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.

Answer:

I started learning programming using C and C++ and continued with Java and Python. Consequently, I was immersed in the statement style programming for awhile. Hoare discusses why arithmetic expressions enjoyed much success first through FORTRAN then again in ALGOL 60, which added conditional expressions. He makes the excellent point that expressions are an advancement because they embody our underlying approach to problem solving: decomposing a problem into subproblems with well-defined interfaces. After learning languages like Haskell and Scala both of which favor the expression style over statements, I noticed expressions decrease the friction in translating the problem and solution from my mind to the computer language. One concrete example here is how `map` and `filter` can be chained to modify data in a very "natural" way. Another example is how easy it is to define and operate on recursive data structures like graphs and trees.

On the other hand, Hoare discusses the need for efficient object code. He makes the point that language designers should continue to focus on the artifacts produced by their compilers. He gives five reasons why the language designer should continue to care, and I generally agree with these reasons. However, he continues to argue that optimizing compilers cannot regain machine efficiency to support his claim. He gives FORTRAN as an example of a success story for compiler optimized languages, yet still maintains optimizations have too many disadvantages. While the three first disadvantages he lists are technically true, they can still be overcome to an impressive extent, as evident in modern day compilers like LLVM. LLVM is fast, mostly correct, reliable and can perform increasingly complex optimizations. Hoare's point here seems rather shortsighted in retrospect. Finally, the last point he makes against optimizing compilers is rather subjective and fails to see the opportunity for robust abstractions that don't need to be broken even for the sake of performance.

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.

Answer:

The abstract syntax for `Aexp` needs to include a new production rule for division. The operational semantics now mainly requires two changes: a description of what should happen when the denominator is not 0, and what should happen when the denominator is 0. In the first case, we could simply divide the numbers in the mathematical sense, but then we risk producing a real number instead of an integer. The solution is to perform integer division or *mod*. In the second case we should describe the "undefined" behavior that results from performing $n_1 \, mod \, n_2$ where $n_2 = 0$. This leads to the third change required, which is defining the semantics of this new undefined term when combined with other `Aexp` expressions. I'm using the shorthand $\{/, *, +, -\}$ to express the rule applies for all the operators in the set.

$$\text{UNDEF}\ \frac{}{\langle \bot, \sigma\rangle \Downarrow \bot} \qquad \text{DIV-ZERO}\ \frac{\langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \Downarrow \bot}$$

$$\textsc{Div}\ \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1/e_2, \sigma\rangle \Downarrow n_1 \ mod \ n_2}$$

$$\bot\text{-EXP}\ \frac{\langle e_1, \sigma\rangle \Downarrow \bot \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 \ \{/, *, +, -\} \ e_2, \sigma\rangle \Downarrow \bot} \qquad \bot\text{-EXP}\ \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow \bot}{\langle e_1 \ \{/, *, +, -\} \ e_2, \sigma\rangle \Downarrow \bot}$$

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.

Answer: The new rule is the following:

$$\text{LET}\ \frac{\langle x := e, \sigma\rangle \Downarrow \sigma' \qquad \langle c, \sigma'\rangle \Downarrow \sigma'' \qquad \langle x, \sigma\rangle \Downarrow n \qquad \langle x := n, \sigma''\rangle \Downarrow \sigma'''}{\langle let \ x := e \ in \ c, \sigma\rangle \Downarrow \sigma'''}$$

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.

Answer:

The redex grammar needs to include one more rule:

```
r ::= | let x = n in c
```

The corresponding reduction rule is:

$$\langle let\ x := n\ in\ c, \sigma\rangle \rightarrow \langle x := n; c; x := \sigma(x), \sigma\rangle$$

The contexts' grammar needs to include:

```
H ::= | let H in c
```

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