Programming in Haskell
CSCI-4966 and CSCI-6966
Fall 2024

www.cs.rpi.edu/~milanova/csci4966/

Ana Milanova
Office: Lally 314
Email: milanova@cs.rpi.edu
Office hours: Wednesdays 1pm-2pm, Fridays after class or by appointment

1

Introductions

- Ana Milanova
- You
  - Tell us
    - Your name
    - Graduate or undergraduate student
    - Concentration, interests, research area
    - Haskell experience?

2

Outline

- Logistics
  www.cs.rpi.edu/~milanova/csci4966/
- Why Haskell?
- Course topics and homework
- Intro to Haskell

3

Logistics

- Course webpage
  http://www.cs.rpi.edu/~milanova/csci4966
  - Schedule, Reading, Notes
    - Schedule, lecture slides and assigned reading
  - Submitty
    - All homework submission and grades, **forum**
    - Check forum regularly for announcements

4

Logistics

- Resources
  - **Programming in Haskell,** Second Edition by Graham Hutton
  - https://www.haskell.org/

Haskell
An advanced, purely functional programming language

5

Logistics

- Syllabus
  www.cs.rpi.edu/~milanova/csci4966/syllabus.htm
  Outcomes, policies and grading
- In-class quizzes (6): 20%
- Programming homework (8-10): 47%
- Project: 25%
- Attendance and participation: 8%

6

Logistics

- Homework is to be completed individually unless otherwise specified
- Project can be completed individually or in teams of two (recommended)
- Quizzes are in-class, open-notes, and may be completed individually or in small groups; we will drop the lowest quiz

7

Logistics

- Graduate students enrolled in CSCI-6966
- Grade breakdown:
  - In-class quizzes (6-8): 17%
  - Homework assignments (8-10): 45%
  - Project: 30%
  - Attendance and participation: 8%
- Some assignments may have additional problems. Graduate students are expected to complete a more challenging project

8

Late Homework

- Homework must be submitted in Submitty by 12pm on the due date
- You have 10 late days for the semester, with a max of 5 late days per assignment
- Exceptions to policy will be granted only in rare cases

9

Academic Integrity

- Trust
- Discussion **is allowed**, even encouraged
- Taking written notes out of discussion **is NOT allowed**. Actual work should be your own
- Posting solutions on public forums (e.g., Discord, Github) is not allowed

10

Why Haskell?

- Acquiring the skill to program in Haskell will make you a better programmer in **any** language
- Haskell has influenced plenty of features now in "mainstream" programming languages
- Haskell is firmly grounded in theory
  - Lambda calculus, type theory, category theory

11

List Comprehensions

- Python:

```
[x for x in range(10)]
[x for x in range(10) if x%2==0]
```

  - An AST construct
- Haskell:

```
[x | x<-[0..9]]
[x | x<-[0..9], x`mod`2==0]
```

  - Syntactic sugar over a monadic computation

12

Pattern Matching

- Python, since Python 3.10:

```
match shape:
  case Triangle(point1=p1,point2=p2,point3=p3):
    print ((p1,p2,p3))
  case Circle(center=(0,0)):
    …
```

- Haskell:

```
case shape of
   Triangle p1 p2 p3 -> (p1,p2,p3)
   Circle (0,0) -> …
```


13

Generators

- Python:

```
def inf():
  a = 1
  while True:
     yield a
     a = a+1
g = inf()
print(next(g),next(g))

import itertools
s = list(itertools.islice(g,10))
```


14

Generators

- Haskell:

```
[1..]
```

-- generates the infinite list of integers 1,2,3…

```
take 10 [1..]
```

-- yields [1,2,3,4,5,6,7,8,9,10]

15

Optional Type, and More!

- Python Optional type hint:

```
def fun(input: int) -> Optional[int]:
  return (1 if f(input) else None)
```

- Haskell's Maybe type:

```
fun::Int -> Maybe Int
fun input =
  if (f input) then Nothing else Just 1
```


16

Outline

- Logistics
  www.cs.rpi.edu/~milanova/csci4966/
- Why Haskell?
- Course topics and homework
- Intro to Haskell

17

Course Topics

- Basics
  - Types, base types, defining functions, recursion
  - List comprehensions, ADTs, pattern matching
  - Higher-order functions and recursion patterns
  - Interactive programming and basic IO

18

Course Topics

- More advanced
  - The lambda calculus and lazy evaluation
  - Parametric polymorphism and type inference
  - Type classes

19

Course Topics

- Advanced
  - Monads
    - Maybe, List, IO, State and Continuation monads
    - Composition
    - Parsing with monads
  - Functors and applicative functors
  - Effectful programming
  - Monoids and foldables
  - Other, TBD

20

Homework Assignments

- There will be 8-10 homework assignments
  - Each makes about 4-5% of your grade
  - A set of programming problems covering material
  - Part of Friday's lecture will be a lab and at least part of the homework should be completed by end of lecture

21

Project

- A larger programming assignment
  - Project proposal
  - Checkpoints
  - Coding (of course!)
  - Present
- Could be anything: code "from scratch" using Prelude or use Hackage
- Complete in groups of 2-3 (recommended) or individually

22

Intro to Haskell (slides due to Graham Hutton, with modifications)

23

What is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments
- A functional language is one that supports and encourages the functional style

24

Example

Summing the integers 1 to 10 in Java:

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
   total = total + i;
```

The computation method is variable assignment

25

Example

Summing the integers 1 to 10 in Haskell:

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

The computation method is function application

26

Another Example

Inner product in a "Von Neuman style" language:

```
c := 0
for i := 1 step 1 until n do
   c := c + a[i]*b[i]
```

Variable assignment and state transition

27

Another Example

Inner product in the functional language FP:

```
Def InnerProduct =
    (Insert +) ∘ (ApplyToAll *) ∘ Transpose
```

Function application, reduction, higher-order programming

28

In Haskell

```
ip = sum . (map product) . transpose
```


29

Historical Background

1930s:

Alonzo Church develops the lambda calculus, a simple but powerful theory of functions

30

Historical Background

1950s:

John McCarthy develops Lisp, the first functional language, with some influences from the lambda calculus, but retaining variable assignments

31

Historical Background

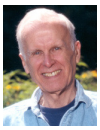
1960s:

Peter Landin develops ISWIM, the first *pure* functional language, based strongly on the lambda calculus, with no assignments

32

Historical Background

1970s:



John Backus develops FP, a functional language that emphasizes *higher-order functions* and *reasoning about programs*

33

Historical Background

1970s:

Robin Milner and others develop ML, the first modern functional language, which introduced *type inference* and *polymorphic types*

34

Historical Background

1970s - 1980s:

David Turner develops a number of *lazy* functional languages, culminating in the Miranda system

35

Historical Background

1987:

Haskell

An advanced, purely functional programming language

An international committee starts the development of Haskell, a standard lazy functional language

36

Historical Background

1990s:

Phil Wadler and others develop *type classes* and *monads*, two of the main innovations of Haskell

37

Historical Background

2003:

The committee publishes the Haskell Report, defining a stable version of the language; an updated version was published in 2010

38

Historical Background

2010-date:

Haskell Platform

Haskell with batteries included

Standard distribution, library support, new language features, development tools, use in industry, influence on other languages, etc

39

A Taste of Haskell

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a ← xs, a ≤ x]
              zs = [b | b ← xs, b > x]
```

?

40

Defining Characteristics of Haskell

- Functional: functions are first-class values, computation method is function application
- Pure: there is no mutation, all function calls are "referential transparency"
- Lazy: expressions are NOT evaluated until they are needed in computation
- Statically typed: every expression has a type determined and checked at compile time!

41

Characteristics of Haskell

- Syntactic sugar
- Abstraction
- Compact code
- Recursion
- Other…

42

Outline

- Logistics
  www.cs.rpi.edu/~milanova/csci4966/
- Why Haskell?
- Course topics and homework
- Intro to Haskell. Now let's get started!

43

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell, and comprises a compiler and interpreter
- The interactive nature of the interpreter makes it well suited for teaching and prototyping
- GHC is freely available from:

  `www.haskell.org/downloads`
- You may find the Get Started page useful as well: `https://www.haskell.org/get-started/`

44

Haskell Programming

- We'll start with the interpreter
  - run GHCi and
  - an Editor (VSCode, Emacs, vim)
- Later (for project) we'll program "in the large":
  - Hackage
  - cabal
  - Modules

45

Starting GHCi

The interpreter can be started from the terminal command prompt by simply typing ghci:

```
$ ghci

GHCi, version X: http://www.haskell.org/ghc/  :? for help

Prelude>
```

The GHCi prompt > means that the interpreter is now ready to evaluate an expression.

46

For example, it can be used as a desktop calculator to evaluate simple numeric expressions:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```


47

The Standard Prelude

Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists

- Select the first element of a list:

```
> head [1,2,3,4,5]
1
```


48

```
> tail [1,2,3,4,5]
?
```

```
> [1,2,3,4,5] !! 2
?
```

```
> take 3 [1,2,3,4,5]
?
```


49

```
> drop 3 [1,2,3,4,5]
?
```

```
> length [1,2,3,4,5]
?
```

```
> sum [1,2,3,4,5]
?
```


50

```
> product [1,2,3,4,5]
?
```

```
> [1,2,3] ++ [4,5]
?
```

```
> reverse [1,2,3,4,5]
?
```


51

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space

```
f(a,b) + c d
```

Apply the function f to a and b, and add the result to the product of c and d.

52

In Haskell, function application is denoted using space, and multiplication is denoted using *.

```
f a b + c*d
```

As previously, but in Haskell syntax.

53

Moreover, function application is assumed to have higher priority than all other operators.

```
f a + b
```

Means (f a) + b, rather than f (a + b).

54

Examples

Mathematics	Haskell
`f(x)`	`f x`
`f(x,y)`	`f x y`
`f(g(x))`	`f (g x)`
`f(x,g(y))`	`f x (g y)`
`f(x)g(y)`	`f x * g y`

55

Haskell Scripts

- As well as the functions in the standard library, you can also define your own functions
- New functions are defined within a script, a text file comprising a sequence of definitions
- By convention, Haskell scripts usually have a .hs suffix on their filename. This is not mandatory, but is useful for identification purposes

56

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x = x + x

quadruple x = double (double x)
```


57

Leaving the editor open, in another window start up GHCi with the new script:

```
$ ghci test.hs
```

Now both the standard library and the file test.hs are loaded, and functions from both can be used:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```


58

Leaving GHCi open, return to the editor, add the following two definitions, and resave:

```
factorial n = product [1..n]

average ns = sum ns `div` length ns
```

Note:

- div is enclosed in back quotes, not forward;
- x `f` y is just syntactic sugar for f x y.

59

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```


60

Demo

61

Useful GHCi Commands

Command	Meaning
:load *name*	load script *name*
:reload	reload current script
:type *expr*	show type of *expr*
:?	show all commands
:quit	quit GHCi

62

Naming Requirements

- Function and argument names must begin with a **lower-case** letter. For example:

```
myFun    fun1    arg_2    x'
```

- Idiomatic Haskell uses **camelCase** for variable names:

```
myFun    maxBound    numDigits
```

- By convention, list arguments usually have an *s* suffix on their name. For example:

```
xs    ns    nss
```


63

The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10
b = 20
c = 30
```

✓

```
a = 10
 b = 20
c = 30
```

✗

```
 a = 10
b = 20
 c = 30
```

✗

64

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

implicit grouping

means

```
a = b + c
    where
      {b = 1;
       c = 2}
d = a * 2
```

explicit grouping

65

To Get Started

- Download and install GHC from haskel.org. Run GHCi. Load and reload a .hs program.
- Fix the syntax errors in the program below and test your solution using GHCi.

```
N = a 'div' length xs
    where
      a = 10
      xs = [1,2,3,4,5]
```


66

To Get Started

Solve the problems below using **only functions and constructs** introduced in this lecture. Hint: list comprehensions may come in handy.

- Define last', your implantation of library function last that selects the last element of a list
- Similarly, define init' that like the library function init removes the last element from a list

To Get Started

- Define sumsq that takes a positive integer n and returns the sum of squares of first n ints:

  `sumsq n = 1² + 2² + ... + n²`

- Write function comb that takes positive integers m and n and returns "n choose m":

  `comb n m = n!/(m!*(m-n)!)`

- Write function prime that takes a positive integer n and returns 0 if n is prime, and some positive integer otherwise