CS 495: Programming Paradigms and Patterns

January 13, 2016

Contents

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]. 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

[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.

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]

We 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 6 presents and highlights relevant topics and learning objectives from the 2013 ACM Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.

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

[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.

- 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 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
- 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
  - unpredictability
  - referential transparency

- Antipatterns
  - repetition (copy/paste coding)
  - accidental complexity
  - circular dependencies
  - action at a distance (via pointers/references)
  - premature optimization

- Language molding
  - "bottom-up" design
  - domain specific languages
  - macros

- Practicum
  - distributed version control
  - module systems, packages, and namespaces
  - instrumentation / profiling

5 ACM Curriculum Guidelines for CS UG Degree Programs

5.1 Relevant Knowledge Areas (KAs)

5.1.1 Programming Languages / Functional Programming

Topics:

[Core-Tier1]

- **Effect-free programming**
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
  - Data can be freely aliased or copied without introducing unintended effects from mutation
- **Processing structured data (e.g., trees) via functions with cases for each data variant**
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Functions defined over compound data in terms of functions applied to the constituent pieces
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- **Function closures** (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition – creating closures at runtime by capturing the environment
  - **Canonical idioms**: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
  - Currying and partial application
- Defining **higher-order operations on aggregates, especially map, reduce/fold, and filter**

Learning outcomes:

[Core-Tier1]

1. **Write basic algorithms that avoid assigning to mutable state or considering reference equality. [Usage]**
2. **Write useful functions that take and return other functions. [Usage]**

3. **Compare and contrast (1) the procedural/functional approach** (defining a function for each operation with the function body providing a case for each data variant) **and (2) the object-oriented approach** (defining a class for each data variant with the class definition providing a method for each operation). Understand both as defining a matrix of operations and variants. [Assessment] This outcome also appears in PL/Object-Oriented Programming.

[Core-Tier2]

1. **Correctly reason about variables and lexical scope in a program using function closures. [Usage]**
2. **Use functional encapsulation mechanisms such as closures and modular interfaces. [Usage]**
3. **Define and use iterators and other operations on aggregates**, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. [Usage] This outcome also appears in PL/Object-Oriented Programming.

5.1.2 Programming Languages / Advanced Programming Constructs

Topics:

- **Lazy evaluation and infinite streams**
- **Control Abstractions**: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
- **Metaprogramming: Macros, Generative programming, Model-based development**
- **Module systems**
- **String manipulation via pattern-matching (regular expressions)**
- **Dynamic code evaluation ("eval")**
- **Language support for checking assertions, invariants, and pre/post-conditions**

Learning outcomes:

1. **Use various advanced programming constructs and idioms correctly. [Usage]**
2. **Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]**
3. **Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]**

5.1.3 Programming Languages / Language Pragmatics

Topics:

- Principles of language design such as orthogonality
- **Evaluation order, precedence, and associativity**
- **Eager vs. delayed evaluation**
- **Defining control and iteration constructs**
- **External calls and system libraries**

Learning outcomes:

1. **Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Familiarity]**
2. **Use crisp and objective criteria for evaluating language-design decisions. [Usage]**
3. **Give an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Usage]**
4. **Show uses of delayed evaluation, such as user-defined control abstractions. [Familiarity]**
5. **Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation. [Familiarity]**

5.1.4 Software Development Fundamentals / Development Methods

Topics:

- Program comprehension
- Program correctness
  - **Types of errors (syntax, logic, run-time)**
  - **The concept of a specification**
  - **Defensive programming (e.g. secure coding, exception handling)**
  - **Code reviews**
  - **Testing fundamentals and test-case generation**
  - **The role and the use of contracts, including pre- and post-conditions**
  - **Unit testing**
- **Simple refactoring**
- **Modern programming environments**
  - Code search
  - Programming using library components and their APIs
- **Debugging strategies**
- Documentation and program style

Learning outcomes:

1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2. **Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]**
3. Identify **common coding errors** that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Usage]
4. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
5. Contribute to a **small-team code review** focused on component correctness. [Usage]

6. Describe how a contract can be used to specify the behavior of a program component. [Familiarity]
7. Refactor a program by identifying opportunities to apply procedural abstraction. [Usage]
8. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
9. **Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers**. [Usage]
10. Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]
11. Analyze the extent to which another programmer's code meets documentation and programming style standards. [Assessment]
12. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]

5.1.5 Software Engineering / Tools and Environments

Topics:

- **Software configuration management and version control**
- Release management
- Requirements analysis and design modeling tools
- Testing tools including **static and dynamic analysis tools**
- Programming environments that automate parts of program construction processes (e.g., automated builds)
  - **Continuous integration**
- Tool integration concepts and mechanisms

Learning Outcomes:

1. **Describe the difference between centralized and distributed software configuration management**. [Familiarity]

2. **Describe how version control can be used to help manage software release management. [Familiarity]**
3. **Identify configuration items and use a source code control tool in a small team-based project. [Usage]**
4. **Describe how available static and dynamic test tools can be integrated into the software development environment. [Familiarity]**
5. **Describe the issues that are important in selecting a set of tools for the development of a particular software system**, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Familiarity]
6. **Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage]**