3.I

Объектно-ориентированном программировании в Scheme состоит из функций и состояний:
У функции есть
- параметры и тело описанные в lambda-выражении
- окружение, где хранятся связывания её имён
Можно использовать функцию для хранения (и сокрытия) данных в локальных переменных и предоставления доступа к ним.
При вызове функции-конструктора создаётся новое окружение.
Нужно иметь доступ из функции-объекта к этому окружению:
- для чтения нужны операции-селекторы (или геттеры);
- для изменения состояния нужны мутаторы;
- состояние - это текущие связывания в кадре, созданном при вызове конструктора

Пример описания нетривиального класса на Scheme
```
(define (create-thing name location)
    (define (this msg)
        (cond
            ((eq? msg 'get-name) name)
            ((eq? msg 'get-location) location)
            (else (error "WRONG MESSAGE"))))
        (begin (location 'put-thing! this) this))
```

На Racket
```
(define thing%
    (class object%
        (super-new)
        (init-field name location)
        (define/public (get-location) location)
        (define/public (get-name) name)
        (send location put-thing! this)
))
```

Основные преимущества ООП:
- Модульность - более структурированный код
- Гибкость - код легче дополнять и понимать
- Экономия времени - благодаря абстрациям и т.п.

3.II
Если в ходе процесса возникает цепочка отложенных вычислений, то процесс рекурсивный.
Если в ходе процесса отложенных вычислений нет, то процесс итеративный

Рекурсивный процесс -
```
(define (my-reverse lst)
    (if (null? lst)
        '()
        (append (my-reverse (cdr lst)) (list (car lst)))))
```

Итеративный процесс -
```
(define (reverse2 lst)
    (define (loop lst result)
```

```
        (if (null? lst)
            result
            (loop (cdr lst) (cons (car lst) result))))
    (loop lst '()))
```

Описание рекурсивного процесса больше, однако с прагматической точки зрения более оправдано использование итеративного процесса, так как во время него не возникает отложенных вычислений. Что в конечном итоге влияет на потребляемый объем памяти

3.III
Макросы syntax-rules
- Закрытость. Макросистема отделена от Scheme. Во время подстановки нельзя запустить какой-либо код или использовать какое-то значение.
- Гигиена. Макрос не портит непредсказуемым образом окружения, в которых происходит подстановка.
- Прозрачность ссылок. Окружение, в котором происходит подстановка не портит макрос.
- Язык образцов и язык шаблонов используются для описания структуры макрокоманд и «тел» макросов, соответственно

уместно использовать макросы:
- макросы для изменения порядка вычислений (определения собственных спецформ).
Ситуации, когда макросы необхоимы:
1) условные вычисления (аналоги cond, case, and, or)
2) циклы (аналоги именованного let, do)
3) связывания (аналоги set!, let, let*)
4) используются не вычисляемые имена (=>)

неуместно использовать макрос:
Всегда, когда без него можно обойтись
1) Макросы в отличие от функций не являются объектами первого класса.
2) Макросы затрудняют отладку.

Попробуем описать функцией обмен значений двух переменных swap:
```
(define (swap! x y)
    (let ((c x)) (set! x y) (set! y c)))
```
У нас ничего не получилось, потому что в функцию передются локальные имена и как раз они меняются

Для того чтобы все получилось воспользуемся макросом
```
(define-syntax swap
    (syntax-rules ()
        ((swap a b)
        (let ((c b))
            (set! b a)
            (set! a c)))))
```