Homework 5: Operational Semantics

Due:

This is an individual homework. When you write λProlog programs, follow the conventions of the previous homework for them.

1. (10 points) Using the computation semantics for the lambda calculus given in lecture, or from the directory

   `~leavens/classes/cs541/lectures/operational-semantics/lambda-in-lambda-prolog`

   show the detailed proofs for reducing the following terms to normal form, if such a normal form exists. (Hint: a good way to do these is to use a text editor. Also consider the order of steps carefully for the second of these.) Don't use λProlog to do these, as you will have to do something like this by hand on the final exam.

   (a) $(\lambda x\,.\,(y\ x))$

   (b) $((\lambda x\,.\,((\lambda y\,.\,x)\ z))\ ((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x))))$

   (c) $((\lambda k\,.\,(((\lambda r\,.\,(\lambda k\,.\,(k\ z)))\ (\lambda i\,.\,n))\ k))\ (\lambda v\,.\,v))$

2. (25 points) Write (either by hand, or using λProlog), an evaluation (i.e., big step) semantics for the lambda calculus. Illustrate how it works by showing proofs (by hand) for the examples in the previous problem. (Hint: as in our computational semantics, use an abstract syntax that doesn't allow any unneeded parentheses if you like; that will make the proofs easier.)

3. (15 points) Write out an operational semantics of the call-by-value lambda calculus. You can write either an evaluation (big step) or a computation semantics (little step) semantics. However, the point of the problem is to change whatever rules are needed to make the lambda functions evaluate their arguments to normal form before doing any application. For example, there should be no value given by your semantics for the term $((\lambda x\,.\,\lambda y\,.\,x)z)((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x)))$.

4. (20 points, extra credit only; semantics, programming) Write out a variant of the lambda calculus, either an evaluation (big step) or a computation semantics (little step) for a dynamically scoped, call-by-value lambda calculus. (This is essentially old versions of LISP.)

5. (15 points) For each of the following lists of programming systems, state what version of the lambda calculus most resembles those in each list: the standard (call by name), the call by value, or dynamic scoping (see the previous problem) version? Briefly explain each answer.

   (a) macros as in a macro assembly language, or the C pre-processor (`#define`),

   (b) the programming language Standard ML (or Scheme),

   (c) Pascal, Basic, FORTRAN, or COBOL.

6. (15 points) Add to either the computation or evaluation semantics of the lambda calculus the nondeterministic operator called (by M. Broy) "angelic choice". The angelic choice operator $\bigtriangledown$ is written as an infix operator; that is, the new term in the abstract syntax has the form $E_1\ \bigtriangledown\ E_2$, where E_1 and E_2 are terms.

   Informally, the meaning of an angelic choice between two terms, $E_1 \bigtriangledown E_2$, is a nondeterministic choice between the meaning of either E_1 or of E_2; however, if only one of the two terms has a normal form, then the meaning is the meaning of the term that has the normal form. For example, the term $x\ \bigtriangledown\ y$, has both x and y as possible results (meanings). The only possible result of the term $x\ \bigtriangledown\ ((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x)))$, however, is x. (The same holds for the term $((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x)))\ \bigtriangledown\ x$.)

The operator is angelic because an expression of the form $E_1 \bigtriangledown E_2$ will always halt (have a normal form) if either E_1 or E_2 always has a normal form. "Operationally", you may think of the angelic choice operator as reducing both expressions in parallel and returning the first normal form. However, you have to specify it using strictly a computation or an evaluation semantics (hint, think about which one would be best!). You can just write down the extra rule or rules for the semantics; or if no such rule exists, say why no such rule exists.

7. (40 points; extra credit only) Consider the statements in a language such as C, Pascal, or Ada, but take out unstructured jumps, such as the `goto`. Suppose you are given an operational semantics for the expressions in the language, which is written using environments and stores as described in class. Write a computation (little step) semantics for these non-jump statements in either C, Pascal, or Ada.

8. (50 points; extra credit) Adapt your semantics from the previous problem to handle unstructured jumps.