Principles of Programming Languages, Spring 2018
Assignment 5
Lazy Lists, CPS, 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**. You can scan hand-drawn trees and graphs and attach them to the submitted work.
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. There are 110 points in this assignment: the question 3c (10 points) is optional.

Part 1: Lazy Lists and CPS

Question 1 (8): Lazy Lists

Using lazy lists you cannot see all the list because sometimes it is infinite. Thus you have learned the "take" function, which receives a lazy list and n, a number of elements, as parameters, and returns a regular list of the first n lazy list elements. Expand the interface of the lazy lists with a new function, take1, which still accepts two parameters, and the first parameter is still the lazy list. But compared to the previous one, the second parameter is a function. This is a predicate, which examines the terms to decide if it should continue. Each element in the lazy list is examined by the predicate. As long as it succeeds, additional elements are taken. Once an element is found that does not pass the predicate, the action stops, and the identified element is not returned. That is, the longest sequence of elements is required to match the condition of the predicate. For example, if we create a lazy list that calculates powers, we can get:

```
> (take (powers 2) 10)
'(1 2 4 8 16 32 64 128 256 512)
> (take1 (powers 2) (lambda (x) (< x 100)))
'(1 2 4 8 16 32 64)
> (take1 (powers 2) (lambda (x) (< x 64)))
'(1 2 4 8 16 32)
> (take1 (powers 2) (lambda (x) (= x 128)))
'()
```

Implement the function take1, adding it to the regular lazy list interface. We give you a *lazy.rkt* file, with the lazy list interface and a few tests. The comments show the expected answers.

Question 2 (17): CPS

Answer in q2.ts

You are giving the following function in scheme

```
;; Purpose: Find the leftmost even leaf of an unlabeled tree
;; whose leaves  are labeled by positives numbers.
;; If no leaf is even, return -1.
;; Type: [List -> Number]
;; Examples: (leftmost-even '((1 2) (3 4 5))) ==> 2
;;           (leftmost-even '((1 1) (3 3) 5)) ==> -1
(define leftmost-even
  (lambda (tree)
    (cond ((empty-tree? tree) -1)
          ((leaf? tree)
           (if (even? tree) tree -1))
          (else
           (let ([ansCar (leftmost-even (car tree))])
             (if (not (eq? ansCar -1))
                 ansCar
                 (leftmost-even (cdr tree)))))))))

(define empty-tree?
  (lambda (tree)
    (null? tree)))

(define leaf?
  (lambda (tree)
    (number? tree)))
```

You are also given the following code in Typescript, which describes the tree data types and its functional interface:

```
interface TreeNode {
  children: Tree[];
}
interface TreeLeaf {
  value: number;
}
type Tree = TreeNode | TreeLeaf;
const isTreeNode = (x:any): x is TreeNode => x.children != undefined;
const isTreeLeaf = (x:any): x is TreeLeaf => x.value != undefined;
const isTree = (x:any): x is Tree => isTreeNode(x) || isTreeLeaf(x);
```

Example values:

```
const t1: Tree = {value: 5};
const t2: Tree = {children: [
                    {children: [{value:1}, {value:7}, {value:5}]},
                    {value:3},
                    {value:10}]};
const t3: Tree = {children: [
                    {children: [{value:20}, {value:5}, {value:50}]},
                    {value:5}]};
```

1. (8) Write a function in Typescript that is equivalent to `leftmost-even`, the signature should be:

```
const leftMostEven1 = (atree: Tree): number => {
  // your code here
}
```

2. (9) Write a CPS function in Typescript that is CPS-equivalent to the one you wrote in 1.

```
const leftMostEven2 = (atree: Tree): number =>
  leftMostEven$(atree,
                (x) => x,
                () => -1);


const leftMostEven$ = <T1, T2>(atree: Tree,
                               succ: ((x:number) => T1),
                               fail: (() => T2)): (T1 | T2) =>
     // your code here
```

Part 2: Logic Programming

Question 3 (17 + optional 10): Relation Logic Programming

Answer in *q3.pl* file.

Relational databases are useful for managing structured information. Each table in the database represents a relation. Elementary structured query operations, such as select, project, join and Cartesian product enable data access. You are allowed to use only Relational LP in this question, unless it is mentioned explicitly. You may use the procedure `not_member`, appearing in the provided file *ex5-aux.pl*.
A relational database for Wikipedia management is given in the file *ex5-aux.pl*. The database consists of four tables, represented as fact-based procedures:

1) The table `page(Page_id, Page_namespace, Page_title, Page_len)/4` contains information about pages in Wikipedia. Each page has an id (that identifies it uniquely), a namespace number (used for article, user page, book, category, help, and so on), a title (where namespace and title pair is a unique key again), and length in bytes. For example, `page(12345, 1, 'DOS', 345).`
2) The table `namespaces(Ns_number, Ns_name)/2` is the namespaces list.
3) The table `category(Cat_id, Cat_title, Cat_hidden)/3` describes the categories. Each category has id (same as in page table), name (also same as in page), and a boolean value that determines if the category name the page belongs to is displayed on the page bottom.
4) The table `categorylinks(Cl_from, Cl_to)`/2 describes the category links. Each inclusion of page to category is characterized by Cl_from (same as Page_id in page table) and Cl_to (which is the category title in page).

a. (8) Write a procedure (i.e. set of axioms) `page_in_category(Name, Id/2)` that defines the relation between a page name and its category id, such that the category is non-hidden only.
b. (9) Write a procedure `splitter_category(Id)`/1 which defines a category with at least two pages. Multiple right answers are allowed if the program does not enter to infinite loop.
c. (optional, 10 points, up to 110 in assignment) Write a procedure `namespace_list(Name, List)/2` which is a relationship between a namespace name and list of all the pages IDs in it.

Question 4 (20): Unification

What is the result of these operations? Provide all the algorithm steps. Explain in case of failure.

1. (4) `unify[t(s(s), G, s, p, t(K), s), t(s(G), G, s, p, t(K), U)]`

2. (4) `unify[g(l, M, g, G, U, g, v(M)), g(l, v(U), g, v(M), v(G), g, v(M))]`

3. (4) `unify[m(M,N), n(M,N)]`

4. (4) `unify[p([v | [V | VV]]), p([[v | V] | VV])]`

5. (4) `unify[g([T]), g(T)]`

In your answer, please follow the pattern from Practice 11 as in this 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 5 (18): Answer-query algorithm

Church numbers (numerals) provide symbolic representation for natural numbers. They are defined inductively:
1. zero is a Church number,
2. For a Church number N, s(N) is a Church number.

The following program defines addition over Church numbers:

```
% Signature: natural_number(N)/1
% Purpose: N is a natural number.
natural_number(zero). %1
natural_number(s(X)) :- natural_number(X). %2

% Signature: plus(X, Y, Z)/3
% Purpose: Z is the sum of X and Y.
plus(X, zero, X) :- natural_number(X). %1
plus(X, s(Y), s(Z)) :- plus(X, Y, Z). %2
```

a. (10) Draw the proof tree for the query below and the given program. For success leaves, compute the substitution composition and report the answer.
```
?- plus(s(s(zero)), s(X), s(s(s(s(zero))))).
```

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

c. (2) Is this a success or a failure proof tree? Explain.

d. (2) Is this tree finite or infinite? Explain.

Question 6 (20): Lists in Prolog and Lazy Lists in Scheme

Answer in file **q6.pl** and **q6.rkt**

1. (8) Write a procedure `sub(Sublist, List)/2` that defines following relation between two lists:

```
% Signature: sub(Sublist, List)/2
% Purpose: All elements in Sublist appear in List in the same order.
% Precondition: List is fully instantiated
% (queries do not include variables in their first argument).
% Example:
% ?- sub(X, [1, 2, 3]).
% X = [1, 2, 3];
% X = [1, 2];
% X = [1, 3];
% X = [2, 3];
% X = [1];
% X = [2];
% X = [3];
% X = [];
% false
```

2. (12) The following Scheme predicate implements one of the modes of this relation:

```
; Signature: sub?(sub, long)
; Type: [List(T) * List(T) -> Boolean]
; Purpose: can we obtain sub by only removing items from long.
; Pre-conditions: -
; Examples:
; (sub? empty empty) -> true
; (sub? empty '(1)) -> true
; (sub? '(1) empty) -> false
; (sub? '(1) '(1 2 3)) -> true
; (sub? '(1 3) '(1 2 3)) -> true
; (sub? '(3 1) '(1 2 3)) -> false
; (sub? '(4) '(1 2 3)) -> false
(define sub?
  (lambda (sub long)
    (cond ((empty? sub) #t)
          ((empty? long) #f)
          (else (let ((suffix (member (car sub) long)))
                  (if suffix
                      (sub? (cdr sub) (cdr suffix))
                      #f))))))
```

Implement the second mode of this relation (that is, the invocation pattern where the first parameter is a free variable as opposed to a constant) as a Scheme procedure which returns a Lazy List of all the sublists that can be obtained from the original list `long`:

```
; Signature: all-subs(long)
; Type: [List(T) -> LZL(List(T))]
; Purpose: compute all lists that can be obtained
; from long by removing items from it.
; Pre-conditions: -
; Tests:
; (take (all-subs '(1 2 3)) 8) ->
; '(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
(define all-subs
  (lambda (long)
    ;; Your code here
))
```