COMP105 Assignment: An Imperative Core

Due Tuesday, September 15 at 11:59PM.

1.1. Getting Started

- To add the course binaries 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 with scp, ssh, git, and rsync.

- **IMPORTANT NOTE**: This assignment is due one minute before midnight on a class day. You may turn it in up to 24 hours after the due date, which will cost you one extension token. If you wish not to spend an extension token, then when midnight arrives submit whatever you have. We are very willing to give partial credit.

1.2. Programming in Impcore

These are finger exercises'' to get you into the swing of the LISP syntax and style of programming. You can start these exercises immediately after the first lecture. If you find it entertaining, you may write very efficient solutionsâ but do not feel compelled to do so. Do not share your solutions with anyone. We encourage you to discuss ideas, but **no one else may see your code**.

Do exercises 3 through 8 on pages 59â 60 of Ramsey's textbook. Place your solutions to problems 3 through 8 in a file called `solution.imp`.

You must use recursion. While loops will be disabled.

You can find an impcore interpreter 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. See its man page. Note that you can run the contents of a file through the interpreter by typing `impcore < file`. You can eliminate unwanted prompts by running `impcore -q < file`.

You must use one or more `check-expect` definitions to test each required function. You should briefly explain the purpose of each check-expect definition and explain why those present are sufficient.

You may find it useful to create additional testing code in a file mytests; you can then check your work by typing

```
cat solution.imp mytests | impcore -q
```

Don't include any such additional testing code in the `solution.imp` file you submit, just the required check-expect definitions. Your solutions **must be valid Impcore**; in particular, they must pass the following test:

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

without any error messages. 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). You may find it more convenient to keep solutions in separate files as you develop them. If so, we recommend that you do so and combine them in the end with cat. For example,

```
cat s2 s3 s4 s5 s6 s7 > solution.imp
```

Use helper functions where appropriate, but do not use global variables.

Below each function, *not* as part of that function's regular documentation, please put a comment that explains what inductive structure that function is imposing on the integers or the natural numbers. For example, I could write the `even?` function this way:

```
(define even? (n)
   (if (= n 0) 1
       (if (= n 1) 0
           (even? (- n 2)))))
   ;; Breaks down the natural numbers into three cases:
   ;;    0
   ;;    1
   ;;    n+2, where n is a natural number
```

The solutions you write for problems 3 through 8 should be in order in the file solution.imp (i.e. problem 3 first, problem 8 last) and each solution should be preceeded by a comment that looks like something like this:

```
;;
 ;; Problem N
 ;;
```

My solutions total 50â 60 lines of Impcore. If you have difficulty, find a TA who can work you through some similar problems.

1.3. How your work will be evaluated

A big part of this assignment is for you to be sure you understand how we expect your code to be structured and organized. There is some material about this on the details page on the course web site. When we get your work, we will evaluate it in two ways:

- About 60% of your grade will be based on our judgement of the structure and organization of your code. To judge structure and organization, we will use the following four dimensions:
  - *Documentation* assesses whether your code is documented appropriately.
  - *Form* assesses whether your code uses indentation, line breaks, and comments in a way that makes it easy for us to read.
  - *Naming* assesses your choice of names. (To people who aspire to be great programmers, names matter deeply.)
  - *Structure* assesses the underlying structure of your solution, not just how its elements are documented, formatted, and named.
- About 40% of your grade 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 relative to the other homeworks in 105, the problems on this assignment are very easy, so they carry less weight.

The detailed criteria we will use to assess your work are as follows:

	Exemplary	**Satisfactory**	**Must improve**
Documentation	• The contract of each function is clear from the function's name, the names of its parameters, and perhaps a one-line comment describing the result. • When names are not enough, each function is documented with a contract that explains	• A function's contract omits some parameters. • A function's documentation mentions every parameter, but does not specify a contract. • A function's documentation includes information that is redundant with the code, e.g., "this	• A function is not named after the thing it returns, and the function's documentation does not say what it returns. • A function's documentation includes a narrative description of what happens in the body of the function, instead of a contract that mentions only the parameters and

	what the function returns, in terms of the parameters, which are mentioned by name. • From the name of a function, the names of its parameters, and the accompanying documentation, it is easy to determine how each parameter affects the result. • The contract of each function is written without case analysis, or case analysis was unavoidable. • Documentation appears consistent with the code being described.	function has two parameters." • A function's contract omits some constraints on parameters, e.g., forgetting to say that the contract requires nonnegative parameters. • A function's contract includes a case analysis that could have been avoided, perhaps by letting some behavior go unspecified.	result. • A function's documentation neither specifies a contract nor mentions every parameter. • There are multiple functions that are not part of the specification of the problem, and from looking just at the names of the functions and the names of their parameters, it's hard for us to figure out what the functions do. • A function's contract includes a *redundant* case analysis. • Documentation appears inconsistent with the code being described.
Form	• All code respects the offside rule • Indentation is consistent everywhere. • In Impcore, 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. • No code is commented out. • Solution file contains no distracting test cases or print statements.	• The code contains one or two violations of the offside rule • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • Solution file may contain clearly marked test *functions*, but they are never executed. It's easy to read the code without having to look at the test functions.	• The code contains three or more violations of the offside rule • The code is not indented consistently. • The closing parenthesis of a multi-line construct is followed by more code (or by an open parenthesis) on the same line. • A closing parenthesis appears on a line by itself. • Solution file contains code that has been commented out. • Solution file contains test cases that are run when loaded. • When loaded, solution file prints test results.
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	• Functions' names contain appropriate nouns and verbs, but the names are more complex than needed to convey the function's meaning. • Functions' names contain some suitable nouns and verbs, but they don't convey enough information about what the function returns or does. • A function that is used as a predicate (for `if` or `while`) does not	• Function's names include verbs that are too generic, like "calculate", "process", "get", "find", or "check" • Auxiliary functions are given names that don't state their contracts, but that instead indicate a vague relationship with another function. Often such names are formed by combining the name of the other function with a suffix such as `aux`, `helper`, `1`, or even

	include `even?` or `prime?`. (Applies only if the language permits question marks in names.) • *Or*, the code defines no predicates. • 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`.	have a name that ends in a question mark. (Applies only if the language permits question marks in names.) • The name of a parameter is a noun phrase formed from multiple words. • Although the name of a parameter is not short and conventional, not an English noun, and not a name from the math or the problem, it is still recognizable---perhaps as an abbreviation or a compound of abbreviations.	_. • Course staff cannot identify the connection between a function's name and what it returns or what it does. • The name of a parameter is a compound phrase phrase which could be reduced to a single noun. • The name of some parameter is not recognizable---or at least, course staff cannot figure it out.
Structure	• 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. • There's only as much code as is needed to do the job. • In every case analysis, all cases are necessary. • In the body of a recursive function, the code that handles the base case(s) appears before any recursive calls. • Solutions are recursive, as requested in the assignment. • Expressions cannot be made any simpler by application of algebraic laws.	• Course staff have to work to tell whether the code is correct or incorrect. • There's somewhat more code than is needed to do the job. • In some case analyses, there are cases which are redundant (i.e., the situation is covered by other cases which are also present in the code). • Code for one or more base cases appears after a recursive call.	• From reading the code, course staff cannot tell whether it is correct or incorrect. • From reading the code, course staff cannot easily tell what it is doing. • There's about twice as much code as is needed to do the job. • A significant fraction of the case analyses in the code, maybe a third, are redundant. • Code uses `while` or `set` (**serious fault**) • Code can be simplified by applying algebraic laws. For example, the code says `(+ x 0)`, but it could say just `x`.
Correctness	• Impcore functions test correct with no faults.	• Testing Impcore solutions identifies a few faults.	• Testing Impcore code shows a preponderance of faults.

	• Or, under test, Impcore functions have only tiny faults, typically arising from problems with arithmetic overflow or from some confusion about exactly what numbers are prime.	• Or, testing Impcore solutions identifies a single fault that shows a lack of understanding.	• Impcore code fails because the names of helper functions are spelled differently in different places (**serious fault**). • When we attempt to load Impcore code, there are errors (**No Credit**).

Exemplary work typically earns a Very Good grade; if exemplary work truly excels, it may earn an Excellent grade. Satisfactory work typically earns a Good grade. Work that must improve typically earns a Fair grade, but work that has a serious fault may earn a Poor grade, and as noted in the table, a very serious fault may result in a grade of No Credit.

1.4. Difficulty Alert

This assignment is three or four times easier than a typical COMP 105 assignment. Its role is to get you acclimated and to help you start thinking systematically about how recursion works. Later assignments get much harder and more time-consuming, so don't use this one to gauge the difficulty of the course.

1.5. How to submit your work

Before submitting your code, test it. We do not provide any tests; you must write your own.

To submit, change into the directory containing your code and run

```
submit105-impcore
```

to submit your work. In addition to file `solution.imp`, please also include a file called README. Use your README file to

- Tell us how to pronounce your name, e.g. “Sam-yoo-el G-eye-er”.
- Tell us how well you think you did on each of the five dimensions: Documentation, Form, Naming, Structure, and Correctness
- Tell us how long it took you to complete the assignment

(If you wish to use PDF, then please submit README.pdf instead of README.)

Created with Madoko.net.