Topics in CS3, Fall 2007

Common to all areas

- define and use procedures
- given an error message, identify its cause
- give a good comment for a mystery procedure
- test a mystery procedure
- devise test suites that exercise all expressions in the program, along with boundary cases
- critique a test suite
- simplify complex code
- find a bug
- fix the bug
- characterize input values that yield symptoms of the bug
- characterize input values that do not yield symptoms of the bug
- compare and contrast procedures

PreRecursion

- translate algebra to Scheme
- work with words and sentences
- predict the result of expressions involving an empty word or an empty sentence
- compare and contrast English words and sentences with Scheme words and sentences
- identify the effect of "shadowing" a procedure name
- supply parentheses and quotes to produce an expression with a given value
- identify misconceptions relating to misuse of parentheses and quotes
- work with conditionals, `and`, `or`, and `not`
- implement a procedure using `and`, `or`, and `not` instead of `if` and `cond`, and vice versa
- use `member?` in place of large `cond` expressions
- check for a valid argument

Recursion

- design a general recursion from individual procedures that handle inputs of size 0, 1, 2, 3, 4...
- supply base cases
- identify and prune redundant base cases
- supply recursive cases
- identify infinite recursions
- identify type mismatches between the value returned in the base case and the value returned in the recursive case
- design a recursion involving both the `butfirst` and the `butlast` of a word or sentence
- design a recursion that builds a sentence or word (front to back, or vice versa)
- design a recursion involving pairs of words in a sentence or characters in a word
- design a recursion with two arguments, both changing in the recursive call
- design an accumulating recursion
- understand tail versus embedded recursion
- design a nested recursion
- design a procedure that involves calls to two different recursive procedures
- provide base cases for a tree recursion
- trace a tree recursion
- count the number of recursive calls in a tree recursion that result from a given call

Case Studies

- summarize the case study
- determine which of two procedures is written according to principles from the case study
- model the development (testing and debugging) of a program

Higher order procedures

- identify the domain and range of a given function
- use the built-in higher-order procedures (`keep`, `every`, and `accumulate`)
- supply the appropriate higher-order procedure to produce a given result
- supply an argument to a given higher-order procedure to produce a given result
- identify errors in the use of the built-in higher-order procedures
- give a good comment for arguments to a higher-order procedure
- compare recursive implementations of the built-in higher-order procedures
- identify which direction `accumulate` accumulates
- supply parentheses to get a given result
- use `lambda` and use `let`
- identify the need to use `lambda`
- implement and use a higher-order procedure that's not built-in

Lists and beyond

- use `car` and `cdr`
- distinguish the effects of `cons`, `list`, and `append`
- supply one of `cons`, `list`, and `append` to produce a given result
- give a combination of uses of these procedures that produces a given result
- add parentheses and quotes to produce a given result
- use `member`
- use and implement a semipredicate
- identify appropriate uses for `member`
- use the built-in higher-order procedures for lists (`map`, `filter`, `reduce`, and `apply`)
- distinguish `reduce`, `accumulate`, and `apply`
- use `map` with multiple list arguments
- use `assoc`
- provide a table for use with `assoc`
- identify appropriate uses for `assoc`
- write a procedure to process a generalized list
- analyze a procedure that processes a generalized list
- Understand the structure of trees, and write a HOF program to process a tree. (Mutual recursion is *not* covered in this course).
- Understand the difference between sequential and functional programming
- Use `begin` to do sequential programming
- Use `show` and `display` to send output to the screen
- Use `random` to generate random numbers

Working with existing programs

(*Difference Between Dates*, *Roman Numerals*, *Tic Tac Toe*, and *Change Making*)

- draw a call tree
- provide sample calls that produce a given result or result in a given number of calls to a given procedure
- identify appropriate arguments for a procedure
- given input values for a procedure, determine the value it returns
- given erroneous input values for a procedure, determine if and where it crashes
- given a category of input values for a procedure, determine all possible values it could return
- use the procedures to implement some other computation
- modify or extend the program
- rewrite one of the procedures
- determine the effect of making a given modification
- given symptoms of a bug resulting from changing a single word, integer, or symbol in the program, identify possibilities for the bug and describe how you determined them
- invent bugs for your partner to find
- invent a test case that exercises as much of the program as possible
- provide a good comment for one of the procedures