PPL242

Assignment 4

Responsible Lecturer: Meni Adler
Responsible TA: Ofir Shahar

Submission Date: 22/7/2024

Submit your answers to the theoretical questions in a pdf file called ex4.pdf and your code for programming questions inside the provided ex4.rkt, ex4.pl files. ZIP those 3 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 - CPS [20 points]

The function `pipe` is implemented as follows:

```
; Signature: compose(f g)
; Type: [T1 -> T2] * [T2 -> T3]  -> [T1->T3]
; Purpose: given two unary functions return their composition, in the
same order left to right
; test: ((compose sqrt -) 16) ==> -4
;       ((compose not not) true)==> true
(define compose
  (lambda (f g)
    (lambda (x)
       (g (f x)))))


; Signature: pipe(lst-fun)
; Type: [[T1 -> T2],[T2 -> T3]...[Tn-1 -> Tn]]  -> [T1->Tn]
```

```
; Purpose: Returns the composition of a given list of unary
functions. For (pipe (list f1 f2 ... fn)), returns the composition
fn(....(f1(x)))
; test: ((pipe (list sqrt - - number?)) 16)) ==> true
;       ((pipe (list sqrt - - number? not)) 16) ==> false
;       ((pipe (list sqrt add1 - )) 100) ==> -11
(define pipe
  (lambda (fs)
    (if (empty? (cdr fs))
        (car fs)
        (compose (car fs) (pipe (cdr fs))))))
```

a. Write a CPS procedure `pipe$`, which gets a list of unary CPS functions and a continuation function and returns their compositions.
(12 points) [ex5.rkt]

The functions in the following examples are defined in ex4-tests.rkt file:

```
; Signature: pipe$(lst-fun,cont)
; Type: [
         [T1 * [T2->T3]] -> T3,
         [T3 * [T4 -> T5]] -> T5,
         ...,
         [T2n-1 * [T2n * T2n+1]]-> T2n+1
        ]
        *
        [[T1 * [T2n -> T2n+1]] -> T2n+1] ->
              [[T1 * [T2n+1 -> T2n+2]] -> T2n+2]
        -> [T1 * [T2n+3 -> T2n+4]] -> T2n+4
; Purpose: Returns the composition of a given list of unary CPS
functions.


(
 (pipe$ (list add1$ square$ div2$)
        id)
 3 id
)
⇒ 8

(
 (pipe$ (list square$ add1$ div2$)
```

```
        id)
 3 id
)
⇒ 5

(
 (pipe$ (list add1$ square$ div2$)
        (lambda (f$) (compose$ f$ add1$ id)))
 3 id
)
⇒ 9

(
  (pipe$ (list square$ add1$ div2$)
         id)
  3 (lambda (x) (* x 10))
)
⇒ 50

(
 (pipe$ (list square$ add1$ div2$)
        (lambda (f$) (compose$ f$ add1$ id)))
 3 (lambda (x) (* x 10))
)
⇒ 60

(
 (pipe$ (list g-0$ bool-num$)
        id)
 3 id
)
⇒ 1
```

b. Prove that `pipe$` is CPS-equivalent to `pipe` (8 points) [ex5.pdf]

Question 2 - Lazy lists [30 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* (taught in class) 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 [50 points]

3.1 Unification [20 points]

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

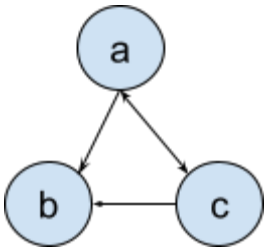
```
1. unify[x(y(y), T, y, z, k(K), y), x(y(T), T, y, z, k(K), L)]

2. unify[f(a, M, f, F, Z, f, x(M)), f(a, x(Z), f, x(M), x(F), f, x(M))]

3. unify[t(A, B, C, n(A, B, C),x, y), t(a, b, c, m(A, B, C), X, Y)]

4. unify[z(a(A, x, Y), D, g), z(a(d, x, g), g, Y)]
```

3.2 Logic programming [20 points]

Let us define a *connected directed graph* by the predicate *edge/2*, which defines an edge from one node(denoted by a symbol) to another (denoted by a symbol). For example:

```
edge(a,b).
edge(a,c).
edge(c,b).
edge(c,a).
```

Represents the graph:



Implement the following procedures (all examples refer to the above graph):

a. path

```
% Signature: path(Node1, Node2, Path)/3
% Purpose: Path is a path, denoted by a list of nodes, from Node1 to
Node2.

?- path(a,b, P)
P = [a,b];
P = [a,c,b];
P = [a,c,a,b];
…
```

b. cycle

```
% Signature: cycle(Node, Cycle)/2
% Purpose: Cycle is a cyclic path, denoted a list of nodes, from
Node1 to Node1.

?- cycle(a,C)
C = [a,c,a]
```

Now, let us alternatively define a *connected directed graph* by a list of pairs, each defining an edge in the graph. The list [[a,b],[a,c],[c,b],[c,a]] defines the above graph. Implement the following procedures:

c. reverse

```
% Signature: reverse(Graph1,Graph2)/2
% Purpose: The edges in Graph1 are reversed in Graph2

?- reverse([[a,b],[a,c],[c,b],[c,a]],Tree2)
Tree2 = [[b,a],[c,a],[b,c],[a,c]]
```

d. degree

```
% Signature: degree(Node, Graph, Degree)/3
% Purpose: Degree is the degree of node Node, denoted by a Church
number (as defined in class)

?- degree(a,[[a,b],[a,c],[c,b],[c,a]],D)
D = s(s(zero))

?- degree(c,[[a,b],[a,c],[c,b],[c,a]],D)
D = s(s(zero))

?- degree(b,[[a,b],[a,c],[c,b],[c,a]],D)
D = zero
```

3.3 Proof tree [10 points]

Draw the proof tree for the query:

```
?- path(a,b, P)
```

Is it a finite or an infinite tree?

Is it a success or a failure tree?