Homework 0: Operational Semantics in λProlog

Due: February 1, 2000

In this homework, you will learn some basics of operational semantics. Logic programming is the essence of the operational semantic techniques we use, so it is useful to try at least once doing this in a language like λProlog.

For all λProlog programs, you must run your code with an interpreter (Terzo is preferred), and 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 λProlog code does not have to run "backwards". (But you may want to see which ones can do that.) In the examples you will see "no more solutions" or "yes" printed; your program does not have to have exactly this output.

You can use cut (`!`) and `not` in your code.

1. Using the computation semantics for the lambda calculus given in lecture (or in my course directory file), 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. Write, using λProlog, an evaluation (i.e., big step) semantics for the lambda calculus. You can use the following as the abstract syntax of lambda calculus terms.

```
module abstract_syntax.   % abstract syntax of the lambda calculus

kind term type.

type var string -> term.
type ap term -> term -> term.
type lambda string -> term -> term.
```

   This, and other useful files are found in the following URL. (Copy and adapt these for your solution.)

   `http://www.cs.iastate.edu/~leavens/ComS641/lectures/operational-sematics/lambda-in-lambda-prolog/`

   Illustrate how your semantics works by showing queries for the two examples in the previous problem, among others.

3. Write out the changed rules for the operational semantics of the call-by-value lambda calculus. You can write either an evaluation (big step) or a computation semantics (little step) semantics. Be sure that you say which of these you are doing!

   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. 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. 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 Haskell,

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

6. 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 $x \bigtriangledown ((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x)))$, 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. If you do a semantics, be sure to say which (big step or little step) you are doing.