Homework 5: λProlog and Logic Programming

Due: December 9, 1999.

In this homework, you will learn a few basic skills in logic programming. 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.

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 Terzo 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"; that is, you do not have to test relations where the last argument is given and other arguments are sought. (But you may want to see which ones can do that, just for fun.) In the examples you will see "no more solutions" or "yes" printed; your program does not have to have exactly this output.

You may use cut (`!`), `not`, and `once` in your code.

You may want to 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.

1 Lists

The next few problems deal with lists in λProlog. See the file

```
/usr/unsup/Terzo/doc/SystemModules/maps.mod
```

for some further examples of (higher-order) operations on lists. You'll often have to parenthesize lists, for example `(1::nil)`, so it's safer to always parenthesize them.

Specify the following relations in λProlog.

1. (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.

```
Terzo> #query find_last.
?- find_last nil 3.

no
?- find_last (1::2::3::nil) 3.

solved
?- find_last (1::2::3::nil) Y.

Y = 3 ;

no more solutions
?- find_last L 3.

L = 3 :: nil ;

L = Y :: 3 :: nil ;

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

?- #end.
Terzo>
```

2. (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,

```
Terzo> #query consecutive.
?- consecutive (1::2::nil) 1 2.

solved
?- consecutive (1::1::2::3::nil) 1 2.

solved
?- consecutive (1::nil) 1 2.

no
?- consecutive (4::1::3::2::nil) 1 2.

no
?- #end.
```

3. (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,

```
Terzo> #query subst_second.
?- subst_second 33 1 (1::2::1::3::4::1::nil) L.

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

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

L = 3 :: 1 :: nil

?- subst_second 33 1 nil X.

X = nil

?- #end.
```

4. (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 the unary.mod file available from the web page and in

```
~leavens/WWW/ComS541/homework/unary.mod
```

(You should copy that file to your directory, and then import it into your answer module.)

5. (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; instead give a direct specification of modulo. For example,

```
Terzo> #query unary, modulo.
?- to_nat 3 X.

X = s (s (s z)) ;

no more solutions
?- to_nat 3 THREE, to_nat 2 TWO, modulo THREE TWO N, to_int N Z.

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

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

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

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

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

?- to_nat 3 THREE, modulo THREE THREE N, to_int N Z.

THREE = s (s (s z))
N = z
Z = 0
?- #end.
```

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

2 Conceptual Modeling

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

6. (20 points) Encode your program of study (the classes you are taking as part of your degree) as a set of λProlog facts and rules. Show how it would be used by writing some queries.

3 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.

7. (42 points) Specify in λProlog a module for homogeneous finite sets. Your module should be called `set.mod`. It should start somewhat like the following.

   ```
   module set.
   import set_syntax.
   has (set_insert Set Elem) Elem.
   % ...
   ```

   The use of the `import` clause imports the following module, which will allow you to avoid retyping the kind and type declarations it contains.

   ```
   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 should make a copy of the `set_syntax` module from the file `set_syntax.mod` available from the web page and from

```
~leavens/WWW/ComS541/homework/set_syntax.mod
```

and put it in the directory where you have the your file `set.mod`.

You are to specify all the "observer" relations whose types are given in `set_syntax.mod`. 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`.

Note that terms of type `(set T)` allow for duplicates; however, you are to specify the mathematical content. For example, you should have:

```
Terzo> #query set_syntax, set.
?- set_size (set_insert (set_insert emptyset 3) 3) Z.

Z = 1
```

Similarly, order should not matter in testing for subsets or equality.

When testing your code, be sure to `load` both of the modules. That is, start your testing session by giving the command.

```
Terzo> #load "set_syntax.mod", "set.mod".
```

8. (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.)

9. (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?

10. (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 available from the web page and from the following.)

```
~leavens/WWW/ComS541/homework/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.

11. (30 points; extra credit) Write the previous problem so that all your relations run backwards. (We will give appropriately scaled partial credit.)