SOLUTION

By

Avi Ferdman

Q1.1

a. primitive atomic expression – #t
b. Non-primitive atomic expression – variable x
c. Non-primitive compound expression – (* 3 8)
d. Primitive atomic value – Booleans value: true
e. Non-primitive atomic value – symbol ‘blue
f. Non-primitive compound value – list (1, 2, 3, 4), the return value of app expression: (+ 2 4) -> **6**

Q1.2

Special form are s-expressions which **processed by their own rules.**

This form is written with parenthesis which the first word is a special operator.

Define is a "special form": it is compound expression which is not calculates like regular compound expressions.

Example:

(define x 3)

Q1.3

A variable x **occurs free** in an expression E if and only if there is some use of x in E that is not bound by any declaration of x in E.

Example: ((lambda (x) x) y) -> y occurs free

Q1.4

S-Exp – is the literal expression in our languages, composed from tokens, that let us define combinations of values and hierarchy between them. Examples for S-Exp in L3:

Example1:

```
['if',['>','y','3'],'#t','#f']
```

Q1.5

Syntactic abbreviation – syntactic shortcut which can be replaced by expressions that already exists in the language ("syntactic sugar").

Examples for syntactic abbreviation in L3:

1) Let expressions

```
(let ((a 1)) (f a))
```

Is interpreted to:

```
((lambda (a)(f a)) 1)
```

2) Cond expressions

```
(if (x>0) (* x 2) (* x 3))
```

Is interpreted to:

```
(cond ((x>0)(* x 2))
 (else (* x 3)))
```

Q1.6

No, **there is no program in L3 which cannot be transformed to an equivalent program in L30.**
The reason for this is that **every operation on lists can be transformed equivalently to operations on pairs** (assuming we have empty list '()), we can do this by look at every list as a pair, the first operand of the list is the first operand in the pair and the rest of the list is the second operand in the pair, and the second cell in the last pair is the empty list.
Therefore, **every list can be interpreted recursively to be represented as pair.**

Q1.7

PrimOp – the evaluation of the operations is faster than in closure, because that we don't need to check the environment every time we calculate the value.

Closure – the implementation is simpler, because it uses an existing structure in the interpreter, therefore in order to change a primitive in a closure, all it takes is to add a primitive to the interpreter and to update the global environment.

Q1.8

Map

Assuming the order of cells in the array in both the operation of activating the list from the beginning and from the end is equivalent the result would be equal, the
reason is that “map” operates on each cell separately and return a new array, there for there is no meaning for the order.

Reduce - there is difference, because every iteration depends on the previous iterations,
for Example:

[10, 15, 20].reduce((acc: number, curr: number)=>acc<20? acc += curr ,0)

If we will operate “reduce” from the beginning we will get the value: 25, however if we will operate it from the beginning, we’ll get the value: 20.

Filter - There is no difference, the “filter” function operates separately on each cell, therefore, there is no meaning for the order of the iteration.

Compose - There is a difference,

Example:

(((x)=>x*0, (x)=>x+2)).compose()

If we will operate “compose” from the beginning on the number 3 for example, we will get 5, however if we will operate from the end, we will get 0.