All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

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.

I absolutely agree to the importance of type declaration. When taking EECS 445 (Introduction to Machine Learning) we did our projects in Python. In that semester, I struggled a lot with the dimensions of matrices. For most of the time I had to explicitly print these matrices out before realizing what was probably going wrong. What's more, Python couldn't tell anything wrong about type before executing, so many times my program studied the training data for half an hour or so only to find out I input a wrong matrix in a function. On the other hand, OCaml is nice. Yesterday when I compiled my program, I got a warning indicating that some expressions should have a different type. I checked the line the warning pointed to, and I found an implementation error that was not exposed by my test.

I don't quite agree with Hoare's stereotype that "it is almost impossible to persuade him to change to a new one". Of course, a new programming language need to be user-friendly (simple, fast-translated, etc.) enough so that programmers would be willing to try it, but people may also give up their most familiar languages because of some special tasks. For example, the programming language I familiar to most is C++. Once I encountered a problem in an interview, which asked me to parse several user inputs in several patterns and store necessary information to the database. I switched to Python immediately, even if I was not so familiar to Python semantics. Given that I was able to use Python dictionary and simple functions like string.split, it just seemed too complicated to parse strings with getc and define vector or unordered_maps with different types like int and string. I believe different languages are used to solve different problems, at least before some "perfect" programming languages exist.

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.

We need to add several rules to support the division operator. In order to keep the type of all IMP Arithmetic expressions integer, we need to first define a new literal `bad_int` with type `int`. We define our new Aexp as follows:

$$\begin{array}{rcll} e & ::= & \texttt{n} & \text{for } n \in \mathbb{Z} \\ & | & \texttt{x} & \text{for } x \in \mathrm{L} \\ & | & \texttt{bad\_int} & \text{of integer type} \\ & | & \texttt{e}_1 + \texttt{e}_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ & | & \texttt{e}_1 - \texttt{e}_2 & \text{for } e_1, e_2 \in \text{Aexp} \\ & | & \texttt{e}_1 * \texttt{e}_2 & \text{for } e_1, e_2 \in \text{Aexp} \end{array}$$

Then we can define our rules for division operation:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad \langle n_2 = 0, \sigma\rangle \Downarrow \texttt{false} \quad \langle n_1 * n_2 \leq 0, \sigma\rangle \Downarrow \texttt{true}}{\langle e_1/e_2, \sigma\rangle \Downarrow \lceil n1/n2 \rceil}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad \langle n_2 = 0, \sigma\rangle \Downarrow \texttt{false} \quad \langle n_1 * n_2 \leq 0, \sigma\rangle \Downarrow \texttt{false}}{\langle e_1/e_2, \sigma\rangle \Downarrow \lfloor n1/n2 \rfloor}$$

However, because of the newly added literal `bad_int`, rules for other operations need to be updated as well. For example, for literal evaluation we need to add:

$$\frac{}{\langle \texttt{bad\_int}, \sigma\rangle \Downarrow \texttt{bad\_int}}$$

For addition we need to add the following two rules:

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 + e_2, \sigma\rangle \Downarrow \texttt{bad\_int}} \qquad \frac{\langle e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 + e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}$$

Similarly, for subtraction and multiplication we also need to add two rules, respectively.

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 - e_2, \sigma\rangle \Downarrow \texttt{bad\_int}} \qquad \frac{\langle e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 - e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 * e_2, \sigma\rangle \Downarrow \texttt{bad\_int}} \qquad \frac{\langle e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}{\langle e_1 * e_2, \sigma\rangle \Downarrow \texttt{bad\_int}}$$

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 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.

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

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.

Define $[e/x]c$ to be: substitute each variable x in command c with expression e. Then we can define the following contexts, redexes and local reduction rules for the `let` command:

$$\texttt{H} ::= \ldots \mid \texttt{let x} = \texttt{H in } \texttt{c}_0$$
$$\texttt{r} ::= \ldots \mid \texttt{let x} = \texttt{n in } \texttt{c}_0$$

$$\langle \texttt{let } x = n \texttt{ in } c_0, \sigma \rangle \to \langle [n/x]c_0, \sigma \rangle$$

Let $c =$ "$[n/x]c_0$" we will have:

$$\begin{array}{rcl} \texttt{c} & ::= & \texttt{skip} \\ & \mid & x' := e' \\ & \mid & c_1; c_2 \\ & \mid & \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \\ & \mid & \texttt{while } b \texttt{ do } c_1 \\ & \mid & \texttt{let } x' = e' \texttt{ in } c_1 \end{array}$$

Following the existing small step rules, we can finally get

$$\ldots \to \langle \texttt{skip}; \sigma' \rangle \to \sigma'$$

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).

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.

1 HW1 (select all pages: your first page has your name and bookkeeping, and all others are anonymous))

- **0 pts** Correct