This assumes you know what the terms "function", "function name", "function definition", "literal", ", argument", "unary", "binary", "parameter name", "expression", "echo", and "call a function" mean.
To understand an assignment, in particular the design process we've been following, understand 
those terms first. When those terms come up here and in the assignment comments, ask us about 
the terms you don't know first (before asking about what to do for the assignment).

This is a summary. If you're tempted to read about a skill more than practicing it, read ahead only 
after you've tried and practiced earlier parts explicitly. Reading ahead can be counter-productive if 
you try too hard to understand parts you haven't ready for. When asking for help, let us know which 
step you're working on, and include the concrete work you have for the earlier steps.

Goal Example: write down at least one example of what you want your new function to accomplish.

```scheme
(same! (new-function-name example-literal-argument etc) 
example-literal-result)
```

After making your example, you notice that you decided:
• a name for your new function (the name you used in your example)
• how many arguments it will take/need (the number of arguments you used in your example)
• in particular, you implicitly decided whether the function will be unary, binary, ternary, etc

That also enables you to see the grammatical form of your future function definition.
With one made-up parameter name per argument, you can write down the form ...
```scheme
(define (new-function-name _ _ _) ; For example, if the function is binary.

_)
```
... and could even use that as a temporary definition, since it's grammatically correct, and even has 
the right number of parameters for the examples.

Problem-Solving: create an expression to produce your example-literal-result from your 
example-literal-argument's.

Copy the example-literal-arguments and example-literal-result into the Interactions area, so you can 
see them together there. Actually entering them and seeing them echoed back as-is, with no errors,
also confirms that you copied the literals and not other parts of your example.

Play around, following the Forward-Backward aspect of problem-solving.
Forward: play around with the arguments, calling functions on them, and/or editing them manually to 
get a feel for the computation, to try to produce results like the example result.
Backward: play around with the result, calling functions to try to produce it form arguments that are 
more like the example arguments, and/or edit it manually to try to break it into simpler parts that you 
might be able to produce from the example arguments eventually and then put together.

Design Record: make a record of your problem-solving, in particular for the example expression(s) 
that can compute the result.
```scheme
(same! (new-function-name example-literal-argument etc) 
one-of-your-expressions-that-computes-the-result)
```
That is known as a "Design" example.
When an expression to compute the result uses the arguments explicitly (unedited as-is), and nowhere 
else implicitly (a part that would need to change if the arguments changed), then it fully expresses the 
computation generally, which is known as a "Full Design". Other designs are known as "Partial".

Function Definition: mechanically turn your Full Design example into a function definition.
```scheme
(define (new-function-name a-parameter-name etc) 
modified-full-design-expression-that-computes-the-result)
```
Turn a Full Design into a correct function definition by copy-pasting it, then simply/mechanically:
• replace "same!" with "define"
• choose a parameter name for each argument
• modify the body by replacing each argument everywhere (so both in the header and body)
• with your chosen name