Introduction to Haskell

COMP360
“Home computers are being called upon to perform many new functions, including the consumption of homework formerly eaten by the dog.”

Doug Larson
Reading Assignment

http://learnyouahaskell.com/introduction
Programming Language Paradigms

- Imperative – Program execution changes the state
  - Procedural – Cobol, Fortran, C
  - Object-Oriented – Java, PHP, C++
- Declarative – Explain what instead of how
  - Functional – Haskell, Scala
  - Logical – Prolog
  - Domain specific – SQL, HTML
Functional Programming

• As the name implies, functional programs compute functions
• Recursion instead of iteration
• Uses pure functions
• Can write first-class functions (functions that take functions as parameters and return functions)
Pure Functions

• Pure functions to not have any side effects
  → no I/O
  → no changing of global variables

• The result is determined completely by the inputs

• You always get the same result from the same input
Get Haskell

• You can download Haskell from www.Haskell.org
• There are many tutorials on Haskell
• A nice tutorial is available at http://learnyouahaskell.com/introduction
• WinGHCi provides a nice interface
Command Line Interface

• The GHCi program allows you to execute a function
• You can write functions in a text file (extension .hs) and load them with

  :load filename
Simple Functions

• Arithmetic statements are functions
  • 5 + 3 will return 8

• Functions are called by typing the name followed by the parameters
  • NO (parenthesis)

• myfunc first second
Defining your own Functions

• You can enter a function on the command line after the `let` command

• You can type a function into a file and load it into Haskell with the `:load filename` command

• Functions are defined by

  functionName parm1 parm2 = something
Return Values

• Functional programs return values
• Imagine the first word in each Haskell function is “RETURN”
• Whatever value the function creates is returned
Try It

• Write a Haskell function called `poly` that calculates $2x^2 - 3x + 1$

• Try it with several numbers
  • `poly 2` returns 3
  • `poly 3` returns 10
  • `poly 1` returns 0
  • `poly 0.5` returns 0
  • `poly -1` gives an error, but `poly (-1)` returns 6
if Statement

• The Haskell \textbf{if} statement has the form
  \begin{verbatim}
  if cat == dog then something else whatever
  \end{verbatim}
• Note there are no (parenthesis)
• The \textbf{else} is required. The function must always return a value
Try It

• Write a Haskell function that returns “root” if the return value of poly is zero and “not zero” otherwise
Write a Recursive Method

• Haskell uses recursion instead of iteration
• Write a method to compute the nth Fibonacci number when $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$
• Haskell allows you to create lists, such as [3, 4, 7]
• All values in a list must be the same type
• Strings (i.e. “COMP360”) are lists of characters
List Operators

• `++` Concatenate two lists into one
  
  
  \[3, 5, 7]++[2,4,8] \text{ gives } [3,5,7,2,4,8]

• `:` Add another element to the beginning of a list
  
  \[42:[3, 5, 7] \text{ gives } [42,3,5,7]
  
  “dog” : [] \text{ gives } [”dog”]

• `!!` returns the n\text{th} element of a list
  
  \[3, 5, 7]!!1 \text{ gives } 5
List Functions

• **head** – returns the first element of a list
• **tail** – returns a list without the first element
• **last** – returns the last element of a list
• **init** – returns a list with everything except the last element
• **length** – returns the length of a list
• **null** – returns true if the list is empty
More List Functions

• **reverse** – reverses a list
• **take** – takes a number and a list and returns that many elements from the beginning of the list
• **drop** – takes a number, n, and a list and returns a list without the first n elements
• **maximum** – returns the largest element in a list
• **minimum** – returns the smallest element in a list
And More List Functions

• **sum** – returns the sum of all elements in a list
• **product** – returns the product of all elements in a list
• **elem** – given a value and a list, returns true if the value is in the list

  elem 3 [3, 5, 7]  returns true
  elem 2 [3, 5, 7]  returns false
Recursion Basics

• A recursive method calls itself with a (usually) smaller parameter
• A recursive method must have some base case that indicates when the recursion stops
• Most recursive functions (any language) start with an if statement to determine if this is the base case
Recursion not Iteration

• Recursion is used to repeat something, not iteration
• A method to sum all elements of a list could be:

```c
mySum str = if length str == 0 then 0
            else head str + mySum (tail str)
```
Write a Haskell Function

• Write a function that takes a list and an integer and returns that element of the list

getone [2,3,5,7,11,13,17] 3 returns 5
List Ranges

• You can create a list by specifying the first and last elements

  \([1..5]\) is \([1,2,3,4,5]\)

The elements can increment by more than one

  \([3,5..12]\) gives \([3,5,7,9,11]\)
Write a Haskell Function

• Write a Haskell function to calculate the sum of the squares of a list of numbers

sqrsum [2, 3, 4] will return 29
Possible Solution

• Write a Haskell function to calculate the sum of the squares of a list of numbers

\[
\text{sumsqr \ lst = if length \ lst == 0 then 0 else (head \ lst) * (head \ lst) + sumsqr (tail \ lst)}
\]
Write a Haskell Function

• Write a function that will multiple all numbers in a list by a constant

```
multby [2,3,4] 7   returns  [14,21,28]
```
Infinite Lists

• You can create an infinite list in Haskell:
  
  \[3,5..\] creates \[3,5,7,9,11,13,15,17 \text{ and so forth}\]

• Haskell uses lazy evaluation. It will not create a list until it needs to do so:
  
  \(\text{take 4 } [3,5..] \text{ returns } [3,5,7,9]\)

• Haskell only had to create the first four elements of the list.
Lists in Lists

• Lists can contain lists
• All list types must be the same
  
  ```python
  [['A', 'E', 'I'], ['B', 'C', 'D']]  
  ```
List comprehensions

• A list comprehension contains
  
  \[ \text{output function} \mid \text{input set}, \text{predicate} \]

• such as
  
  \[ [x^2 \mid x < [0..5]] \text{ returns } [0,1,4,9,16,25] \]

• or
  
  \[ [x^2 \mid x < [0..10], x^2 < 50] \text{ returns } [0,1,4,9,16,25,36,49] \]
Creating a List

• A list comprehension creates a new list

\[
\text{upchar dog} = [\ \text{succ cat} \mid \text{cat} <- \text{dog}]
\]

\[
\text{upchar "Aggies" returns "Bhhjft"}
\]
Write a Haskell Function

• Using a list comprehension, write a function that will multiple all numbers in a list by a constant

\[ \text{multby} \{2,3,4\} \times 7 \quad \text{returns} \quad \{14,21,28\} \]
Write a Haskell Function

• Write a function that takes a string and returns a string with only the vowels of the input string

vowels "This is a sentence." returns "iiaaeee"

elem is a useful function that given a value and a list, returns true if the value is in the list
Possible Solution

• Write a function that takes a string and returns a string with only the vowels of the input string

vowels st = [c | c <- st, elem c "aeiouyAEIOUY"]
Tuples

• Tuples are like list but they can have different data types
• Tuples are written with parenthesis
  (5,"cow")   (1,"aardvard")
• When using multiple tuples, they must have elements of the same type
Tuple Functions

• **fst** – returns the first element of a tuple
• **snd** – returns the second element of a tuple
• **zip** – given two lists, returns a list of tuples with elements from the first list matched with the second list

    zip [1,2,3] [ 'X', 'Y', 'Z' ] returns [(1,'X'),(2,'Y'),(3,'Z')]