CSCI 5535, ECEN 5533: Homework 1

Due September 1

Exercise 1: Bookkeeping. Indicate in a sentence or two how much time you spent on this homework, how difficult you found it subjectively, and what you found to be the hardest part. Also, indicate one thing you like about the class so far and one thing you would change about it. Any non-empty answer will receive full credit.

Exercise 2: Language Design. On the google group, 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. I should be able to identify your main claim, the arguments you are bringing to bear, and your conclusion.

Exercise 3: Simple Operational Semantics. Consider the IMP language discussed in class, with the Aexp sub-language extended with an integer division operator.

\[
e ::= \ldots \\
| e_1 / e_2
\]

Explain what changes must be made to the operational semantics (big-step only). Write out formally any new rules of inference you introduce.

Exercise 4: Language Feature Design. 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 to “parenthesize” commands):

1
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 big-step operational semantics judgment $< c, \sigma > \Downarrow \sigma'$ 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.

- 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 so that it contains a “tricky” IMP command that can be parsed by our IMP test harness (e.g.,

  ./imp < example.imp

should not yield a parse error).

- Rename hw1.ml to your_identkey-hw1.ml and rename example.imp to your_identkey-example.imp for submission. Do not modify any other files. Your submission’s grade will be based on how many of the submitted example.imps it interprets correctly (in a manner just like the “make test” trials). If your submitted example.imp breaks the greatest number of interpreters (and more than 0!), you will receive extra credit. If there is a tie, all those in the tie will receive the extra credit.