Exercise 1F-2. Language Design [5 points]. Comment on some aspect from Hoare's *Hints On Programming Language Design* that relates to your programming experience. Provide additional evidence in favor of one his points and against one of his points. Do not exceed three paragraphs. Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter. Readers should be able to identify your main claim, the arguments you are bringing to bear, and your conclusion.

Solution: Hoare's description of convincing a programmer to switch languages as a nearly impossible task exactly mirrors my own gravitation towards familiar languages. While various classes have enabled me to work with OCaml, and internship experiences have prompted me to pick up Python, I always default to my mother tongue (the first programming language I learned), C++. Hoare goes on to describe how programmers have adapted to the deficits of their current programming language, just as I have accepted the extra lines of code needed to accomplish the same task in C++ rather than Python.

Moreover, Hoare's reverence for proper documentation should be engrained in CS coursework. Rather than just learning about data structures and algorithms and the intracasies of a language, students should be taught best practices for developing code that can be easily parsed by others –and their future selves. At the introductory level I never adopted the habit of properly documenting my code; in upper level coursework on Operating Systems, my group appended minimal comments to our code, after our entire project was working, directly contradciting Hoare's guidance on documenting along the way. Our group thought our code was so clean and readable it didn't need comments, everything seemed obvious to us. A year later we bit our tongues, as my group members went on to become instructors for the class. Attempting to review the projects they referred back to our code, they found a fully functioning product with minimal and useless comments. It took more time to re-derive our previous intentions and naming schemes, than it would have to properly document everything the first time.

While I agree with Hoare's point on the importance of documentation and properly commenting one's code, I disagree with his strong preference for low level style commenting. Hoare describes how in low level programs, comments come after an instruction, are started with a special character, and end with the end of a line. He praises how this style of commenting is less susceptible to errors from other techniques like using "/* */" to denote the beginning and end of comments. I argue instead, that the low level style of commenting is unfit for longer comments. In both my classes and work expereience, I've been required to keep all lines of code, including comments, to 80 characters maximum. This could be unfeasible when using the low level style Hoare prefers, and attempting to write a more in depth comment. The 80 character maximum allows programmers to easily peruse code without scrolling from left to right, and as Hoare asserts, readability is far more important than write ability.

Exercise 1F-3. Simple Operational Semantics [3 points]. Consider the IMP language discussed in class, with the Aexp sub-language extended with a division operator. Explain what changes must be made to the operational semantics (big-step only). Write out formally any new rules of inference you introduce.

Solution: Since IMP only supports integers, we introduce integer division. If we divide by zero we reach an error, else we proceed with division as normal and truncate any fractional part of the remainder. Since we don't have a representation of an error state, below we map division by zero to false. If there was some sort of error state, we would replace "false" with its representation.

Divide by zero: $$\frac{\langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \Downarrow false}$$

Integer division : $$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad \langle \neg(e_2 = 0), \sigma\rangle \Downarrow True}{\langle e_1/e_2, \sigma\rangle \Downarrow \lfloor n_1/n_2 \rfloor}$$

Exercise 1F-4. Language Feature Design, Large Step [10 points]. Consider the IMP language with a new command construct "`let` $x = e$ `in` c". The informal semantics of this construct is that the Aexp

Peer Review ID: 302543792 — enter this when you fill out your peer evaluation via gradescope

Questions assigned to the following page: 4 and 5

e is evaluated and then a new local variable x is created with lexical scope c and initialized with the result of evaluating e. Then the command c is evaluated. We also extend IMP with a new command "`print` e" which evaluates the Aexp e and "displays the result" in some un-modeled manner but is otherwise similar to `skip`.

We expect (the curly braces are syntactic sugar):

```
x := 1 ;
y := 2 ;
{ let x = 3 in
  print x ;
  print y ;
  x := 4 ;
  y := 5
} ;
print x ;
print y
```

to display "3 2 1 5".

Extend the natural-style operational semantics judgment $\langle c, \sigma \rangle \Downarrow \sigma'$ with one new rule for dealing with the `let` command. Pay careful attention to the scope of the newly declared variable and to changes to other variables.

Solution:
$$\frac{\langle x, \sigma \rangle \Downarrow x_0 \quad \langle e, \sigma \rangle \Downarrow n \quad \langle c, \sigma[x := n] \rangle \Downarrow \sigma'}{\langle let\ x = e\ in\ c, \sigma \rangle \Downarrow \sigma'[x := x_0]}$$

Exercise 1F-5. Language Feature Design, Small Step [10 points]. Extend the set of redexes, contexts and reduction rules for the contextual-style operational semantics that we discussed in class to account for the `let` command introduced above.

Solution: We extend the redexes to also include:

$$r ::= let\ y\ := n\ in\ c$$

Then we extend the reduction rules to include:

$$\langle let\ y\ := n\ in\ c, \sigma \rangle \rightarrow \langle y := n; c; y := \sigma[y], \sigma \rangle$$

Exercise 1C. Language Feature Design, Coding. Download the Homework 1 code pack from the course web page. Modify `hw1.ml` so that it implements a complete interpreter for IMP (including `let` and `print`). Base your interpreter on IMP's large-step operational semantics. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

Solution: See autograder

Peer Review ID: 302543792 — enter this when you fill out your peer evaluation via gradescope

No questions assigned to the following page.

Submission. Turn in the formal component of the assignment (1F-1 through 1F-5) as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment (1C) via the `autograder.io` website.

Peer Review ID: 302543792 — enter this when you fill out your peer evaluation via gradescope