Principles of Programming Languages, Spring 2019
Assignment 5
Lazy Lists, Logic Programming

Submission instructions:

a. Submit an archive file named ***id1_id2.zip*** where *id1* and *id2* are the IDs of the students responsible for the submission (or ***id1.zip*** for one student in the group).
b. Use exact procedure and file names, as your code will be tested automatically.
c. Answer theoretical questions in *ex5.pdf*.
d. Answer coding questions in their specified files.
e. Your implementation will be held correct if it returns answers in any order given a query, as long as all the correct answers are returned, and only the correct answers are returned, and infinite computations are avoided.
f. You can choose to draw your tree carefully on paper, scan and add to the submission as *tree*<*maybesometext*>.<*extension*> file or files.

Part 1: Lazy Lists

Question 1: Lazy Lists (20 points)

a. (7 points) Write the function **interleave3** that accepts three lazy lists and interleaves them (answer in *lazylists.rkt* file).

```
;; Signature: interleave3(lz1, lz2, lz3)
;; Type: Lzl(T)*Lzl(T)*Lzl(T)->Lzl(T)
```

For example:

```
(define ones (cons-lzl 1 (lambda() ones)))
(define twos (cons-lzl 2 (lambda() twos)))
(define threes (cons-lzl 3 (lambda() threes)))

> (take (interleave3 ones twos threes) 6)
'(1 2 3 1 2 3)
```

b. (5 points) Write the function **biasedInterleave** that accepts two lazy lists and interleaves them two items for one list and then one item from the second (answer in *lazylists.rkt* file).

```
;; Signature: biasedInterleave(lz1, lz2)
;; Type: Lzl(T)*Lzl(T)->Lzl(T)
```

For example:

```
> (take (biasedInterleave ones twos) 6)
'(1 1 2 1 1 2)
> (take (biasedInterleave ints ones) 6)
'(0 1 1 2 3 1)
```

c. (8 points) A lazy tree is a step forward beyond lazy lists, in the form of a tree, as seen in class. Write the function **take-lzt**, that can be used for lazy-trees as "take" for lazy lists: it returns the list of the first n nodes from the lazy-tree when it is traversed in breadth-first order (answer in *lazytrees.rkt* file).

```
;; Signature: take-lzt(lzt, n)
;; Type: Lzt(T)*Number->List(T)
```

For example:

```
> (define testsmall
   (lambda (root)
     (expand-lzt root
                 (lambda (node)
                   (list (+ 1 node) (+ 1 node))))))
> (define smalltree (testsmall 0))
> (take-lzt smalltree 8)
'(0 1 1 2 2 2 2 3)
```

Part 2: Logic Programming

Question 2: Relational Logic Programming (20 points)

Recall the parent and ancestor relations from class, assume the existence of the following relations male/1, female/1, as well as more content to the parent, male and female relations.

parent(abraham, isaac).
parent(isaac, jacob).
parent(sarah, isaac).
parent(jacob, joseph).
parent(rebbeca, esav).
parent(rebbeca, jacob).
parent(isaac, esav).
male(abraham).
male(isaac).
male(joseph).
female(sarah).
female(rebbeca).

a. (5 points) Define the relation **siblings**/2: two people are siblings if they have the same father and the same mother. Father has to be male, and mother has to be female (answer in *family.pl* file).
b. (10 points) Define the relation **relatives**/2 as two people that share a common ancestor. Hint: make sure that your relation does not create an infinite loop. You can return the same answer multiple times, if all the expected results are returned at least once, and there are no unexpected ones (answer in *family.pl* file).
c. (5 points) Write a query that returns all the female relatives of joseph (answer in *ex5.pdf* file).

Question 3: Unification (20 points)

Answer in *ex5.pdf* file.
What is the result of these operations (4 points each)? Provide all the algorithm steps. Explain in case of failure.

```
a. unify[mn(ttt,M,mn,V,HO,mn,br(M)),mn(ttt,br(HO),mn,br(M),b
   r(V),mn,br(M))]
b. unify[m(M,N),n(M,N)]
```

```
c. unify[kmr(pt(pt),V,pt,g,kmr(TF),pt),kmr(pt(V),V,pt,g,kmr(
   TF),HO)]
d. unify[mn([T]), mn(T)]
e. unify[g([br | [KK | K]]), g([[br | KK] | K])]
```

In your answer, please follow the pattern from Practice 11 – for example:
A = tree_member(tree(X, 10, f(X)), W)
B = tree_member(tree(Y, Y, Z), f(Z))

1. s={X=Y}	A°s= tree_member(tree(Y, 10, f(Y)), W)
	B°s= tree_member(tree(Y, Y, Z), f(Z))
2. s={X=**10**, Y=10}	A°s= tree_member(tree(10, 10, f(10)), W)
	B°s= tree_member(tree(10, 10, Z), f(Z))
3. s={X=10, Y=10, Z=f(10)}	A°s= tree_member(tree(10, 10, f(10)), W)
	B°s= tree_member(tree(10, 10, f(10)), f(f(10)))
4. **s={**X=10, Y=10, Z=f(10), W=f(f(10))}	
	A°s= tree_member(tree(10,10, f(10)), f(f(10)))
	B°s= tree_member(tree(10,10,f(10)), f(f(10)))

Answer: s={X=10, Y=10, Z=f(10), W=f(f(10))}

Question 4: Answer-query algorithm (25 points)

Answer in *ex5.pdf* – you can add more image files for the trees as explained in the last preamble paragraph.
Given the following relations:

```
parent(abraham, isaac).        %1
parent(isaac, jacob).          %2
parent(sarah, isaac).          %3
parent(jacob, joseph).         %4

% Signature: ancestor(Ancestor, Descendant)/2
% Purpose: Ancestor is an ancestor of Descendant.
ancestor(Ancestor, Descendant) :-
    parent(Ancestor, Descendant).  %1
ancestor(Ancestor, Descendant) :-
    parent(Ancestor, Person),      %2
    ancestor(Person, Descendant).
```

a. Draw the proof trees for the following queries, explicitly mention all substitutions:

```
1. ?- parent(isaac, X).          % (5 points)
2. ?- ancestor(abraham, X).      % (10 points)
```

b. (4 points) What are the answers of the answer-query algorithm for these queries?

c. (3 points) Are these trees finite or infinite? Explain.

d. (3 points) Are they success or failure proof trees? Explain.

Question 5: Grammars (8 points)

The following context-free grammar describes a very partial yet infinite subset of Scheme. SC, as in "S Category", is the start nonterminal (variable). Other non-terminals represent syntax components such as numbers, booleans or function applications.

```
SC --> AtomC | CompoundC
AtomC --> NumberC | BooleanC | StringC
CompoundC --> RegularFormC
RegularFormC --> '(' StringC SC* ')'
NumberC --> <Number>
BooleanC --> #t | #f
StringC --> <String>
```

Prolog has been used extensively to program formal language constructs using the following approach:

- A formal language is defined by its membership predicate: that is, a predicate which holds for a list of tokens when the tokens belong to the language.
- For a context free grammar (CFG), each syntactic category is implemented by a logical predicate which holds when a list of tokens belongs to the category. For example, `sc(['#t']).`
- A context-free rule of the form "LHS → RHS1 RHS2 RHS3" is implemented by a logical rule which states that a list of tokens L belongs to LHS if there exist three smaller lists of tokens L1, L2, L3 which belong respectively to RHS1, RHS2 and RHS3 and such that their concatenation is equal to the list L. This is implemented as:

  ```
  LHS(L) :- append([L1, L2, L3], L), RHS1(L1), RHS2(L2), RHS3(L3).
  ```

  which relies on the append/2 primitive predicate. append/2 holds when L is the concatenation of all the lists held in the first parameter. For example:

```
?- append([[1,2], [3], [4,5,6], []], L).
L = [1,2,3,4,5,6]
```

append/2 can be used as a generator of splits of a given list:

```
?- append([L1, L2, L3], [1,2,3]).
L1 = [], L2 = [], L3 = [1,2,3];
L1 = [], L2 = [1], L3 = [2, 3];
L1 = [], L2 = [1,2], L3 = [3];
…
```

The file *grammar.pl* includes a Prolog program that implements the above grammar. Read the contract and run the examples.

a. (3 points) Extend the grammar above to support the **lambda** special form. Modify the derivation rules in the grammar above accordingly – you will need to add a new nonterminal and possibly change some others (answer in *ex5.pdf* file).

b. (5 points) Implement the change in the Prolog program (answer in *grammar.pl* file).

Question 6: Lists (7 points)

Answer in file *contained.pl*.

Write a procedure contained(List1, List2)/2 that defines the following relation between two lists:

```
% Signature: contained(List1, List2)/2
% Purpose: All elements in List1 appear in List2 in some
% (possibly different) order.
% Precondition: List2 is fully instantiated.
```

For example:

```
?- contained(X, [1, 2]).
X = [1, 2];
X = [2, 1];
X = [1];
X = [2];
X = []
```