Homework 5: λProlog, Logic Programming, and Operational Semantics

Due:

In this homework, you will learn basic skills in logic programming and operational semantics. One reason logic programming is important is because it resembles formal specifications; it is useful to know how these work, even if one does not specify formally. Some of the problems are geared towards this end. One of the more complex tasks in specification is precisely describing a programming language; logic programming is the essence of the operational semantic techniques we use. The small operational semantics problems will give you some experience with these techniques.

Since the purpose of this homework is to ensure basic skills in logic programming and operational semantics, this is an individual homework. That is, for this homework, you are to do the work on your own, not in groups.

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 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 Literature

1. (suggested practice) Read a tutorial on the concepts of logic programming. For example, chapter 14 in Watt's *Programming Language Concepts and Paradigms* is a (very brief) introduction. You might also look in the first few chapters of Sterling and Shapiro's *The Art of Prolog*, which is on reserve at the library. Then you should read the first 5 chapters of Dale Miller's monograph as minimal background on λProlog.

2. (varying amounts of points; extra credit) If you find any errors in Dale Miller's draft, please pass them along to me, and I'll send a batch of them to Dale. To encourage this, I'll give you extra credit points for suggestions and errors you find on the following scale. spelling or punctuation errors (1pt) confusing points, with corrections (2-10 points each) Please send your suggestions or corrections in the following format:

   ```
   p. 5, line 7, change "foo" to "bar"

   p. 5, line -3, the paragraph starting here isn't clear.
   Can it use an example?
   ```

   where negative line numbers refer to counting from the bottom of the page.

3. (40 points; extra credit) Read an article in a journal or conference proceedings that is related to logic programming languages (such as λProlog) from the "Introduction to the Literature" handout. Again please find an article that is either a survey article or one that describes research in programming languages. Don't report on a chapter from a book, or an article on software engineering or hardware. See Gary if you have questions about the suitability of an article. Write a short (1 or 2 page) review of the article.

   If you read a survey article, your review should clearly state: (a) the subarea surveyed and the major problems covered, (b) the main points (ideas) in the work surveyed, and (c) what you learned from it.

If you read a research article, your review should clearly state: (a) what the problem is that the article claims to solve, (b) the main points including any claimed contribution vs. related work, (c) your opinion as to how good the solution to the problem was, and (d) what you learned from it.

Please, be nice to the class by not checking out a proceedings or journal volume from the library, but only making a copy of a relevant article, or reading it in the library.

2 Lists

The next few problems deal with lists in λProlog. Try typing `describe '::'.` to the interpreter for a description of the list building operation. You'll often have to parenthesize lists, for example `(1::nil)`, so it's safer to always parenthesize them. See also the file `/usr/unsup/elp/elp/lib/lists.mod` for some examples of operations on lists. The operations in that file are not built-in to λProlog, but you can use them by putting `import lists.` in your module.

Specify the following relations in λProlog.

4. (5 points) The relation `find_last` such that

   ```
   type find_last  (list T) -> T -> o.
   ```

   and `(find_last L X)` holds iff `X` is the last item in the list `L`. For example.

   ```
   ?- find_last nil 3.
   no
   ?- find_last (1::2::3::nil) 3.
   solved
   ;
   no more solutions
   ?- find_last (1::2::3::nil) Y.

   Y = 3.
   ;
   no more solutions
   ?- find_last L 3.
   find_last L 3.

   L = 3 :: nil.
   ;

   L = Y :: 3 :: nil.
   ;

   L = Y :: Y1 :: 3 :: nil.

   yes
   ```

5. (10 points) The relation `consecutive` such that

   ```
   type consecutive  (list T) -> T -> T -> o.
   ```

   where `consecutive L X Y` holds iff in the list `L`, the items `X` and `Y` appear in that order. For example,

```
?- consecutive (1::2::nil) 1 2.
solved
;
no more solutions
?- consecutive (1::1::2::nil) 1 2.
solved
;
no more solutions
?- consecutive (1::nil) 1 2.
no
?- consecutive (4::1::3::2::nil) 1 2.
no
```

6. (15 points) The relation `subst_second` such that

   ```
   type subst_second T -> T -> (list T) -> (list T) -> o.
   ```

   and `(subst_second New Old L1 L2)` holds if `L2` is like `L1` except that the second occurrence of `Old`, if any, has been replaced by `New`. For example,

   ```
   ?-  subst_second 33 1 (1::2::1::3::4::1::nil) L.

   L = 1 :: 2 :: 33 :: 3 :: 4 :: 1 :: nil.

   yes
   ?- subst_second 55 1 (3::1::nil) L.

   L = 3 :: 1 :: nil.

   yes
   ```

7. (5 points extra credit only) Using `not` and cut (!), but a minimum number of times, can you program `subst_second` so that when run forwards, it only gives one answer?

In the following problems, you are to build on the code in `~leavens/src/lambda-prolog/unary.mod`. (You should copy that and import it into your answer module.)

8. (10 points) Write a relation `modulo` such that

   ```
   type modulo nat -> nat -> nat -> o.
   ```

   and such that `(modulo X Y R)` holds if `R` is `X` modulo `Y`. Do not use the λProlog built-in integer arithmetic, give a direct specification of `modulo`. For example,

   ```
   ?- use "modulo".
   ?- use "unary".
   ?- to_nat 3 THREE, to_nat 2 TWO, modulo THREE TWO Z, to_int Z N.

   N = 1,
   Z = s z,
   TWO = s (s z),
   THREE = s (s (s z)).

   yes
   ?- to_nat 3 THREE, to_nat 7 SEVEN, modulo SEVEN THREE Z, to_int Z N.
   ```

```
N = 1,
Z = s z,
SEVEN = s (s (s (s (s (s (s z))))))),
THREE = s (s (s z)).

yes
?- to_nat 3 THREE, to_nat 7 SEVEN, modulo THREE SEVEN Z, to_int Z N.

N = 3,
Z = s (s (s z)),
SEVEN = s (s (s (s (s (s (s z))))))),
THREE = s (s (s z)).

yes
```

(Hint: you may need to think about the order of clauses if your program seems to be in an infinite loop.)

9. (15 points) Look at chapter 6 (especially section 6.3) in Dale Miller's notes, and write a relation `iterate` such that

   ```
   type iterate nat -> (T -> T) -> T -> T -> o.
   ```

   and such that `(iterate N Fun X Z)` holds if `Z` is the result of iterating the function `Fun N` times on `X`. The following examples use lambda abstraction.

   ```
   ?- use "unary".
   ?- iterate z (x \ x) z Ans.

   Ans = z.
   ;
   no more solutions
   ?- iterate (s (s z)) (x : nat \ (s x)) z Ans.

   Ans = s (s z).
   ;
   no more solutions
   ?- iterate (s (s z)) (x : nat \ (s (s (s x)))) z Ans.

   Ans = s (s (s (s (s z)))).
   ;
   no more solutions
   ```

3 Conceptual Modeling

The following explore the conceptual modeling (or database) aspects of λProlog.

10. (10 points) Encode your weekly schedule as a set of λProlog facts and rules. Show how it would be used by writing some queries.

11. (20 points, extra credit) Write a λProlog program to keep track of information about some sporting league, such as the baseball or football league standings.

12. (30 points, extra credit) Encode your family tree as a set of λProlog facts and rules. Can you make the program answer questions like "is Sue my second cousin once removed"? Can you make the program answer questions like what is the relationship between Sue and Fred"? (Hint: see Dale Miller's notes)

13. (50 points, extra credit) Write a theorem prover for predicate logic that returns a proof of a theorem when it can be proved.

4 Describing Abstract Values

In this section we'll look at how to use λProlog to specify abstract data types. The idea is much like that in Chapter 6 of Watt's *Programming Languages: Syntax and Semantics*, which you might want to read.

14. (42 points) Specify in λProlog a module for homogeneous finite sets. Your module should be called `set.mod` and should import the following module, which is found in the following file.

    `~leavens/src/lambda-prolog/set_syntax.mod`

```
module set_syntax.  % types for homogeneous finite sets.

kind set                type -> type.

% set terms
type emptyset           set T.
type set_insert         set T -> T -> set T.

% set observers
type has                set T -> T -> o.
type set_delete         set T -> T -> set T -> o.
type set_union          set T -> set T -> set T -> o.
type set_intersect      set T -> set T -> set T -> o.
type set_size           set T -> int -> o.
type subseteq           set T -> set T -> o.
type set_equal          set T -> set T -> o.
```

    You are to specify all the "observer" relations. The meaning of these is intended to be the usual thing for sets. For example, the relation `(has S E)` holds iff `E` is an element of `S`. The relation `(set_delete X E Z)` holds iff `Z` has all the elements of `X` except `E`. The relation `(subseteq X Y)` holds iff each element of `X` is also an element of `Y`.

15. (30 points; extra credit) Write the previous problem so that all your relations run backwards. (We will give appropriately scaled partial credit for doing part of this.)

16. (30 points; extra credit) Add the constructor

```
type set_singleton      T -> set T.
```

    to sets and rewrite all of the observers to deal with singleton sets constructed using `set_singleton`. Why is this so painful? What could be done to make using singleton sets possible without having to go through the pain?

17. (45 points; extra credit) Specify in λProlog a module for homogeneous finite doubly-ended queues. Your module should be called `dequeue.mod` and should import the following module. (The file is in `~leavens/src/lambda-prolog/dequeue_syntax.mod`.)

```
module dequeue_syntax. % syntax for doubly ended queues

kind deq                type -> type.

% dequeue terms
type emptydeq           deq T.
type deq_precat         deq T -> T -> deq T.
type deq_postcat        deq T -> T -> deq T.

% deq observers
type deq_has            deq T -> T -> o.
type deq_size           deq T -> int -> o.
type deq_head           deq T -> T -> o.
type deq_last           deq T -> T -> o.
type deq_init           deq T -> deq T -> o.
type deq_tail           deq T -> deq T -> o.
type deq_count          deq T -> T -> int -> o.
type deq_isEmpty        deq T -> o.
type deq_equal          deq T -> deq T -> o.
```

You are to specify all the "observer" relations. The meaning of these is intended to be the usual thing for doubly-ended queues. The relation `(deq_has Q E)` holds iff `E` is in the deque `Q`. The relation `(deq_init Q1 Q2)` holds iff `Q2` has all the elements of `Q1` except the last. The relation `(deq_tail Q1 Q2)` holds iff `Q2` has all the elements of Q1, except the first. Etc.

18. (30 points; extra credit) Write the previous problem so that all your relations run backwards. (We will give appropriately scaled partial credit.)

4.1 Operational Semantics

19. (10 points) 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) $(((\lambda x\,.\,(\lambda y\,.\,x))z)((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x))))$

20. (35 points) 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/ComS541/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.

21. (15 points) 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 $(((\lambda x.(\lambda y.x))z)((\lambda x.(x\ x))(\lambda x.(x\ x))))$

22. (20 points, extra credit only) 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.)

23. (15 points, extra credit only) 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.

24. (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 $x \bigtriangledown ((\lambda x\,.\,(x\ x))(\lambda x\,.\,(x\ x)))$, 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.