CS 111 - Exam 1 Review Suggestions - Fall 2016

last modified: 2016-09-29

- You are responsible for material covered in class sessions, lab quizzes, lab exercises, and homeworks; but, here's a quick overview of especially important material.
- You are responsible for the material covered through the end of the Week 6 Labs (Friday, September 30) and through and including the Week 6 Lab Exercise and Homework 5.
- You are permitted to bring into the exam a single piece of paper (8.5" by 11") on which you have **handwritten** whatever you wish on one or both sides. This paper must include your name, it must be handwritten by you, and it will **not** be returned.
  - Other than this piece of paper, the exam is closed-note, closed-book, and closed-computer.
- This will be a pencil-and-paper exam, but you will be writing and reading Racket expressions and functions in this format. You will be answering questions about concepts, expressions, and functions as well.
- Your studying should include careful study of posted examples and notes as well as the lab quizzes, lab exercises, and homeworks thus far.

Expression Basics

- What is meant by syntax? What is meant by semantics?
- In Racket, what is a simple expression? What is a compound expression? How do you write simple and compound expressions in Racket?
  - Given an expression, you should be able to tell if it is a "valid" Racket expression.
  - You should be able to tell if an expression is simple or compound; you should be able to write a simple or compound expression.
- What Racket data types have we discussed so far?
  - How can you write simple expressions that are literals (simply values) of most of these data types? How can you write a simple expression in each of these data types?
  - How can you write compound expressions of each of these data types?
  - For the functions we have discussed so far, you should know what type(s) of expressions each expects, and what type of value it produces.
  - Given a simple or compound expression, you should be able to give its type (that is, to give the data type of its value).
  - How can certain predicates/boolean functions be used to verify the data type of an expression?
- How can you write a comment in Racket?
- What are some of the arithmetic operations/functions provided by Racket? Given a numeric operation in "algebraic" form, you should be able to write it as an equivalent Racket expression.

- How can you give a name to a value in Racket? Recall that when such a name represents an unchanging value in a program, we call it a **named constant**; you should be able to define and use named constants appropriately.
  - Why is it good to use named constants within functions? (You should know at least two reasons.)
  - Note that it is a course style standard (on exams as well as homeworks and labs) to write a named constant in all-uppercase.
  - Given an example of Racket code, you should be able to name the named constants in that code.
  - You could be asked to define a named constant; you could be asked to use a named constant that has been previously defined.
- Recall that an **identifier** is any name chosen by a programmer; named constants, function names, and parameter names are all examples of kinds of identifiers.
  - In terms of style, how should identifier names be chosen?
  - You should be comfortable with the basic syntax for Racket identifiers (for example, what should identifier names start with? Can you have blanks in identifier names?)
  - You should be able to choose identifier names that meet Racket syntax and course style guidelines; you should be able to tell if something is not a syntactically-correct identifier for Racket.
- We know that an identifier, once defined, is considered to be a simple expression. You should be able to tell the difference between a simple expression that is an identifier and a simple expression that is a literal.
  - Given a simple expression, you should be able to tell if it is a "valid" Racket identifier, or if it is a "valid" Racket literal.
- You should be very comfortable reading and writing Racket expressions, function definitions, named constant definitions, and function calls.
  - You should be comfortable with both simple and compound expressions.
  - Note that `define` expressions are unusual in that they do not themselves produce a value; a `define` expression does not really have a type! But it has an important side-effect, giving a value to an identifier or making an identifier a function.
- Given a signature and purpose statement for a function, you should be able to make appropriate use of that function; you should be able to write compound expressions using that function.
  - You also should be able to understand and use other functions we have not yet used if provided with signatures and purpose statements for such functions.

Function Basics

- What is the syntax for defining a new function in Racket?
- What is a function header? What is a function body?
  - Given a function definition, you should be able to tell what part is its header, and what part is its

body.

  - CS 111 course style: you are expected to write a function body on a separate line from the function header, slightly indented under the function header.
  - You should be able to write a function header if asked; you should be able to write a function body if asked.

- A **parameter** is a name defined in a function header, after the name of the function being defined, to stand in for an argument expression that this function will expect when it is called. Parameters are then used in the function body to determine what the value of that function will be when it is called.
  - In terms of style, what kind of a name should be chosen for each parameter?
  - Given an example of Racket code, you should be able to name the parameters in that code.
- When you use a function (when you write a compound expression using that function), the expressions you put after that function's name in that compound expression are called its **arguments**.
  - (Each argument becomes the value of a parameter in that function call.)
  - You should know the difference between parameters and arguments.
  - Given an example of Racket code, you should be able to tell the difference between parameters and arguments in that code; you should be able to give examples of parameters and arguments in that code.
- You should be very comfortable reading and writing Racket expressions, function definitions, named constant definitions, and function calls.
  - You should be comfortable with both simple and compound expressions
  - You should be comfortable with predicate/boolean functions
- You should be comfortable with the `2htdp/image` and `2htdp/universe` teachpack functions we have used so far; you could be asked to write specific expressions using these functions.
  - You should be comfortable with how the `2htdp/universe` teachpack can be used to set up and animate a world, using `big-bang`
  - Be sure you are comfortable with the difference between a scene, an image, and a world value
  - You should be able to read and create expressions involving images and scenes
  - You should be comfortable with what kinds of functions need to be provided to `on-tick`, `to-draw`, and `on-key`; you should be able to read and write such functions, and appropriate `big-bang` expressions.
  - Given such functions, named constant definitions, and `big-bang` call involved, you should be able to describe the animation that should result, including being able to describe what should happen if the user presses certain keys.
- You should be comfortable reading and writing expressions involving `modulo` to make sure that some expression results in a value in a desired interval
  - you should be comfortable reading and writing expressions involving `random`

  - you should be comfortable with colors, whether represented as strings or as `make-color` expressions

The Design Recipe

- You should be comfortable with the design recipe (since you should have been using it for all of the numerous functions that you have written up to this point).
  - Note that there are very likely to be questions about the design recipe itself; you should also be very familiar with the expected **order** of the steps in this process.
  - There WILL be questions where you will have to produce/perform specified steps within the design recipe.
- Give a scenario for a problem, you should be able to follow the steps of the design recipe for each function involved in solving that problem.

Step 1 - problem analysis and data definition

- Analyze the problem and determine what types of data are involved; (for some programs, you actually determine if you would like to "define" new data types for use in solving this problem, as we did when we wrote data definition comments for enumerations).
  - Note that there is not always a visible result to this step, especially for the kinds of functions we are writing at this point, although even for our simple functions at this point you might define a data definition for an enumeration or some named constants as part of this step. But you might also notice that you would like a named constant at other stages of the design recipe as well.

Step 2 - signature/purpose/header

- Write the **signature** comment for the function you are designing.
  - Remember: for CS 111, signatures are written as a comment with the following format:

```
; signature: funct-name: param-type param-type ... -> type-returned
```

  - Remember: in the signature, after the function's name, you only put types -- the types for each parameter expected, the type of the value produced by that function. You do not put parameter names or named constant names or any other descriptions of those values.
- Write an appropriate **purpose** comment for that function.
  - Each purpose comment needs to describe the values **expected** by that function, and needs to describe the value **returned** by that function.
  - The key word here is **describe** -- you do not just give types, since the signature already provides that information.
- Write the **function header** for that function.
  - Notice that the "new action" being done at this design recipe stage is to determine appropriate, descriptive names for each of this function's parameters

  - Although the header is really just:

    ```
    (define (funct-name param-name ... param-name)
    ```

  - ...we go ahead and add a simple template body, `...)`, to stand in for its eventual body at this point:

    ```
    (define (funct-name param-name ... param-name)
        ...
    )
    ```

Step 3 - develop specific examples/tests

- Develop `check-` (`check-expect`, `check-within`, etc.) expressions with specific examples/test cases for that function, including specific example arguments and including specifically what that function should produce when called with those arguments.
- Remember to include at least one `check-` expression for each "case" possible -- for each "kind" or category of data involved (although additional examples are always welcome), and, if applicable, for each "boundary" between different "cases"/"kinds" of data.
  - If there is only one kind of data, we still write at least two `check-` expressions, to better determine "patterns" for the eventual function body and to better test the eventual function.
- Expect to have to write some specific examples/test cases (in the form of `check-` expressions!)
- Expect to have to determine the appropriate minimum number of `check-` expressions needed for a given function being developed.
- Be able to write a string literal describing the expected value for an example call when a `check-` expression is not possible; be able also to write a string literal describing expected side-effects for an example call as needed (as for describing file input/output side-effects).

Step 4 - decide which body template is appropriate

- Determine if there are any templates that might be useful for helping to develop the body of a function involving the types of data involved in this function.
- e.g., templates for interval data, enumeration data, and itemization data; template for a function "walking through" a list

Step 5 - develop/complete the function's body

- Finish designing the function's body, using what you have learned/determined from the previous steps.

Step 6 - run the tests

- Run your code, and determine whether your tests succeeded.
- Feel free to add additional example calls of your function as desired, to see the values produced.

- How can you tell whether your tests succeeded?
- How would you debug your code if any tests failed?

The `cond` expression/branching

- You should be very comfortable with reading and writing the Racket `cond` (conditional) expression
- You should be comfortable with interval notation for expressing ranges of numbers
- You should be comfortable with boolean values, boolean operators (and relational operators), boolean expressions, and boolean functions
  - How can you see if one expression has the same value as another expression? How can you see if an expression is of a particular type?
- You should be able to determine the appropriate number of and kinds of test cases/specific examples you need, based on the kinds/categories of data involved in a problem.
- Expect to have to read, write some `cond` expressions!

Intervals, Enumerations, and Itemizations

- You should know what kind of data each of these terms describes; you should be able to write a data definition for these kinds of data (particularly for enumerations and itemizations).
  - Given a description of data or a data definition, you should be able to determine if it is interval data or enumeration data or itemization data.
- You should know what template is appropriate for the body of a function that expects one of these types.
- Given such a data definition:
  - ...what should you then include in a function's signature?
  - ...how can you write functions handling such data?
  - You should be able to read and write such functions.

Programs: Functions + Definitions

- We combine function definitions and named constant definitions to create programs.
- You should be comfortable with how the `2htdp/image` and `2htdp/universe` teachpack functions can be used to create an animation.
  - That is, you should be comfortable with how you can use them to write a collection of function definitions and named constant definitions that result in an animation using `big-bang`.
  - You should be able to recognize and create images and scenes.
- What is an auxiliary or helper function? Within a Definitions file containing multiple functions, in what relative order must the auxiliary functions be?

Lists

- What is the data definition for a list? How do you write a list in Racket? What are the list-related built-in functions? What is the template for a function that expects a list as a parameter (and "walks though" it)?
- I may ask you to give the value of expressions involving lists, or ask you to write an expression involving a list.
  - You definitely need to be comfortable with using `cons` to construct a list; you should be familiar with the list-shortcut `list` as well.
  - (Make sure you understand that `list` is just *syntactic sugar* to make writing a list expression easier -- "under the hood", a list is still made up of `cons`'ing a first item and a list, even when written using `list`.)
- I could ask you to give a data definition for a specialized list; I could ask you to give the template for a function that expects that specialized list type as a parameter and "walks through" it.
- Expect to have to read a function involving a list; expect to have to write at least one function involving a list. (Note that this might or might not involve "walking through" the whole parameter list...)
- When you are writing a recursive function - one that calls itself - what do we mean when we say that it needs one or more **base** cases? What do we mean when we say it has one or more **recursive** cases? What needs to be true about the recursive calls in the recursive cases?
  - You should be able to identify base cases and recursive cases in a given function.
  - You should be able to tell, in reading a function, whether it is recursive or not, and if it is "good" recursion (has at least one base case, and each recursive call is on a "smaller" part of the data such that a base case MUST eventually be reached).

File input/output

- How can you write a string to a file in BSL Racket?
- How can you read, from a file:
  - ...that file's contents as a single string?
  - ...that file's contents as a list of strings, 1 string per line in the file?
  - ...that file's contents as a list of strings, 1 string per white-space-separated "word" in the file?
- Why might it be useful for a program to be able to read from a file?
- Why might it be useful for a program to be able to write to a file?
- Be sure to be able to tell the difference between a function's parameters, and what it reads from a file -- and be sure to be able to tell the difference between what a function returns, and side-effects such as reading from or writing to a file.