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

Albert Camus
## Remaining Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Class</th>
<th>Date</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friday, April 7</td>
<td>Haskell</td>
<td>Friday, April 14</td>
<td>Good Friday (no classes)</td>
</tr>
<tr>
<td>Monday, April 10</td>
<td>Logic Programming</td>
<td>Wednesday, April 12</td>
<td>Prolog</td>
</tr>
<tr>
<td></td>
<td>read chapter 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, April 12</td>
<td></td>
<td>Wednesday, April 19</td>
<td>Drag &amp; drop programming</td>
</tr>
<tr>
<td>Wednesday, April 12</td>
<td>Prolog</td>
<td>Friday, April 21</td>
<td>Drag &amp; drop programming</td>
</tr>
<tr>
<td>Monday, April 17</td>
<td>Types</td>
<td>Wednesday, April 19</td>
<td></td>
</tr>
<tr>
<td>Monday, April 19</td>
<td>Drag &amp; drop programming</td>
<td>Friday, April 21</td>
<td>Drag &amp; drop programming</td>
</tr>
<tr>
<td>Wednesday, April 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, April 24</td>
<td>Concurrent Programming</td>
<td>Wednesday, April 26</td>
<td>Concurrent Programming</td>
</tr>
<tr>
<td></td>
<td>read chapter 13</td>
<td>Friday, April 28</td>
<td>Concurrent Programming</td>
</tr>
<tr>
<td>Monday, May 1</td>
<td>Exam 3</td>
<td>Wednesday, May 3</td>
<td>final review</td>
</tr>
<tr>
<td>Tuesday, May 9</td>
<td>Final Exam</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10:30am – 12:30pm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
\text{mySum str } = \begin{cases} 
0 & \text{if } \text{length str } == 0 \\
\text{head str + mySum (tail str)} & \text{otherwise}
\end{cases}
\]
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 in Haskell:
  \[3,5..] \text{ creates } [3,5,7,9,11,13,15,17 \text{ and so forth}]\]

• Haskell uses lazy evaluation. It will not create a list until it needs to do so:
  \text{take 4 [3,5..] 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
  \[ \text{[ output function | input set , predicate]} \]
- such as
  \[ [x^2 | x \leftarrow [0..5]] \text{ returns } [0,1,4,9,16,25] \]
- or
  \[ [x^2 | x \leftarrow [0..10], x^2 < 50] \text{ returns } [0,1,4,9,16,25,36,49] \]
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

```haskell
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
## Remaining Schedule

<table>
<thead>
<tr>
<th>Monday, April 10</th>
<th>Wednesday, April 12</th>
<th>Friday, April 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Programming</td>
<td>Prolog</td>
<td>Good Friday</td>
</tr>
<tr>
<td>read chapter 12</td>
<td></td>
<td>(no classes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monday, April 17</th>
<th>Wednesday, April 19</th>
<th>Friday, April 21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Drag &amp; drop programming</td>
<td>Drag &amp; drop programming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monday, April 24</th>
<th>Wednesday, April 26</th>
<th>Friday, April 28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Programming</td>
<td>Concurrent Programming</td>
<td>Concurrent Programming</td>
</tr>
<tr>
<td>read chapter 13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Monday, May 1</th>
<th>Wednesday, May 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exam 3</strong></td>
<td>final review</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuesday, May 9</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Final Exam</strong></td>
<td>10:30am – 12:30pm</td>
<td></td>
</tr>
</tbody>
</table>