CIS 130 - Exam 1 Study Suggestions

last modified: 02-20-10

- Anything covered in class or on a homework is fair game for the exam.
- You are responsible for the material covered through the end of Week 4 and through and including HW #3
  - (`cond` expressions and conditional functions will not be on Exam 1)
- These are simply suggestions of some especially important concepts, to help you in your studying.
- 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.
- You will be writing and reading expressions and functions on this exam; you will be answering questions about concepts, expressions, and functions as well.

Expression Basics

- What is meant by syntax? What is meant by semantics?
- In Scheme, what is a simple expression? What is a compound expression? How do you write simple and compound expressions in Scheme?
  - Given an expression, you should be able to tell if it is a "valid" Scheme 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 Scheme data types have we discussed so far?
  - How can you write simple expressions that are literals (simply values) of each of these data types (except for image)? 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 do you write a comment in Scheme?
- What are some of the arithmetic operations/functions provided by Scheme? Given a numeric operation in "algebraic" form, you should be able to write it as an equivalent Scheme expression.
- How can you give a name to a value in Scheme? 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 preferred style to write a named constant in all-uppercase.
  - Given an example of Scheme code, you should be able to name the named constants in that code.
- 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 Scheme 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 Scheme syntax and course style guidelines; you should be able to tell if something is not a syntactically-correct identifier for Scheme.
- 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, can you tell if it is a "valid" Scheme identifier? if it is a "valid" Scheme literal?
- You should be very comfortable reading and writing Scheme 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 a name or making a name a function.
- Given a contract 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 should be comfortable with the universe.ss and fabric-teachpack.scm functions we have used so far; you could be asked to write specific expressions using these functions.
  - You also should be able to understand and use other functions we have not yet used if provided with contracts and purpose statements for such functions.

Function Basics

- What is the syntax for defining a new function in Scheme?
- 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;
  - CIS 130 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 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 Scheme 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 also called **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 Scheme 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.

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 are very likely to be questions where you will have to produce/perform certain 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 would actually determine if you would like to define new data types for use in solving this problem, although that is beyond the scope of CIS 130);
  - 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 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 - Contract/Purpose/Header

- Write the **contract** for the function you are designing.

  - Remember: for CIS 130, contracts are written as a comment with the following format:

```
; contract: function-name: param-type param-type ... -> type-produced
```

  - Remember: in the contract, 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 statement** for that function.
  - Each purpose statement needs to describe the values **expected** by that function, and needs to describe the value **produced** by that function.
  - The key word here is **describe** -- you do not just give types, since the contract already provides that information.
- Write the **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 (`...`) to stand in for its eventual body at this point, and finish the define expression with a closing parenthesis:

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

Step 3 - Develop specific examples/test cases

- Develop `check-expect` or `check-within` 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-expect` or `check-within` expression for each "case" possible -- for each "kind" of data involved (although additional examples are always welcome), and, if applicable, for each "boundary" between different "cases"/"kinds" of data.
- Expect to have to write some specific examples/test cases (in the form of `check-expect` or `check-within` expressions!)

Step 4 (not applicable yet for Exam 1) - use a template if applicable

- 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; this step is not applicable yet for Exam 1, although it will be for Exam 2.

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?

Programs: Functions + Definitions

- We combine function definitions and named constant definitions to create programs.
- You should be comfortable with how the universe.ss 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;
  - You should be able to recognize and create images and scenes.