Homework 6: Introductory λProlog Programming

Due: 1-6, Friday, March 11; 7-8, Friday, March 25.

Since the purpose of this homework is to ensure basic skills in logic programming, 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.

The first 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.

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

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.

```
?- 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
```

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,

```
?- 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
```

3. (15 points) The relation `delete_second` such that

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

and `(delete_second L1 X L2)` holds if `L2` is like `L1` except that the second occurrence of `X` has been deleted. For example,

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

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

yes
?- delete_second (3::1::nil) 1 L.

L = 3 :: 1 :: nil

yes
```

4. (5 points extra credit only; programming, semantics) Using `not` and cut (!), but a minimum number of times, can you program `delete_second` so that when run forwards, it only gives one answer?

5. (15 points) The relation `subst_all` such that

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

and `(subst_all New Old L1 L2)` holds if `L2` is like `L1` except that every occurrence of `Old` has been replaced by `New`.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

```
?-  subst_all 5 6 (6::6::3::4::6::nil) L.

L = 5 :: 5 :: 3 :: 4 :: 5 :: nil.
;
no more solutions
?- subst_all 7 8 (9::nil) L.

L = 9 :: nil.
;
no more solutions
```

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

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

and `(subst_last New Old L1 L2)` holds if `L2` is like `L1` except that the last occurrence of `Old` has been replaced by `New`.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

```
?-  subst_last 54 1 (1::1::3::4::1::nil) L.

L = 1 :: 1 :: 3 :: 4 :: 54 :: nil.

yes
?- subst_last 22 7 (3::nil) L.

L = 3 :: nil.
;
no more solutions
```

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.)

7. (10 points) Write a relation `expt` such that

```
type expt nat -> nat -> nat -> o.
```

and such that `(expt X Y Z)` holds if `Z` is `X` raised to the `Y` power. For example,

```
?- to_nat 3 THREE, to_nat 2 TWO, expt THREE TWO Z, to_int Z N.

N = 9,
Z = s (s (s (s (s (s (s (s (s z)))))))),
TWO = s (s z),
THREE = s (s (s z)).
;
no more solutions
```

8. (15 points) Look at chapter 6 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.

```
?- 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 (s z))))).
;
no more solutions
```

1 Extra Credit

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

9. (10 points, extra credit; programming) Encode your weekly schedule as a set of λProlog facts and rules. Show how it would be used by writing some queries.
10. (20 points, extra credit; applications, programming) Write a λProlog program to keep track of information about the Computer Science Department racketball standings.
11. (20 points, extra credit; applications, semantics) 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)
12. (30 points, extra credit; semantics) Write a theorem prover for predicate logic that returns a proof of a theorem when it can be proved.