Programming with Induction, Recursion, and Algebraic Laws

CS 105 Assignment

Due Tuesday, September 13, 2022 at 11:59PM

Contents

This homework will help you

- Practice using the 9-step design process (especially naming, contracts, and algebraic laws)
- Think explicitly about induction and recursion
- Get used to fully parenthesized concrete syntax as used throughout the course
- Practice using the course software

Like all homeworks, **this one is preceded by comprehension questions** that help you focus your reading. Before starting the homework problems, download the comprehension questions and tackle the reading as they direct. Answer the comprehension questions on your own.

The homework itself has two parts:

- *Programming exercises* that reinforce your programming process and that develop skills with induction and recursion
- A *photograph* for the private use of the course staff, which will help us learn to recognize you and call you by name

The programming exercises must be done as individual work.

NOTE: This assignment is due one minute before midnight. You may turn it in *up to 24 hours after the due date*, which will cost you one extension token, as described in the syllabus. If you wish not to spend an extension token, then when midnight arrives submit whatever you have. We give partial credit.

ALERT: This assignment is significantly easier than a typical CS 105 assignment. Its role is to get you acclimated and to help you start thinking systematically about writing code from scratch, especially using algebraic laws to write recursive functions. Later assignments get harder and more time-consuming, so don't use this one to gauge the difficulty of the course.

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module. It is also OK to alternate between reading-comprehension questions and related homework questions.

Programming in Impcore (90%)

You can start these exercises immediately after the first lecture. You may write very efficient solutions—but do not feel compelled to do so. Just make sure that your recursive functions terminate!

Do not share your solutions with anyone. We encourage you to discuss ideas, but

- **No other student may see your code**.
- **No other student may see your algebraic laws**.
- **No other student may see the contracts for your helper functions**.

Getting set up with the software

The interpreter you need to run Impcore code is provided as part of the course. To add course software to your execution path, run

```
use -q comp105
```

You may want to put this command in your `.cshrc` or your `.profile`. The `-q` option is needed to prevent `use` from spraying text onto standard output, which may interfere with `scp`, `ssh`, `git`, and `rsync`.

The `impcore` interpreter is in `/comp/105/bin`; if you have run `use` as suggested above you should be able to run it just by typing

```
ledit impcore
```

The `ledit` command gives you the ability to retrieve and edit previous lines of input.

If your code and unit tests are in file `solution.imp`, you can load and run them by typing

```
impcore -q < solution.imp
```

Unit testing

The special "extended-definition forms" `check-expect`, `check-assert`, and `check-error` are part of every language in the book. For example, as described in section~1.2.1 of the book (page~17), they are part of the Impcore language. These forms serve both as unit tests and as documentation. **Every function you write must be tested and documented using `check-expect` or `check-assert`**, and possibly also `check-error`. The number of unit tests must be appropriate to the function's contract and to the structure of its input, as described in the first lesson on program design.

How do you know when to use `check-expect` and when to use `check-assert`? Like this: `check-expect` tells if two expressions evaluate to the same value, and `check-assert` checks to see if a single expression evaluates to truth or falsehood. To be confident you are getting things right,

- If you are writing a `check-expect` that something is 0 or 1, meant as a Boolean, then probably you should be writing `check-assert`. (If checking for 0, use `check-assert` with `not`, as shown below.)
- If you are writing a check-assert of an expression written with `=`, you should *definitely* write `check-expect` instead. It's more readable, and if it fails, you get a more useful error message.

Expectations for your software-design process

To write code from scratch, we recommend the nine-step design process that is presented in the introduction to *Seven Lessons on Program Design*. In this homework, the process is primarily for practice—many students will be able to do the problems without the help that the process provides. In a couple of weeks, though, you will need to have a good process, or you will find that the homeworks take too much time.

Our process has nine numbered steps. On this homework, here's what we expect for each step:

1. All data are integers, and most are natural numbers. We expect you to be able to break natural numbers down into the forms described in the first lesson on program design. Those forms will appear on the left-hand sides of the algebraic laws you write for step 6.
2. We expect you to be able to choose example inputs, which we expect to see in your unit tests.
3. We expect you to choose names only for any helper functions that you choose to write. Most problems, but not all, can be solved without helper functions. When there are helper functions,

they will be scrutinized carefully, and their names will be judged according to the general coding rubric.

4. We expect each function you submit to be preceded by a short contract that appears in a comment.

    - Contracts for the assigned functions don't require much thought; language from the book (or this homework) is fine.

    - Contracts for helper functions are another story. For each helper function you define, *choose a good name*, and write a *specific, accurate contract.* Use the coding guidelines. Your contract must meet the expectations laid out in the Documentation section of the general coding rubric.

    The contracts for the helper functions will weigh more heavily toward your grade.

    Here's a model contract:

    ```
    ;; (has-digit? n d) returns 1 if and only if the decimal
    ;; representation of positive integer `n` includes a decimal
    ;; digit that is equal to `d`; it returns 0 otherwise.
    ```

5. We expect unit tests to be submitted with each function, and we expect them to *follow* the function and to be indented eight spaces. We expect only the basics: one or two unit tests for each form of input.[1] Additional unit tests are acceptable, but they must be separated from the basic tests by a blank line.

    Here are two example unit tests:

    ```
    (check-assert (not (has-digit? 123 7)))
    (check-assert      (has-digit? 123 2))
    ```

6. We expect you to write algebraic laws for any function that contains an `if` expression or a recursive call. Algebraic laws should appear between a function's contract and its definition.

7. We expect the body of each function to have at most one case per algebraic law.

8. We expect the body of each function to be consistent with the function's algebraic laws.

9. We expect each function's unit tests to be indented eight spaces, and to pass. When a function is meant to return only 1 or 0, coding for "true" or "false," we expect that function's unit tests to use `check-assert` and to test *both* outcomes.

Help with the process: a solution template

To help you organize and present the results of your work, we provide a solution template. Copy the template into file `solution.imp`. The template provides placeholders for contracts (design step 4), unit tests (step 5), algebraic laws (step 6), and function definitions (steps 7 and 8).

In the template, you will see that most functions are followed by a single unit test that uses `check-error`. The test is a placeholder. Remove the `check-error` and replace it with unit tests that use `check-expect` or `check-assert`, which you must write. (For most functions, you will need multiple unit tests: at least one per algebraic law.)

[1]For ordinary results, we expect one test per form of input. For Boolean results, we expect one "true" test and one "false" test for each form of input, when possible.

Indent all unit tests by 8 spaces.

A placeholder function definition has body `(/ 1 0)`. Evaluating this code divides 1 by 0, which causes an error. The solution template should pass all unit tests, as reported by

```
impcore -q < solution.imp
```

The template does not include placeholders for helper functions. If you write any helper functions, *supply each helper function with a contract, laws, and unit tests*.

Quick help with Impcore

The concrete syntax of Impcore can be found on page~18 of *Programming Languages: Build, Prove, and Compare*. As a quick summary, the following code uses every possible syntactic form of expression. But on this assignment, **you must not use the `while` and `set` forms**, and the code you *submit* must not print.

```
(define even? (n) (= (mod n 2) 0))

(define 3n+1-sequence (n) ; from Collatz
  (begin
    (while (!= n 1)    ;; don't use `while` on your homework!
       (begin
         (println n)
         (if (even? n)
             (set n (/ n 2))    ;; don't use `set` on your homework!
             (set n (+ (* 3 n) 1)))))
    n))
```

The "initial basis" of a programming language is a fancy technical name for the names that are already defined. In Impcore, the initial basis comprises primitive and predefined functions, which have these names:

```
mod      or       println  /
!=       and      =        *
>=       printu   >        -
<=       print    <        +
not
```

The problems you must solve

You will solve three groups of ordinary problems and one challenge problem. Do the problems in the order in which they appear below, not in book-numbering order.

Direct applications of the design method

Each problem in the first group can be solved by direct application of the methods sketched in the first lesson on program design: choose a proof system, write algebraic laws, design the code.

The problems are as follows:

- **DD.** Define a function `double-digit`. When given a positive integer less than 20,000, `double-digit` returns a positive integer whose decimal representation is the same as the decimal representation of the input, except each digit appears twice. For example, `(double-digit 123)` is `112233`.
- **C.** Define a function `population-count`, which when given a nonnegative integer, returns the number of 1 bits in the binary representation of that integer. (This function is named for a machine instruction found on Intel CPUs.)
- **8**. Define the function `binary` described in exercise~8 on page~78 of *Build, Prove, and Compare*.

Generalization of the design method

Each problem in the second group can be solved by generalizing the methods in the first lesson on program design.

- **2.** Define the function `sigma` that is specified in exercise~2 on page~76 of *Build, Prove, and Compare*. Although m and n are not guaranteed to be natural numbers, this problem specifies $m \leq n$, so the difference $n - m$ is a natural number. Your algebraic laws should handle cases for $n - m = 0$ and $n - m = k + 1$, where k is a natural number. The laws should be written like this:

  ```
  ;; laws:
  ;;   (sigma m m) == ...
  ;;   (sigma m n) == ..., where n > m
  ```

  If you make a recursive call for the case $n-m = k+1$, the recursive call should satisfy $n-m = k$.

Recursion that is not structural

A recursive computation that is driven by the structure of the data, as in the first two groups of problems, is called *structural recursion*. Not all recursions are structural.

- **5**, part one. Define the function `prime?` that is specified in exercise~5 on page~76 of *Build, Prove, and Compare*. I recommend searching for a divisor. This search will require a recursive helper function, which is not structural.

  All helper functions must begin with algebraic laws. When the recursion is not structural, algebraic laws do not discriminate on the *forms* of the input. Instead, they discriminate on *properties* of the input. For this problem, one interesting property is that one number evenly divides another. There is also one other interesting property. In each law, write the interesting properties for that law as a *side condition*, which should appear after the right-hand side of the law, following a comma and the word “where.” As a model, look at the side condition for the property $n > m$ in the laws for `sigma` above.

  For the function `prime?` itself, I recommend discriminating using three mutually exclusive properties: either $n < 2$, $n = 2$, or $n > 2$. (Function `prime?` expects a nonnegative integer.)

A challenge problem

- **5**, part two. Define the function `nthprime` that is specified in exercise~5 on page~76 of *Build, Prove, and Compare*. This one should be implemented using structural recursion, and I recommend a structurally recursive helper function. The challenge here is to come up with a good name and a clear contract for the helper function.

Note that function `nthprime` expects a *positive* integer.

In addition to unit tests of individual functions, you may wish to consider unit tests for *properties* of combinations of functions, like these:

```
(check-assert (prime? (nthprime 6)))
(check-assert (not (prime? (+ 1 (nthprime 7)))))
```

Other expectations for your solutions

For this assignment, we expect you to apply the recommended design process and to deliver working functions with good names, clear contracts, algebraic laws, and unit tests. We also expect the following:

- Your solutions **must be valid Impcore**; in particular, they must pass the following test:

  ```
  /comp/105/bin/impcore -q < solution.imp > /dev/null
  ```

  with no error messages and no unit-test failures. If your file produces error messages, we won't test your solution and you will earn No Credit for functional correctness (you can still earn credit for readability).

- Your solutions must load and complete within 250 CPU milliseconds. If you write long-running tests, don't include them in `solution.imp`; instead, create a file `extra-tests.imp`. This file should include a line `(use solution.imp)`.

- On this assignment, as on several assignments to come, **your code must use recursion.** Code using `while` loops will receive No Credit.

- Code you submit must not call `print`, `println`, or `printu`. Code that prints is likely to fail our automated tests.

- You may use helper functions where appropriate, but **your code must not define or use global variables**.

- Your code must be your own work. **Do not share your solutions with anyone.** We encourage you to discuss ideas, but
  - **No other student may see your code.**
  - **No other student may see your algebraic laws.**
  - **No other student may see the contracts you write for your helper functions.**

Frequently asked questions

Here are some questions we are frequently asked about this assignment:

- When a contract gives an edge case condition, do we have to account for that? For example, in `double-digit`, do we need a condition for > 20,000?

- Should we return an error for invalid input? For example, introduce a division-by-0 error when input is invalid?

All these questions are really the same question: *am I obligated to detect a contract violation, and if so, how should my code respond?* The answer is no, your code is not obligated to detect a contract violation—and in most cases, it *should* not.

The contract is one of the premier tools we have to manage complexity in large systems. The beginner's mindset is, "every function must handle every possible case." In a real system, this mindset leads to complexity and code duplication—high costs. It is time to shift to a mindset that says, "every function need handle only the cases stated in its contract." Bugs still occur, and things go wrong, in which case the question becomes, "who violated their contract? Was it a caller or a callee?"

Every function should be as simple as it can be while satisfying a contract. If it so happens that you can detect a contract violation "for free," it is OK to halt the program with a checked run-time error. But this situation is rare. It is far more common that detecting a contract violation requires extra case analysis. In CS 105, this is never OK. Part of the purpose of a function's contract is to simplify that function's implementation.

Note: this mindset assumes that whatever is calling the function is trusted code. If a function's inputs ultimately come from a user (web form or whatever), then somebody needs to validate those inputs to be sure they conform to the contract, but that job is a separate concern, to be dealt with in another function.

A photograph we can use to learn your name (10%)

If it is safe to be in the classroom without masks, I hope to learn the name of every student in the class. The teaching assistants and recitation leaders would also like to learn your name. But we need your help. Part of the assignment, **for 10% of the grade on the assignment**, is to submit a photograph that will help us learn to recognize you. I've consulted with a skilled portrait photographer to prepare guidelines for producing a good, easily recognizable photograph, even if all you have is a cellphone camera. You'll submit the photo as `photo.jpg`.

What to submit and how to submit it

Before submitting the homework problems, you'll **submit the reading-comprehension questions**:

- Download file `cqs-impcore.txt`, edit in your answers, copy it to the server, and submit it using the script `submit105-cqs`.

To submit the main questions, choose a directory for your homework, in which you will create or copy these files:

- `solution.imp` will contain your code, with its documentation and unit tests. Problems will appear in the order in which they are listed above, which is also the order of the solution template.
- `photo.jpg` will contain a recognizable photograph of you.
- `README` will identify anyone with whom you have collaborated or discussed the assignment and will include any other information you wish to pass on to us. We provide a `README` template; please use it.

As soon as you have the files listed above, run `submit105-impcore` to submit a preliminary version of your work. The submit script checks your work and runs `provide` on your behalf. Do submit early, then keep submitting until your work is complete; we grade only the last submission.

The submit script will ask you some questions:

- Your preferred first and last names
- How we should pronounce your name, as in "kaeth-LEEN FI-shur" or "NORE-muhn RAM-zee."
- ~~How many hours you worked on the assignment~~

You may also submit `extra-tests.imp`, which should contain only test code, unit tests (`check-expect` or `check-error`), and the line `(use solution.imp)`. You can run the tests using the Unix command

```
impcore -q < extra-tests.imp
```

How your work will be evaluated

How your code will be evaluated

In this assignment, you learn how we expect your code to be structured and organized. Our expectations are presented on the coding-style page. When we get your work, we will evaluate it in two ways:

- About 50% of your grade for the assignment will be based on our judgement of the structure and organization of your code. To judge structure and organization, we will use the following dimensions:
  - *Documentation* assesses whether your code is documented appropriately.

    We expect you to document each function such that someone else could use your code and reason about its result without having to see the code itself. In particular, every function must be documented with a contract, and the contract must mention each parameter by name.
  - *Algebraic laws* are also documentation, but they serve primarily as a design tool. Your algebraic laws must demonstrate your understanding of the forms of input: we expect one law per form of input. If any form of input requires more than one law, we expect the laws to be disambiguated with mutually exclusive side conditions.

    We expect the laws to be the foundation for your code: your code must begin by examining inputs to determine which algebraic law governs the given input.

    Write your algebraic laws before you start coding. In the first half of the course, this "design first" practice might save you forty or fifty hours.
  - *Unit testing* assesses whether your code is appropriately tested by some combination of `check-expect`, `check-assert`, and/or `check-error`. Appropriate testing exercises every form of input and every part of your code. We expect a test for every algebraic law.
  - *Form* assesses whether your code uses indentation, line breaks, and comments in a way that makes it easy for us to read.

    We expect you to use consistent indentation, to obey the offside rule described in the coding-style guidelines, and to limit the use of inline comments in the body of each function.

    We expect you to indent each block of unit tests by eight spaces.
  - *Naming* assesses your choice of names. To people who aspire to be great programmers, names matter deeply.

We give you a hand here by providing a template in which the names of top-level functions and their arguments are already chosen for you. For helper functions, you will choose your own names. Look at the general coding rubric and choose wisely.

  - *Structure* assesses the underlying structure of your solution, not just how its elements are documented, formatted, and named.

    We expect that your solutions will use recursion and function calls, not loops and assignments. Additionally, we expect the case structure of each function to follow that function's algebraic laws.

- About 30% of your grade for the assignment will be based on our judgement of the correctness of your code. We often look at code to see if it is correct, but our primary tool for assessing correctness is by testing. On a typical assignment, the correctness of your code would carry more weight, but this assignment is all about following the recommended software process and using algebraic laws effectively, so test results carry less weight.

The detailed criteria we will use to assess your code are found at http://www.cs.tufts.edu/comp/105/coding-rubric.html. Though things may be worded differently, most of these criteria are also applied in our introductory courses (CS 11, 15, and 40)—they make explicit what we mean by "good programming practice." But as you might imagine, there is a lot of information there—probably more than you can assimilate in one reading. The highlights are

- *Documentation*
  - Each function is documented with a contract that explains what the function returns, in terms of the parameters, which are mentioned by name. From documentation, it is easy to determine how each parameter affects the result.
  - The contract makes it possible to use the function without looking at the code in the body.
  - If not all inputs are permissible, the contract determines what inputs are and are not OK.
  - The contract appears consistent with the laws, code, and unit tests.
  - Each parameter is mentioned in its function's contract.
- *Form*
  - Code fits in 80 columns.
  - Code respects the offside rule.
  - Code contains no tab characters.
  - Indentation is consistent everywhere.
  - If a construct spans multiple lines, its closing parenthesis appears at the end of a line, possibly grouped with one or more other closing parentheses—never on a line by itself.
  - No code is commented out.
  - Solutions load and run without calling `print`
- *Naming*
  - Each function is named either with a noun describing the result it returns, or with a verb describing the action it does to its argument. (Or the function is a predicate and is named as suggested below.)
  - A function that is used as a predicate (for if or while) has a name that is formed by writing a property followed by a question mark. Examples might include `even?` or `prime?`. (Applies only if the language permits question marks in names.)
  - In a function definition, the name of each parameter is a noun saying what, in the world of ideas, the parameter represents.
  - Or the name of a parameter is the name of an entity in the problem statement, or a name from

    the underlying mathematics.
  - Or the name of a parameter is short and conventional. For example, a magnitude or count might be n or m. An index might be i, j, or k. A pointer might be p; a string might be s. A variable might be x; an expression might be e.
- *Laws*
  - Each left-hand sides applies the defined function to forms of input.
  - Laws are distinguished by distinct forms of input, or (only when necessary) by mutually exclusive side conditions.
  - Every permissible input is handled by exactly one law, and no forms of input are omitted.
  - Each law specifies a result that is consistent with the function's contract, and it does not say anything about inputs that are forbidden by the contract.
  - The right-hand side of a law refers only to those variables that appear on the left-hand side of that law.
  - Variables are used consistently between the left-hand side and the right-hand side of each law.
- *Structure*
  - Solutions are recursive, as requested in the assignment.
  - Any case analysis begins by identifying which law applies to the input.
  - There's only as much code as is needed to do the job.
  - In the body of a recursive function, the code that handles the base case(s) appears before any recursive calls.
  - The code of each function is so clear that, with the help of the function's contract, course staff can easily tell whether the code is correct or incorrect.
  - Expressions cannot easily be simplified.
- *Unit Testing*
  - Every test uses inputs whose behavior is constrained by contract.
  - There is a unit test for each form of input.
  - There is one unit test for each algebraic law (should overlap with input tests).
  - If a function returns a Boolean, its tests are written using `check-assert` and possibly `not`.
  - If a function returns a Boolean, then for each form of input that can return either true or false, there is both a truth test and a falsehood test.

How your photograph will be evaluated

If your photograph is clear, makes it easy to recognize you, and is not ridiculous in size, it will earn a grade of Very Good (the top grade). If you're not sure how to take a photograph that makes you easy to recognize, consult the table below. Aim for an "Exemplary" photograph (the left column), be willing to settle for "Satisfactory" (the middle column), and avoid "Must Improve" (the right column).[2]

[2]Yes, a student once sent me a photograph of two people.

	Exemplary	Satisfactory	Must Improve
Composition	• Head and shoulders fill 2/3 to 3/4 of the frame. • The shot is taken from a distance of 4 to 6 feet, and the camera is zoomed as needed to include just head and shoulders. • *Or*, the shot is taken from a distance of 4 to 6 feet, then cropped to include just head and shoulders. • The subject is looking at the camera with a normal look on the face. • The background around the subject's face is one color, and there are no visual features that would distract a viewer from the subject.	• Face fills the frame; shoulders not visible. • The subject is not looking at the camera, but there is a normal look on the face. • The background around the subject's face contains distracting visual features, but the distraction is minimal (or is mitigated by soft focus).	• Photo down to waist; full-body photo. • More than one person visible in photo. • Eyes are closed. • The camera is too close to the subject. (This will happen if you compose the shot by moving the camera toward the subject until the subject's head and shoulders fill the "viewfinder"; you'll get perspective distortion.) • The subject is making a strange face (eye rolls, etc) • The background around the subject's face contains distracting visual features, making it difficult for a viewer to identify the subject without concentrating.
Lighting	• The subject is illuminated by two or more light sources, such that one side of the subject's face is noticeably brighter than the other (about 2 to 1). • The main sources of light are soft and diffuse: overcast sky, indirect daylight, daylight reflected off a wall or building, and so on.	• The subject's face is lit evenly. • The subject's face is lit by ambient light, plus flash bounced off a ceiling or wall (possible only with a real camera)	• The background is significantly brighter than the subject. • There is light behind the subject aimed at the camera. • The sun is shining in the subject's face. • The subject is illuminated by a camera flash.
Focus	• The subject is in sharp focus, while the background is a little blurry (possibly only with a real camera or with *very* sophisticated software). • The subject's face is in sharp focus.	• Some part of the subject is in sharp focus, or something near the subject is in sharp focus. The subject's face is not in sharp focus but is still easy to recognize.	• The subject's face is out of focus.

	Exemplary	Satisfactory	Must Improve
File	• The uploaded image file is from 300KB to 1.2MB in size. • Resolution of the uploaded file is high enough that there's no pixelation. • The uploaded image is at least as tall as it is wide (portrait orientation)	• The uploaded image file is at most 2MB in size. • When shown at a few inches high, the uploaded image file is pixelated or has compression blur.	• The uploaded image file is more than 2MB in size. • At its natural resolution, the uploaded image file shows pixels or compression artifacts. • The uploaded image is wider than it is tall (landscape orientation)