**CS 240 : Principles of Functional Programming**

**1. Overview**

After completing the introductory programming sequence (CS 115 & CS 116 or CS 201), students remain ill-equipped to begin programming in the large[[1]](#footnote-1). For one, they are unfamiliar with how to use and combine basic data structures such as lists and hashmaps. They have also not been taught to recognize the dangers of misusing programming concepts such as mutable and shared state. Additionally, because of the strong focus of the introductory sequence on the imperative and object-oriented programming paradigms, students have not been exposed to a number of important programming concepts and abstractions including lexically scoped closures and higher order functions[[2]](#footnote-2).

We therefore propose a new required course, to be taken by all CS undergraduate students after the completion of CS 116 / CS 201, which will focus on ideas and techniques that will help give students the confidence to construct large programmed systems. The class aims to teach concepts related to managing complexity in programs, building and reusing composable modules, and choosing the right paradigms and tools for the problem at hand. Also, to balance the strong imperative bent of the introductory sequence, the class will make use of a language with a functional core (e.g., Scheme).

Another motive for teaching the functional programming paradigm stems from the increasingly urgent need to prepare students to write programs that leverage multi-core processors, i.e., concurrent programs. Abstractions derived from the functional world and related techniques for isolating state mutations are very useful for dealing with concurrency and non-determinism, and can greatly reduce the complexity of concurrent programs. These, in turn, clearly complement the core objective of the class.

In section 2 we list the ``big picture ideas'' students are expected to internalize after completing the class; section 3 lists secondary (mostly pragmatic) learning objectives; section 4 contains a detailed list of topics to be covered; section 5 breaks down the amount of academic hours to be apportioned to each topic.

**2. Big picture ideas**

* records (aka structs) let us build complex data structures
* data types and ``shapes'' drive function design and implementation
* higher order functions let us build abstractions and manage complexity
* lexically scoped closures are a core programming concept
* mutable state increases complexity and must be carefully managed
* languages can be molded to suit our needs
* antipatterns increase complexity: avoid them!

**3. Secondary learning objectives**

* learn how to organize large programs using a module system
* learn how to use a REPL and debugger to help test and debug code
* learn how, where, and when to write and run tests
* understand and apply the practice of iterative development
* learn how to use a distributed version control system

**4. Detailed topics**

* Records as building blocks
* vectors
* self-referential structures → lists, association lists, trees
* hashmaps
* Data driven design
* simple records → destructuring / reconstitution
* self-referential structures → recursion
* primitive recursion and accumulation
* type descriptions
* polymorphism
* Higher order functions
* composition
* partial application
* mapping & reducing
* towards aspect-oriented programming
* Closures as a core abstraction
* as functions
* as classes & objects
* for lazy evaluation
* Mutable state hazards & management
* unpredictability
* referential transparency
* Language molding
* ``bottom-up'' design
* domain specific languages
* macros
* Practicum
* distributed version control
* module systems, packages, and namespaces

**5. Hourly breakdown**

Links:

* Brown <http://www.cs.brown.edu/courses/cs019/2012/readme>
* CMU [www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf](http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf)
* MIT <http://j.mp/TWp7ip>
1. We use ``programming in the large'' to refer to the design and implementation of programmed systems made up of a large number of functions and/or modules, requiring a high degree of abstraction, reuse and composability. [↑](#footnote-ref-1)
2. This is partly due to the inordinate amount of time spent teaching the syntax of the Java programming language. It is not advisable to switch languages at this time, however, due to both industry demand and that of the College Board. [↑](#footnote-ref-2)