Homework 8: Operational Semantics

Due: April 25
This homework is to be done individually.
This homework is about operational semantics. In it you will study more about the lambda calculus, something about parallelism, and more about mini-cecil.
For all λProlog programs, you must run your code with the elp interpreter. You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing.
Unless we specifically request, your code does not have to run "backwards". (But you may want to see which ones can do that.) You can use cut (`!`) and `not` in your code.

1. (10 points) Using the computation semantics for the lambda calculus given in lecture (or in my course directory file), with the rule [extras] that we added in class, 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.)

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

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

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 two examples in the previous problem. (You can 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, extra credit only; applications, programming) For each of the following programming systems, state what version of the lambda calculus most resembles it: 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 the meaning of either E_1 or of E_2, but 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. The only possible result of the term $x \ \triangledown \ ((\lambda x \,.\, (x\ x))(\lambda x \,.\, (x\ x)))$, however, is x. (The same holds for the term $((\lambda x \,.\, (x\ x))(\lambda x \,.\, (x\ x))) \ \triangledown \ x$.)

The operator is angelic because an expression of the form $E_1 \ \triangledown \ E_2$ 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. (25 points) As noted in class, in Cecil, one cannot assign to the formal parameters of a method. Modify the semantics of mini-Cecil as found in the file

   `~leavens/classes/cs541/lectures/operational-semantics/mini_cecil_semantics.mod`

   (and other imported modules in that directory) so that the semantics reflects this restriction. That is, change the calling mechanism for methods in the semantics to be call by constant. Hand in any code you modified, along with test cases.

8. (40 points) In Cecil, there is no if-then-else statement. Instead, like Smalltalk, Cecil provides a way to make closures and like Smalltalk provides primitive classes `True` and `False` that implement if-then-else expressions. Since doing this right relies on getting method lookup implemented, we won't go that far in this problem. Instead your task is to modify the mini-Cecil operational semantics, to:

   (a) add abstract syntax for closure expressions (anonymous, non-recursive procedures, i.e., λ), to mini-Cecil,

   (b) add semantics for closure expressions.

   Your modified semantics should be able to work with curried procedures! Hand in any code you modified, along with test cases.

9. (50 points, extra credit only; semantics, programming) Continue the previous problem by adding abstract syntax to mini-Cecil as in problems 5 and 6 of the previous homework. Make each object keep track of its class. Make method lookup work correctly by choosing the unique most specific method for each message send and giving an error otherwise. Change the built-in algebra for mini-Cecil so that it has a type `Bool` and two classes `True` and `False` as in Smalltalk, and take out the `if` statement. Show how your semantics works. (If you wish, this might be developed into a paper to publish.)