Principles of Programming Languages - Spring 2011
Exercise no. 6 – Logic programming

Guidelines:

1. Submit in the submission system, a single zipped file, ex6.pl, with the answers to programming questions 1-3. Submit in the course mailbox (37/ground floor), a paper with answers to theoretical questions 4-6. Do not forget to write your submission group no. on the paper.
2. You are required to write a contract (in comments) above every procedure you write in ex6.pl.

Tracing and printing in Prolog:

In order to trace your program you can use the 'writeln' primitive predicate, to follow the instantiations of your procedures. For example,

```
?- writeln(term(3)).
term(3)
?- writeln(term(X,3)).
term(_G572, 3)
```

In addition, you can trace the calls along the evaluation of your proof tree. For example,

```
?- trace.
Unknown message: query(yes)
[trace] 16 ?- recommend(10, RRProductID,0).
   Call: (7) recommend(10, _G16922, 0) ? creep
   Call: (8) recommend_product(10, _G16922) ? creep
   Call: (9) order_details(_L192, 10, _L212, _L213) ? creep
   Exit: (9) order_details(10524, 10, 2, 0) ? creep
...
```

Press "enter" to get the next step in the trace.

Question 1: Relational Logic Programming

This example is based on the Northwind database, which is a Microsoft sample database. The database contains the sales data for Northwind Traders, a fictitious specialty foods export-import company. It describes how customers order products. A customer can order multiple products in a single order and he can specify a quantity for each product he orders. Each product has a unit price, but a customer can get a discount on a product in a specific order.

In the given file, the tables of the Northwind relational database are represented by Prolog facts, defining the relations customer, product, order and order_details. Here are their schemas:

customer(CustomerID, CompanyName, ContactName, ContactTitle, Address, City, PostalCode, Country) product(ProductID, ProductName, UnitPrice)

order(OrderID, CustomerID)

order_details(OrderID, ProductID, Quantity, Discount)

Part 1: Natural Join and Product: Write a procedure
`total_price_customer_product(CustomerID, ProductName, Discount)/3`
that gives for each customer, the name of the products he ordered in all orders he placed with the discount he got on that product. Notice the connection between the relevant tables should be done according to CustomerID, ProductID and OrderID fields.

- Use anonymous variables (starting with underscore) for variables that are not used more than once in the body of the procedure and are reported by the query.

For example,

```
?- total_price_customer_product(Customer, Product, Discount).
```

gets this answer among others:

```
   Customer = aLFKI, Product = chang, Discount= 0.25 ;
```

meaning that customer aLFKI ordered the product 'chang' and got a discount of 25%.

Part 2: Selection and Production: Write a procedure
`europe_ordered_products(ProductName, ProductID)/2`
that gets the names and ids of products that were ordered by European customers.
You can define an auxiliary procedure to identify Europian countries.

- Some answers appear more than once. For example,

  ```
  ?- europe_ordered_products(ProductName, ProductID).
  ...
  ProductName = uncleBobOrganicDriedPears, ProductID = 7 ;
  ProductName = uncleBobOrganicDriedPears, ProductID = 7 ;
  ...
  ```
  Explain why this happens. Answer in a comment below the procedure.

Part 3: Transitive closure: Write a procedure
`recommend_product(ProductName, RecommendedProductName)/2`
that recommends some products according to a given product `ProductName`.
Recommended products are products that were bought by the same customer that bought the given product. Customers who bought product p1, also bought products p1', p2',etc. So products p1', p2', etc. may be recommended as desirable to whomever purchased p1. For example,

```
?- recommend_product(ikura, RecommendedProductName).
RecommendedProductName = chai ;
false.
```

- Notice that products appearing in different orders of the same customer who bought p1, should also be recommended.
- The recommended product must not equal the given product. Use \= for this purpose. X\=Y means that X and Y cannot be unified.

After products of first closeness are recommended, products of second closeness can be reported. That is, customers who bought p1',p2'. etc. also bought products p1'', p2'',etc. So products p1'', p2'', etc. may also be recommended (but, the recommendation is less significant). Thus, recommendations for pi' may also become recommendations for p1. This recommendation policy can be continued to any degree of closeness to the given product.

Write a procedure `recommend(ProductName, RecommendedProductName, Level)/3` that gives a recommendation `RecommendedProductName`, of a different product with level of closeness `Level`, according to given product `ProductName`. In order to specify the closeness level, use the following representation of the natural numbers: s(0) represents the number 1, s(s(0)) represents 2, and so on. For example,

```
    ?- recommend(ikura, RRProductName,s(0)).
    RRProductName = chai ;
    false.
    ?- recommend(ikura, RRProductName,s(s(0))).
    RRProductName = aniseedsyrup ;
    RRProductName = chefAntonCajunSeasoning ;
    RRProductName = grandmaBoysenberrySpread ;
    false.
```

Question 2: Lists

The primitive procedure append is defined as follows:

```
% Signature: append(List1, List2, ListResult)/3
% Purpose: ListResult is a list starting with the elements of List1
%      and ending with the elements of List2, in the original order.
append([],X,X).
append([X|Xs],Y,[X|Zs]):- append(Xs,Y,Zs).
```

In this question, we want to generalize the definition of append. Write the procedure `super_append(SuperList, ListResult)/2`, such that SuperList is a list of lists and ListResult contains all the elements of all the lists in SuperList in the order or their appearance. For example,

```
?- super_append([], ListResult).
ListResult=[]

?- super_append([[a,b,c],[d],[],[e]], ListResult).
ListResult=[a,b,c,d,e]

?- super_append([[a,[b,c]],[d],[],[[e]]], ListResult).
```

```
ListResult=[a,[b,c],d,[e]]
```

- Does the procedure work “in the other direction”? Explain.
  ?- `super_append(SuperList, [a,b,c,d]).`
  Answer in a comment below the procedure.

Question 3: Context-Free Grammar parsing and syntax tree

The following Context-Free Grammar specifies a simplified Scheme language: The lambda expression has only one parameter and there is only one variable symbol ‘x’. There are no numbers. Booleans are marked by the constants ‘yep’ and ‘na’.

```
        S -> Exp
        Exp -> AtomicFormula | Composite
        AtomicFormula -> Boolean | Variable
        Composite -> Form | Lambda | If | Symbol
        Boolean -> yep | na
        Variable -> x
        Form -> ( ExpSequence )
        ExpSequence -> Exp+
        Lambda -> ( lambda (Variable) ExpSequence )
        If-> (if Exp Exp Exp)
        Symbol -> (quote X) where X is unrestricted.
```

Part 1: Language recognition:
Write a procedure `exp(Expression)/1` that recognizes of expressions of this language. You can add auxiliary procedures, if necessary. Here are some examples of queries that return true.

```
?- exp(na).
?- exp(x).
% Lambda
?- exp([lambda, [ x ], x ]).
?- exp([lambda, [ x ], x, yep]).
?- exp([lambda, [ x ], x, [lambda, [ x ], x ] ]).
% Form (application)
?- exp([ x, x, na]).
?- exp([ [ lambda, [ x ], x ], yep ]).
% If
exp([ if, yep, [lambda , [ x ], x ], na]).
exp([ if, yep, [lambda , [ x ], [if, x, yep, na] ], na]).
```

Part 2: Expression generation:
The procedure `exp` can also be used to generate all the expressions in the language, by posing the query `?- exp(X).`

- Give the first 5 answers of this query.
- How many answers do you expect this query gives? Explain.

Answer in a comment below the procedure.

Part 3: Parse tree generation:
Write a procedure `parse(Expression, ParseTree)/2`, that gives the relation between an expression in the language above and a parse tree of the expression. The parse tree has functors for labeling different types and parts of expressions. Here are some examples:

```
?- parse(na, boolean(na)).
true.
?- parse(x, variable(x)).
true.
?- parse([lambda, [ x ], x, yep], T).
T = lambda(parameter(x), body([variable(x), boolean(yep)]))
?- parse([lambda, [ x ], x, [lambda, [ x ], x ] ], T).
T = lambda(parameter(x),
           body([variable(x),
                    lambda(parameter(x),
                          body([variable(x)]))])) ;
?- parse([ x, x, na], T).
T = form([variable(x), variable(x), boolean(na)]).
?- parse([ [ lambda, [ x ], x ], yep ], T).
T = form([lambda(parameter(x),
                body([variable(x)])),
          boolean(yep)]) ;
?- parse([ if, yep, [lambda , [ x ], x ], na], T).
T = if( condition(boolean(yep)),
        consequent(lambda(parameter(x), body([variable(x)]))),
        alternative(boolean(na))) ;
?- parse(Exp, if(condition(boolean(yep)),
                consequent(lambda(parameter(x),
                                 body([variable(x)]))),
                alternative(boolean(na)))).
Exp = [if, yep, [lambda, [x], x], na] ;
```

Question 4: Concepts

1. Suggest an algorithm for deciding the question: "Is a query Q provable from a program P?" in Relational Logic Programming (RLP).
2. What is a most general unifier? In which cases does the Unify algorithm for Full Logic Programming (FLP) fail to compute it?
3. Compare the role of:
   a. Identification labels as we used in representing ADTs in Scheme.
   b. Value constructors in ML.
   c. functors in FLP.

3. Can a success/failure proof tree be finite/infinite? For each combination, if it is possible, demonstrate it with a simple example: Give a program and query for which the evaluation creates a proof tree with the demonstrated properties. Drawing the trees is not required.

4. FLP has the ability to create compound data using functors. In contrast, RLP does not have this ability.
   1. Are all proof trees in RLP finite?
   2. Is RLP decidable?
   3. Isn't there a contradiction between the above?
   4. If RLP is decidable, can every TM computation be performed in RLP?

Question 5: Unification

Apply the unification algorithm, step by step, on the following inputs. If the algorithm fails give the reason, if it succeeds give the substitution it computes.

1. unify[isTree(tree(X, tree(s(0), empty, empty), X)),
   isTree(tree(s(Y), tree(Y, empty, empty), W))]

2. unify[isTree(tree(X, tree(s(0), empty, empty), X)),
   isTree(tree(s(Y), tree(Y, empty, empty), empty))]

3. unify[isTree(tree(X, tree(s(X), empty, empty), X)),
   isTree(tree(s(Y), tree(Y, empty, empty), W))]

Question 6: Meta-circular interpreter

Consider the following interpreter, similar to the one presented in class.

```
% Signature:  solve(Q)/1
% Purpose: Check if Q is provable with respect to program P.
% Pre-conditions: The program P was translated into program P’,
%                    including a single procedure rule/2.
% Signature:  solve(Goal, Rest_of_goals)/2
1.    solve( [], [] ).
2.    solve( [], [G | Goals] ) :- solve(G, Goals).
3.    solve( [B1|Bs], Goals) :-  append(Bs, Goals, Goals1),
                                  solve(B1, Goals1).
4.    solve( A, Goals) :- rule(A, Bs), solve(Bs, Goals).
```

Part 1: Preprocessing. The following program P is given:

```
1. pa(a, b).
2. pa(b, c).

1. grandpa(X,Y):- pa(X,Z),pa(Z,Y).
```

Translate P into P' by using the 'rule/2' predicate, as explained in class.

Part 2: Draw the proof tree, according to the interpreter above, for the program P and the query Q:

```
?- solve(grandpa(A,B),[]).
```

- Draw the tree up to and including the first success leaf and the first failure leaf, then stop.
- For the success path: compute the combination of the substitutions along the path. Restrict the result of the combination to the query variables, to get the answer of the query corresponding to the path.
- Use the rule selection policy of Prolog, that is, in increasing order of appearance in P.
- Treat append as a primitive procedure: it is computed in one step, no need to apply its axioms!

Part 3: The primitive procedure reverse is defined as follows:

```
% Signature:  reverse(List1, List2)/2
% Purpose: List2 is the reverse of List1.
   1. reverse([], []).
   2. reverse([H|T], R) :- reverse(T, S), append(S, [H], R).
```

Suppose we try to replace rule 3 in the interpreter above, with the following rule:

```
   3. solve( [B1|Bs], Goals) :-
                     reverse([B1|Bs], [RB1|RBs]),
                     append(RBs, Goals, Goals1),
                     solve(RB1, Goals1).
```

Explain the impact of this change on the computation and on structure of the proof tree. In general, will the interpreter return the same results? How will it work?

Part 4: Suppose we try to replace rule 4 in the interpreter above, with the following rule:

```
4.    solve( A, Goals) :- rule(A, Bs),
                          reverse(Bs, RBs),
                          solve(RBs, Goals).
```

Explain the impact of this change on the computation and on structure of the proof tree. In general, will the interpreter return the same results? How will it work?