret' :: Int
tripleSecret' = 3 * secret + other
                      where secret = mySecretNum
                           other  = myOtherNum
```
TUPLES AND LISTS
BUILDING TUPLES

• Tuples are pairs/triples/...

myTuple2 :: (Int, Int)
myTuple2 = (7, 42)

myTriple :: (Int, Int, Int)
myTriple = (7, 42, 108)
MORE TUPLES

- Tuples can mix and match different types

```haskell
myMixedTuple :: (Int, Int, Bool)
myMixedTuple = (7, 42, false)
```

- Empty tuple is *unit* type, only one possible value

```haskell
emptyTuple :: ()
emptyTuple = ()
```
WORKING WITH TUPLES

• Get first or second elements:

\[
fstInt :: (\text{Int, Int}) \to \text{Int}
fstInt (x, y) = x
\]

\[
sndInt :: (\text{Int, Int}) \to \text{Int}
sndInt (x, y) = y
\]

-- In standard library:
\[
fst :: (a, b) \to a
fst (x, y) = x
\]

\[
snd :: (a, b) \to b
snd (x, y) = y
\]
WORKING WITH TUPLES

• Swap elements of tuple

```
swapInt :: (Int, Int) -> (Int, Int)
swapInt (x, y) = (y, x)
```

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```
LIST OF THINGS OF SAME TYPE

This is a list of four integers:

```haskell
myList :: [Int]
myList = [1, 2, 3, 4]
```

Lots of operations on lists:

```haskell
myList'  = 0 : myList  -- [0, 1, 2, 3, 4]
myFirstElem = head myList  -- 1
myLength   = length myList  -- 4
myBigList  = myList ++ myList  -- [1, 2, 3, 4, 1, 2, 3, 4]
doubleSmalls = [ 2 * x | x <- myList, x < 3 ]  -- [2, 4]
```
PATTERN MATCHING

Define functions on list by case analysis:

```haskell
listPrinter :: [Int] -> String
listPrinter [] = "Empty list :(")
listPrinter (x:xs) = "List: " ++ (show x) ++ " and " ++ (show xs)
```

Underscore _ matches any value:

```haskell
listPrinter' :: [Int] -> String
listPrinter' [] = "Empty list :(" 
listPrinter' _ = "List with something :)")"
```
MORE ABOUT FUNCTIONS
INPUTS TO OUTPUTS

- Same input always leads to same output
  - No hidden dependence/effects
  - Think: “functions in math class”
- No side effects! This always returns same value:

constFun :: () -> Bool
-- Either always returns True, or always returns False
INFIX FUNCTIONS

• Often convenient to write binary functions *infix*

```haskell
myAppend :: [Int] -> [Int] -> [Int]
myAppend list list' = list ++ list'
```

• Can turn any binary function into infix operator:

```haskell
myLists = [1, 2] `myAppend` [3, 4] -- = myAppend [1, 2] [3, 4]

-- Symbol function names can be used infix by default
(@@) :: [Int] -> [Int] -> [Int] -> [Int]
list @@ list' = myAppend list list'

myLists' = [1, 2] @@ [3, 4] -- = (@@) [1, 2] [3, 4]
```
ANONYMOUS FUNCTIONS

• Can define function without giving a name
• Useful for small, one-off functions

```
plusFour = doTwice (∧x → x + 2)
-- \x looks like λx
```

• Can take multiple arguments

```
plus = \x y → x + y
-- SAME AS: plus      = \x → \y → x + y
-- SAME AS: plus x   = \y → x + y
-- SAME AS: plus x y = x + y
-- SAME AS: plus     = (+)
```
What does ext refer to below?

ext :: Int
ext = 42

myFun = \arg -> arg + ext

• Anonymous function can use outside variables
• If myFun called elsewhere, remembers value of ext
• This kind of function is also called a closure
MORE ABOUT VARIABLES
What is the result of the following program?

```
let foo = 1 in
let foo = 2 in
foo
```

Answer: 2. Looks like `foo` was updated...
VARIABLES ARE NEVER "UPDATED"

What about the following programs?

```plaintext
let foo = 1 in
  (let foo = 2 in foo) + foo

let foo = 1 in
  foo + (let foo = 2 in foo)
```

Answer: 3. Inner `foo` has nothing to do with outer `foo`!