Principles of Programming Languages, Spring 2017
Assignment 4
Promises, Generators, Lazy Lists, CPS, Logic programming

Submission instructions:

Do NOT run your asynchronous JS programs in Jupyter, only in Node.

1. 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).
2. Write your answers to each of the programming questions below in the corresponding files, ex4.rkt, ex4.pl, ex4.js.
3. Answer theoretical questions in *ex4.pdf*. You can scan hand-drawn trees and add them to the submitted work.
4. Use exact procedure and file names, as your code is tested automatically. Use the provided submission template.
5. Scheme: Make sure your .rkt files are in the correct format (see the announcement about "WXME format" in the course web page).
6. Prolog: You are not allowed to use Prolog's arithmetic or negation in this assignment. Also, you are not allowed to use functors or lists in questions marked as Relational LP. All these constraints will be checked. You may use Relational LP (only) primitive procedures and add auxiliary procedures, if needed. Any order of answers of a query is acceptable, as long as all the correct answers are returned, only correct answers are returned and infinite computations are avoided. Also avoid duplicate answers unless they are allowed explicitly.
7. Before you start implementing, read all the contracts provided in the source files.

Note: The total number of points for this assignment are 110

Question 1: Promises (10 points)

Complete the above implementation of 'all' procedure. The procedure gets an array of Promises, and returns one new Promise. If all of promises are fulfilled, the returned Promise is fulfilled and its value is an array of their result values. If any of the given promises fails, the returned promise should fail. [it is allowed, in your implementation, to modify a state in the memory].
Write your solution in the file ex4.js. (10 points)

```
function all(promises : Array<Promise<any>>) : Promise<Array<any>> {
  return new Promise( (resolve, reject) => {
    // @TODO
  });
}


//Test
function p1() { // always succeeds, with content 1
  return new Promise((resolve, reject) => {
    setTimeout(() => { resolve(1); }, Math.random() * 1000);
  });
}

function p2() { // always succeeds, with content 2
  return new Promise((resolve, reject) => {
```

```
  setTimeout(() => { resolve(2); }, Math.random() * 1000);
 });
}

function p3() {  // always fails, with err 3
 return new Promise((resolve, reject) => {
  setTimeout(() => { reject(3); }, Math.random() * 1000);
 });
}

all([p1(),p2()])
.then(content => {
  if (JSON.stringify([1,2]) === JSON.stringify(content))
    console.log("Test 1 Succeeded: ", content);
  else
    console.log("Test 1 Failed: ", content);
})
.catch(err => { console.log("Test 1 Failed: ", err); });

all([p1(),p3()])
.then(content => { console.log("Test 2 Failed: ", content); })
.catch(err => {
 if (err === 3) console.log("Test 2 Succeeded: ", err);
 else console.log("Test 2 Failed: ", err); });
```

Question 2 – Generators / Lazy Lists (25 points)

a. [1] Define an equivalence criterion for two given generators / lazy lists.
[2] Show that the following **evenSquares1** and **evenSquares2** generators equivalent according to your definition.
[3] Show that the following **fibs1** and **fibs2** lazy-lists equivalent according to your definition.
(15 points)
Answer in 'ex4.pdf' file.

```
function* naturalNumbers() {
  for (let n=0;; n++) {
    yield n;
  }
}


function* mapGen(generator, mapFunc) {
  for (let x of generator) {
    yield mapFunc(x);
  }
}

function* filterGen(generator, filterFunc) {
  for (let x of generator) {
    if (filterFunc(x)) {
      yield x;
    }
  }
```

```
}

const evenSquares1 = filterGen(mapGen(naturalNumbers(), x=> x*x), x=> (x % 2) === 0);
const evenSquares2 = mapGen(filterGen(naturalNumbers(), x=> (x % 2) === 0), x=> x*x);
```

```
; Signature: fibs1
; Type: [Void -> LzL(Number)]
(define fibs1
  (letrec ((fibgen
            (lambda (a b)
              (cons-lzl a
                        (lambda () (fibgen b (+ a b))))) ))
    (fibgen 0 1)))

; Signature: fibs2
; Type: [Void -> LzL(Number)]
(define fibs2
  (cons 0
        (lambda ()
          (cons 1
                (lambda () (lz-lst-add (tail fibs2) fibs2))))))

; Signature: lz-lst-add(lz1,lz2)
; Type: [LzL(Number)*LzL(Number) -> LzL(number)]
(define lz-lst-add
  (lambda (lz1 lz2)
    (cond ((empty-lzl? lz1) lz2)
          ((empty-lzl? lz2) lz1)
          (else (cons-lzl (+ (head lz1) (head lz2))
                          (lambda () (lz-lst-add (tail lz1) (tail lz2))))))))
```

b. Implement the 'sieve' lazy-list, defined bellow (taught in class), as a JS generator [note, the remainder in JS is given by applying (n1 %n2)]. Write your answer in the file ex4.js (10 points)

```
; Signature: sieve(lz)
; Type: [LzL(Number) -> LzL(Number)]
; Purpose: returns a lazy list of prime numbers by removing the multiples of the given lazy list values
(define sieve
  (lambda (lz)
    (cons-lzl (head lz)
              (lambda ()
                (sieve (lz-lst-filter
                        (lambda (x) (not (= (remainder x (head lz)) 0)))
                        (tail lz)))))))
```

```
function* sieve(generator) {
  //@TODO
}
```

Question 3 – CPS programming (35 points)

a. Recursive to Iterative CPS Transformations:

The following implementation of the procedure append, presented in class, generates a recursive computation process:

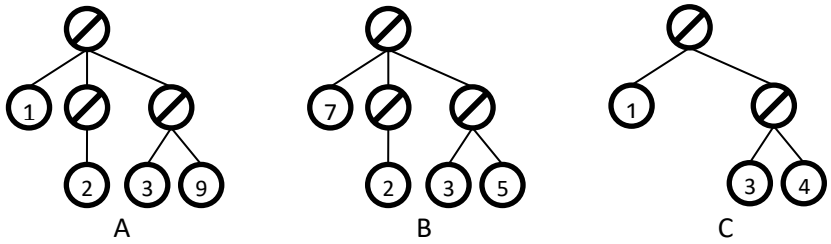
```
; Signature: append(list1, list2)
; Purpose: return a list which the arguments lists, from left to right.
; Purpose: Append list2 to list1.
; Type: [List * List -> List]
; Example: (append '(1 2) '(3 4)) => '(1 2 3 4)
; Tests: (append '() '(3 4)) => '(3 4)
(define append
   (lambda (x y)
      (if (empty? x)
          y
          (cons (car x)
                (append (cdr x) y)))))
```

[1] Write a CPS style iterative procedure append$, which is CPS-equivalent to append. Implement the procedure and complete its associated contract in the file *ex4.rkt*.

[2] Prove that append$ is CPS-equivalent to append. That is, for lists lst1 and lst2 and a continuation procedure cont, (append$ lst1 lst2 cont)=(cont (append lst1 lst2)).
Prove the claim by induction (on the length of the first list), and using the applicative-eval operational semantics. Write your proof in the file *ex4.pdf*.

(15 points)

b. Structure identity:



We regard type-expressions as leaf-valued trees (in which data is stored only in the leaves). Trees may differ from one another both in structure and the data they store. E.g., examine the following leaf-valued trees:

Trees A and B have different data stored in their leaves. Both A and B have a different structure than C.

The CPS procedure equal-trees$ receives a pair of leaf-valued trees, t1 and t2, and two continuations: succ and fail and determines their structure identity as follows:

- If t1 and t2 have the same structure, equal-trees$ returns a tree with the same structure, but where each leaf contains a pair with the leaves of the original two trees at this position (no matter whether their values agree or not).
- Otherwise, equal-trees$ returns a list with the first conflicting sub-trees in depth-first traversal of the trees.

Trees in this question are defined as an inductive data type:

- Empty tree
- Atomic tree (number or boolean or symbol)
- Compound tree: no data on the root, one or more children trees.

For example:

```
> (define id (lambda (x) x))
> (equal-trees$ '(1 (2) (3 9)) '(7 (2) (3 5)) id id)
‘((1 . 7) ((2 . 2)) ((3 . 3) (9 . 5)))
> (equal-trees$ '(1 (2) (3 9)) '(1 (2) (3 9)) id id)
‘((1 . 1) ((2 . 2)) ((3 . 3) (9 . 9)))
> (equal-trees$ '(1 2 (3 9)) '(1 (2) (3 9)) id id)
‘(2 2)  ;; Note that this is the pair '(2 . (2))
> (equal-trees$ '(1 2 (3 9)) '(1 (3 4)) id id)
‘(2 3 4) ;; Note that this is the pair '(2 .  (3 4))
> (equal-trees$ '(1 (2) ((4 5))) '(1 (#t) ((4 5))) id id)
‘((1 . 1) ((2 . #t)) (((4 . 4) (5 . 5))))
```

Implement the procedure equal-trees$ and write its contract in *ex4.rkt*.
Make sure you define the type of the procedure accurately.
(20 points)

Question 4 - Logic programming: Lists and functors, Unification, Answer-query algorithm (40 points)

a. Write a procedure `noDups(List, WithoutDups)/2` that succeeds if and only if `WithoutDups` contains the same elements of `List` with duplications removed. Meaning, for every element `X` in `List` you should keep the first appearance of `X` and remove the rest. Answer in 'ex4.pl' file.
For example:

```
?- noDups([1,2,3,4,5],X).
X = [1,2,3,4,5].
?- noDups([1,2,3,4,5,3,4,5],X).
X = [1,2,3,4,5].
?- noDups([1,2,3,4,5,3,4,5],[1,2,3,4,5]).
true.
?- noDups([1,2,3,4,5,3,4,5],[1,2,4,5,3]).
false.
```

(15 points)

b. What is the result of these operations? Provide all the algorithm steps. Answer in 'ex4.pdf' file. Explain in case of failure.
   a. `unify[p(v(v(d(1), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), KtM))]`
   b. `unify[p(v(v(d(1), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), ntuf3))]`
   c. `unify[p(v(v(d(M), M, ntuf3), X)), p(v(d(B), v(B, ntuf3), KtM))]`
   d. `unify[p(v(v(d(1), M, p), X)), p(v(d(B), v(B, ntuf3), KtM))]`
   (10 points)

c. *Unary numbers* provide symbolic representation for the natural numbers. They are defined inductively as follows: zero is the atom `[]`, and for a Unary number `c`, `[1|c]` represents the number `c+1`. The numbers 0, 1, 2, 3 etc. are represented by the terms `[], [1], [1, 1], [1, 1, 1]`, and so forth. The following program is given.

```
% Signature: unary_number(L)/1
% Purpose: L is a unary number
```

```
unary_number([]).         %1
unary_number ([1|A]) :-   %2
    unary_number (A).

% Signature:  unary_plus(X,Y,Z)/3
% Purpose: X append Y = Z
unary_plus(X, [], X) :-             %1
    unary_number(X).
unary_plus (X, [A|Y], [A|Z]):-      %2
    unary_plus(X, Y, Z).
```

[1] Draw the proof tree for the query below and the given program. For success leaves, calculate the substitution composition and report the answer at each success leaf.

```
?- unary_plus ([1|X], [1,1|Y], [1,1,1|X]).
```

[2] Is this a success or a failure proof tree?

[3] Is this tree finite or infinite?

Answer in 'ex4.pdf' file
(15 points)