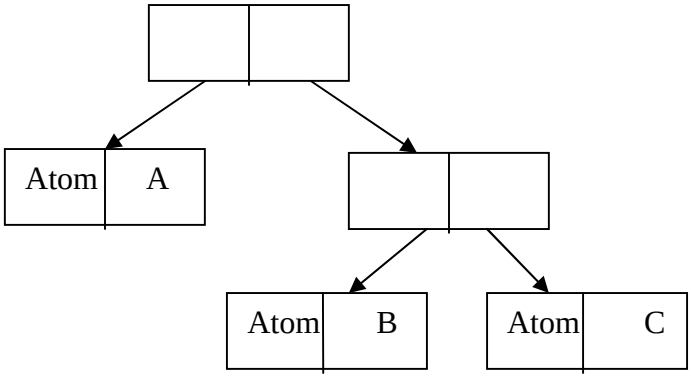
1.

a)



b)

(cons(cons ‘A (cons ‘B ’C))(cons ‘B ‘C))

At first, the most inner con cell will be made. Secondly this expression : (cons ‘A (cons ‘B ’C) and at the end the whole expression will be made.

c)

$((lambda\ (x)\ (cons\ (\ cons'A\ x)(x))\ (cons'B'C))$

Firstly, $(cons'B'C)$ will be evaluated . After that, function will apply on it.

2.

a) It's not possible, because there is no way to understand that an expression is undefined. We should continue executing, if an expression is undefined the system will halt and we won't get that this expression is undefined to skip.

b) In this method we must evaluate all expressions in a parallel way. If one of them evaluates true, it will determine the return value of cond and if none of them returns true the result will be undefined. Thus, we can do each evaluation in different threads and if one of the threads got stuck or halts, others are evaluating an expression and also returning one of the expressions.

c)

```
(defun Odd(x)
(cond ((eq x 0) nil)
((eq x 1) true)
((> x 1) (Odd (- x 2)))
((< x 0) (Odd (+ x 2)))
) )
```

d)

It's better to use “a” for “short circuit or”, because expressions evaluated sequentially there, however “b” is not suitable since even one true expression causes the whole expression evaluates true, which is not fine because it might be undefined.

It is much easier to implement POR using “b” since expressions are evaluated in parallel there. Nevertheless, it is difficult using “a” because by using “a” if “e1” evaluated undefined , the whole expression will evaluate undefined, however we do not expect it in POR.

4)
a)

```
(define compose2
        (lambda (g h)
                (lambda (f xs)
                        (g (lambda (xs) (f (h xs))) xs )
)))
```

b.) g replaced by maplist , and “h” by “car”

c.) lambda (xs) (f (h (xs)))) = compose (f h)

5)

a) No, imagine a list that is not accessible by our current program from a specific point, however there are some programs which have access to it . In this scenario, according to Mccarthie's algorithm , this list will not recognize as garbage , but according to formal definition of garbage , the "list" is a garbage.

b)

Yes, imagine we have a garbage location based on McCarthie's definition. It cannot be reached by a car-cdr chain from any base register .Moreover its contents can no longer be found by any possible program so no continued execution of any program from this point can access it.Thus, it considered as garbage in our definition too.

c)

No, because we cannot predict which one of the memory locations will not be accessed in different scenarios of a given program in compile time.

6)

a) All of the three cons cells generated by the given expression are garbage.

b) ((Replaca (cons (cons (a b)cons (c d)) (e f))) , In this example cons(a b) has a counter and its counter does not change after applying "replaca" to it. Thus , reference-counting algorithm does not work properly on this structure.

8)

a) $t(n) = \max (t(n-1), t(n-2)) + 3$

$= t(n-1) + 3$

$\Rightarrow t \in O(n)$

b) cons(rplcd x 'a) (cdr x)

cons(future(rplcd x 'a)) (cdr x)

In the second expression "cdr x" is evaluated earlier than "rplcd x 'a" , however in the first one it is vice versa , which means firstly cdr of the list "x" will change and it affects the following expressions.

c) In the following example that is written by pure lisp , although we have not got any side effect expression , in (I)

I. cond((true P)(true V)) → P
II. cond((future(true) P)(true V)) → V

d)

1. we cannot consider a priority to different errors
2. Imagine lots of errors exist in a given expression e. So, in different scenarios, it might generate a different error, that is, in scenario ‘a’ when the program executes, handler ‘A’ will execute, and in scenario ‘b’ when the program executes, handler ‘B’ will execute.