Advance Programming Languages — Assignment 2
CS 6110 Spring 2010 — Due Friday, 26 February 2010

In this assignment, you may work alone or in groups of two or three. If you work in a group, please form a group in CMS as before. Only one person needs to submit the assignment.

1. Observational Equivalence
   Prove Lemma 1 and Theorem 2 from Lecture 4. Prove Theorem 2 for CBV only.

2. For Loops
   We would like to add `for` loops to the IMP language. The syntax is

   $$c \;::=\; \cdots \mid \mathsf{for}\ a\ \mathsf{do}\ c$$

   where a is an arithmetic expression and c is a command. Intuitively, the for loop works as follows. Upon entering the loop, the expression a is evaluated in the current state, yielding an integer n. If $n \leq 0$, the loop body is not executed at all. If $n > 0$, then the loop body c is executed n times. No action in the body of the loop, such as assigning to a variable, can change the number of times the loop is executed, nor does executing the body alone change the value of any variable except by explicit assignment.

   Write (a) small-step and (b) big-step rules for this new command in the style of Lecture 6.

3. Gödel's Mistake
   Consider the language IMP with for loops but without while loops. Despite the fact that we can write many useful programs with this language—it can compute all *primitive recursive functions*—the language is not universal. Show that it is not universal by demonstrating that all programs terminate. Use well-founded induction. Say explicitly what your well-founded relation is, and argue that it is indeed well-founded.

4. Combinatory Logic
   A *combinator* is a closed λ-term of the form $\lambda x_1, \ldots, x_n.\, t$, where t is a t is a term built from $x_1, \ldots, x_n$ and the application operator alone. For example,

   $$\mathsf{S} \triangleq \lambda xyz.\, xz(yz) \qquad \mathsf{K} \triangleq \lambda xy.\, x \qquad \mathsf{I} \triangleq \lambda x.\, x$$

   Curry experimented with combinators as a way to eliminate the need for λ-abstraction and substitution in the λ-calculus. Instead of β-reduction, he axiomatized their behavior in terms of rewrite rules. For example,

   $$\mathsf{S}XYZ \;\to\; XZ(YZ) \qquad \mathsf{K}XY \;\to\; X \qquad \mathsf{I}X \;\to\; X.$$

   He proved a remarkable fact: *any* closed λ-term can be simulated by some combination of S, K, and I. For example, the B combinator $\mathsf{B}XY \to Y$ (that is, $\lambda xy.\, y$) is equivalent to SK:

   $$\mathsf{SK}XY \;\to\; \mathsf{K}Y(XY) \;\to\; Y.$$

   (a) Show that the I combinator is redundant by constructing a term from S and K with the same behavior.

   (b) Build a term from S, K and I with no normal form. (*Hint*: Ω.)

5. Implementing IMP

The archive `imp.zip` contains a partial implementation of the IMP imperative language. We have provided all the boring infrastructure (lexer, parser, and read-eval-print loop) which should compile and run right out of the box. We have also supplied some sample IMP programs.

(a) Right now you can load `fact.txt` and `fib.txt`, but you cannot run them, because the evaluation functions in `eval.ml` are not implemented. Please implement them. Use the big-step operational rules as described in Lecture 6. There are a few extra features besides those described in the lecture, namely reading from the standard input, writing to the standard output, and some extra arithmetic operations. Once you have done this, you should be able to run `fact.txt` and `fib.txt`.

```
IMP interactive interpreter version 2010.0
>> help
Available commands are:
load <file>, list, run, help, quit
>> load etc/fib.txt
>> run
? 10
89
>> list
n := input;
f0 := 1;
f1 := 1;
k := 2;
while (k <= n) {
  if (k % 2 = 0) f0 := f0 + f1 else f1 := f0 + f1;
  k := k + 1;
}
if (n % 2 = 0) print f0 else print f1;
>> load etc/fact.txt
>> run
? 5
120
>> list
n := input;
f := 1;
while (n > 1) {
  f := f * n;
  n := n - 1;
}
print f;
>> quit
bye
```

(b) Add the `for` loop for which you wrote the SOS rules in Exercise 2. Use the big-step rules. You will also have to add this to the lexer and parser, but you can use the `while` statement for a model, since the concrete syntax is very similar; see the files `fact2.txt` and `fib2.txt` for examples. Modify the specification files `lexer.mll` and `parser.mly`, and use OCamllex and OCamlyacc to build the lexer and parser. See the supplied Makefile for the appropriate incantations.