Avishai Yaniv 307916023
Alex Abramov 314129438

Question 1

1.1:

- **Primitive atomic expression:** `the expression 1`
- **Non-primitive atomic expression:** `the expression x in:`
```
(define x 1)
x
```
- **Non-primitive compound expression:** `(define yy (lambda(x) (+ x x )))`
- **Primitive atomic value:** `the number 1`
- **Non-primitive atomic value:** `1 is the value in:`
```
(define x 1)
x -> 1
```
- **Non-primitive compound value:** `'hello (value of 'hello)`

1.2:

- A special form is a compound expression which is not evaluated like regular compound expressions, thus they have their own rules.
  - (if cond then-part else-part) is a special form: `(if (= x 0) 1 (/ 1 x))`

1.3:

- x is called a free variable in expression E iff x has reference in E and there is no declaration for x in E.
  - y is a free variable in: `(lambda (x) (+ x y))`

1.4:

- Symbolic expression is a nested list of data which is used by some programming languages to represent programs data.
  - ['1','2'] is the s-exp of (1,2)

1.5:

- Syntactic abbreviation is a syntax which makes code easier to read or to express.
  - let- `(let ((a 5) (b 6)) (+ a b))`
  - define- `(define (average x y) (/ (+ x y) 2))`

1.6:

- Assuming p1 is a program written in L1. Let us define p0, a program that has no "define" expressions and that for every input x, if p1 halts, throws exception, or doesn't halt, p0 throws exception, or doesn't halt respectively and if halts, p1(x) = p0(x).
  Basically, all left to prove is that there is a way to transform every "define" statement into the appropriate solution in L0. As defined in previous question, define is just making code easier to read. For each '(define x codeX)' statement in p1, we replace 'x' instances related to this x (not including bounded variables named x) with codeX. Now we have the same program equivalent to p1, named p0 written in L0.

1.7:

- We will show a contradictory example.
  Let us define fibonacci procedure in L2, which can get any natural number as an argument.

```
(define fibo
  (lambda (n)
    (if (eq? 0 n) 0
        (if (eq? 1 n) 1
          (+ (fibo (- n 1)) (fibo (- n 2)))))))
```

As we can see, in the last line we have a recursive call. To make this call done properly, there is a binding of *fibo* onto the defined procedure. Let us assume that there's a way to solve Fibonnaci number calculation in L20. It can be made by calculating iteratively the numbers, since there are no calls to other procedures/variables (no define arguments). Assuming we want to calculate fibo(5), we will have to calculate fibo(5) + fibo(4) + fibo(3) + … + fibo(0) = 0. In L20 we'll have whether to make an assumption (to know the values of prior calculations) or to calculate 0 + 1 +... + 5. Let us assume that fibo(n) in L20 can calculate fibo(5), and now we want to calculate fibo(6), we will need to change the procedure, and therefore fibo(n) in L20, it's not capable of making calculation for each n without changing the procedure for each calculation.

1.8:

- PrimOp advantage- supported operators which are part of the language and do not need any new procedures addings and definitions.
- Closure advantage- special and complicated procedures can be formed into a lambda expression to make code readable, reusable and effective.

1.9:

- In case of map, and under assumption the the original order is being kept, it makes no difference iterating from beginning or ending since we only apply the given procedure on all list indexes. So for each list l1, original map(l1) equals to the modified map(l1).
- In case of reduce we will show an example for different returned values.

```
(define li1 (list 1 2 3) )
(define val 0)
(define proc (lambda(x  z) (/(- z 1 ) x)))

(reduceB proc val li1)  ;; -2/3
(reduce proc val li1)   ;; -5/3
```

As we can see, we have chosen a mathematical phrase which has order importance, thus the differ in evaluation returns different values.

Question 2 - Contracts

2.1:

- ; Signature: empty?(lst)
- ; Type: [List(Symbol) -> Boolean]
- ; Purpose: Check whether a given list is empty
- ; Pre-conditions: true
- ; Tests: (empty? ‘()) -> true
- (empty? ‘(1 2 3)) -> false

2.2:

- ; Signature: list?(lst)
- ; Type: [List(Symbol) -> Boolean]
- ; Purpose: Check whether a given argument is a list
- ; Pre-conditions: true
- ; Tests: (list? ‘()) -> true
- (list? 1) -> false

2.3:

- ; Signature: equal-list?(lst1 lst2)
- ; Type: [List(Symbol) * List(Symbol) -> Boolean]
- ; Purpose: Check if two lists are equal
- ; Pre-conditions: true
- ; Tests: (equal-list? ‘() ‘()) -> true
- (equal-list? ‘() ‘(1 2)) -> false

2.4:

- ; Signature: append(lst1 lst2)
- ; Type: [List(Symbol) * List(Symbol) -> List(Symbol)]
- ; Purpose: Create a new list from two given lists, containing both of them
- ; Pre-conditions: lst1 is a list && lst2 is a list
- ; Tests: (append '(1 2 3) '(4 5 6)) -> ‘(1 2 3 4 5 6)

2.5:

- ; Signature: append3(lst1 lst2 num)
- ; Type: [List(Symbol) * List(Symbol) * Number -> List(Symbol)]
- ; Purpose: Create a new list from two given lists and a symbol, containing three of them
- ; Pre-conditions: lst1 is a list && lst2 is a list && num is number
- ; Tests: (append '(1 2 3) '(4 5 6) 7) -> ‘(1 2 3 4 5 6 7)

2.6:

- ; Signature: pascal(n)
- ; Type: [Number -> List(Number)]
- ; Purpose: Calculate Pascal’s Triangle
- ; Pre-conditions: n >= 0
- ; Tests: (pascal 5) -> '(1 4 6 4 1)

- ; Signature: choose(n k)
- ; Type: [Number -> Number]
- ; Purpose: Calculate Binomial coefficient
- ; Pre-conditions: n >= 0
- ; Tests: (choose 5 2) -> 10

- ; Signature: better_pascal(n k)
- ; Type: [Number * Number -> List(Number)]
- ; Purpose: Calculate Pascal's Triangle
- ; Pre-conditions: n >= 0 && k>= 0
- ; Tests: (better_pascal 4 0) -> '(1 4 6 4 1)