עבודת בית 2

מגישים: עומרי גולדברג 208938985, נעה חרדון 312164668

Part 1:

Q1.1- Special forms are s-expressions which are processed by they own rules, i.e indicates that a special operation must be performed by the interpreter to evaluate the form. It is a compound expression which is not evaluated like regular compound expressions.

If we evaluate a special form in standard way, then it might lead to errors while evaluating the expressions. For example, the expression (define x 10). If we evaluate the define in a standers way, then the variable x would have to be evaluated- which could lead to an error (because the whole point of define is to give a value to the variable!).

Q1.2- Expressions that can be done in parallel:

(define x 10) (define y (+ 6 7))

(- 10 1)

Expressions that can't be done in parallel: when there are dependencies between expressions

(define x 10)

(define y 5)

(define z (+ (* x 2) y))

Q1.3- Every program in L1 can be transformed to a program in L0 in, we can use values directly instead of defining them. The transformation will be the following way: We can replace every instance of define x y in L1 by passing over all of the program and replacing x with y. We will do it until we have passed over all of the defines from top to bottom (every variable will be replaced by its value from the define form). This will work because we don't have closures in L1 and every L1 program is finite.

Q1.4- Not every program in L2 can be transformed to a program in L20. Example :

(Define x (lamda(y) (x y))) (x x), or a recursive function (for example: the Fibonacci recursive function)

Q1.5- Map- If there is a dependency between the elements in the list, or a dependency between the list and the function that is giving to map, we should execute the procedure sequential.

Example: (define map (lambda (f s) (if (empty? s) '() (cons (f (car s)) (map f (cdr s))))))

(define f(lambda (x)(x+s[0])))

But if there are no dependencies between the elements or the function we should execute the procedure parallel.

Filter- should be executed parallel- because for each item in the list the pred return true or false , it doesn't change the elements in the list.

Reduce- should be executed sequential because in this function there is a dependency between the elements. For each element the function is based on the acc which is calculated from the elements before

All- should be executed parallel-similar to filter, because for each item in the list, the pred return true or false, it doesn't change the elements in the list.

Compose – should be executed sequential, because they might be dependences between the procedures.

Q1.6- The output is- 12, the line ((lambda (c) (p34 'f)) 5) returns the procedure lambda () (+ a b c) and the values are: a = 3, b = 4 from the definition of the pair (define pair (class (a b) …) and the value of c is 5 because we run the lambda procedure with the value 5 so it does 3+4+5 = 12.

Part 2:

Q2.1

```
; Signature: append(lst1, lst2)
; Type: [list<T1>*list<T2>->list<T3>]
; Purpose: Returns the concatenation of lst1 and lst2.
; Pre-conditions: lst1 and lst2 are lists.
; Tests: (append '(1 2 3) '(4 5)) ->'(1 2 3 4 5)

(append '(1 2 3) '()) ->'(1 2 3)
```

Q2.2

```
; Signature: reverse (lst)
; Type: [list<T> -> list<T>]
; Purpose: Returns the elements of lst in reversed order.
; Pre-conditions: lst is a list.
; Tests: (reverse '(1 2 3)) ->'(3 2 1)
```

Q2.3

```
; Signature: duplicate-items (lst, dup-count)
; Type: [list<T>*list<number> -> list<T>]
```

; Purpose: duplicates each item of lst according to the number defined in the same position in dup-count.
; Pre-conditions: lst is a list, dup-cound is a list of non-negative numbers and can't be empty.
; Tests: (duplicate-items '(1 2 3) '(2 2 2)) ->'(1 1 2 2 3 3)

(duplicate-items '(1 2 3) '(1 2)) ->'(1 2 2 3)

; Signature: duplicate-item-helper (element, n)
; Type: [T*number -> list<T>]
; Purpose: Returns element duplicated n times as a list
; Pre-conditions: n is non- negative number (n>=0)
; Tests: (duplicate-item-helper 1 2) ->'(1 1)

Q2.4

; Signature: payment (n, coins-lst)
; Type: [number*list<number> -> number]
; Purpose: Returns the number of possible ways to get the sum- n with the list of coins.
; Pre-conditions: n is a number, coins-lst is a list of numbers
; Tests: (payment 5 '(1 1 1 2 2 5 10)) ->3

; Signature: remove-dup (element, lst)
; Type: [number*list<number> -> list<number>]
; Purpose: Returns the lst without the element.
; Pre-conditions: element is a number, lst is a list of numbers
; Tests: (remove-dup 1 '(1 1 1 2 2 5 10)) ->'(2 2 5 10)

Q2.5

; Signature: compose-n (f, n)
; Type: [[number->number]*number -> [number->number]]
; Purpose: Returns the closure of the n-th self-composition of f.
; Pre-conditions: f is an unary function, element n is a non negative number (n>0)
; Tests: (define mul8 (compose-n (lambda (x) (+ 2 x)) 2))
(mul8 3) → 7