Principles of Programming Languages 202
Assignment 5

Ben Gindi - 205874142
Shira Segev - 208825349

Question 1- Lazy Lists

a. We say that two lazy lists, lzl1 and lzl2, are equivalent if: for each x such that $x \geq 0$, $lzl1[x] \ === \ lzl2[x]$.

b. Now, Given **even-squares-1** and **even-squares-2**, we can notice few attributes:

   i. both lazy lists contains all even numbers, starting from 0, to the power of 2.

   ii. an even number in square is an even number.

   iii. an even odd in square is an odd number (so we don't have to worry about "sneakers").

   relaying on those observations, we will now prove by induction on i that **even-squares-1** and **even-squares-2** are equivalent:

   base:

   - $x = 0$: In that case, $x\%2 \ = \ 0$, so x "passes" our predicate successfully (regarding **`even-square-2`**, $x^2 = 0^2 = 0$, so the value that being check is the same). For both lists, 0 is in.
   - $x = 1$:
     - **`even-square-1`**: first we examine whether $x\%2 \ = \ 0$. The answer is no, so that's the end for that odd number.
     - **`even-square-2`**: first we calculate $x^2 = 1^2 = 1$, than we examine whether $x\%2 \ = \ 0$. The answer here is no too of course, and both lists still includes only 0 at this point.

- We assume, that for each $x = n - 1$, **even-squares-1** and **even-squares-2** contain the same values.

  Induction step: now, we want to prove that after the nth iteration, both lists remain identical.

  - If x is odd, than according to our previous observations, x won't "pass" the predicate at all (not at it's original value, neither at it's aqare value).
  - Else, x is even, and for the same reasons, x^2 will join both lists.

```
(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))))
```

Question 2- CPS

a. Let $f\,(T)\ :\ T_1\,|\,T_2$ be a procedure such that: T is it's argument (can be a tuple), T_1 is a **success**, and T_2 is a **failure**.
   Let $f\,\$\,(T)\ :\ T_1\,|\,T_2$ be a Success-Fail-Continuations version procedure such that:
   - when T is a function which our procedure execute in case of a **success**, it's return value is of type T_1.
   - else, T is a **failure**, and in this case, f's type will be $[Empty\ \rightarrow\ T_2]$.

   Let x be an input parameter and y the return value of the procedure. We say that a procedure f and its Success-Fail-Continuations version $f\$$ are equivalent when:
   - On success- $if\,(f(X)\ ===\ y)\,s.t\,y\ :\ T_1$ and y is a success, than there is a success function that can be given to $f\$$ as an input such that by executing this function, we will get the same output $y\ :\ T_1$.
   - On failure- $if\,(f(X)\ ===\ y)\,s.t\,y\ :\ T_2$ and y is a failure, than there is a failure function that can be given to $f\$$ as an input such that by executing this function, we will get the same output $y\ :\ T_2$.

d. According to the equivalence criterion we described above, we will now prove by <u>induction on the length of the list</u> (The same way as we saw on practical session), that the procedure **get-value** and **get-value$** are equivalent.

<u>base</u>- $len = 0$:

`a-e[(get-value$ '() key success fail)] ==>* fail () =`

`= a-e[get-value '() key] ==>** fail`

- We assume, that for each $len = k \in N$, our claim is correct for each positive k, i.e:

```
if (get-value k-list key === 'fail')  ===  (get-value$ list k-key success fail)
    fail()
    success(get-value k-list key)
```

<u>Induction step</u>: now, we want to prove that for each $len = k + 1 \in N$:

`a-e[(get-value$ (k+1-list) key success fail)] ==>*`
`success(car(assoc-list)) =`

`= a-e[get-value '() key] ==>** success(car(assoc-list))`

`*` = according to the procedure definition, if the predicate is true, the return value will be: `success(car(assoc-list))`.

`**` = according to the procedure definition, if the predicate is true, than the `if` condition is applied too, and the return value will be: `success(car(assoc-list))` as well.

Otherwise (predicate is false), according to the recursion, we are calling with: `(cdr assoc-list) key success fail`. By sending `cdr` of our list, its' length is now $len - 1$, which means len is k now. So, according to the recursion assumption, the argument is applied.

```
(define get-value
 (lambda (assoc-list key)
   (if (empty? assoc-list)
     'fail
     (if (eq? (car (car assoc-list)) key)
       (cdr (car assoc-list))
       (get-value (cdr assoc-list) key)))))
```

```
(define get-value$
 (lambda (assoc-list key success fail)
   (if (empty? assoc-list)
     (fail)
     (if (eq? (car (car assoc-list)) key)
       (success (cdr (car assoc-list)))
       (get-value$ (cdr assoc-list) key success fail)))))
```

Another way to prove that the procedure **`get-value`** and **`get-value$`** are equivalent:

- On success- let's assume that for x, there is a function $(f(X) === y)\ s.t\ y : T_1$ and y is a success.
  We want to show that there is a success function that can be given to $f\$$ as an input such that by executing this function, we will get the same output $y : T_1$.
  We defined this function as the identity method. That way, same output is guaranteed.
  - Let's examine a case which our input is a non empty list and a valid key (the key appears in the list). Than, at some iteration of the procedures, the conditions `(eq? (car (car assoc-list)) key)` and `(eq? (car (car assoc-list)) key)` result will be true. For f, the procedure will be finished and the wanted output will be received, and for $f\$$, the success method will be executed, and same output is expected.
- On failure- let's assume that for x, there is a function $(f(X) === y)\ s.t\ y : T_2$ and y is a failure,
  We want to show there is a failure function that can be given to $f\$$ as an input such that by executing this function, we will get the same output $y : T_2$.
  We defined this function to be $g() : 'fail$ (g doesn't get any input). That way, same output is guaranteed (fail). we'll divide to cases:
  - If the input is an empty list- for each key, both **`get-value`** and **`get-value$`** will return fail, and the procedure $f\$$ will execute this failure.
  - Another predicted failure is when the list is not empty, but the key is not in it. That case, both procedures will end in the recursive condition and will return a failure. Same procedure will happen for $f\$$ (the failure procedure will be executed).

Question 3- Logic Programing

3.1 Unification

a. `Unify [ t ( s (s), G, H, p, t (E), s ),`
    `t ( s (H), G, p, p, t (E), K )]`

Initialization:

sub: substitution = $\{\}$ // Empty substitution

equations: Equation[] = $[t\,(s(s),\ G,\ H,\ p,\ t(E),\ s)\ = t\,(s(H),\ G,\ p,\ p,\ t(E),\ K)]$

i. Now we examine $s(s) = s(H)$ (according to line no. 7 at the algorithm)

sub: substitution = $\{\}$

equations: Equation[] = $[G = G,\ H = p,\ p = p,\ t(E) = t(E),\ K = s,\ H = s]$

ii. Now we examine $G = G$ (according to line no. 4.3 at the algorithm)

sub: substitution = $\{\}$

equations: Equation[] = $[H = p,\ p = p,\ t(E) = t(E),\ K = s,\ H = s]$

iii. Now we examine $H = p$ (according to line no. 4.1 at the algorithm)

sub: substitution = $\{H = p\}$

equations: Equation[] = $[p = p,\ t(E) = t(E),\ K = s,\ H = s]$

iv. Now we examine $p = p$ (according to line no. 6 at the algorithm)

sub: substitution = $\{H = p\}$

equations: Equation[] = $[t(E) = t(E),\ K = s,\ H = s]$

v. Now we examine $t(E) = t(E)$ (according to line no. 7 at the algorithm)

sub: substitution = $\{H = p\}$

equations: Equation[] = $[K = s,\ H = s,\ E = E]$

vi. Now we examine $K = s$ (according to line no. 4.1 at the algorithm)

sub: substitution = $\{H = p\} \circ (K = s)$

equations: Equation[] = $[H = s,\ E = E]$

vii. Now we examine $H = s$ (according to line no. 4.1 at the algorithm)

sub: substitution = $\{H = p,\ K = s\} \circ (H = s) \Rightarrow$ *this fails*

equations: Equation[] = $[E = E]$

Operation result is: FAIL (H gets two different symbols as a substitution).

b. `Unify [ g ( c, v (U), g, G, U, E, v (M)),`
`g ( c, M, g, v (M), v (G), g, v (M)]`

Initialization:

substitution = $\{\}$ // Empty substitution

Equation[]= $[g(c,\ v(U),\ g,\ G,\ U,\ E,\ v(M))\ = g(c,\ M,\ g,\ v(M),\ v(G),\ g,\ v(M)]$

i. Now we examine $c = c$ (according to line no. 6 at the algorithm)

   substitution = $\{\}$

   Equation[]= $[v(U) = M,\ g = g,\ G = v(M), U = v(G),\ E = g,\ v(M) = v(M)]$

ii. Now we examine $v(U) = M$ (according to line no. 4.1 at the algorithm)

   substitution = $\{\} \circ (v(U) = M)$

   Equation[] = $[g = g,\ G = v(M), U = v(G),\ E = g,\ v(M) = v(M)]$

iii. Now we examine $g = g$ (according to line no. 6 at the algorithm)

   substitution = $\{v(U) = M\}$

   Equation[] = $[G = v(M), U = v(G),\ E = g,\ v(M) = v(M)]$

iv. Now we examine $G = v(M)$ (according to line no. 4.1 at the algorithm)

   substitution = $\{v(U) = M\} \circ (G = v(M))$

   Equation[] = $[U = v(G),\ E = g,\ v(M) = v(M)]$

v. Now we examine $U = v(G)$ (according to line no. 4.1 at the algorithm)

   substitution = $\{G = v(v(U))\} \circ (U = v(G))$

   Equation[] = $[E = g,\ v(M) = v(M)]$

vi. Now we examine $E = g$ (according to line no. 4.1 at the algorithm)

   substitution = $\{G = v(v(v(G)))\} \circ (E = g) \Rightarrow$ *this fails*

   Equation[] = $[v(M) = v(M)]$

Operation result is: FAIL (G is a circular expression and a substitution will end with an infinite loop).

c. `Unify [s ([v | [[v | V] | A]]),`
`s ([v | [v | A]])]`

Initialization:

sub: substitution = $\{\}$ // Empty substitution

equations: Equation[] = $[s\ ([v|[[v|V]|A]])\ =\ s([v|[v|A]])]$

i. Now we examine $v|[[v|V]|A] = v|[v|A]$

substitution = $\{\} \circ ([[v|V]|A] = [v|A])$

Equation[]= $[v|[[v|V]|A]] = [v|[v|A]]$

ii. substitution = $\{[[v|V]|A] = [v|A]\}$

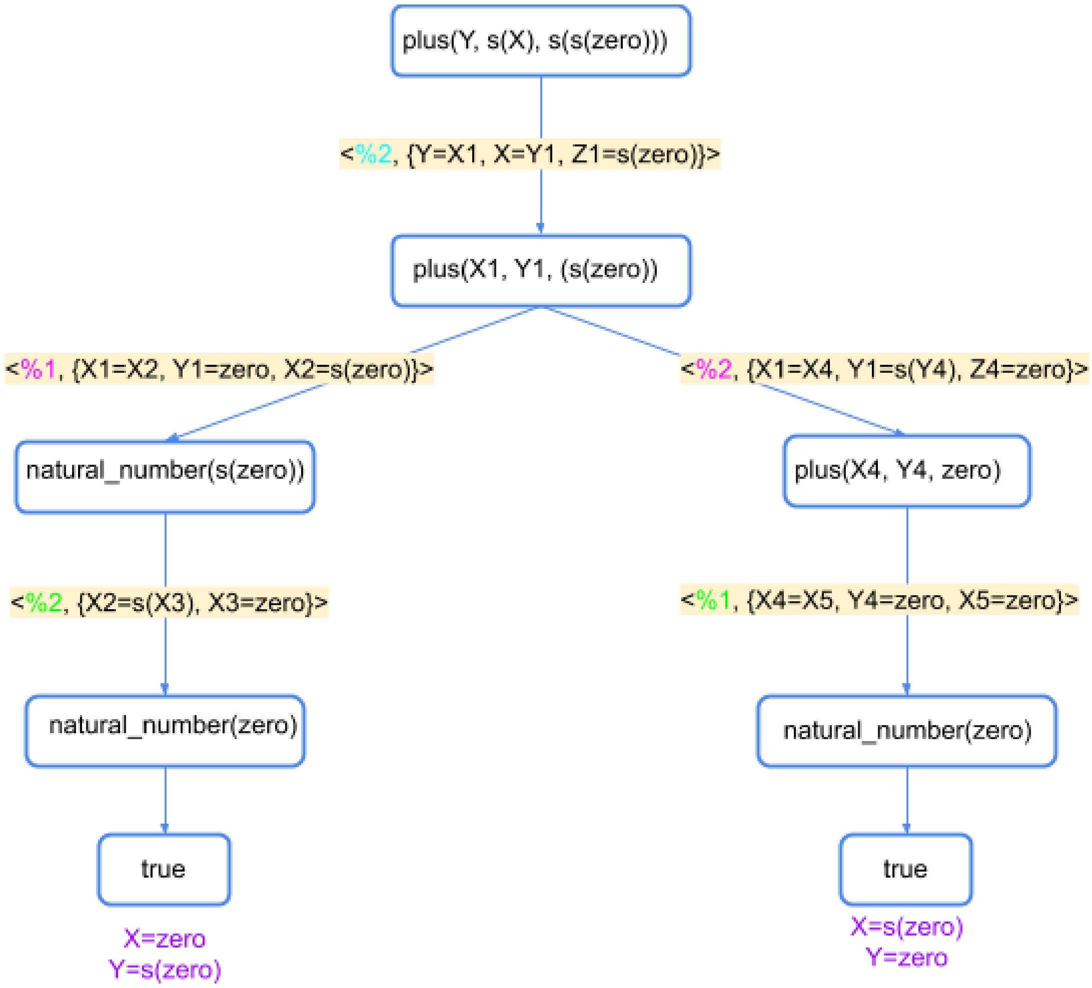
Equation[] = $[]$

iii. substitution = $\{[v|V] = v\}$ ⇒ $this\ fails$

Operation result is: FAIL ($v = [v|V]$ is illegal. A constant variable can't be a functor).

3.3 Proof Tree

a.



b. The answer of the answer-query algorithm is:
   - For the left path we got: $X = zero,\ Y = s(zero)$.
   - For the right path we got: $X = s(zero),\ Y = zero$.

   But in general, we wanted to find two natural numbers $X,\ Y$, such that our answer will be (Y = X's successor + zero's successor's successor).
   Zero's successor is 1, so 1's successor is 2 (according to church numbers).

c. This is a **success** proof tree.
   Explanation: we defined in practical session that a success tree is a tree that has at least one success path. In our case, all paths end with a success.

d. This tree is **finite**.
   Explanation: we defined in practical session that an **infinite** tree is a tree that has at least one infinite path (caused as a result of the order of goals). In our case, all paths are finite.