Start with a wish list for Scripts or Big-Bang programs. For Scripts you can add a main function for doing IO to the wish list. For Big-Bang programs you can add relevant event handling functions.

Maintain a list of function headers that must be designed to complete a program. Writing down complete function headers ensures that you can test those portions of the programs that you have finished, which is useful even though many tests will fail. When coding an incomplete portion, add auxiliary functions to your wish list as needed. Auxiliary functions are often needed when one piece of a data definition refers to another data definition (i.e. dealing with lists of structs). Of course, when the wish list is empty, all tests should pass and all functions should be covered by tests.

The design process for functions (so far) is:

1. Formulate a data definition and interpretation for the kinds of values flowing in and out of the functions in your program. Formulate data examples for use in testing. For structs, you construct smaller data examples for the different fields and then construct the example for the struct from the subparts. For itemizations, you want data examples for each item. For intervals, you want data examples on the boundaries of the interval, as well as in the middle. For arbitrarily large data, ensure that your specification has an item that contains atomic data.

2. Provide a signature, a statement of purpose, and a function header. A signature specifies all valid data flowing to and from a function. A statement of purpose should give a *high level* description of the purpose of the function. A function header stubs out relevant parameters and provides a temporary return value.

3. Provide functional examples to say what your functions expect as output when given a certain input. Use the data examples as input to your function. You can start by writing examples as comments, but eventually format them as tests.

4. Provide a skeleton for your code. Functions that accept itemizations should cond over the different items. When dealing with structs, select out a field at each level of nesting. For arbitrarily large data, you should skeleton out the natural recursion over the different structures.

5. Code out your definition. When dealing with structs, call functions on relevant fields. If you have a template function in your code, assume it does what you think and keep filling out the code in the current definition. When dealing with itemizations, let the functional examples drive the coding of the different branches in your cond. For recursive functions, design your combinator function from concrete examples and use the tabular method if stuck.

6. Test your code. If a test fails, determine if the test or code is wrong. Fix the faulty party by returning to step 3 or step 5 in the design process. Re-test, afterwards.