PPL252

Assignment 4

Responsible Lecturer: Meni Adler
Responsible TA: Stas Rodov

Submission Date: 26/6/2025

Submit your answers to the theoretical questions in a pdf file called ex4.pdf and your code for programming questions inside the provided ex4.ts, ex4.rkt, ex4.pl files. ZIP those 4 files together into a file called id1_id2.zip.
You can add any function you wish to the code.
Do not send assignment related questions by e-mail, use the forum instead.
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 - Promises [8 points]

Complete the above implementation of ‘all’ procedure. The procedure gets an array of Promises, and returns one new Promise. The returned Promise should wait for the completion of all given promises. If all of the promises are fulfilled, the value of the returned promise 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.ts

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

In order to run the tests:
```
npm test
```

Question 2 - Generators [12 points]

a. Define (at ex4.pdf) an equivalence criterion for two given generators (2 points)

b. Implement two generators, `Fib1` and `Fib2`, which provide the Fibonacci number sequence, one based on recurrence relation ($F_n = F_{n-1} + F_{n-2}$; $F_1 = F_2 = 1$) and the other on a closed-form formula, e.g. Binet's method.
Write your solution in the file ex4.js. (6 points)

In order to run the tests:

```
npm test
```

c. Prove (at ex4.pdf) that `Fib1` and `Fib2` are equivalent (4 points)

Tests for questions 1 and 2 are in `test/ex4.test.ts`.

Question 3 - CPS [25 points]

3.1 Recursive to Iterative CPS Transformations [10 points]

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

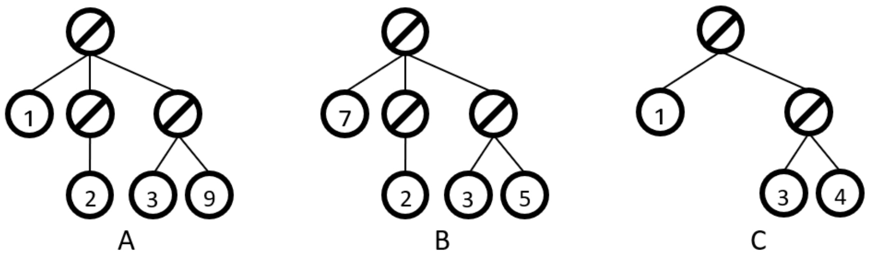
```
; Signature: append(list1, list2)
; Purpose: Append list2 to list1.
; Type: [List<T> * List<T> -> List<T>]
; Example: (append '(1 2) '(3 4)) => '(1 2 3 4)
; Tests: (append '() '(3 4)) => '(3 4)
(define append
   (lambda (x y)
       (if (empty? x)
          y
          (cons (car x)
                (append (cdr x) y)))))
```

a. Write a CPS style iterative procedure append$, which is CPS-equivalent to append.
Implement the procedure in ex4.rkt.
(5 points)

In order to run the tests, open `test/ex4-tests.rkt` with DrRacket and click the green 'run' button. If no error message is written you passed all tests.

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 ex4.pdf.
(5 points)

3.2 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 ex4.rkt).

In order to run the tests, open `test/ex4-tests.rkt` with DrRacket and click the green ‘run’ button. If no error message is written you passed all tests.

Question 4 - Lazy lists [25 points]

In mathematics, a *Cauchy sequence* is a sequence whose elements become arbitrarily close to each other as the sequence progresses.
Real numbers can be represented by a Cauchy *sequence* which converges to its ‘real’ value. PI, for example, can be represented by the infinite *Cauchy sequence* (3, 3.1, 3.14, 3.141, 3.1415...)
Real numbers can be also represented by a Cauchy sequences of *rational* numbers, *i.e.*,an infinite lazy list of rational numbers which converges to the ‘real’ value: `Real = Lzl(Number)`
A rational number `x` can be represented by a Cauchy sequence as well: since `x` is already a ‘converged’ rational number, we simply use a constant infinite lazy list of `x`, in all of its components.
Real functions can be represented as functions which take infinite lazy lists which represent numbers and return an infinite lazy list which represents the result number.
Note that the operators (addition, subtraction, multiplication, and division) for this case are defined pointwise - we add/sub/mult/div the two given numbers according to their same level of ‘precision’, for example: `(x+y)[n] = x[n] + y[n]`

4.1. Implement the following 5 procedures (in ex4.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
; Example: (take (-- (integers-from 10) (integers-from 2)) 5) ⇒ '(8
8 8 8 8)

; Signature: **(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Multiplication of real numbers
; Example: (take (** (integers-from 10) (integers-from 2)) 4) ⇒ '(20
33 48 65)

; Signature: //(x, y)
; Type: [ Lzl(Number) * Lzl(Number) -> Lzl(Number) ]
; Purpose: Division of real numbers
; Example: (take (// (integers-from 10) (integers-from 1)) 5) ⇒
'(10/1 11/2 12/3 13/4 14/5)
```

(5 points)

4.2. 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, each represented by an infinite list.

In order to convert it to one 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,` each represented as a *Cauchy sequence,* and returns the sequence $\{y_n\}_{n=0}^{\infty}$ of real numbers (each represented as a *Cauchy sequence*) 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:

```
[ x11 , x12 , x13 , x14 , … ]
[ x21 , x22 , x23 , x24 , … ]
[ x31 , x32 , x33 , x34 , … ]
[ x41 , x42 , x43 , x44 , … ]
```

Would yield:

```
[ x11 , x22 , x33 , x44 , … ]
```

```
; 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 5 - Logic programing [30 points]

5.1 Unification [5 points]

What is the result of these operations? Provide algorithm steps, and explain in case of failure (in ex4.pdf).

a. `unify[t(s(s), G, s(U), p, t(K), s), t(s(G), G, K, p, t(K), U)]`
b. `unify[p([[W | V] | [V | k]]), p([[v | V] | W])]`

5.2 Logic programming [15 points]

We would like to write a database for books. It will be composed of three types of fact:

1. *author(Id, Name)*
2. *genre(Id, Name)*
3. *book(Name, AuthorId, GenreId, Length)*

Implement the following predicates:

a. **max_list**

```
% Signature: max_list(Lst, Max)/2
% Purpose: true if Max is the maximum church number in Lst, false if
Lst is empty.

?- max_list([], Max)
false

?- max_list([s(zero),zero,s(s(zero)),s(zero)],Max)
Max = s(s(zero))
```

b. **author_of_genre**

```
% Signature: author_of_genre(GenreName, AuthorName)/2
% Purpose: true if an author by the name AuthorName has written a
book belonging to the genre named GenreName.


author(a, asimov).
author(h, herbert).
author(m, morris).
author(t, tolkien).

genre(s, science).
genre(l, literature).
genre(sf, science_fiction).
genre(f, fantasy).

book(inside_the_atom, a, s, s(s(s(s(s(zero)))))).
book(asimov_guide_to_shakespeare, a, l, s(s(s(s(zero))))).
book(i_robot, a, sf, s(s(s(zero)))).
book(dune, h, sf, s(s(s(s(s(zero)))))).
book(the_well_at_the_worlds_end, m, f, s(s(s(s(zero))))).
book(the_hobbit, t, f, s(s(s(zero)))).
book(the_lord_of_the_rings, t, f, s(s(s(s(s(s(zero))))))).

?-author_of_genre(fantasy,Author)
Author = morris;
Authos = tolkien
```

c. longestBook

```
% Signature: longest_book(AuthorName, BookName)/2
% Purpose: true if the longest book that an author by the name
AuthorName has written is titled BookName.


?-longest_book(tolkien, Book)
Book = the_lord_of_the_rings
```

Hint: Prolog has built-in predicates findall/3 (another source).

Write your solution in the file ex4.pl

5.3 Proof tree [10 points]

Draw the proof tree for the query (in ex4.pdf):

```
% 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).  %3
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).   %4

?- plus(s(s(zero)), X, s(s(zero))).
```

Is it a finite or an infinite tree?

Is it a success or a failure tree?