Project 1

Due: November 4, 2008, 5:30 pm

Some rules and conditions:

1. You can work either in singles or in pairs for this project.
2. You are **not** allowed (and, really, do not need) to use Prolog's built-in predicates, except those that we discussed in the class. If in doubt, email me, I will try to reply quickly.
3. The reports are to be typed.
4. Late submission penalty: 25% off the grade for every 24 hours or part of thereof.
5. As usual, cheating will not be tolerated. Copied code is even easier to detect than copied paper-and-pencil assignments (we have software that does marvelous job at this).
6. If anything is unclear, email me. Frequently asked questions will be posted on the website.

Project Description

In this project we are going to implement Robinson's unification algorithm that works on the special type of terms, called l-terms[1]. Informally, an l-term is a Prolog term written in the list representation, and in which all variables are enclosed in single quotes. As an example, the l-terms that correspond to Prolog terms $f(X, g(12))$ and $f(a, Y)$ are $[f,' X', [g, 12]]$ and $[f, a,' Y']$, respectively, and our unification algorithm will output $['X'/a,' Y'/[g, 12]]$ as an m.g.u. of these two l-terms[2]. Formally, l-terms can be described as follows: a *l-term* is either:

(i) an *l-variable* – any sequence of characters that starts with a capital letter or an underscore, enclosed in single quotes ($'$), or

(ii) an *l-atom* – any Prolog atom which is not an l-variable, or

(iii) an *l-number* – any Prolog number, or

(iv) a *compound l-term* – an expression of the form $[f, a_1, \ldots, a_n]$, where f is a l-atom (functor), and $a_1, \ldots, a_n$ are l-terms (arguments).

[1] l- is for "list".

[2] Note that as far as Prolog is concerned, these two l-terms are not unifiable, because everything that is enclosed in single quotes is considered to be an atom.

Some examples:

- an l-term that corresponds to a Prolog term $f(X, a)$, is a list $[f,' X', a]$;
- an l-term that corresponds to a Prolog term $f(X, g(h(Y), 12))$, is a list $[f,' X', [g, [h,' Y'], 12]]$;
- an l-term that corresponds to a Prolog term Y is simply $'Y'$.

Your task for this project is to complete the implementation of the unification algorithm for l-terms that I have started in file `unify_lterms.pl` (available on course website). To get things going I've implemented the following predicates for you (you may not need all of these, but will probably need some):

> `lterm(+T)` – true if `T` is an l-term.
>
> `lvar(+T)` – true if `T` is a l-variable.
>
> `latom(+T)` – true if `T` is an l-atom.
>
> `lnumber(+T)` – true if `T` is a l-number.
>
> `latomic(+T)` – true if `T` is l-atomic.
>
> `lcompound(+T)` – true if `T` is a compound l-term.

Note: The `+` and `?` signs in front of the arguments is the standard convention to indicate the *instantiation pattern* of the arguments. These signs have the following meaning:

> `+` sign means that the argument must be fully instantiated, i.e. bound to a ground term. In other words, this is an input argument.
>
> `?` sign means that there is no restriction on the argument, i.e. it could be either fully bound, or partially bound, or unbound. In other words, this is argument can be either input or output, depending on what the programmer wants to achieve.
>
> `-` sign means that the argument must be unbound, i.e. this it is an output-only argument.

This is the standard convention used in Prolog documentation.

Your goal in this project is to implement the predicates described below. You can assume that all predicates will be invoked with correct input arguments, that is, for example, you do not need to check that the second argument of `occurs/2` described below is indeed an l-term.

(a) `occurs(+V, +T)` – true if `V` is an *l*-variable that occurs in *l*-term `T`. For example,

```
-?  occurs('X', [f, 'X']).
Yes
-?  occurs('X', [f, [h, 'Y']]).
No
```

(b) `apply_subst(+[V/T], +T_in, ?T_out)` – true if `T_out` is the result of substitution of *l*-term `T` for *l*-variable `V` in the l-term `T_in`. For example,

```
?- apply_subst(['X'/[f, 'Y']],[g, 'X'], T).
T = [g, [f, 'Y']] ;
No
?- apply_subst(['X'/12],[h, 'Y'], T).
T = [h, 'Y'] ;
No
```

(c) `compose_subst(+S_in, +[V/T], ?S_out)` – true if `S_out` is a substitution obtained by the composition of substitution `S_in` with a substitution `[V/T]`. `S_in` is given as a (possibly empty) list of *l*-variable/*l*-term pairs: `[V_1/T_1, V_2/T_2, ... , V_n/T_n]`. Be careful: if `V` is one of `V_i`, `[V/T]` is not part of `S_out`. For example,

```
?- compose_subst(['X'/[f, 'Y'], 'Z'/[g, 'X', 'Y']], ['Y'/12], S).
S = ['Y'/12, 'X'/[f, 12], 'Z'/[g, 'X', 12]] ;
No ?- compose_subst(['X'/[f, 'Y'], 'Z'/[g, 'X', 'Y']], ['X'/12], S).
S = ['X'/[f, 'Y'], 'Z'/[g, 12, 'Y']] ;
No ?- compose_subst([], ['X'/12], S).
S = ['X'/12] ; No
```

Note that in the first example, the new mapping (`['Y'/12]`) has been appended to the front of `S_in` – this is just an artifact of my implementation, you can add the new substitution wherever you want.

(d) `diff_lterms(+T1, +T2, ?D1, ?D2)` – true if two given *l*-terms `T1` and `T2` are different, and `D1` and `D2` are the first, from the left, (sub)terms that differ. For example,

```
?- diff_lterms([f,[g,[a, 23]]], [f,[g,'Y']], D1, D2).
D1 = [a, 23]
D2 = 'Y' ;
No
?- diff_lterms([f,[g, 'Y'], 23, a], [f,[g,'Y'], [h, 'Z'], b], D1, D2).
D1 = 23
D2 = [h, 'Z'] ;
No
?- diff_lterms('X', [f,[g,'Y']], D1, D2).
D1 = 'X'
D2 = [f, [g, 'Y']] ;
No
?- diff_lterms([f, 'X'], [f, 'X'], D1, D2).
No
```

(e) `unify_lterms(+T1, +T2, ?S_out)` – true if *l*-term `T1` is unifiable with the *l*-term `T2`, and `S_out` is the substitution which is the m.g.u. of `T1` and `T2`. For example,

```
?- unify_lterms([f, 'X'], [f, [g, 'Y']], S).
S = ['X'/[g, 'Y']] ;
No
?- unify_lterms([p, a, 'X', [h, [g, 'Z']]], [p, 'Z', [h, 'Y'], [h, 'Y']], S).
S = ['Y'/[g, a], 'X'/[h, [g, a]], 'Z'/a] ;
No
?- unify_lterms([g, a, 'X'], [g, a, [h, 'X']], S).
No
?- unify_lterms([g, a, 'X'], [h, 'X'], S).
No
```

Note that the order in which the mappings are listed in the second example is not important (i.e. your implementation may give the mappings in different order). Also, note that the second last example fails because of occurs-check.

What to Submit, How, and When

Your implementation of the predicates described above should be placed inside the `unify_lterms.pl` file (from the website), and submitted using the following command:

```
submit 3401 p1 unify_lterms.pl
```

The submission deadline is 5:30pm, November 4, 2008. Note that your submissions are automatically time-stamped – I will use these time-stamps to determine the late penalties. I expect your code to be reasonably well-commented.
In addition to the code, I ask you to submit a short (2 pages or so), typed, report which describes in detail the way you implemented each of the predicates above. In addition,

attach a full listing of your version of `unify_lterms.pl` to the end of your report. **The report should be handed in to me at the beginning of the class on November 4, 2008**.

The level of detail of the description of your predicates should be such that it indicates to me that you understand what you are doing. If your predicate is implemented using multiple clauses, describe the purpose of each clause. Try to be concise, but do not omit important details. Here is a description of the predicate `lshift/2` (from the last class) as an example (you do not need to cut-and-paste your code into the main part of the report though):

```
lshift([], []).
lshift([X], [X]).
lshift([X,Y|T], [Y|T2]) :- lshift([X|T], T2).
```

Description: the left shift of the list with 0 or 1 elements is the list itself – this case is handled by the first two clauses. For lists that contain at least two elements X and Y, we place Y on the first position of the result, and append to it the result of the recursive left-shift of the list $[X|T]$.