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.

My Answer:
Evidence in favor of one his points:

Point: (page 24) "Sane languages ... prefer the dangerous tolerance of machine code ... disadvantages: (1) The result will often be "nearly" right, so that the programmer has less warning of his error."

Evidence: Python doesn't require us to set type for each variables, which sounds convenient. But it is sometimes hard for debugging, because you have to manually check the type for each command.

Evidence against one of his points:

Point: (page 4 & 5) "In fact, a programmer's need for an understanding of his language is so great, that it is almost impossible to persuade him to change to a new one."

Evidence: As far as I can tell, nowadays, programmers are happy switching programming languages for new features. For example, `Taichi` gains lots of programmers quickly as a new powerful tool for computer graphics.

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.

My Answer:
Unlike multiplication, we have to make sure that the divisor is not zero. We need to add a state $\sigma_{\texttt{err}}$ to show if the program should stop and report the error.

$$\frac{< e_2, \sigma > \Downarrow 0}{< e_1/e_2, \sigma > \Downarrow < 0, \sigma_{\texttt{err}} >}$$

$$\frac{< e_1, \sigma > \Downarrow n_1 \quad < e_2, \sigma > \Downarrow n_2 (n_2 ! = 0)}{< e_1/e_2, \sigma > \Downarrow \lfloor n_1/n_2 \rfloor}$$

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 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.

My Answer:
To have the required features, we have to store the original x to a temp variable x_t, set $x := e$, run c and then set x back to its original value. Steps can also be describe as $x_t := x; x := e; c; x := x_t$. We can then write the following large step semantics judgment.

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

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.

My Answer:
Context:

$$\begin{aligned} H &::= \cdots \\ &\mid \texttt{let}\ H\ \texttt{in}\ c \end{aligned}$$

Redex:

$$\begin{aligned} r &::= \cdots \\ &\mid \texttt{let}\ x = n\ \texttt{in}\ c \end{aligned}$$

Local Reduction rule:

$$< \texttt{let}\ x = n\ \texttt{in}\ c, \sigma > \to < x := n; c; x := \sigma(x), \sigma >$$

Notice that $x = n$ is a terminal programs.

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).

My Answer:
Submitted.

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