COURSE NAME: SOFTWARE FOUNDATIONS 558

DATE: 8/21/18

1. Lecture
   1. Professor passed out syllabus.
   2. Specific announcements:
      1. There will be 2 in-class midterms and 1 final exam.
      2. Check UNM email for class announcements.
      3. Everything will be uploaded to Learn follow each lecture session.
      4. Assignments will also be posted on Learn.
   3. Professor passed out additional handout on academic integrity.
      1. This form must be signed and returned to professor.
   4. Textbooks:
      1. For part 1 of the course on Haskell programming, you will need the Lipovaca book. It’s available online (see syllabus).
      2. For part 2, you will need the Pierce book. You don’t need to buy this book, however.
      3. **By next Tuesday, read Chapters 1-3.**
2. Imperative vs. Declarative Programming Pardigms
   1. Imperative languages 🡪 when you write programming language, you are saying that there is a specific sequence of actions you want to occur.
   2. Declarative languages 🡪 these specify what the results should be, and not necessarily the steps that need to be taken.
      1. This includes the Haskell functional language.
      2. Professor showed example of C/Java:
         1. This uses an index list to produce a value for res. The programming tells you exactly how to compute the result.
         2. The example of Haskell is much shorter and only asks for a sum value (the same intention as C/Java example).
   3. Haskell:
      1. Functions are first class objects 🡪 this makes it functional. The function can be used and then passed to other functions.
      2. It’s pure, meaning there are no side effects.
         1. In the C/Java example, the value stored is changed each time. Haskell does not have that happen.
      3. It has referential transparency meaning the variables can be easily referenced because they do not change.
      4. Powerful type system
      5. Can be run using either ghci interpreter or compiled using a ghc.
   4. Base types:
      1. Int (machine dependent);
      2. Integer (arbitrary);
      3. Float (floating point);
      4. Double (double precision floating point);
      5. Bool (true or false);
      6. Char (character);
         1. Must capitalize first letters!
   5. Simple arithmetic
      1. Professor showed example using HP. It automatically makes conversions between int, float.
      2. Bad code: 3 + ‘c’
         1. HP will not allow an integer be added to a character. The types MUST match.
      3. Boolean:
         1. && and
         2. || or
         3. /= inequality
         4. == equality
         5. <= less than or equal
         6. >= greater than or equal
      4. Example: x13 = succ 7 🡪 will add 1 successor
      5. Example: x15 = div 7 3 🡪 this says 7 divided by 3
   6. Defining simple functions
      1. Start by saying what type it is (example: int)
         1. Example: square : : Int -> Int 🡪 this takes an int and returns an int.
      2. Then define what it actually does:

Square n = n \* n

* + 1. Do the fourth power by:

fourthpower : : Int -> Int

fourthpower n = square (square n)

So we use the square power twice for the fourth power.

* + 1. Function composition takes two functions as operations and then applies them twice:

fourthpower = square . square

This declares the same thing as above.

* + 1. The input and output do not have to be the same. For example:

greaterthanZero : : Float -> Bool

* + 1. Applying a function twice 🡪 this example applies a function twice. f is the function. It takes x, passes it to the function f, and then passes it again to f.

twice : : (Int -> Int) -> Int -> Int

twice f x -= f (f x)

* + - 1. You can also use the twice function for the fourthpower:

fourth n = twice square n

* + 1. Recursive functions
       1. In imperative programming, these would be iterations. But in HP, we use recursion.
          1. The function calls itself using a smaller argument and then constructs res from the recursive calls.
       2. Defining:

fact : : Integer -> Integer

fact n = if n==0 then 1 else n\*fact (n-1)

* + 1. Lists
       1. Example of how to write:

nums, range1, range2, irange : : [Int]

nums = [4, 8, 15, 16, 23, 42]

ramge1 = [1 . . 10]

range 2 = [100,98 . . 0]

irange = [1. . ]

* + - 1. You can add elements to list by adding elements to the empty list using the “cons” operator (:)
      2. First element of the list is the head. The rest is the tail.
      3. You can define functions on the lists. One example is by computing the length:

(length [2, 3, 5]) = = 3

length : : [a] -> Int

length [] = 0

length (x:xs) = 1 + length xs

* + 1. Strings
       1. In the previous example, we have lists of a. So a is the type variable that can stand for any type. It’s polymorphic.
       2. We’ll finishing next time.

##end notes##