Introduction and Haskell I

Dr. Isaac Griffith
Idaho State University

Outcomes

After today's lecture you will:

- Introduce the course and review the syllabus
- Learn the basics of Haskell
    - Running Haskell Scripts and using the GHCi
    - Haskell Types and Expressions
    - Basic Haskell Data Structures
        - Lists and List Comprehensions
        - Tuples
    - Functions

Syllabus Review

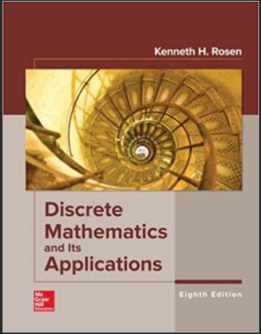
CS 1187

Course Introduction

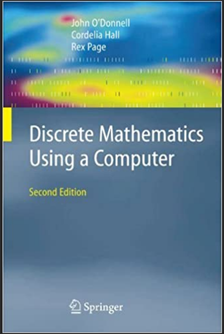
CS 1187

Idaho State University | Computer Science

What is Discrete Structures/Mathematics?



Discrete Mathematics
and Its Applications, 8th Edition
Rosen 2019



Discrete Mathematics Using
a Computer, 2nd Edition
O'Donnell, Hall and Page (2006)

ROAR

Course Introduction

- **Goals of this course**:
  - To understand and be able to apply the following concepts of discrete mathematics to computing
    - Logic
    - Set Theory
    - Relations and Functions
    - Counting
    - Number Systems and Modular Arithmetic
    - Recursion and Induction
    - Graphs and Trees
  - To be able to use symbolic mathematics, logical laws, and logical inference in mathematical reasoning
  - Assess mathematical and logical arguments for validity, construct mathematical arguments and simple proofs, and apply definitions and theorems

Course Introduction

- In considering this topic:
  - I wanted to better connect the goals of the previous slide to computing
  - I thought: “What better way than through programming itself?”
  - Additionally: “What if the language was a paradigm the students haven’t seen and one which is very close to the math itself?”
- Towards this, I opted to select the Haskell language, and then found the course book which already put all of this together.
- This adds four additional goals for this course, provide by Haskell:
  - Be able to understand and use the functional programming paradigm.
  - Be able to use the computer to help you to learn and understand mathematics.
  - Be able to use software tools to make it possible to use the mathematics more effectively.
  - Understand the widespread application of mathematics to computing.

Haskell Introduction

- Why a functional language? Why Haskell?
  - Allows you to compute directly with fundamental objects of discrete mathematics
  - Haskell is powerful yet expressed simply
  - We reason about Haskell programs in the same way we reason about mathematics
  - Haskell is excellent for *rapid prototyping*
  - Haskell is stable, standard, and well-documented
  - Haskell implementations are free and available on most OSs
  - Haskell can be used interactively

REPL

- What is a ***REPL***?
  - ***R*ead** - reads in code from command prompt
  - ***E*valuate** - evaluates the code read in
  - ***P*rint** - prints the result of the evaluation to the terminal
  - ***L*oop** - loops on REP until the user exits

- Several REPLs exist for Haskell
  - **GHCi**
  - Hugs
  - nhc

Running Haskell

Idaho State University | Computer Science

- To start haskell interactively (with GHCi), do the following (assuming you have installed Haskell):
  1. Open a terminal (Linux/MacOS) or Command Prompt (Windows)
  2. At the prompt, execute the following command (Note the `$` is the prompt not a part of the command)

     ```
     $ ghci
     ```

  3. This should start the interactive haskell system. Additionally it should give you an intro message followed by a prompt, for example:

     ```
     GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
     Prelude>
     ```

     where `Prelude>` is the prompt.

  4. Personally, I don't like the `Prelude>` prompt, so I set it to `ghci>` using the following command:

     ```
     :set prompt "ghci> "
     ```

Running Haskell

Idaho State University | Computer Science

- As noted previously, this prompt is the REPL
  - We can type in *expressions* and see their results (similar to a calculator)

- For example:

```
ghci> 1 + 2
3
ghci> 3 * 4
12
ghci>
```


ROAR

Loading Files

- When working with haskell, we often need to use definitions stored in scripts
  - Haskell scripts end with the **.hs** extension
  - To load a file while in GHCi, we use the load file command:

    `:load <module>` or `:l <module>`

    Where `<module>` is the name of the file to load **(without the .hs extension)**.

- Throughout this course, we will be using the book's `stdm.hs` file, so you should download it
  - Additionally, I would suggest creating a folder for your Haskell work
  - In this folder should be the `stdm.hs` file
  - This folder is the working directory where you should be prior to starting your Haskell interactive sessions

ROAR

Working with definitions

- Along with loading prior definitions created by others, we will also want to create our own code.

- Using a text editor, create a new file `mydefs.hs` and add the following to it:

```
y = x + 1
x = 2 * 3
```

- Save the file to your Haskell working folder. Then load it as follows:

```
ghci> :l mydefs
[1 of 1] Compiling Main             ( mydefs.hs, interpreted )
Ok, one module loaded.
ghci>
```

Working with Definitions

- After loading `mydefs.hs` both `x` and `y` in the file are definitions, which are now available to us
  - So we could do something like:

```
ghci> x
6
ghci> y
7
ghci> x * y
42
ghci>
```

  - If we make any changes to the file
    - then we would need to reload it for those changes to take affect in GHCi
      - using the `:reload` or `:r` command, or
      - load again with the `:load` or `:l` command

GHCi Commands

- All GHCi commands start with ":" which tells the system that we are executing a command for the environment not entering an expression

- The commands we will be using the most are:
  - `:load <module>` or `:l <module>` which loads the specified module (or file in our case)
  - `:reload` or `:r` which reloads the current module set
  - `:type <expr>` or `:t <expr>` which identifies the type of the given expression
  - `:cd <dir>` change the current working directory to the one specified
  - `:set prompt <prompt>` which sets the prompt to the given string
  - `:quit` or `Ctrl+D` (on some systems) which exits GHCi

Haskell Code Structure

- Haskell has an apparent lack of structure
  - No extra punctuation (colons, semicolons, braces, begin, end, etc.)
  - Instead, structure is controlled by line endings and indentation
    - Haskell will figure the rest out for us

- Comments
  - **line comments** start with -- and everything after it is ignored

```haskell
x = 2 + 2 -- the result should be 4
```

  - **multiline comments**: `{- text -}` where the `text` is the comment, and anything between `{-` and `-}` will be ignored

```haskell
{-
This is a multiline comment
-}
x = 2 + 2
```


ROAR

Haskell Expressions

CS 1187

Expressions

Idaho State University | Computer Science

- We can do a lot with just expressions in Haskell
- The following discusses useful kinds of expressions organized according to value ***type***
  - `Integer` and `Int` expressions
  - Rational and Floating Number Expressions
  - Boolean
  - Character and String Expressions

Integer and Int

Idaho State University | Computer Science

- Integer constants are simply a sequence of digits:
  `2, 0, 12345, -72`

- **Operators:**
  - Addition `(+)` : `4 + 3`
  - Subtraction `(-)` : `4 - 3`
  - Multiplication `(*)` : `2 * 3`
  - Exponentiation `(^)` : `2^3`
  - Division `` `div` `` : ``4 `div` 2`` (note the backticks)

`Integer` and `Int`

- Haskell has two Integer types
  - `Int` - a whole number whose maximum size fits within a *word* in memory (i.e., 64 bits on a 64bit machine)
  - `Integer` - a type representing mathematical integers
    - Using `Integer` type allows arithmetic operations to satisfy algebraic laws
- The *has type* operator `(::)`
  - Can be used to force the type specification rather than allow Haskell to infer the type
  - Example: `2::Int` or `2::Integer`

ROAR

Floating Point Numbers

- Types of Floating Point numbers:
  - `Float` - single precision
  - `Double` - double precision

- Operators:
  - Addition `(+)`
  - Subtraction `(-)`
  - Multiplication `(*)`
  - Division `(/)`
  - Exponentiation `(**)` : `4.0 ** 2`

Floats are Approximations

- Floating points are approximations
  - Cannot guarantee satisfaction of algebraic laws
  - Cannot directly compare two floating point numbers for equality

- **Procedure to compare Floating Point numbers:**
  - for two floating point numbers x and y
  - compare the absolute value of the difference: $|x - y|$ to some small error tolerance
  - Mathematically: $|x - y| < 0.001$
  - In Haskell:

```haskell
isEqual :: Float -> Float -> Bool
isEqual x y = (abs (x - y)) < 0.001
```

Rational Numbers

- To get around the limitations of Floating Point arithmetic, Haskell also supports exact arithmetic on rational numbers
- To use the exact form we must work with fractions in which both the numerator and denominator are integers
- The type of these numbers is `Fractional` and is written as `num/denom`, for example:
  - `2/3`
  - `2/3 + 1/6` → `5/6`
- **Note:** Haskell will automatically reduce fractions

Booleans

- Booleans are represented by the `Bool` type which can be one of two values (note the capitalization)
  - `True`
  - `False`

Comparing Two Numbers

- `==   -- equality`
- `/=   -- not equal`
- `<    -- less than`
- `<=   -- less than or equal`
- `>    -- greater than`
- `>=   -- greater than or equal`

Boolean Logic

- `&&  -- Boolean and (True only if both are True)`
- `||  -- Boolean or (False only if both are False)`
- `not -- Boolean not (Returns opposite truth value)`

Characters

- Characters have type `Char`
- Constants are written using single quotes: `'a'`
- **Useful Operations**
  - Comparison operators can be used
  - `toUpper` - converts lowercase to uppercase (must `import Data.Char`)
  - `toLower` - converts uppercase to lowercase (must `import Data.Char`)
  - Examples:

```
ghci> 'c' < 'Z'
False
ghci> import Data.Char
ghci> toUpper 'w'
'W'
ghci> toLower 'Q'
'q'
```

- Special Character: *newline* which causes a line break when printed
  - Written as `'\n'`

Strings

- A `String` is a sequence of zero or more characters.

- Constants are written inside double quotes:
  - `"tree"`

- **Useful Operators and Operations**
  - Concatenation `(++)` - joins two strings together
    - Example: `"abc" ++ "defg" => "abcdefg"`
    - Example: `"Here is a line" ++ "\n"`
    - **Note:** will not join a `String` to any other type
  - `length` - operation which counts the length of the string
    - Example: `length "abc"` $\rightarrow$ 3
    - Example: `length ""` $\rightarrow$ 0

Basic Data Structures

CS 1187

Tuples

- Tuples provide the capability to store multiple values in a single variable
  - `(1, 2)`
  - `('a', 2, "cab")`
- A tuple type is defined by two characteristics
  - The number of items to be stored, which is fixed once defined
  - The order of types to be stored, types do not need to be homogenous
- Examples:

```
("dog", "cat") :: (String, String)
(True, 5) :: (Bool, Int)
('a', "b") :: (Char, String)
("bat", (3.14, False)) :: (String, (Double, Bool))
```

Tuples

Idaho State University | Computer Science

- The general name is an *n-tuple* where *n* the number of components (i.e., 4-tuple)
  - a 2-tuple is also called a *pair*
  - a 3-tuple is also called a *triple*
  - no such thing as a 1-tuple
  - However, there is a special 0-tuple in Haskell
    - Written as `()`
    - Often used as a dummy value

- Pairs are a commonly used data structure, common operations include:
  - `fst (a, b)` - where the argument to the function is the tuple `(a, b)`, will return the value `a`
  - `snd (a, b)` - where the argument to the function is the tuple `(a, b)`, will return the value `b`

Lists

- Lists are the most common data structure used in functional programming
  - Size: unlimited
  - Type: all elements must be the same type
  - Examples:

```haskell
[1, 2, 3]
['c', 'a', 'b']
[] -- empty list
```

- Type is written as: `[A]` where `A` is the type of the contained elements

```haskell
[13,9,-2,100] :: [Int]
["cat", "dog"] :: [String]
[[1,2], [3,7,1], [], [900]] :: [[Int]]
```


ROAR

Sequences

Idaho State University | Computer Science

- If we recall, a String is a sequence of characters, well this is just a list:

```
"string" == ['s','t','r','i','n','g']
```

- Additionally, we define a list using a range:

```
[1..10] == [1,2,3,4,5,6,7,8,9,10]
[0,2..10] == [0,2,4,6,8,10]
[10,9..0] == [10,9,8,7,6,5,4,3,2,1,0]
```

- Ranges also work for characters

```
['a'..'z'] == "abcdefghijklmnopqrstuvwxyz"
['0'..'9'] == "0123456789"
```

List Notation and `(:)`

The **Cons Operator** - `(:)`

- We can construct new lists with the `:` operator

- This is an infix binary operator
    - Left argument is an element to add to the list
    - Right argument is a list
    - Type: `(:) :: a -> [a] -> [a]`

- Examples:

```
1 : [2, 3] => [1, 2, 3]
1 : [] -> [1]
```

List Notation and `(:)`

- Thus, we can write a list a series of cons operations:

```
[1, 2, 3, 4] == 1 : (2 : (3 : (4 : [])))
"abc" == 'a' : ('b' : ('c' : []))
```

- However, since `(:)` is right-associative, we can drop the parentheses

```
[1, 2, 3, 4] == 1 : 2 : 3 : 4 : []
"abc" == 'a' : 'b' : 'c' : []
```

- **Note:** that the end of the cons sequence is always the empty list

ROAR

List Comprehensions

- **List Comprehension** - a simple but powerful syntax to directly define a list
  - based on set comprehensions from mathematics
    - Example set comprehension: $\{x^2 | x \in \mathcal{S}\}$
  - does not require a program to build a list

List Comprehension Syntax

General Form: `[expression | generator, ..., filter, ...]`

- Read the `|` as *such that*
- `generator` - defines a sequences of values that a variable will take on and is written in the form `var <- list`
  - **Note:** there may be more than one generator, one for each variable in the expression (acts like loop nesting)
- `expression` - evaluated for each value that the generator variable(s)
- `filter` - are `Bool` expressions that apply to one or more generator variables in order to determine if the value will be included or not
  - If the expression evaluates to `False` then the value is thrown out
  - Filters are optional and there can be more than one filter, each separated by commas

List Comprehension Examples

- List of the product of each pair from another list

```
[a * b | (a, b) <- [(1, 2), (10, 20), (6, 6)]]
       => [2, 200, 36]
```

  - *expression* - `a * b`
  - *generator* - `(a, b) <- [(1, 2), (10, 20), (6, 6)]`

- List of numbers that are divisible by 5 from the range 1 to 1000

```
[a `mod` 5 | a <- [1..1000]]
        => [5, 10, 15, 20, ...]
```

  - *expression* - ``a `mod` 5``
  - *generator* - `a <- [1..1000]`

ROAR

List Comprehension Examples

```
[(x, y) | x <- [1,2,3], y <- ['a', 'b']]
    => [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

```
[x | x <- [0..100], x `mod` 2 == 0 && x `mod` 7 == 0]
    => [0,14,28,42,56,70,84,98]
```

```
[x | x <- [1..12], y <- [1..12], x*y == 12]
    => [1,2,3,4,6,12]
```

Exercises

Idaho State University | Computer Science

Work out the values of the following list comprehensions; then check your results by evaluating them with the computer

```
[x | x <- [1,2,3], False]
```

```
[not (x && y) | x <- [False, True],
                y <- [False, True]]
```

```
[x || y | x <- [False, True],
          y <- [False, True],
          x /= y]
```


ROAR

Exercises

Idaho State University | Computer Science

Work out the values of the following list comprehensions; then check your results by evaluating them with the computer

```
[x | x <- [1,2,3], False]
    => []
```

```
[not (x && y) | x <- [False, True],
                y <- [False, True]]
    => [False, False, False, True]
```

```
[x || y | x <- [False, True],
          y <- [False, True],
          x /= y]
    => [True, True]
```


ROAR

Functions

CS 1187

Function Application

- **Function Application** - when an expression uses a function
  - A function is **applied** to its arguments
  - Syntax: `func arg1 arg2`
    - `func` is the function name
    - `arg1` and `arg2` are the arguments (may be 0 or more of these all separated by spaces)

- Example:

```
sqrt 9.0
3.4 + sqrt 25 * 100
2 * sqrt (pi * 5 * 5) + 10
```

Function Types

- Functions like data have types, and these types are quite important

- **Function type** - for a function, *f*, with an argument of type *a* and a return type of *b* has a *function type* of $a \rightarrow b$ (read *a* arrow *b*).
  - Thus we would write: `f :: a -> b`

- **Example Function Types**

```
sqrt :: Double -> Double
max :: Integer -> Integer -> Integer -- first two are the args
not :: Bool -> Bool
toUpper :: Char -> Char
```

Operators and Functions

- **Operator** - a function which takes exactly two arguments
  - **Infix notation** - when the operator is written between the arguments
    - `(+)` operator - `2 + 4`
    - `min` function - ``2 `min` 4`` the backticks allow it to act as an operator
  - **Prefix notation** - when the operator
    - `(+)` operator - `(+) 2 4` when used like this or by itself it must be enclosed in parentheses
    - `min` operator - `min 2 4`

- All operators are functions, and thus have a function type
  - `(+) :: Integer -> Integer -> Integer`
  - `(&&) :: Bool -> Bool -> Bool`

Function Definitions

- A function definition has two parts:
  1. **Type declaration** which has the following form:

     $function\_name :: argType_1 \rightarrow argType_2 \rightarrow \ldots \rightarrow argType_n \rightarrow resultType$
  2. **Defining Equation** which has the following form:

     $function\_name\ arg_1\ arg_2\ \ldots\ arg_n = expression\ using\ the\ arguments$

- Function definitions should be written in a Haskell script file
  - To use the functions, the file should be loaded into GHCi

- Example Function Definition

```
square :: Integer -> Integer
square x = x * x
```

For Next Time

- Review DMUC Chapter 1.1 - 1.5
- Review this Lecture
- Come To Lecture
- Read DMUC Chapter 1.6 - 1.10

Are there any questions?