PPL212

Assignment 5

Responsible Lecturer: Meni Adler
Responsible TA: Eyal Segal

Submission Date: 20/6

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 [30 points]

1.1 Recursive to Iterative CPS Transformations [15 points]

The following implementation of the procedure append, presented in class, 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
```

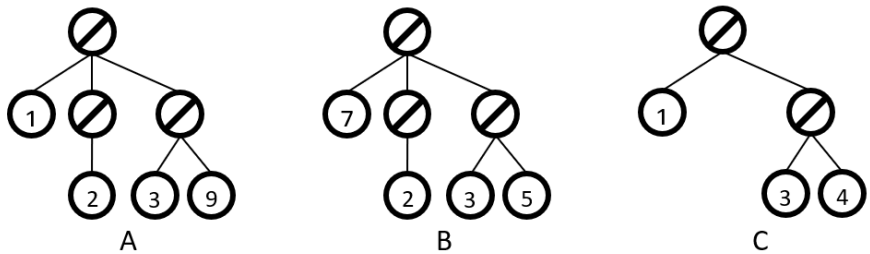
```
(cons (car x)
      (append (cdr x) y)))))
```

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.p
df.

(10 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]

a. Implement the *reduce1-lzl* procedure (in ex5.rkt), which gets a binary function, an init value, and a lazy list, and returns the reduced value of the given list (where the items are reduced according to their list order).

```
> (reduce1-lzl + 0
    (cons-lzl 3 (lambda () (cons-lzl 8 (lambda () `()))))))
11

> (reduce1-lzl / 6
    (cons-lzl 3 (lambda () (cons-lzl 2 (lambda () `()))))))
1   ;;; [6 / 3 / 2]
```

b. Implement the *reduce2-lzl* procedure (in ex5.rkt), which gets a binary function, an init value, a lazy list and an index n, and returns the reduce of the n first items of the given list (where the items are reduced according to their list order).

```
> (reduce2-lzl + 0 (integers-from 1) 5)
15

> (reduce2-lzl + 0
    (cons-lzl 3 (lambda () (cons-lzl 2 (lambda () `()))))) 5)
5

> (reduce2-lzl / 6 (integers-from 1) 2)
3    ;;; [6 / 1 / 2]
```

c. Implement the *reduce3-lzl* procedure (in ex5.rkt), which gets a binary function, an init value and a lazy list, and returns a lazy-list which contains the reduced value of the first item in the given list, the reduced value of the first two items in the given list, and so on.

```
> (take (reduce3-lzl + 0 (integers-from 1)) 5)
'(1 3 6 10 15)
```

d. For which cases will you use each of the above reduce1-lzl, reduce2-lzl and reduce3-lzl procedures?

e. Implement the *integers-steps-from* procedure (in ex5.rkt), which returns a lazy list of integers from a given number with jumps according to a given number..

```
>(take (integers-steps-from 1 3) 5)
'(1 4 7 10 13)

>(take (integers-steps-from 4 -2) 5)
'(4 2 0 -2 -4)
```

f. Use *reduce3-lzl*, *map-lzl* and *integers-steps-from* procedures in order to implements generate-pi-approximations procedure, which returns a lazy list composed of the approximations to pi according to this formula (taught in class), which converges to pi/8 when starting from a=1:

1/a×(a+2) + 1/(a+4)×(a+6) + 1/(a+8)∗(a+10) +…

The first item in the returned lazy list is the approximation according to 1/a×(a+2), the second item is the approximation according to 1/a×(a+2)+1/(a+4)×(a+6), and so on

```
>(take (generate-pi-approximations) 10)
'(2 2/3 2 94/105 2 3382/3465 3 769/45045 3 608747/14549535 3
19543861/334639305 3 352649347/5019589575 3
357188479553/4512611027925 3 388444659833/4512611027925 3
15298116214421/166966608033225)
```

g. What is the advantage and the disadvantage of *generate-pi-approximations* implementation, w.r.t. the *pi-sum* implementation taught in class.

Question 3 - Logic programing [45 points]

3.1 Unification [10 points] [ex5.pdf]

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

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

3.2 Logic programming [20 points] [ex5.pl]

We would like to write a database for books. It will be composed of three types of fact: *author*(Id, Name), *genre*(Id, name) and *book*(Name, AuthorId, GenreId, Length).

Implement the following predicates:

I. **authorOfGenre**(GenreName, AuthorName), that is true if an author by the name {*AuthorName}* has written a book belonging to the genre named {*GenreName}*

II. **longestBook**(AuthorId, BookName), that is true if the longest book that an author with the ID {*AuthorId*} has written in titled {*BookName*}

III. ***versatileAuthor****(AuthorName),* that is true if an author by the name {*AuthorName*} has written books in *at least* three different genres

Hint: Prolog has a built-in predicate findall(3) that you will find useful.

3.3 Proof tree [15 points] [ex5.pdf]

Draw the proof tree for the query:

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

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

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