PPL202

Assignment 5

Responsible Lecturer: Meni Adler
Responsible TA: Guy Erez

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", do not put titles with inflammatory tone (such as "MISTAKE", "CONTRADICTION", "CATASTROPHE"). We will not answer questions in the forum on the last day of the submission.

Question 1 - Lazy lists [30 points]

1.1 Theoretical [ex5.pdf]

a. Define an equivalence criterion for two lazy lists (when do we say that two lazy lists are equivalent)

b. Show that the following **even-squares-1** and **even-squares-2** lazy lists are equivalent according to your definition.

```
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda () (integers-from (+ n 1))))))

(define lzl-map
  (lambda (f lzl)
    (if (empty-lzl? lzl)
        lzl
        (cons-lzl (f (head lzl))
                  (lambda () (lzl-map f (tail lzl)))))))
```

```
(define lzl-filter
  (lambda (p lzl)
    (cond ((empty-lzl? lzl) lzl)
          ((p (head lzl)) (cons-lzl (head lzl)
                                    (lambda () (lzl-filter p (tail lzl)))))
          (else (lzl-filter p (tail lzl))))))

(define even-square-1
  (lzl-filter (lambda (x) (= (modulo x 2) 0))
              (lzl-map (lambda (x) (* x x))
                       (integers-from 0))))

(define even-square-2
  (lzl-map (lambda (x) (* x x))
              (lzl-filter (lambda (x) (= (modulo x 2) 0))
                       (integers-from 0))))
```

1.2 Practical [ex5.rkt]

In class, we implemented the procedure *sqrt,* which gets a number and returns its square root according to Newton's method:

```
(define sqrt (lambda (x init epsilon) (sqrt-iter x init epsilon)))

(define sqrt-iter
  (lambda (x guess epsilon)
    (if (good-enough? guess x epsilon)
        guess
        (sqrt-iter x (improve guess x) epsilon))))

(define abs (lambda (x) (if (< x 0) (- x) x)))
(define square (lambda (x) (* x x)))

(define good-enough?
  (lambda (guess x epsilon)
    (< (abs (- (square guess) x)) epsilon)))

(define average
  (lambda (x y) (/ (+ x y) 2)))
```

```
(define improve
  (lambda (guess x)
    (average guess (/ x guess))))

(sqrt 2 1 0.0001)
→
577/408
```

In this version, the improve-guess step is repeated until a 'good-enough' approximation is obtained. In contrast, the procedure *sqrt-lzl* returns **all** guesses, including those which are not 'good-enough' yet, and those which are beyond the 'good-enough' criterion.
The procedure *sqrt-lzl*, gets a number and a non-zero initial guess for its square root, and returns a lazy list composed of all its approximated roots, each represented by (guess . accuracy) pairs. The 'accuracy' value at each pair indicates the proximity of the given guess to the root of the given number x: (abs (- (square guess) x))

For example:
```
(take (sqrt-lzl 2 1) 5)
→
'((1 . 1) (3/2 . 1/4) (17/12 . 1/144) (577/408 . 1/166464) (665857/470832 . 1/221682772224))
```

a. Implement the *sqrt-lzl* procedure.

```
Signature: sqrt-lzl(x, init)
Purpose: Generate a lazy list of approximations (pairs of
<guess, accuracy>) of the square root of the given number x,
according to Newton method, starting from init guess.
Type: [Number * Number -> LzlList<Pair<Number,Number>>
Pre-condition: init =/= 0
Tests: (take (sqrt-lzl 2 1) 3) →  '((1 . 1) (3/2 . 1/4) (17/12
. 1/144))
```

b. Implement the procedure *find-first* which gets a lazy list and a predicate, and returns the first item in the list which satisfies the predicate. In case the item was not found (for the case of finite lazy list) return 'fail.

```
(find-first (integers-from 1) (lambda (x) (> x 10)))
→
11
(find-first (cons-lzl 1 (lambda() (cons-lzl 2 (lambda () '())))) (lambda (x) (> x 10)))
→
'fail
```

```
Signature: find-first(lzlst, p)
Purpose: Return the first item in the given lazy list which
satisfies the given predicate.
Type: [[LzlList<T> * T->Boolean] -> T | {'fail} ]
Pre-condition: /
Tests: (find-first (integers-from 1) (lambda (x) (> x 10))) →
11; ;;Tests: (find-first (integers-from 1) (lambda (x) (> x
10))) →  11; (find-first (cons-lzl 1 (lambda() (cons-lzl 2
(lambda () '()))))) (lambda (x) (> x 10))) → 'fail
```

c. Implement the procedure sqrt2, which gets the same parameters of the above sqrt procedure and returns the same value, by applying *sqrt-lzl* and *find-first*: it should first generate the lazy list of all approximations, and then select the first approximation which fit the 'epsilon' measure.

(sqrt2 2 1 0.0001)
→
577/408

```
Signature: sqrt2(x,init,epsilon)
Purpose: return approximation of the square root of the given
number x, according to Newton method, starting from init guess
with epsilon threshold. The procedure uses sqrt-lzl and
find-first procedures.
Type: [Number * Number * Number -> Number]
Pre-condition: init =/= 0
Tests: (sqrt2 2 1 0.0001) → 577/408
```

Question 2 - CPS [20 points]

a. Define an equivalence criterion for a procedure and its Success-Fail-Continuations version (when do we say that a procedure and its Success-Fail-Continuations version are equivalent). It helps to specify the type of the function to express this criterion. **[ex5.pdf]**

b. Write a value-retrieval procedure **get-value** for association lists: **[ex5.rkt]**

```
Signature: get-value(assoc-list, key)
Purpose: Find the value of 'key'. If 'key' is not found, return
'fail.
Type: [List<Pair(Symbol,T)> * Symbol -> T | 'fail]
```

```
Tests: (get-value '((a . 3) (b . 4)) 'b) → 4,(get-value '((a .
3) (b . 4)) 'c) → `fail
```

c. Write a Success-Fail-Continuations version of get-value: **[ex5.rkt]**

```
Signature: get-value$(assoc-list, key, success, fail)
Purpose: Find the value of 'key'. If 'key' is found, then apply
the continuation 'success' on its value val. Otherwise,
apply the continuation 'fail'.
Type: [ List<Pair<Symbol,T>> * Symbol * [T->T1] *
[Empty->T2] ] -> T1 | T2)
Tests: > (get-value$ '((a . 3) (b . 4)) 'b (lambda(x) (* x x ))
(lambda()#f)) → 16, (get-value$ '((a . 3) (b . 4)) 'c
(lambda(x) (* x x)) (lambda()#f)) → #f
```

d. Show that get-value and get-value$ are equivalent according to your definition. **[ex5.pdf]**

e. The procedure collect-all-values gets a list of associated lists and a key, and returns a list composed of the values for this key in the various associated lists. **[ex5.rkt]**

```
Signature: collect-all-values(list-assoc-lists, key)
Purpose: Returns a list of all values of the first occurrence
of 'key' in each of the given association lists. If there's no
such value, return the empty list.
Type: [List<List<Pair<Symbol,T>>> * Symbol -> List<T>]
Tests:
> (define l1 '((a . 1) (e . 2)))
> (define l2 '((e . 5) (f . 6)))
> (collect-all-values (list l1 l2) 'e) → `(2 5)
> (collect-all-values (list l1 l2) 'k)--> `()
```

Implement two versions of collect-all-values:
collect-all-values-**1** - based on **get-value**
collect-all-values-**2** - based on **get-value$**

Question 3 - Logic programing [50 points]

3.1 Unification [15 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 , H , p, t (E) , s) ,

t (s (H) , G , p , p, t (E) , K)]

b. Unify [g (c , v (U) , g , G , U , E , v (M)) ,
   g (c , M , g ,v (M) , v (G) , g , v (M))]
c. Unify [s ([v | [[v | V] | A]]) ,
   s ([v | [v | A]])]

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

The country club runs several sports activities for children. You are given the country database which includes information about the different activities, the registered kids and their parents. In your answer you may use the procedure ***not_member***, appearing in the provided ex5.pl file. A database for the country club management is given in the file ex5.pl.
The database consists of four 'tables', represented as fact-based procedures:

- The table ***activity(Name,Day)/2*** contains information about the activities the country club maintains and the day in which it takes place.
- The table ***parents(Child,Parent1,Parent2)/3*** describe the parents of each child. Each child will appear once while parents of several kids will appear several times (once for each kid).
- The table ***participate(Child,Activity)/2*** describe the relation of participating in an activity.
- The table ***parent_details(Parent,Phone,Has_car)/3*** describe the parent details, his\her name, phone number and a boolean value that determines if the parent has a car or not.

a. Write a procedure (i.e. set of axioms) ***pick_me_up(Child,Phone)/2*** that defines the relation between a child name and its parent phone number, when the parent has a car.

b. Write a procedure ***active_child(Name)/1*** which is true when a child participates in at least two activities. Multiple right answers are allowed if the program does not enter in an infinite loop.

   For parts c+d, you can use *findall/3* of Prolog.

c. Write a procedure ***activity_participants_list(Activity, List)/2*** which is a relationship between an activity name and list of all the children's names that participate at this activity (without repetition).

d. Write a procedure (i.e. set of axioms) ***can_register(Child,Activity)/2*** that defines the relation between a child name and an activity that the child can register to (the child is available at the day the activity takes place).

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

Church numbers (numerals) provide symbolic representation for natural numbers. They are defined inductively:

- zero is a Church number,
- 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. Draw the proof tree for the query below and the given program. For success leaves, compute the substitution composition and report the answer.

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

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

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

d. Is this tree finite or infinite? Explain.