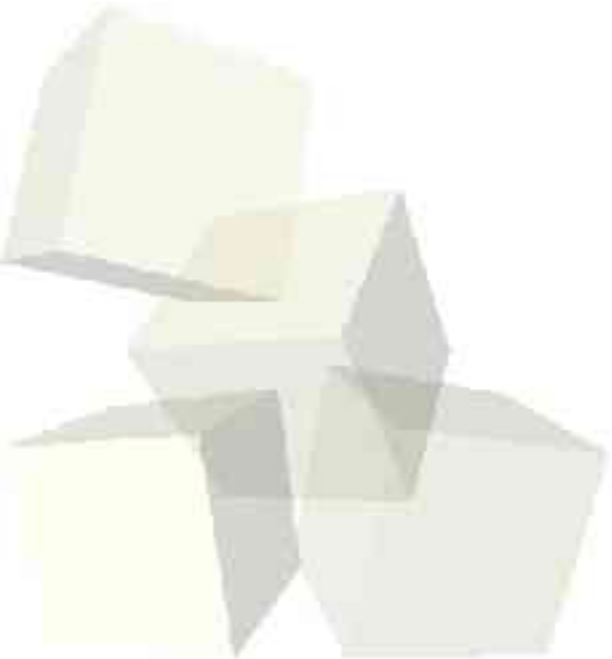
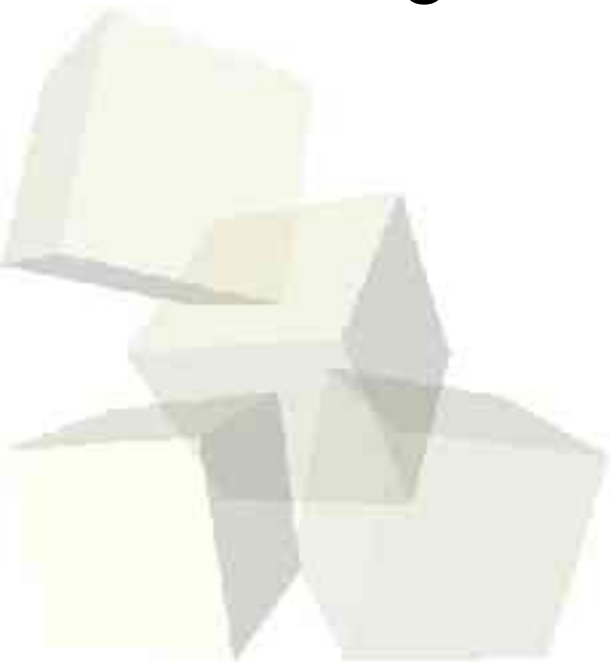
Quick Introduction to Haskell

12-3-2004



Opening Discussion

- Do you have any questions about the quiz?
- What are some of the features of O'Caml that we discussed last time?
- Do you have any questions about the assignments?



Basics

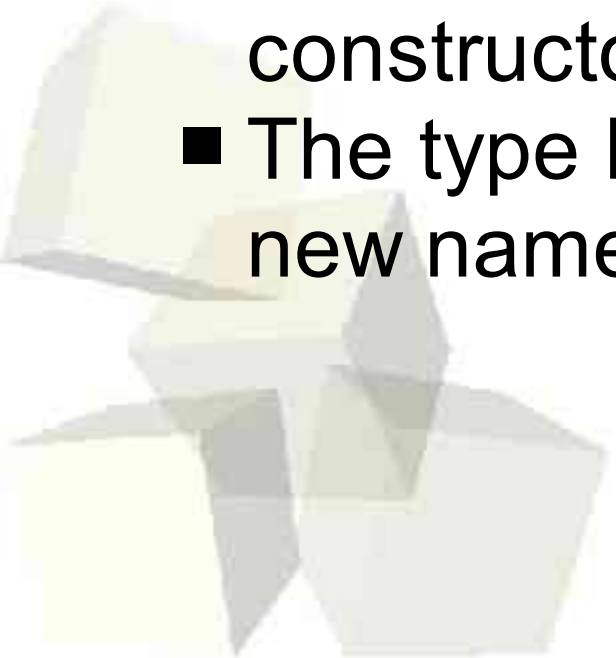
- Haskell is “pure” functional language that has roots in ML, but is more distant than O'Caml.
- Haskell can infer types like ML and O'Caml, but they also suggest users provide intended types. The compiler will check if they are valid and give an error if not.
- Haskell has lazy evaluation which has some really major implications.
- You never use fun, val, or the O'Caml style let. Declarations are just written as equations.

Simple Changes

- Haskell doesn't use ;.
- They use :: for type and : for cons. List concatenation is done with ++. Lists otherwise look like ML, but they have some pretty impressive abilities we'll talk about later.
- Haskell also had regular looking tuples.
- For some reason the interactive environment doesn't let you define functions. That's just one environment though. Other implementations are optimizing compilers with no interactive environment.

User Types in Haskell

- We can define our own type in Haskell with the “data” declaration. The syntax looks a lot like ML, but has some difference. Largely, the parameters follow the type name and no “of” is used after the constructors.
- The type keyword works like in ML to give a new name to a type.



List Comprehensions

- This is something that is unique to Haskell as far as I know. Basically, we can construct lists in a way that reads very much like set theory in math. For example
  - [f x | x <- lst]
- This makes a list of elements f(x) for all x in the list lst.
- The code to the right of the | is called a generator and we can have multiple of them with comma separation.
  - [(x,y) | x <- lst1, y <- lst2]

More List Power

- You can also put guards on the list generation. These are boolean expressions that go with the generators in comma separations. Only combinations that satisfy the boolean are selected.
- Let's look at code for quicksort using this power.
- Lists can also be made with sequences like [1..10] or [3..99]. What is more, Haskell allows infinite lists such as [5..], but I don't recommend doing that at the top level.

Strings and Lambda Abstractions

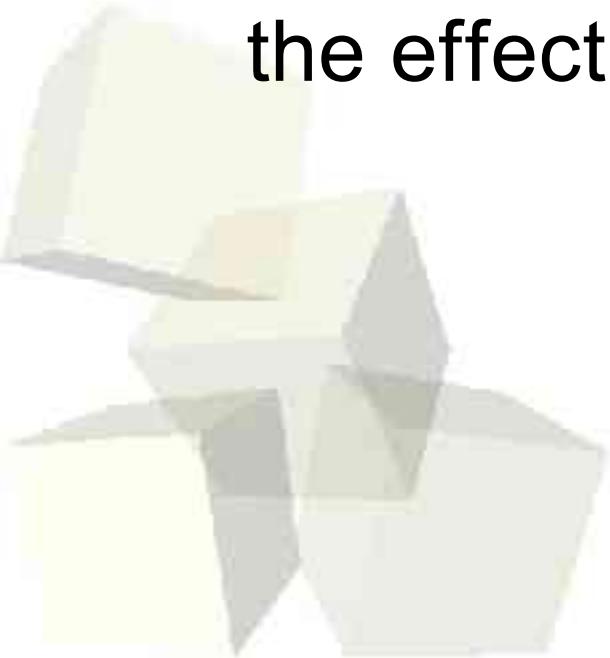
- In Haskell, strings are nothing more than shorthand for lists of characters.
- Thankfully, characters use the syntax of a single character in single quotes.
- A “lambda abstraction” is basically an anonymous function. We define them as follows.
  - \<args> -> <expression>
- Haskell allows the definition of infix operators as long as they contain only symbols.

Sections and Prefix Operators

- Haskell allows the partial evaluation of infix operators (since they are curried).
  - (+x) is like \y -> y+x
- Similarly, by putting an operator in parentheses we get a prefix operator. This is just like putting “op” before an operator in ML.
- A prefix operator can be changed to an infix operator by putting backwards single quotes around it.
  - 5 `elementOf` lst
- Precedence of infix ops can be set.

Lazy Evaluation

- Haskell can have the infinite lists we saw before because it uses lazy evaluation. That implies that a function doesn't evaluate its arguments unless it actually needs them.
- Let's look at some code that demonstrates the effect of lazy evaluation.

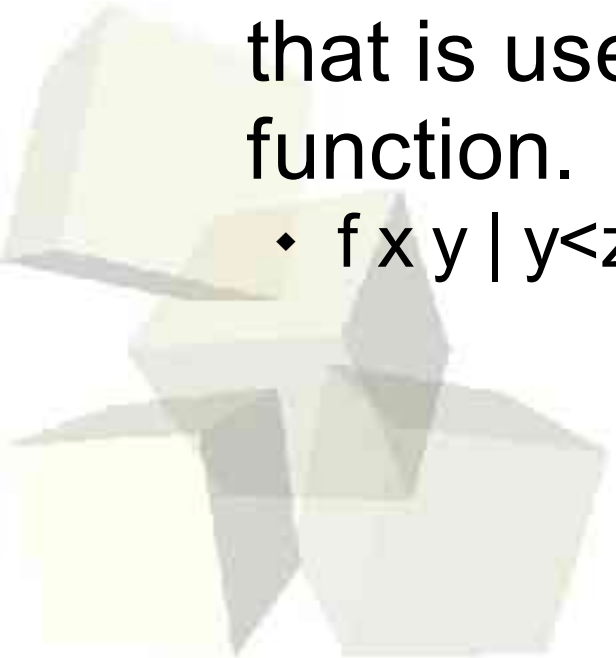


Patterns in Haskell

- Haskell has patterns very much like ML or O'Caml with _ as a wildcard. They use @ for “as”.
- Patterns in Haskell can have guards, multiple boolean tests on each pattern.
- Haskell has a case-of expression that does a match like in ML.
- By putting a ~ in front of a pattern we make it lazy. It now matches anything, but won't be processed until needed.

let and where

- Haskell and a let similar to ML, but without the end.
- Haskell also has a where clause that can go after all the matches in a function or case statement. This allows you to define a value that is used across many patterns in a function.
  - f x y | y<z = ... | y==z = ... | y>z = ... where z=x*x

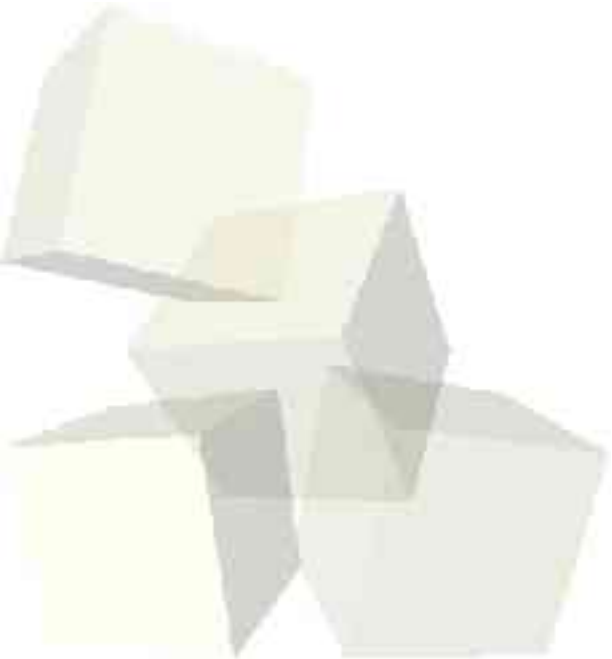


Type Classes

- Haskell allows you to define things called type classes. A type class produces a type of polymorphism that is more restrictive than the parametric polymorphism of the parameterized types.
- You define a set of functions that must be defined for types in that class.
- You can also use inheritance to make subtypes of type classes.

Modules and I/O

- Haskell also provides functionality for doing I/O as well and for creating modules. We don't have the time to get into the details of those systems though.



Minute Essay

- So what are your thoughts about Haskell? What would you think about it being taught in future offerings of functional?
- I like the idea of teaching at least two functional languages in this course. What languages would you teach and how long would you spend on each one? What extra topics would you cover? Which ones would you throw out?