Advanced Programming Languages
Homework Assignment 1

Wes Weimer

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. Any non-empty answer will receive full credit. If you are working in pairs, each person must respond to this prompt. If your uniquename is `mst3k`, upload a single ZIP file containing `mst3k.pdf` (your answers to the written parts of this assignment) as well as `mst3k-hw1.ml` and `mst3k-example-imp-command` (see below).

Exercise 2: Language Design. 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. If you are working in pairs, each person must complete this exercise separately.

Exercise 3: Simple Operational Semantics. 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.

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):

```
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 $< c, \sigma > \Downarrow \sigma'$ 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.

- 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.

- 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" IMP command that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

- Rename `hw1.ml` to `your_uniquename-hw1.ml` and rename `example-imp-command` to `your_uniquename-example-imp-command` and submit them. Do not modify any other files (such as the expected output of your example-imp-command). Your submission's grade will be based on how many of the submitted `example-imp-command`s it interprets correctly (in a manner just like the "`make test`" trials). If your submitted `example-imp-command` breaks the greatest number of interpreters (and more than 0!), you will receive extra credit. If there is a tie all tiers will receive the extra credit.