PPL222

Assignment 5

Responsible Lecturer: Meni Adler
Responsible TA: Ariel Grunfeld

Submission Date: 22/6

General Instructions

Submit your answers to the theoretical questions in a pdf file called ex5.pdf and your code for programming questions inside the provided ex5.rkt, ex5.pl files. ZIP those 3 files together into a file called id1_id2.zip.
Do not send assignment related questions by e-mail, use the forum instead. For any administrative issues (milu'im/extensions/etc) please open a request ticket in the Student Requests system.
Use the forum in a responsible manner so that the forum remains a useful resource for all students: do not ask questions that were already asked, limit your questions to clarification questions about the assignment, do not ask questions "is this correct". We will not answer questions in the forum on the last day of the submission.

Question 1 - CPS [25 points]

1.1 Recursive to Iterative CPS Transformations [10 points]

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

```
; Signature: append(list1, list2)
; Purpose: Append list2 to list1.
; Type: [List<T> * List<T> -> List<T>]
; 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)
```

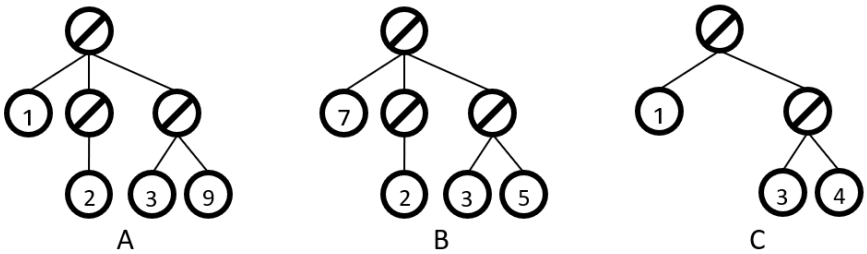
```
                    (append (cdr x) y)))))
```

a. Write a CPS style iterative procedure append$, which is CPS-equivalent to append. Implement the procedure in ex5.rkt.
(5 points)

b. 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 ex5.pdf.

(5 points)

1.2 b. Structure identity [15 points]



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 leaf-valued trees above.

Trees A and B have different data stored in their leaves, but they have the same structure. 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 pair 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.

(See lecture notes example https://bguppl.github.io/interpreters/class_material/4.2CPS.html#using-success-fail-continuations-for-search)

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$` (in ex5.rkt).

Question 2 - Lazy lists [25 points]

Cauchy Reals

Usually, when we use numbers in a program, we only use rational numbers.
However - in principle we can actually use real numbers in full generality:
Real numbers can be represented by using Cauchy sequences of rational numbers, so we'll use the definition:

```
Real = Lzl(Number)
```

That is - we'll represent each real number as an infinite lazy list of rational numbers.

Each rational number `x` can be represented as a Cauchy sequence which converges into it, so since `x` is already a rational number, we can simply use the constant infinite lazy list which has `x` in all of its components.

All the real field operators (addition, subtraction, multiplication, and division) are defined pointwise - for example: (x+y)[n] = `x[n] + y[n]`

Similarly - real functions can be represented as functions which take infinite lazy lists of numbers and return infinite lazy lists of numbers.

Note that by using Cauchy representation you can turn any **approximation** of a real number, to an **actual** real number.

2.1. Implement the following 5 procedures (in ex5.rkt):

```
; Signature: as-real(x)
; Type: [ Number -> Lzl(Number) ]
; Purpose: Convert a rational number to its form as a
; constant real number
; Example: (take (as-real 4) 10) => '(4 4 4 4 4 4 4 4 4 4)

; Signature: ++(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Addition of real numbers
; Example: (take (++ (as-real 4) (as-real 3)) 10) => '(7 7 7 7 7 7 7
7 7 7)

; Signature: --(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Subtraction of real numbers

; Signature: **(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Multiplication of real numbers

; Signature: //(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Division of real numbers
```

(5 points)

2.2. To represent the square root function, we can use Newton-Raphson approximation. Given the sequence:

$$y_{n+1} = \frac{y_n^{\,2} + x}{2y_n}$$

$\{y_n\}_{n=0}^{\infty}$ converges to $\sqrt{x}$

Note, however, that $\{y_n\}_{n=0}^{\infty}$ is a sequence of **real numbers**, so to convert it to a real number, we need to **diagonalize** the sequence - take the first component of the first component, the second component of the second component, and so on.

a. Write a procedure `sqrt-with` which takes two real numbers `x` and `y` and returns the sequence $\{y_n\}_{n=0}^{\infty}$ of real numbers which converge into $\sqrt{x}$ by using the Newton-Raphson approximation, using `y` as y_0.

```
; Signature: sqrt-with(x y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Lzl(Number)) ]
; Purpose: Using an initial approximation `y`, return a
; sequence of real numbers which converges into the
; square root of `x`
```

(9 points)

b. Implement the procedure `diag` which takes an infinite lazy list of infinite lazy lists and returns a lazy list in which the first component is the first component of the first component of the input, the second component is the second component of the second component, and so on…
Think of it as enumerating all the sequences and taking the diagonal:
[x_{11} , x_{12} , x_{13} , x_{14} , …]
[x_{21} , x_{22} , x_{23} , x_{24} , …]
[x_{31} , x_{32} , x_{33} , x_{34} , …]
[x_{41} , x_{42} , x_{43} , x_{44} , …]
Would yield:
[x_{11} , x_{22} , x_{33} , x_{44} , …]

```
; Signature: diag(lzl)
; Type: [ Lzl(Lzl(T)) -> Lzl(T) ]
; Purpose: Diagonalize an infinite lazy list
```

(9 points)

c. Using `diag` and `sqrt-with`, implement the procedure `rsqrt` which takes a real number and returns its square root by using the Newton-Raphson approximation, using the input itself as the initial approximation y_0.

; Signature: rsqrt(x)
; Type: [Lzl(Number) -> Lzl(Number)]
; Purpose: Take a real number and return its square root

; Example: (take (rsqrt (as-real 4.0)) 6) => '(4.0 2.5 2.05 2.0006097560975613 2.0000000929222947 2.000000000000002)

(2 points)

Question 3 - Logic programing [50 points]

3.1 Unification [20 points]

What is the result of these operations? Provide algorithm steps, and explain in case of failure.

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

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

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

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

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

3.2 Logic programming [20 points]

a. Write a procedure sub_list(Sublist, List)/2 that defines following relation between two lists:

```
% Signature: sub_list(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 second argument).
% Example:
% ?- sub_list(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
```

b. Write a procedure sub_tree(Subtree, Tree)/2 that defines following relation between two binary trees:

```
% Signature: sub_tree(Subtree, Tree)/2
% Purpose: Tree contains Subtree.
% Example:
% ?- sub_tree(X, tree(a, tree(b,void,void), tree(c,void,void))).
% X = tree(b,void,void);
% X = tree(c,void,void);
% X = tree(a, tree(b,void,void), tree(c,void,void)));
```

C. Write a procedure swap_tree(Tree, InversedTree)/2 that defines the following relation between two binary trees:

```
% Signature: swap_tree(Tree, InversedTree)/2
% Purpose: InversedTree is the 'mirror' representation of Tree.
% Example:
% ?- swap_tree(tree(4,tree(2,tree(1,void,void),
tree(3,void,void)),tree(5,void,void)),T).
% T =
tree(4,tree(5,void,void),tree(2,tree(3,void,void),tree(1,void,void)))
;
```

3.3 Proof tree [10 points]

```
% Signature: natural_number(X)/1
% Purpose: X is a natural number
natural_number(zero).                                  %n1
natural_number(s(X)) :- natural_number(X).             %n2

% Signature: plus(X, Y, Z)/3
% Purpose: Z is the sum of X and Y.
plus(X, zero, X) :- natural_number(X).                 %p1
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).                  %p2

% Signature: times(X,Y,Z)/3
% Purpose: Z = X*Y
times(zero, X, zero) :- natural_number(X).             %t1
times(s(X), Y, Z) :- times(X, Y, XY), plus(XY, Y, Z).  %t2

% Signature: le(X,Y)/2
% Purpose: X is less or equal Y.
```

```
le(zero, X) :- natural_number(X).                    %l1
le(s(X),s(Z)) :- le(X, Z).                           %l2
```

Draw the proof tree for the query:

```
?- le(X,s(zero)),times(X,s(s(zero)),Y).
```