Haskell Review

COMP360

“Some people talk in their sleep. Lecturers talk while other people sleep”

Albert Camus

Remaining Schedule

		Friday, April 7 Haskell
Monday, April 10 Logic Programming read chapter 12	Wednesday, April 12 Prolog	Friday, April 14 *Good Friday* *(no classes)*
Monday, April 17 Types	Wednesday, April 19 Drag & drop programming	Friday, April 21 Drag & drop programming
Monday, April 24 Concurrent Programming read chapter 13	Wednesday, April 26 Concurrent Programming	Friday, April 28 Concurrent Programming
Monday, May 1 **Exam 3**	Wednesday, May 3 final review	
Tuesday, May 9 **Final Exam** 10:30am – 12:30pm		

Return Values

- Functional programs return values
- Imagine the first word in each Haskell function is "RETURN"
- Whatever value the function creates is returned

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:

```
mySum str = if length str == 0 then 0
                   else head str + mySum (tail str )
```

Logical NOT

- The built-in not function in Haskell performs a logical inverse

not (4 < 5) returns False

not (6 > 9) returns True

Infinite Lists

- You can create an infinite list is Haskell

  [3,5..] creates [3,5,7,9,11,13,15,17 *and so forth*]

- Haskell uses lazy evaluation. It will not create a list until it needs to do so

  take 4 [3,5..] returns [3,5,7,9]

- Haskell only had to create the first four elements of the list

List Comprehensions

- List comprehensions build lists
- You can specify the requirements for the elements of the list
- The elements of the list can come from one or more existing lists
- Sometimes it is useful to build lists from newly created lists, such as [1,2..N]
- Recursion is usually not required with list comprehensions

List comprehensions

- A list comprehension contains

  [output function | input set , predicate]

- such as

  [x*x | x <- [0..5]] returns [0,1,4,9,16,25]

- or

  [x*x | x <- [0..10], x*x < 50] returns [0,1,4,9,16,25,36,49]

Multiple Predicates

- You can specify multiple restrictions on the elements of a list
- An element is included only if it meets all predicates
- Think of the predicates as being ANDed together

```
tryit cat = [ x | x <- cat, x > 5, x < 30]
```

Multiple Lists

- A list comprehension may have input from multiple lists
- Assume you want to create a list of tuples with all combinations of two lists

```
allPair cow bull = [(x,y) | x <- cow, y <- bull ]
```

allPair ["COMP", "ECE", "MATH"] [1,2] returns

[("COMP",1),("COMP",2),("ECE",1),("ECE",2),("MATH",1),("MATH",2)]

Infinite Lists

- You need to be careful with infinite lists in comprehensions

- Assume you want to create a list of tuples numbering all the items in a list

allPair cow = [(x,y) | x <- cow, y <- [1..]]

allPair ["COMP", "ECE", "MATH"] returns

[(1,"COMP"),(2,"COMP"),(3,"COMP"),[(4,"COMP"), (5,"COMP"),(6,"COMP"), … *forever*

Parameter Values

- You can specify values in the parameters of a function
- If the function is called with that specific parameter, that definition of the function will be used

```
roll 7 = "you win"
roll 11 = "you win"
roll 2   = "you lose"
roll x   = "you continue"          -- matches anything
```

Matching List Parameters

- You can match an empty list
- The form [h:tl] matches a list where h is the head and tl is the tail

```
doubleSum :: (Num a) => [a] -> a
doubleSum [] = 0
doubleSum (h:tl) =  2 * h + doubleSum tl
```

Observations

- Do not use "a" or "t" as variable names
- Avoid duplicating existing function names
- Do not use tab in a function definition
- Functions should start in column 1 of a file

Functional Programming Advantages

- lack of side effects makes programs easier to understand
- lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
- lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
- programs are often surprisingly short
- language can be extremely small and yet powerful

Functional Programming Problems

- difficult (but not impossible!) to implement efficiently on von Neumann machines
- lots of copying of data through parameters
- (apparent) need to create a whole new array in order to change one element
- heavy use of pointers (space/time and locality problem)
- frequent procedure calls
- heavy space use for recursion
- requires garbage collection
- requires a different mode of thinking by the programmer
- difficult to integrate I/O into purely functional model

ELSEVIER

Remaining Schedule

		Friday, April 7 Haskell
Monday, April 10 Logic Programming read chapter 12	Wednesday, April 12 Prolog	Friday, April 14 *Good Friday* *(no classes)*
Monday, April 17 Types	Wednesday, April 19 Drag & drop programming	Friday, April 21 Drag & drop programming
Monday, April 24 Concurrent Programming read chapter 13	Wednesday, April 26 Concurrent Programming	Friday, April 28 Concurrent Programming
Monday, May 1 **Exam 3**	Wednesday, May 3 final review	
Tuesday, May 9 **Final Exam** 10:30am – 12:30pm		